Coordinated Scheduling and Dynamic Performance Analysis in Multiprocessor Systems

Julita Corbalán González

Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya (UPC)

Barcelona (SPAIN)

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor per la Universitat Politècnica de Catalunya

Coordinated Scheduling and Dynamic Performance Analysis in Multiprocessor Systems

Author: Julita Corbalán González

Advisor: Jesús José Labarta Mancho

Co-Advisor: Xavier Martorell Bofill

Approved, Thesis Committee

.....

.....

.....

.....

.....

Its time...

to design and build computing systems capable of running themselves, adjusting to varying circumstances, and preparing their resources to handle most efficiently the workloads we put upon them. These autonomic systems must anticipate needs and allow users to concentrate on what they want to accomplish rather than figuring how to rig the computing systems to get them there.

AUTONOMIC COMPUTING
IBM's Perspective on the State of Information Technology
IBM 2001

Abtract

The performance of current shared-memory multiprocessor systems depends on both the efficient utilization of all the architectural elements in the system (processors, memory, etc), and the workload characteristics. This Thesis has the main goal of improving the execution of workloads of parallel applications in shared-memory multiprocessor systems by using real performance information in the processor scheduling.

In multiprocessor systems, users request for resources (processors) to execute their parallel applications. The Operating System is responsible to distribute the available physical resources among parallel applications in the more convenient way for both the system and the application performance.

It is a typical practice of users in multiprocessor systems to request for a high number of processors assuming that *the higher the processor request, the higher the number of processors allocated, and the higher the speedup achieved by their applications*. However, this is not true. Parallel applications have different characteristics with respect to their scalability. Their speedup also depends on run-time parameters such as the influence of the rest of running applications.

This Thesis proposes that the system should not base its decisions on the users requests only, but the system must decide, or adjust, its decisions based on real performance information calculated at run-time. The performance of parallel applications is an information that the system can dynamically measure without introducing a significant penalty in the application execution time. Using this information, the processor allocation can be decided, or modified, being robust to incorrect processor requests given by users. We also propose that the system use a target efficiency to ensure the efficient use of processors. This target efficiency is a system parameter and can be dynamically decided as a function of the characteristics of running applications or the number of queued applications.

We also propose to coordinate the different scheduling levels that operate in the processor scheduling: the run-time scheduler, the processor scheduler, and the queueing system. We propose to establish an interface between levels to send and receive information, and to take scheduling decisions considering the information provided by the rest of levels. In particular, we propose that the processor scheduler decides when a new application can be started and let the decision about which application to start to the queueing system.

The evaluation of this Thesis has been done using a practical approach. We have designed and implemented a complete execution environment to execute OpenMP parallel applications. We have introduced our proposals, modifying the three scheduling levels (run-time library, processor scheduler, and queueing system): At the run-time level we have implemented some techniques to improve the run-time behavior in a multiprogrammed multiprocessor system, including the coordination with the O.S. scheduler. We have also proposed a mechanism to dynamically measure the performance of parallel applications. At the processor scheduling level, we have mainly proposed several scheduling policies to include the performance information in the processor allocation policy. We have also specified the mechanism to coordinate the processor scheduler with the queueing system. At this level we have also done proposals to work both in space-sharing and in gang scheduling policies. At the queueing system we have mainly incorporated the coordination with the processor scheduling level.

Results show that the ideas proposed in this Thesis of (1) measuring the applications performance at run-time to decide and / or adjust the processor allocation, (2) imposing a target efficiency to ensure the efficient use of resources, and (3) coordinating the different scheduling levels, significantly improve the system performance. If the evaluated workload has been previously tuned, in the worst case, we have introduced an slowdown around 5% in the workload execution time compared with the best execution time achieved. However, in some extreme cases, with a workload and a system configuration not previously tuned, we have improved the system performance in a 400%, also compared with the next best time.

The main results achieved in this Thesis can be summarized as follows:

- The performance of parallel applications can be measured at run-time. The requirements to apply the mechanism proposed in this Thesis is to have malleable applications and shared-memory multiprocessor architectures.
- The performance of parallel applications must be considered to decide the processor allocation. The system must use this information to self-adjust its decisions based on the achieved performance. Moreover, the system must impose a target efficiency to ensure the efficient use of processors.
- The different scheduling levels must be coordinated to avoid interferences between levels.
- Malleability is a desired application characteristic that benefits both the application and the system. The application because applications do not have to wait for a certain amount of resources to become available, and to the system because it can better distribute the available resources among applications.

Dedicatoria

A mis padres y a Jose

Agradecimientos

Es viernes, son las siete, y llevo desde las tres mirando ficheros en C. Creo que lo mejor es que lo deje y dedique la poca capacidad que me queda a escribir algo que sea fácil, como por ejemplo agradecerle a todas las personas que han hecho que esta tesis tenga sentido.

Podría empezar por mis directores de tesis como la mayoría de la gente pero creo que el primer lugar se lo merecen mis padres y mi marido. Ellos son mi vida y por lo tanto esta tesis es tan suya como mía. A ellos no se lo agradezco, se la dedico. A mi familia en general, a mis hermanos, a mis suegros, a mis cuñados, en fin, a todos ellos gracias por hacer que todo esto tuviera sentido y apoyarme siempre, espero que sientan que en parte esta tesis es suya también. A algunos amigos muy especiales, a Julio, Mari, y Rubén (su peque), ellos también forman parte de mi familia.

Ahora si que podemos pasar al ámbito más "profesional". Por supuesto se lo quiero agradecer a mis dos directores de tesis, a Jesús y a Xavi. A Jesús más que agradecerle que confiara en mi, que sería lo típico que podría decir, le agradezco que no me haya hecho perder el tiempo, y el poder trabajar con él, ha sido muy constructivo. A Xavi tendría que dedicarle un capítulo especial en mis agardecimientos, porque cuando llegué a las mil horas que me había dedicado dejé de contar.

También, aunque no sea mi jefe, me gustaría agradecerle a Nacho que confiara en mi, ya que es posible que sea gracias a él que yo haya acabado siendo "de sistemas". Me gustaría agradecerle también a Mateo que me echara una mano en algunos momentos, y también algunas palmaditas en la espalda que a veces vienen bien.

Y si repaso por orden cronológico podría nombrar a mis compañeros del proyecto NANOS, a todos en general, Marc, Toni, Eduard, Nacho, Jesus, Xavi, empiezo a repetirme pero es que a algunos tengo mucho que agradecerles. A todos ellos tengo que agradecerles que siempre han estado dispuestos a escuchar mis rollos, a veces incluso voluntariamente :-). A algunos creo que incluso les debo parte del nombre de mi tesis. Si no recuerdo mal Toni me inspiró el nombre de mi primer artículo publicado.

A partir del momento en que me decidí a hacer la tesis también empecé a tratar con otros compañeros que al final se han convertido en una constante diaria: Ernest, Xavi, Jesus (Corbal), Daniel, Ernest, Fernando, Ayose, Oliver, Fran, Xavi (Verdu). Algunos hace poco tiempo que los conozco pero otros han dejado ya su huella. Por ejemplo Jesus, gracias a él se que existe la palabra *hilarante* (y que sus gustos por el cine no coinciden con

los mios). A Daniel le quiero agradecer que haya aumentado mi paciencia hasta niveles que rozan el pasotismo, ahora la vida es más fácil. Ernest, otro "de sistemas", un apoyo a la hora de comer para que "los otros" no nos coman. Xavi es el mismo Xavi de siempre es que no se como me aguanta hasta para comer, creo que queda claro que le debo mucho :-). Todos son buenos amigos.

Algunos compañeros también han contribuido con su granito de arena, Por ejemplo Álex, que me dejó una herramienta de lo mas útil para hacer gráficas, ay, si la hubiera conocido antes. Felix, que también ha puesto su granito de arena con otra útil librería. Otros que ya no están, no es que les haya pasado nada es que se han ido a la privada,como Albert, a él le debo un trocito de mi tesis, sus *dituls* son muy útiles, si algún dia lee esto que lo sepa. También a otros que han desarrollado algunas herramientas que sin ellas hubiera sido una tarea mucho más complicada evaluar la tesis, como a Nacho, gracias a su scpus he podido sacar unas trazas muy útiles. Y como no a Jordi Caubet, que ha tenido una paciencia infinita conmigo.

A mi amigo Pepe, que cuando estuvo aquí haciendo tesis fue el mejor amigo, cuando se fue a Murcia fue el mejor amigo y cuando ha vuelto sigue siendo el mejor amigo,pero ahora tiene más secretos porque es un chico Intel :-). En fin, gracias Pepe.

También por supuesto a gente de esa que está en la sombra, que parece que no tienen nada que ver, pero que si no fuera por ellos sería más díficil. En general a todos los de sistemas, en especial a Victor, que son mucho años :-), y a los del CEPBA, que han tenido mails mios a diario durante unos cuantos años.

This work has been supported by the Spanish Ministry of Education under grant CYCIT TIC98-0511, TIC2001-0995, the ESPRIT Project NANOS (21907) and the Direcció General de Recerca of the Generalitat de Catalunya under grant 1999FI 00554 UPC APTIND. The research described in this work has been developed using the resources of the European Center for Parallelism of Barcelona (CEPBA).

Index

CHAPTER 1

Introduction

Abstract

This Thesis focuses on the efficient execution of workloads of parallel applications in multiprogrammed multiprocessor environments. In particular, we will defend two main ideas: the first one is that all the scheduling levels must be coordinated to achieve a good system performance. The second one is that the processor scheduling must consider real performance information to decide the processor allocation, and to impose a target efficiency to running applications to ensure the efficient use of resources.

In this Chapter, we introduce the main subjects of this Thesis, its motivation, and our contributions.

1.1 Introduction

Multiprocessor architectures appeared as the natural extension to uniprocessor systems. The common characteristics among all the multiprocessor systems is that they have multiple processors that may be used to execute multiple applications at the same time, one application in multiple processors (parallel application), or combinations of the two cases (multiprogrammed systems that execute parallel applications).

The first approach to schedule concurrent applications on these systems was to directly apply uniprocessor policies extended to the case of several processors. However, users, system administrators, and researchers, quickly observed that uniprocessor policies do not exploit the potential of these systems.

The problem was to consider that a multiprocessor system had the same characteristics and problems that a uniprocessor system. Multiprocessor systems have their own goals, applications, and architectural characteristics, and they must be taken into account to schedule, not only processors, but any physical resource. *Goals*, because multiprocessor systems are oriented to increase the throughput of the system and the speedup of individual applications. *Applications*, because parallel applications have frequent synchronizations. These synchronizations imply that the delay of some of the processes can result in a delay of the complete application. *Architectural characteristics*, because multiprocessors have two elements that in most of the cases determine the application and the system performance: the memory system and the interconnection network. In multiprocessor systems, the different memory and network organizations have a direct effect in how applications must be scheduled.

For these reasons, since multiprocessor systems appeared, the scheduling of applications has been an important research subject in this kind of systems, from the point of view of job scheduling, to the point of view of run-time libraries that schedule parallel loops.

As we have commented, all the components of the system (processors, memory, network, and I/O) influence in the performance of a multiprocessor system. In this Thesis, we will focus in the problem of how to schedule workloads of parallel applications in shared-memory multiprocessor systems, taking into account all the elements of the system, but focusing in the processor scheduling.

1.2 The scheduling problem

The scheduling problem consists of how to assign physical processors to application threads. The scheduling problem can be divided in three levels: job scheduling, processor scheduling, and loop scheduling.

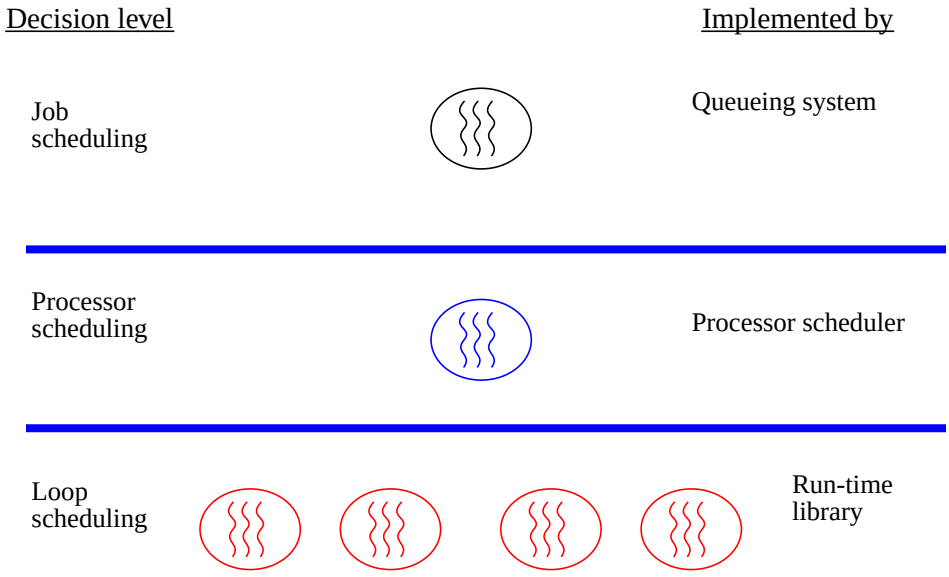


Figure 1.1: Scheduling levels

Figure 1.1 shows the three scheduling levels. The first one, the most external, is the job scheduling. At this level the problem consists of deciding which job should be executed. This level of decision is implemented by the queueing system. These decisions are taken at a low frequency compared with the other levels. The second decision level is the processor scheduling. It decides how many processors to allocate at any moment to each running application. This level of decision is implemented by the processor scheduler. The last one is the loop scheduling problem, and it decides how to distribute the computation among the processors allocated to the application. It is implemented by the run-time library that controls the application parallelism. This very short-term scheduler is normally not considered by the O.S, but it is a responsibility of the application itself.

In commercial systems, these three levels are loosely coordinated. Decisions taken by each level are taken without cooperation with the others levels. This behavior generates situations such as applications running with more kernel threads than available processors, or that there are free processors and queued applications at the same time.

In some previous research works, it was proposed an interface between the processor scheduling level and the loop scheduling level to adjust the number of running threads to the number of physical processors. In some of these proposals, the run-time level

informs the processor scheduler about the number of processors requested, and the processor scheduler informs the run-time about the number of processors allocated to the application. With this first level of interaction, the overall performance was significantly improved. These proposals are described in Chapter 2.

1.3 Our Thesis

In this Thesis, we propose to extend other proposals with three main points:

- To achieve the best overall system performance, and the best individual application performance, **all the scheduling decisions must be coordinated**. That means, to provide an interface between the queueing system and the processor scheduler, and an interface between the processor scheduler and the run-time library allowing such coordination.
- **It is necessary to include real performance information** in the processor scheduling decisions. We consider real performance information those values measured at run-time.
- **It is necessary to impose a target efficiency to running applications** to ensure the efficient use of processors.

The first point is the coordination between levels. Coordination means that each scheduling level will provide information about its internal status to levels that communicate with it, and that it will receive information from them. Coordination also means that scheduling decisions taken at each level will consider all this information. Coordination will allow the system to avoid incorrect situations that degrade the system performance such as the ones commented previously, where there can be free processors and queued applications at the same time. Or the inverse situation, where the queueing system can start a new application when the system is heavily loaded.

The second point, the use of real performance information in addition to other user provided information, will allow the processor scheduler to improve its scheduling decisions. Our third point is not only to consider the application performance, but also to impose a target efficiency to be achieved by running applications. The goal of this target efficiency is to ensure the efficient use of resources. Not to consider the application performance could result in an inefficient processor allocation such as to allocate a small number of processors to a parallel application that scales very well and a lot of processors to a parallel application that does not scale at all. This last case even can result in an increment in the execution time of the application.

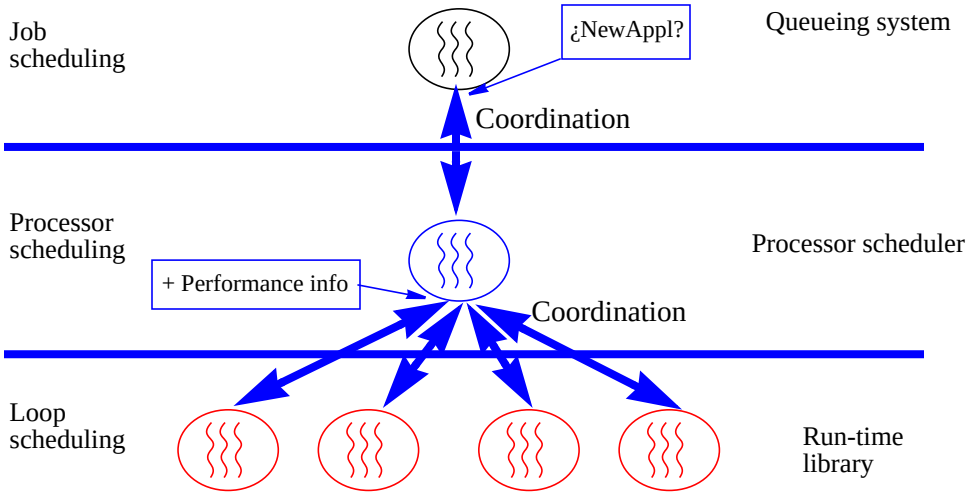


Figure 1.2: Coordinated scheduling

Figure 1.2 shows the main proposals of this Thesis: coordinate the three scheduling levels to improve the system performance, and include performance related information to decide the processor scheduling.

Some works have previously proposed to allocate processors as a function of the application performance. They always assume that application performance is known before the application execution, *a priori*. Details about these proposals are presented in Chapter 5. However, we believe that the system can not rely on users be neither experts nor honest. There are also some other related problems to the use of *a priori* information:

- Sometimes, it is not possible to evaluate parallel applications because their performance depends on input data and the number of combinations makes it impossible to calculate all the possibilities.
- Sometimes, the optimal number of processors for a particular application may not be optimal for the overall system performance, for instance if the load is very high.
- The performance of a high number of applications depends on run-time parameters such as the memory mapping, the number of process migrations, or the influence of the rest of running applications (concurrently executed).

To demonstrate our Thesis, we propose an execution environment with the following characteristics: The performance of parallel applications will be measured at run-time, the processor scheduler will impose a target efficiency to running applications to receive processors, and finally, parallel applications will be malleable to be able to react to the processor scheduling decisions.

1.4 Contributions of this Thesis

To demonstrate our ideas, we present a practical approach based on implementing mechanisms and policies in a real system. The particular contributions that demonstrate the main points of this Thesis are divided into three parts.

In the first part of this Thesis, we propose a complete execution environment that includes a run-time library that performs a dynamic performance analysis, and a new scheduling policy that incorporates the concepts of: use of real performance information, impose a target efficiency, and coordination with the queueing system.

The dynamic performance analysis is implemented by the **SelfAnalyzer**. SelfAnalyzer is a run-time library that measures the speedup of parallel applications at run-time, and also estimates the execution time of parallel applications. SelfAnalyzer exploits the iterative behavior that have a lot of parallel applications, which have a predictable behavior since they repeat the same code several times. The SelfAnalyzer measures the execution time of several iterations with different number of processors and calculates the speedup as the ratio between two of these measurements.

The new coordinated scheduling policy is called **Performance-Driven Processor Allocation Policy (PDPA)**. PDPA takes two decisions: the processor allocation and the multiprogramming level. Regarding the processor allocation, PDPA is a dynamic space-sharing policy that decides a processor allocation based on the performance of running applications, and imposes a target efficiency. With respect to the multiprogramming level, PDPA decides to increment the multiprogramming level when there are free processors and all the running applications have an stable allocation. PDPA will show us the potential and the benefits of a policy that considers application performance and ensures a target efficiency in parallel applications in front of policies that do not consider this point.

In the second part of this Thesis, we present a new methodology to incorporate these three points to any previously proposed processor scheduling policy. The goal of this methodology is to incorporate the concepts of (use real performance information / ensure target efficiency / coordinated scheduler) to other criteria exploited by other policies. With this aim, we present **Performance-Driven Multiprogramming Level** (PDML). We have applied PDML to two space-sharing policies: Equipartition and Equal_efficiency. We have named the resulting policies Equip++ and Equal_eff++. Results will show that after applying PDML, the resulting policies detect and correct situations where applications with bad performance were using a high number of processors due to inefficient allocations decided by the original policies.

In the last part of this Thesis, we incorporate the concepts of (use of real performance information / ensure target efficiency / coordinated scheduler) to a different set of policies, gang scheduling policies. Gang scheduling policies perform time-sharing among applications. Applications are grouped into slots, and at each quantum expiration the scheduler selects a new slot to execute in a round-robin way. Traditionally, these policies apply a

simple dispatch, that means that applications receive as many processors as they request. We will show that the ideas proposed in this Thesis are also valid in gang scheduling policies.

We propose two contributions to this kind of policies. In the first one, we propose to apply the PDML methodology to a traditional gang scheduling policy. We call the resulting policy **Performance-Driven Gang Scheduling** (PDGS). PDGS evaluates the performance of active applications every a certain quantum, and adjust their allocation if they do not reach the target efficiency.

The second proposal to improve gang scheduling is a new re-packing algorithm. Repacking algorithms decide how applications are grouped in the different slots. We propose the **Compress&Join** algorithm. The goal of Compress&Join is to reduce the number of slots, which is one of the main sources of overhead of gang scheduling policies. Compress&Join reduces the processor allocation of applications in a proportional way to the application performance. With this reduction in the processor allocation, it is possible to fit the same number of applications in a small number of slots. In both cases, PDGS and Compress&Join, we coordinate the processor scheduler with the queueing system to decide when a new application can be started.

1.5 Overview of the Thesis environment

Since we will demonstrate our ideas using a practical approach, we believe that it is important to briefly describe the characteristics of the Thesis execution environment. Figure 1.3 shows its main elements, and it is fully described in Chapter 3.

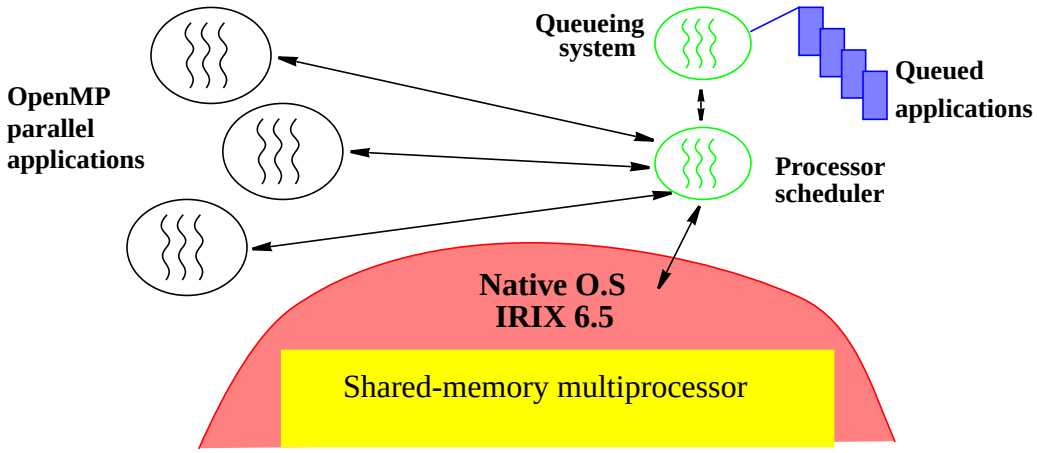


Figure 1.3: Thesis execution environment

This Thesis has been developed in an shared-memory multiprocessor environment. In particular, in a CC-NUMA machine with 64 processors. We have selected this architecture because of its availability, and because it is representative of systems with a medium-high number of processors used in commercial environments and supercomputing centers.

One of the characteristics of our proposed environment is that applications must be malleable to adapt their parallelism to the number of processors available. In this Thesis, we have used the OpenMP programming model because in OpenMP the parallelization does not only depend on the number of physical processors, but also it depends on the algorithm and the loop scheduling policy applied. With the OpenMP model, and the support of the run-time library, applications can be malleable. In this Thesis, we have used the NthLib as run-time library. The NthLib uses the processor scheduler interface to request for processors and to check the number of processors allocated to the application. The NthLib is able to react to changes in the number of processors allocated to the application.

The last elements in the execution environment are the processor scheduler and the queueing system. These two elements have been implemented at user level to implement and evaluate all the proposals presented in this Thesis. The processor scheduler implements the processor scheduling policy and enforces its decisions by means of the native operating system. As we have commented, it provides an interface used by the run-time library. In this Thesis, the processor scheduler also provides another interface used by the queueing system. The queueing system implements the job scheduling policy, that

decides when to start the execution of jobs submitted to the system. In our proposed execution environment, the processor scheduler will inform the queueing system about the convenience of starting a new application, and the job scheduling policy (implemented by the queuing system) will decide which particular application to start. In this Thesis, the job scheduling policy implemented is a FIFO, and the job selected is always the first queued job.

1.6 Organization of the Thesis document

This Thesis is organized as follows: Chapter 2 describes the main elements of a multiprocessor system: multiprocessor architectures, scheduling policies, and programming models. We focus on elements that are more related to this Thesis such as CC-NUMA architectures or space-sharing policies.

Chapter 3 presents the particular characteristics of our execution environment. We describe the queueing system, ***Launcher***, the processor scheduler, ***CPUManager***, and the improvements introduced in this Thesis in the run-time library, ***NthLib***.

Chapter 4 presents ***SelfAnalyzer***, a run-time library that dynamically calculates the performance of parallel applications.

Chapter 5 presents ***Performance-Driven Processor Allocation (PDPA)***, a coordinated scheduling policy that decides both the processor allocation and the multiprogramming level.

Chapter 6 describes ***Performance-Driven Multiprogramming Level (PDML)***, a new methodology that transforms previously proposed processor allocation policies to include job performance analysis and coordination with the queueing system.

Chapter 7 presents two new techniques based on the use of job performance analysis and job malleability to improve gang scheduling policies: ***Performance-Driven Gang Scheduling (PDGS)*** and ***Compress&Join*** algorithm.

Finally, Chapter 8 presents the conclusions and the future work of this Thesis.

CHAPTER 2

General Overview of Shared-Memory Multiprocessor Systems

Abstract

The performance of a multiprocessor system is determined by all of its components: architecture, operating system, programming model, run-time system, etc. In this Chapter, we focus on the topics related to multiprocessor architectures, scheduling policies and programming models.

We analyze the different proposals in multiprocessor architectures, focusing on the CC-NUMA SGI Origin2000, the platform for our work.

We also describe and classify the scheduling policies previously proposed in the literature. Some of them will be presented in more detail since they are more related with the work done in this Thesis. Others are presented only for completion.

Finally, we analyze the standard programming models adopted by users to parallelize their applications.

2.1 Introduction

In this Chapter, we give a general overview about three of the main elements in a multiprocessor system: architecture, operating system, and programming model.

We will give an insight about some of the multiprocessor architectures, focusing on the SGI Origin 2000 [93][93], which has been the target architecture in this Thesis. This machine is a shared-memory multiprocessor architecture and it is representative of multiprocessor machines with a medium-high number of processors used in commercial environments and supercomputing centers.

The second component, and the main point of this Thesis, is the operating system. In particular, this Thesis focuses on the part of the operating system that is in charge of distributing processors among applications: the scheduler. In this Chapter, we briefly present some of the proposals made in the last years on processor scheduling. We want to give a general overview about processor scheduling. More specific proposals related with the topic of this Thesis are distributed among the different chapters.

Finally, we also describe standard programming models used to parallelize applications: MPI, OpenMP, and the hybrid model MPI+OpenMP. In this Thesis, we have adopted the OpenMP programming model.

This Chapter is organized as follows. Section 2.2 describes the SGI Origin 2000 architecture. Section 2.3 presents the related work concerning to proposals to share processors among applications. Section 2.4 presents related work to coordination of scheduling levels. Section 2.5 presents related work to job scheduling policies and Section 2.6 related work to processor scheduling policies. Section 2.7 presents the standard programming models used to parallelize applications: MPI and OpenMP. Finally, Section 2.8 presents the summary of this Chapter.

2.2 Multiprocessor architectures

In this Section, we present a brief description about some multiprocessor architectures. A multiprocessor is a system composed by more than one processor. They appeared with the goal of improving the throughput of the system, executing more applications per unit of time, and to improve the execution time of individual applications, executing them with multiple processors.

There are two main trends in multiprocessor architectures: architectures that implement a global address space, shared-memory multiprocessors, and architectures that implement several separated address spaces, distributed-memory multiprocessors.

Shared-memory multiprocessors are the direct extension from uniprocessors to multiprocessors. It has been shown a difficult task to have shared-memory multiprocessors with a high number of processors, mainly because of the cost of maintain the memory coherence. The alternative, systems with separated address spaces, scale better than shared-memory multiprocessors because they do not have to maintain the memory coherence. However, they require more effort by the programmers. In this kind of architectures, the programmer must explicitly re-design the algorithms to execute in different address spaces.

2.2.1 Shared-memory architectures

Shared-memory multiprocessor architectures are systems with the common characteristic that any processor can access any memory address. In general, advantages of these systems are that they are easy to use and to program because they are a direct evolution from uniprocessor architectures. That means that user applications and system policies can be used in these systems without modification. However, to exploit as much as possible the potential of multiprocessors, it is convenient to modify applications and policies to consider particular characteristics of each architecture.

The main problems related with shared-memory architectures are the memory bandwidth, the cache coherence mechanism, and the semantics of memory consistency. These problems imply that shared-memory multiprocessors are difficult to scale (large shared-memory multiprocessors are composed by 64, 128, 512 or 1024 processors). The memory bandwidth limits the number of simultaneous accesses to memory that applications can do.

The memory coherence generates the false sharing problem. It appears when two different data are accessed by two processors and these data are in the same cache line. In this case, cache coherence protocols generate that the cache line repeatedly travels across the multiprocessor from one processor to the other.

There are two main types of shared-memory architectures: symmetric multiprocessors and CC-NUMA architectures.

Symmetric Multiprocessors, SMP´s.

Figure 2.1 shows a typical configuration of a SMP machine. SMP´s are architectures with a limited number of processors, typically from 2 to 16 processors. Processors are usually inter-connected through a crossbar or a bus. The main characteristic of SMP´s is that the cost to access memory is the same for all the processors. This kind of architecture receives the name of CC-UMA, **C**ache-**C**oherent **U**niform **M**emory **A**ccess.

SMP´s have the advantage of their accessible cost, then their use have been extended as web servers and database servers. Their main problem is their limited scalability due to the interconnection mechanism used (between memory and processors). SMP´s that use crossbars are fast, but crossbars supporting many processors are very expensive. Buses are cheaper than crossbars, but they have a limited bandwidth and do not scale at all for more than around 12 processors.

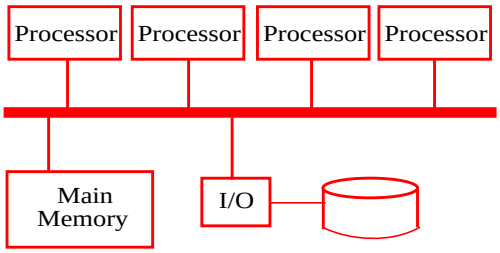


Figure 2.1: SMP architecture

Cache-Coherent Non-Uniform Memory Access, CC-NUMA.

The other group of architectures are CC-NUMA architectures. CC-NUMA machines consist of many processors, and they can scale up to 1024 processors. These machines are the biggest ones that implement shared-memory with cache coherence implemented by hardware. In these architectures, processors are usually grouped and interconnected through a more complex network. The goal of these architectures is to hide memory latency to scale. The main characteristic of CC-NUMA machines is that the cost to access memory is not always the same. It depends on which processor is doing the access and where the data is placed. In CC-NUMA machines, the memory is virtually shared but physically distributed.

The main advantage of CC-NUMA respect to SMP´s is that they scale better, and they can have many more processors. However, CC-NUMA machines have a much more sophisticated design. That implies that they are much more expensive than SMP´s, and then not so accessible. The other problem related to CC-NUMA machines is the memory access. As we have commented yet, the cost to access memory depends on the processor and the memory node. This will imply that the scheduling should be designed and implemented “with care”, taking into account this characteristic.

2.2.2 Distributed-memory architectures

Distributed-memory architectures have the common characteristic that they offer separated address spaces. Compared with shared-memory machines, distributed-memory machines have the advantage that they scale better, mainly because they do not have to maintain information related to each cache line distribution and status. For this reason, typically distributed-memory machines are systems with a high number of processors.

We can differentiate two main types of distributed-memory machines: Massively Parallel Processors and Networks of workstations.

Massively Parallel Processors systems, MPP´s.

MPP systems consist of a high number of processors connected through a fast (generally proprietary) interconnection network. Processors can explicitly send / receive data from / to remote memory nodes by special instructions or through the network API. In any case, the hardware does not provide any memory consistency mechanism. In these architectures, the cost to access to remote data also depends on the distance between processors and memory, and on the path to access it. Due to this hardware simplification, MPP´s can scale to a very high number of processors.

The main advantage of this architecture is the scalability. Nevertheless, MPP's have architectural design problems such as the interconnection network designs or the data injection mechanism design. One of the disadvantages of these systems are that they are expensive, mainly because of the interconnection network.

In MPP´s, each processor executes its own copy of the operating system, that implies that the resource management is more complicated that in shared-memory machines.

Networks of Workstations, NOW´s.

Figure 2.2 shows a NOW. They usually consist of several uniprocessor servers connected over a network. The main difference with MPP systems is the interconnection network. In NOW's, different nodes are usually interconnected with a commodity network and each node is a commodity processor. In fact, each workstation could be a different system with different characteristics.

These kind of systems have a clear advantage compared with MPP´s, they are more economic. In an environment where there are several users, each one with its own workstation, if we connect all of them with a general purpose network, and we install a resource manager, we achieve a NOW. These systems can grow in an easy way. Problems related with these architectures are that they do not offer good performance to individual parallel applications because the processor interconnection is not designed with this goal, and the resource management is not easy because we can have heterogeneous workstations.

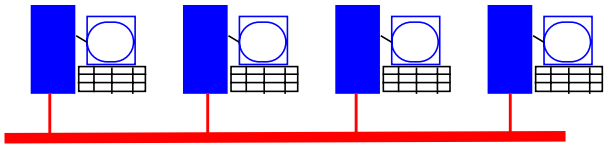


Figure 2.2: Networks of workstations

2.2.3 Mixed approaches

The two previously presented groups of architectures have been combined with the aim of including the best of each option. One of these approaches are clusters of SMP´s.

Clusters of SMP´s consist of several SMP nodes interconnected. Figure 2.3 shows the typical configuration of a cluster of SMP´s, several SMP´s machines connected through an interconnection network. Inside each SMP node we have shared-memory, and between nodes we have distributed-memory. This configuration has become very popular last years because small SMP´s have proliferated due to their prices, and connecting several of them we can achieve a system with a high number of processors. Every day is more usual that users have SMP´s at their work rather than workstations, then clusters of SMP is a natural evolution.

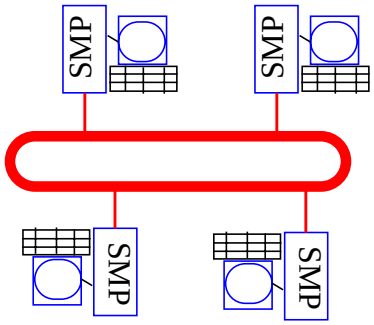


Figure 2.3: Clusters of SMP

Another different approach is the presented by some architectures such as the T3E [19]. The T3E is essentially a distributed-memory machine, however, it provides special instructions to specify shared-memory regions. These memory regions can be shared between more than one processor, but in some cases with a limited access (read only/ write only). The main difference with shared-memory machines is that, in this case, the hardware does not provides any support to ensure the memory consistency.

2.2.4 CC-NUMA architecture: SGI Origin 2000

Since we have adopted a practical approach, it is very important to understand the architecture of the system. The SGI Origin2000 [93] is a shared-memory multiprocessor system, that means that it has a global shared address space. We have selected the SGI Origin 2000 by several reasons. The first one is the availability, we have received the support to evaluate our contributions in this system[1]. Moreover, we consider that the SGI Origin 2000 is a modern architecture, quite extended, and the most used shared-memory multiprocessor system in supercomputing centers. Finally, the number of processors available, 64, is quite enough to evaluate our contributions in a real multiprocessor system.

The SGI Origin2000 is a scalable architecture based on the SN0 (Scalable Node 0), which is, in his turn, composed by two superscalar processors, the MIPS R10000.

The MIPS R10000 CPU

The MIPS R10000 CPU[110][68] is the CPU used in the SN0 systems. The MIPS R10000 is a "four-way"[2] superscalar RISC processor. Superscalar means that it has enough independent, pipelined execution units that it can complete more than one instruction per clock cycle. The main features of the MIPS R10000 are a nonblocking load-store unit to manage memory accesses, two 64-bit ALU's for address computation, arithmetic, and logical operations, see Figure 2.4.

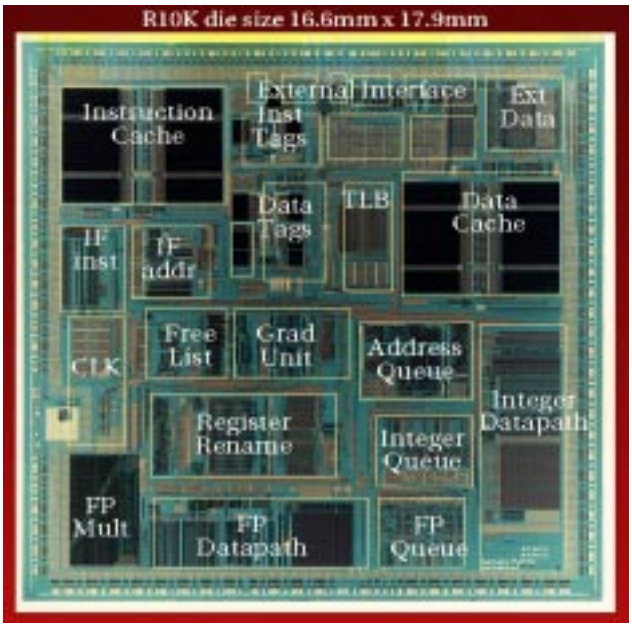


Figure 2.4: MIPS R10000 [68]

1. We have evaluated our contributions using all the processors in the system.
2. It can fetch and decode four instructions per clock cycle.

The MIPS R10000 uses a two-level cache hierarchy. A Level-1 (L1) internal to the CPU chip (separated data and instructions), and a second level cache (L2) external to it. Both of them are non-blocking caches, that means that the CPU does not stall on a cache miss. Both the L1 and L2 uses a Least Recently Used (LRU) replacement policy. The R1000 is representative of the most powerful processors available at the moment this Thesis was started.

SN0 Node Board

Each SN0 (Scalable Node 0)[93] contains two MIPS R10000 CPU's. Figure 2.5 shows a block diagram of a node board.

Each memory DIMM contains the *cache directory*. The use of the cache directory implies that the main memory DIMMs must store additional information. This additional information is proportional to the number of nodes in the system. The SN0 uses a scheme based on a *cache directory*, which permits cache coherence overhead to scale slowly.

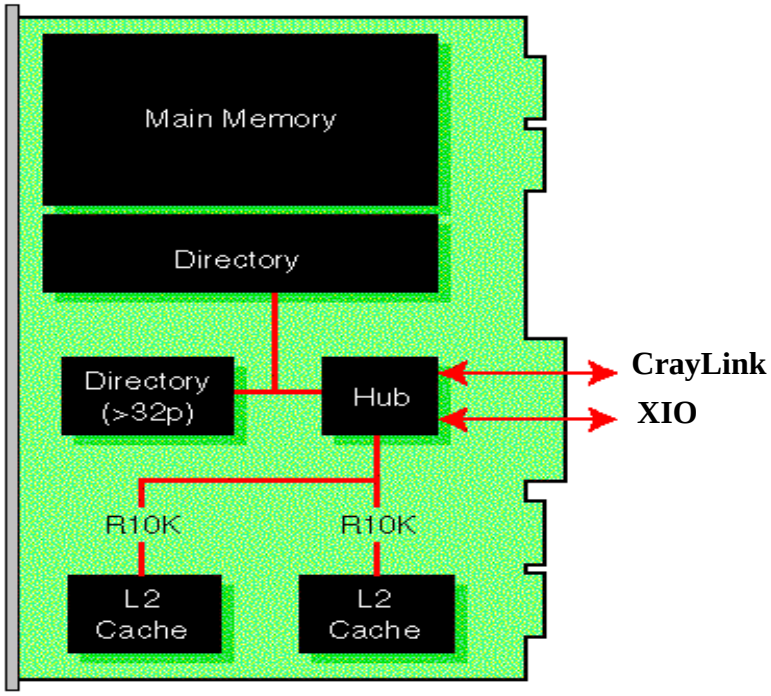


Figure 2.5: SN0 Node Board [93]

The final component is the hub. The hub controls data traffic between CPUs, memory, and I/O. The hub has a direct connection to the main memory on its node board. This connection provides a raw memory bandwidth of 780 MBps to be shared by the two CPUs on the node board. Access to memory on other nodes is through a separated connection called the CrayLink interconnect, which attaches to either a router or another hub.

When the CPU must refer data that is not present in the cache (L2 cache), there is a delay while a copy of the data is fetched from memory into the cache. In the SN0, the cache coherence is the responsibility of the hub chip. The cache coherence protocol allows the existence of multiple readers or only one writer (which will *"own"* the cache line). This protocol implies that, depending on the algorithm, multiple cache lines can travel trough the different nodes to maintain the data coherence. It is important to know the memory behavior to understand the application performance, very influenced by architectural configurations.

SN0 Organization

SN0's are interconnected to compound the complete system [93]. Figure 2.6 shows the high-level block diagrams of SN0 systems. Each "N" box represents a node board. Each "R" circle represents a router, a board that routes data between nodes. Through the hub, memory in a node can be used concurrently by one or both CPU's, by I/O attached to the node, and - via router - by the hubs in other nodes.

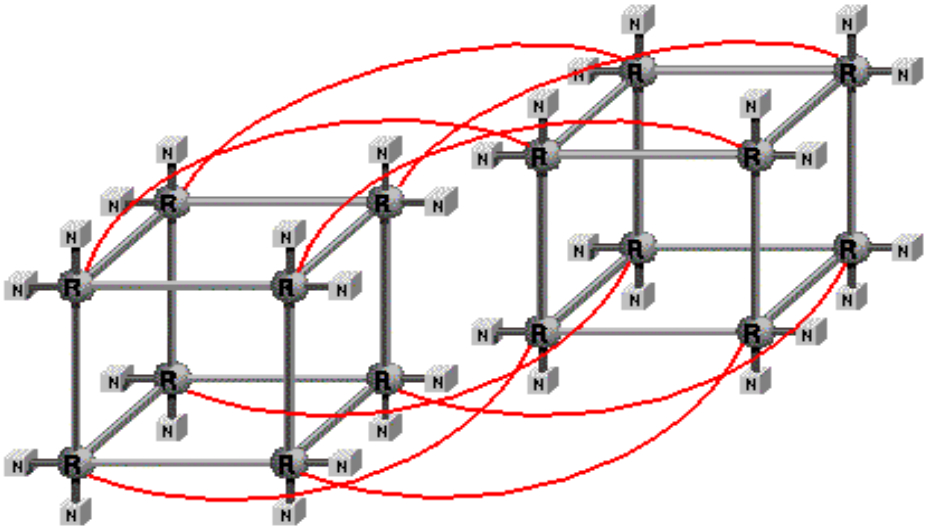


Figure 2.6: SGI Origin 2000 organization [93]

The hub determines whether a memory request is local or remote based on the physical address of the data. Access to memory on a different node takes more time than access to memory on the local node, because the request must be processed through two hubs and possibly several routers. Table 2.1 shows the average latencies to access the different memory levels [53].

Table 2.1: SGI Origin2000 average memory latencies

Memory level	Latency (ns)
L1 cache	5.1
L2 cache	56.4
Local memory	310
4 cpus remote memory	540
8 cpus remote memory	707
16 cpus remote memory	726
32 cpus remote memory	773
64 cpus remote memory	867
128 cpus remote memory	945

The hardware has been designed so that the incremental cost to remote memory is not large. The hypercube router configuration implies that the number of routers information must pass through is at most n+1, where n is the dimension of the hypercube.

This configuration, and the different cost to access memory, implies that it is important to consider the memory placement. Data are not replicated, it is in one memory cache, or memory node. If the processor, or processors, that are accessing to these data are in the same node, accesses to these data will be very fast. However, if data have to travel through the interconnection network, the cost to get the data could be three times slower.

The particular characteristics of our SGI Origin2000 are the following:

- 64 250 MHZ IP27 Processors
- CPU: MIPS R10000 Processor Chip Revision 3.4
- FPU: MIPS R10010 Floating Point Chip Revision 0.0
- Main memory size: 12.288 Mbytes
- L1: Instruction cache size: 32 Kbytes
- L1: Data cache size: 32 Kbytes
- L2: instruction / data cache size: 4 Mbytes

2.3 Scheduling policies

In this Section, we will present the related work in processor scheduling. We will follow the scheduling classification presented in the introduction of this Thesis, see Figure 2.7.

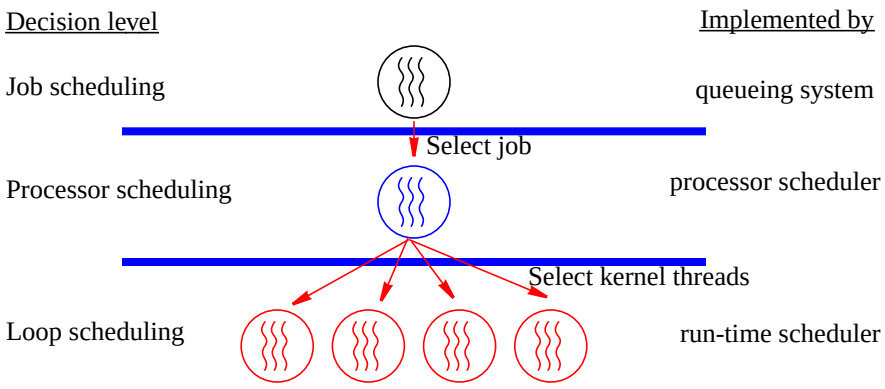


Figure 2.7: Decision levels

The first level, job scheduling, decides which job must be started at any moment. It selects one job for execution, from a list of queued jobs. In a distributed system, the job scheduling policy could include a decision related to where, in which node, to start the application. The job scheduling policy is implemented by the queueing system, and it is activated at a low frequency in the system.

The second level, processor scheduling, decides the scheduling between processors and kernel threads of running applications. The processor scheduling policy is implemented by the processor scheduler in the O.S. kernel. It is activated at a medium frequency in the system, a typical value is between 10 and 100 ms.

Finally, the loop scheduling policy decides which user-level threads to execute from each application. The loop scheduling policy is implemented by the run-time library and it is always implemented at the user-level.

In this Chapter, we focus on the processor scheduling and job scheduling level because they are implemented by the system. Loop scheduling policies are more related to the particular algorithm, the user, and the compiler, and out of the focus of this Thesis.

There exist several possibilities to execute a workload of parallel jobs in a multiprocessor system: from executing one job at any time to executing all the jobs together. Multiple combinations of different policies at each level have been proposed. Some of then, by default, eliminate some of the other decision levels. For instance, most of the job scheduling policies assume that there is not processor scheduling policy in the system (a simple dispatch).

In this Section, we present some related work with our proposal of coordinating the different scheduling levels, and related work with job scheduling and processor scheduling. Related to the scheduling policies, we will try follow a similar classification to the proposed by Feitelson in [32]. However, it does not classify scheduling policies in these three levels, for this reason we will re-organize its classification to match with our classification.

Feitelson classifies in [32] the scheduling policies as single-level and two-level policies, see Figure 2.8. In single-level scheduling, the O.S directly allocates processors to kernel threads. In two-level scheduling, the O.S separates the scheduling decision in two steps: (1) it allocates processors to jobs, and (2) inside each job several kernel threads are selected to run. Single-level policies are also called time-sharing policies, and two-level policies are also called space-sharing policies.

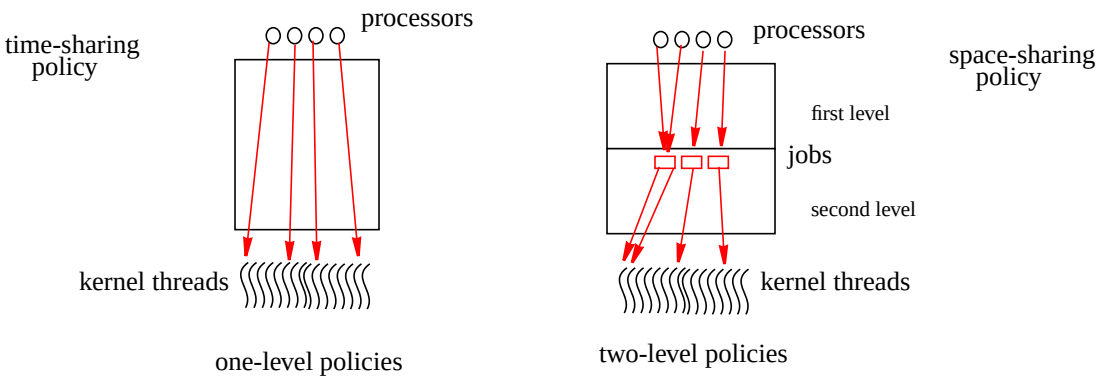


Figure 2.8: One-level policies vs. two-level policies

Feitelson classifies space-sharing policies into four classes: Fixed, Variable, Adaptive, and Dynamic, see Table 2.2. The decisions taken by Fixed and Variable policies are only related to job scheduling. On the other hand, decisions taken by Adaptive and Dynamic policies are related to processor scheduling.

Moreover, Feitelson classifies applications into three types: rigid, moldable, and malleable. Rigid are those applications that can only be executed with the number of processors requested. Moldable are those applications that can be executed with any number of processors, and different from the number of processors requested, but it must be fixed during the complete execution of the application. And malleable are those applications that can be executed with a varying number of processors.

Table 2.2: Space-sharing classification proposed by Feitelson

Type	Hardware	Decision level	Application
Fixed	Determines the partition size	job scheduling	Rigid parallelism
Variable	Does not determine the partition size	job scheduling	Rigid parallelism
Adaptive	Does not determine the partition size	processor scheduling	Moldable
Dynamic	Does not determine the partition size	processor scheduling	Malleable

We classify policies based on the capacity of decision that they have. Then, we classify them in job scheduling policies (Fixed and Variable), and processor scheduling policies (Adaptive and Dynamic). Job scheduling policies only decide which application to execute, not the number of processors to allocate to each one. Processor scheduling policies can apply a time-sharing (one-level), space-sharing (two-levels), or gang scheduling policies (time-sharing and space-sharing).

2.4 Coordinating scheduling levels

2.4.1 Coordinating the processor scheduler and the run-time

The coordination between processor scheduler and run-time has been directly or indirectly used in other proposals. In the NANOS project [63][64], Martorell et al. use an interface between the processor scheduler and the run-time. This interface includes information from the processor scheduler to the run-time such as the number of processors allocated, and information from the run-time to the scheduler such as the number of processors requested. In fact, the interface between processor scheduler and run-time used in this Thesis is an extension of the NANOS interface. In *Process Control* [105], the run-time is informed when it has too many kernel threads activated, and it reduces the number of kernel threads. In *Scheduler Activations* [2], Anderson et al. propose a mechanism to manipulate threads and to communicate the processor scheduler with the run-time library. McCann et al. also propose in [65] an interface between the processor scheduler and the run-time. They also propose some processor scheduling policies that will be commented in next Section.

Rather than to have an interface or a mechanism to communicate different levels, a different approach consists of deducing this information by observing the other scheduling level decisions. For instance, in ASAT [91] the run-time observes the number of processes in the system to deduce if the load is very high. In that case, the run-time decides to reduce its parallelism. This approach is also followed by the native IRIX run-time library.

As we have commented before, we have mainly completed and extended the NANOS interface. In Chapter 3 we describe the main modifications introduced in the NANOS interface.

2.4.2 Coordinating the queueing system and the processor scheduler

The explicit coordination between the queueing system and the processor scheduler has not been explicitly proposed. At this level, the typical interaction between levels consist of the observation to deduce information. A typical practice is to periodically evaluate the load of the system and to decide whether any parameter, such as the multiprogramming level, must be modified. This evaluation can be automatically done by the queueing system, or by the system administrator.

There are three main traditional execution environments. In the first one, the system applies a job scheduling policy, but not a processor scheduling policy. In that case, the multiprogramming level is determined by the number of applications that fill in the system. In that case, there is no coordination because there is no processor scheduling level. However, the multiprogramming level and the load of the system are indirectly controlled by the queueing system. This execution environment has the problem of the

fragmentation. In this environments, jobs are rigid and they only can run with the number of processors requested. In these systems, fragmentation becomes an important issue because they are typically non-preemptive.

The second one is an execution environment without job scheduling policy, and with a processor scheduling policy. In these systems, the main problem is that the system can become overloaded, because there is no control about the number of applications executed. If at any moment, a high number of jobs arrive to the system, the system performance will degrade.

The third one is an execution environment that applies a job scheduling policy and a processor scheduling policy. In this case, depending on the granularity that the queueing system modifies its parameters, such as the multiprogramming level, the system frequently end up being low loaded (with unused processors but with queued applications), or overloaded (with much more processes than processors).

To summarize, there are two main problems related to the no coordination between processor scheduler and queueing system: If the system implements some mechanism to control the multiprogramming level, the main problem will be fragmentation. In the second case, if the system is uncoordinated, the problem will be that the system can saturate due to an excessive use of resources.

2.5 Job scheduling policies

Job scheduling policies decide which job should be executed, and in which order. Depending on the system. The job scheduling policy can also include the decision about in which node to execute it.

Job scheduling policies include Fixed and Variable policies. In fact, we consider that a job scheduling policy can be classified in the two classes, it depends more on the hardware than in the job scheduling policy itself. Feitelson considers as Fixed those policies that are executed in a system where the hardware determines partition sizes and only one application can be executed per partition. And, as Variable, those policies executed in a system where partition sizes can be specified by the queuing system.

Figure 2.9 shows some of the job scheduling policies presented in this Section. We differentiate between those policies that do not use application information, and those policies that use *a priori* information (execution time estimation).

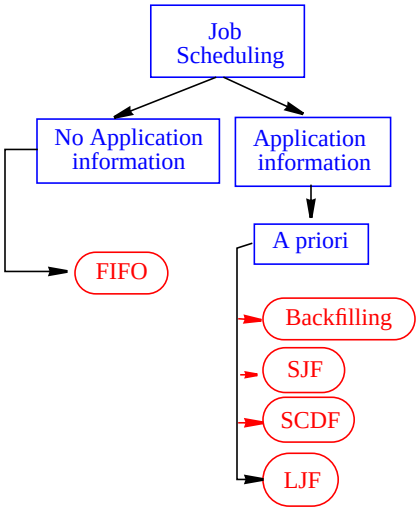


Figure 2.9: Job scheduling policies

2.5.1 Fixed policies

In fixed policies, the partition size is limited by the hardware, and set by the system administrator, typically for reasons related to access control or hardware limitations. This allows certain parts of the machine to be dedicated to certain groups of users. The main problem related to fixed partitioning is internal fragmentation. If a job requires a small number of processors, the rest are left unused. A partial solution is to create several partitions with different sizes, then users can choose the most suitable.

2.5.2 Variable policies

Variable partitioning is similar to fixed partitioning except that partition sizes are not predefined, partitions are defined according to application request. Partitions can be previously created with different sizes or arbitrary created. However, internal fragmentation may occur in cases where the allocation is rounded up to one of the predefined sizes, but the job does not use the extra processors. If the architecture does not impose any restriction, internal fragmentation can be avoided by creating arbitrary partitions. However, external fragmentation remains an issue because a set of processors can remain idle because they are not enough to execute any queued job.

The simplest approach is a *First Come First Serve* (FCFS or FIFO) approach, jobs are executed in the same order of arrival to the system. The favorite heuristic in uniprocessor systems is the *Shortest Job First* (SJF) [49]. However, this is not a good approach in multiprocessors because it needs to know the execution time in advance and long jobs might be starved. In parallel systems, jobs may be measured by the number of processors they request, which is suposed to be known in advance. Using this information, the *Smallest Job First* [60], and the *Largest Job First* [114] have been proposed. The first one is motivated by the same reason that the SJF. However, it turns out to perform poorly because jobs that require few processors do not necessarily terminate quickly [55][50]. To solve this problem a new approach appear, the *Smallest Cumulative Demand First* [55][60][89]. In this policy, jobs are ordered considering the product of number of processors and expected execution time. However, this policy did not show better results that the original *Smallest Job First* policy [50], and also suffers from the same problem than the *Shortest Job First*: it needs to know the application execution time in advance.

Finally, a job scheduling policy that has shown to perform well is called *Backfilling* [56]. Backfilling approach also needs a job execution time estimation. In *Backfilling,* jobs are executed in a first come first serve order but it allows to advance small jobs in the queue if they do not delay the execution of previously queued jobs [56][97]. Feitelson at al. show in [67] that the precision of user run-time estimations do not significantly affect the performance of Backfilling.

2.6 Processor scheduling policies

Processor scheduling policies decide how many processors to allocate to each job. At this level of decision, the processor scheduler can implement three types of policies: time-sharing, space-sharing, and gang scheduling. In time-sharing, the processor scheduler considers the kernel thread as the unit of scheduling. Space-sharing policies consider the job as the unit of scheduling. They divide the processor scheduling policy into two-levels: processors to jobs, and processors allocated to jobs to kernel threads. Gang scheduling is a combination of time-sharing and space-sharing: it implements a time-sharing among jobs, where each job implements a one-to-one mapping between processors and kernel threads. Figure 2.10 shows the three processor scheduling options.

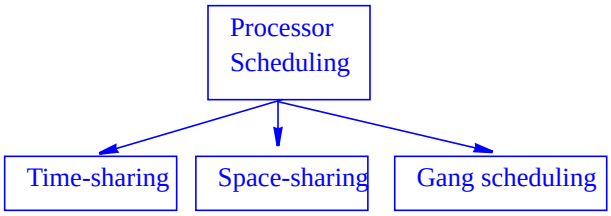


Figure 2.10: Processor scheduling policies

2.6.1 Time-sharing policies

Time-sharing policies are the automatic extension from uniprocessor policies to multiprocessor policies. Time-sharing policies do not consider grouping of threads. These policies are usually designed to deal with the problem of many-to-few mapping of threads to processors. There are two main approaches to deal with this problem: local queues (typically one per processor) or a global queue (shared by all the processors). Figure 2.11 shows the main approaches proposed in time-sharing policies.

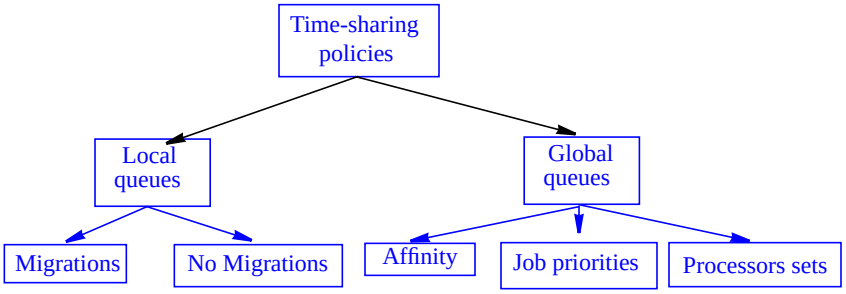


Figure 2.11: Time sharing policies

Local queues

To use local queues is the natural approach for distributed memory machines, but it is also suitable in shared-memory machines. Since normally each processor gives work for its own queue, there is no need for locks (no contention). In shared-memory machines, local queues also helps to maintain the memory locality. However, this approach has the problem that it needs a good load balancing mechanism to map threads to processors.

The use of local queues implies that threads are directly mapped to processors. This provides threads a sense of locality and context. Threads can keep data in local memory, supporting programming models that exploit locality. For example, a very useful programming paradigm is to partition the problem data set, and perform the same processing on the different parts, using some communication for coordination. In SPMD programming, this data partitioning is explicitly part of the program.

The main problem related to local queues is the load balancing. The load balancing was shown crucial to achieve an adequate performance [28], mainly in distributed systems. There is some controversy in the literature in whether the correct load balancing of new threads is enough [29], or maybe thread migration (“preemptive load balancing”) is also required [22][51].

There are several proposals to deal with the problem of thread placement. In the case where no migrations are considered, one simple approach is to use a random mapping[3][48]. Obviously, this scheme has a bad worst-case, because it is possible to choose the most highly loaded processor. A design that has been suggested a number of times is to partition the system using a buddy-system structure. A buddy-system creates a hierarchy of control points: the root controls the whole system, its two children, each controls half of the system, and so on. This control is used to maintain data about load on different parts of the system.

The second issue is load balancing where thread migrations are allowed. Important characteristics of load balancing algorithms are that they distribute the load quickly and that they be stable. Quick load distribution is achieved by non-local communication to disseminate information about load conditions[57]. Stability means that the algorithm

should not over-react to every small change. It is achieved by balancing only if the difference in loads achieves a certain threshold [70], or by limiting the number of times a thread can migrate.

Global queues

A global queue is easy to implement in shared-memory machines but not a valid choice for distributed memory machines. Threads in a global queue can run to completion or can be preempted. In preemptive systems, threads run for a time and then return to the queue. In this Section, we will comment preemptive systems. The main advantage of using a global queue is that it provides automatic *load sharing*[3][32]. Drawbacks are possible contention in the queue and lack of memory locality.

A global queue is a shared data structure that should be accessed *with care*. The main methods used are through locks and using wait-free primitives. There is also some work on fault tolerance [111]. The direct implementation of a global queue uses locks to protect global data and allow multiple processors to add and delete threads. A lot of work has been done regarding the efficient implementation of locks. A global queue is mainly the same as in uniprocessor systems. This typically implies that entries are sorted by priorities, and multi-level feedback is used rather than maintaining strict FIFO semantics. The problem of using locks is the contention [10][44]. Indeed, measurements on a 4 processors machine show that already 13.7% of the attempts to access the global queue found it locked, and the trend indicated that considerable contention should be expected for large number of processors [103]. The alternative is use a bottleneck-free implementation of queues, that does not require locking such as the fetch-and-add primitive [73].

The approach of partitioning and then scheduling using a global queue shared by all the processors in the partition was implemented in the Mach operating system [11][12]. The partitioning is based on the concept of a processor set, in this case a global priority queue is associated with each processor set.

The order in which threads are scheduled from the global queue is determined by their relative priorities. Most implementations use the same algorithms that were used in uniprocessor systems.

Another important consideration in thread scheduling is matching threads with the most appropriate processor for them to run on. This is normally equivalent to the question of data that may reside in the processor cache. The scheduling policy that tries to schedule threads on the same processor on which they ran most recently, under the assumption that this particular processor may still have some relevant data in its cache is called *affinity scheduling* [4][8][23][98][106].

3. The term “load sharing” originated in distributed systems where it describes systems that do not allow a processor to be idle if there is work waiting at another processor.

All the mentioned scheduling schemes operate on individual threads, and do not consider the fact that these threads belong to competing jobs. As a result, jobs tend to receive service proportionally to the number of threads they spawn. An alternative approach is to give jobs equal degrees of services [55], threads belonging to a job with lots of threads should have a lower priority than threads belonging to jobs with few threads. It is also possible to prioritize jobs. This can be done by keeping the threads in separate per-job queue. Processors serve these queues in a round robin way, and set the time quantum for each scheduled thread according to the priority of the job to which this thread belongs [83].

2.6.2 Space-sharing policies

Space-sharing is done by partitioning the machine among applications that run side by side. Applications run in these partitions as in a dedicated machine. This approach is motivated by the desire to reduce the operating system overhead on context switching [105]. With space-sharing, the system is more involved in allocating processors to jobs. The application run-time system may then schedule user threads on these processors[104][113].

In *adaptive partitioning*, the partition size is defined when application is launched, and fixed during the complete execution time. In *dynamic partitioning*, the processor allocation can change at run-time. Adaptive and dynamic partitioning are both considered as processor scheduling policies because they are involved in processor scheduling decisions, not job scheduling.

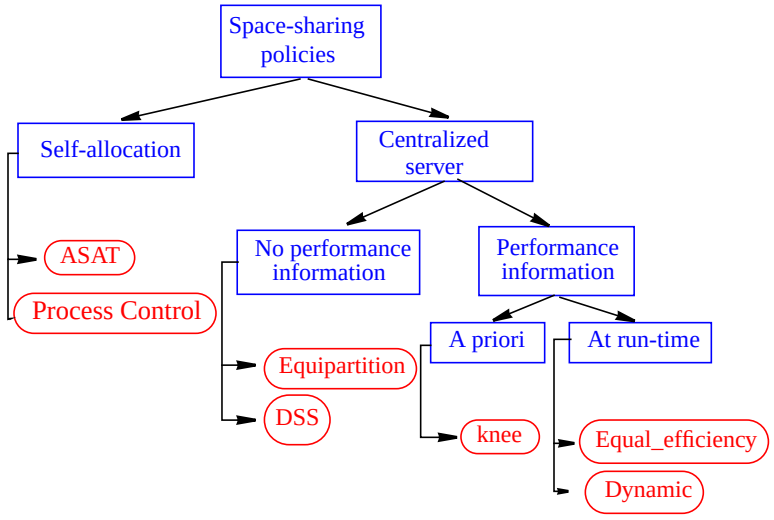


Figure 2.12: Space-sharing policies classification

In adaptive and dynamic partitioning, the system decides how many processors to allocate to each application. In adaptive partitioning, processor allocation must be maintained because the applications are moldable, not malleable. Moldable applications

are those than can be executed with any number of processors, but are not able to adapt their parallelism to changes on the processor allocation. Then, the processor allocation must be maintained during the complete execution of the application. In dynamic partitioning, the processor allocation can change at run-time. However, most of the adaptive policies can be considered also as dynamic processor allocation policies since they depend more on the running application than on the policies themselves.

It is important to comment that, the ability to apply a dynamic processor scheduling policy does not necessarily imply frequent processor re-distribution. The processor scheduling policy could decide the same distribution each time that it is re-applied. Such a stable allocation is, in fact, a desirable situation to avoid processor migrations. It is important to be able to react to changes in both the applications or the system, but also to maintain the allocation if conditions are stable.

In this Section, we focuses in the first level of decision of the processor scheduler, processors to jobs. This decision is also called the *processor allocation policy*. The second level can be implemented by the processor scheduler or by the run-time itself. In our case, the processor scheduler takes this decision. We have followed the proposal made by Tucker and Gupta in *process control* [105]. They propose to do a one-to-one mapping to achieve the optimal execution point. This mechanism is presented in Chapter 3.

Adaptive policies

In adaptive partitioning, the partition size can be decided by the system (on its own), or in cooperation with the application. If the system decides the partition, a simple approach is giving each job as many processors as it requests. However, if the sum of processors requested is greater than the total number of processors, jobs will receive less processors than their request [113][89]. In the extreme case, processor allocation could be reduced to one processor. This approach increases the execution time with respect to the case that each job uses as many processor as it requests, but guarantees that short jobs do not have to wait for large jobs.

One approach is to perform an Equipartition [54][55] among jobs. Equipartition tries to allocate an equal-size partition for all current jobs (limited by the job request). Jobs can run simultaneously and do not have to wait in the queue. Equipartition achieves good performance in both minimizing response time and maximizing throughput.

A few other algorithms have been proposed that do not derive directly from the equipartition. For instance limiting the maximum partition size that a single job can obtain [86]. Another approach is based on a state machine. At each instant the system is in a certain state, which identifies the ideal partition size [86]. With a system with P processors, the states are {P, P/2, P3,..., 1}. Jobs are allocated to partitions whose size is determined by the state. A more sophisticated approach is to base the decisions on predictions of the queuing time, and of how long current jobs will run [26]. These policies do not consider the application characteristics, just the job request.

Another alternative is to decide the partition size in collaboration with the application. One option is to consider the *execution signature*, i.e. how much time it would require on different partition sizes [59][79]. The system combines this information with the load information to derive the optimal partition size. In general, information about the total work and the efficiency of the job is beneficial [13]. The problem of this approach is that it assumes that the information is precise and that the application cooperates with the system. One alternative is that the system collects the information itself, based on previous executions of the job [41]. Eager *et al.* propose in [30] to allocate the number of processors that achieve the *Knee* of the curve execution time vs. efficiency.

Dynamic policies

Dynamic partitioning modifies the processor allocation of applications at run-time to react to different events such as the system load, the application request, performance information, etc. For example, when an application enters in a sequential phase its processors can be reallocated to another application [113][106][112].

The main reason to change the partition sizes at runtime is the desire to improve fairness and resource utilization. Thus, when a new job arrives, a fair share of processors should be preempted from running jobs, and given to the new job. When a job terminates, its processors should be divided among the other jobs. In essence, this is the *Equipartition* policy [105][55][65]. *Dynamic Space Sharing* (DSS) is a dynamic processor allocation proposed by Polychronopoulos et al. in [84] that decides an allocation proportional to the number of requested processors.

Efficiency can also become an issue. Consider a job that it is written so that the work is divided into 8 equal pieces. Running such a job on a partition of 7 processors would not provide any additional benefit respect to a partition of 4 processors, leading to waste of resources [113]. Nguyen et al. propose *Equal_efficiency* in [74][75][76]. The goal of the Equal_efficiency is to achieve an equal efficiency in all the processors. In that case, when a job is submitted the system executes it for short periods on different partition sizes and measures the job efficiency. The policy allocates processors (one by turn) to those applications that achieve the higher efficiency. The main problem of the Equal_efficiency is that it allocates processors tho those applications that achieve the best efficiency, however, the “best” is not a synonym of good efficiency. Another related problem to this policy is the way the estimate the efficiency achieved by running applications.

McCann propose in [65] *Dynamic*, a dynamic space-sharing policy that reallocates processors from one parallel job to another based on changes in the parallelism. Jobs inform the scheduler about the number of processors they could use. They also informs the scheduler when there are no ready application threads to run. Processors are then moved from jobs that do not use them to processors that could use them. Dynamic calculates the “use of processors” considering the idle periods of the application, resulting in a great number of reallocations, as it was shown in [76].

Some versions of dynamic partitioning operates in the following way: instead of explicitly allocating processors to jobs, jobs create threads that execute on available processors. If there are too many threads in the system, jobs voluntarily reduce the number of threads. If there are free processors, jobs create new threads to use them. Examples of this approach includes *Process Control* [105] and ASAT [91].

The processor scheduling policy implemented by the native operating system IRIX 6.5 is a combination of a priority based time-sharing policy with an affinity scheduling.

2.6.3 Gang Scheduling

Gang Scheduling is a technique proposed by Ousterhout in [78] that combines space and time sharing and it was presented as the solution to the problems of static space-sharing policies. We define gang scheduling as a scheme that combines three features [32]:

- Application threads are grouped into gangs (typically all the threads of the application in the same gang)
- Threads in each gang are executed simultaneously, using a one-to-one mapping
- Time slicing is used among gangs (all the threads in the gang are preempted at the same time)

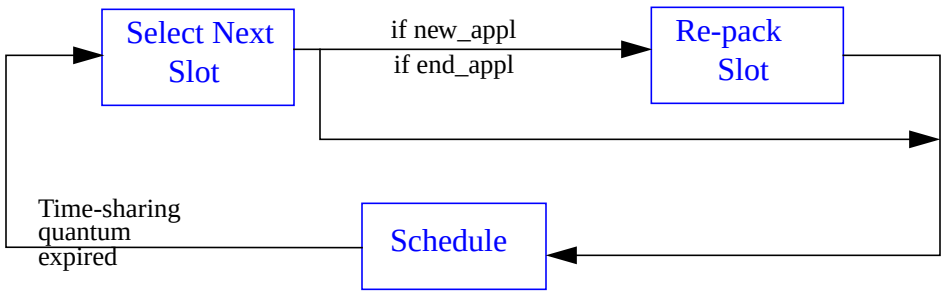


Figure 2.13: Gang scheduling policies diagram

Figure 2.13 shows the diagram that represents the gang scheduling policies behavior. Periodically, at each time-sharing quantum expiration, the system selects a new slot to execute. A slot is a set of applications that will be concurrently executed. If during the last quantum, any new job has arrived to the system, or any job has finished, the slot will be re-organized. The algorithm to re-organize one slot (or the complete list of slots) is called the re-packing algorithm. The re-packing algorithm is applied to migrate some job from any other slot to the currently selected. Traditionally, the scheduling phase is traduced by a single dispatch, to allocate as many processors to jobs as they request. Gang scheduling policies mainly differs in the re-packing algorithm applied.

Figure 2.14 shows a taxonomy as a function of the three elements that define a gang scheduling policy: the re-packing algorithm, the job mapping, and the scheduling algorithm applied.

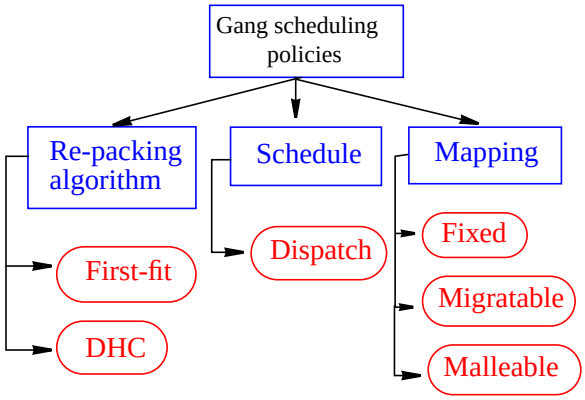


Figure 2.14: Gang scheduling taxonomy

The simplest version of gang, where threads are always rescheduled in the same set of processors is the most popular one. However there have been proposed more flexible versions [81]. One of this is migratable preemptions, where threads are preempted in one set of processors and resumed in another to reduce the fragmentation.

Other similar scheme to gang scheduling is Coscheduling[78]. Coscheduling was originally defined by Ousterhout to describe systems where the operating system attempts to schedule a set of threads simultaneously, as in gang scheduling. However, if it cannot, it will resort to scheduling only a subset of the threads simultaneously.

Feitelson and Jette argue in [34] that Gang Scheduling solves the problem of taking incorrect decisions while performing job scheduling. Feitelson and Rudolph comments in [35] that academically speaking Gang Scheduling is inferior to dynamic partitioning, but that dynamic partitioning is difficult to implement and that drawbacks of gang are not so critical. They conclude that "*the advantages of Gang Scheduling generally outweigh its drawbacks*".

Several works have analyzed the problem of fragmentation in Gang Scheduling and have proposed several re-packing algorithms as solution to this problem. Feitelson in [33] analyzes several algorithms of job re-packing for Gang Scheduling and concludes that the best option is a buddy system or to use migration to re-map the jobs (based on a first-fit algorithm). Feitelson and Rudolph propose and evaluate Distributed Hierarchical Control (DHC) in [35][37]. DHC is a design using a hierarchy of controllers that dynamically re-partitions the system according to changing job requirements. They show, through simulations, that DHC achieves performance comparable to off-line algorithms.

Zhou *et al*. [115] also attack the fragmentation problem and present ideas such as job re-packing, running jobs on multiple slots, and minimizing the number of time slots in the system to improve the buddy scheme for Gang Scheduling. Setia [88] shows through simulations that Gang Scheduling policies that support job migration offer significant performance gains over policies that do not use remapping. Other authors have analyzed other aspects of the performance of Gang Scheduling under different kind of applications. Silva and Scherson propose Concurrent Gang [94] to improve the performance of I/O intensive applications. Using simulations they show that Concurrent Gang combines the advantages of Gang Scheduling for communication and synchronizations intensive applications with the flexibility of a Unix scheduler for I/O intensive applications. They also classify applications through run-time measurements in [95] to provide better service to I/O bound and interactive jobs under gang. They propose to improve the utilization of idle times (idle slots and blocked tasks) and to control the spinning time of tasks. Gare and Leutenegger [40] analyze the relation between job size and quantum allocation. They allocate a number of quanta inversely proportional to the number of processes per job to reduce the slowdown.

Traditionally, systems that apply a gang scheduling policy do not include a job scheduling policy because one of the goals of gang scheduling is to reduce the impact of incorrect job scheduling decissions.However, Zhang et al. [116] propose Backfilling gang scheduling to improve the system performance. They show that, even theoretically gang scheduling policies allows the execution of any new job, due to resource limitations some jobs must remain queued until some running job finishes. Combining Backfilling and Gang scheduling the queued time can be reduced.

Polychronopoulos et al. attack gang scheduling problems by proposing Sliding Window-DSS (SW-DSS) and Variable Time Quanta-DSS (VTQ-DSS) [85], two variations of their implementation of the Dynamic Space-Sharing (DSS) policy. SW-DSS and VTQ-DSS introduce a time-sharing among jobs, allocating a number of processors proportional to the number of requested processors. These approaches do not consider the performance achieved by running applications, and the number of processors received is proportional to the number of processors requested.

2.7 Programming models

There are two main programming models to parallelize sequential programs: MPI and OpenMP.

2.7.1 Message Passing Interface: MPI

MPI [66] is based on the idea of an execution environment with different address spaces. The MPI programming model divides the algorithm in a fixed number of tasks, and each task is executed in a process. MPI forces that the programmer divides the complete algorithm, it does not allow a progressive parallelization. Moreover, since the different tasks will be executed in different address spaces, data communication must be done by explicit message passing.

Most of the parallel applications are iterative, then the typical SPMD behavior of an MPI application is the following (see Figure 2.15):

- The master task reads the data and distributes the data among the rest of tasks.
- Each task calculates one iteration of the loop.
- Each task sends the data to its neighbours
- Each task receives the data from its neighbours
- Each task executes the next iteration and so on.

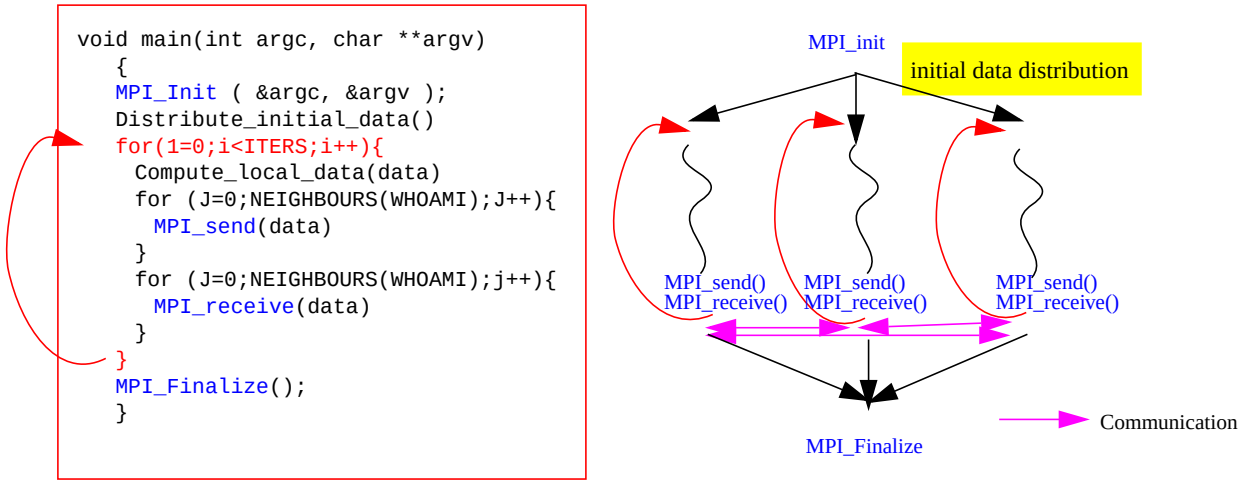


Figure 2.15: MPI example

This behavior implies few points of data communication per task and no task re-scheduling. Since the number of tasks is fixed during the complete execution of the application, no application malleability is allowed in MPI applications. In addition, since the data communication is done explicitly and at a few points, load balancing is not easily implementable.

2.7.2 OpenMP

OpenMP [77] is based on the idea of an execution environment with a global (shared) address space. The OpenMP programming model does not explicitly divide the algorithm in tasks. Programmers define which loops, and Sections of code, can be executed in parallel. The compiler and the run-time, as a function of the number of processors, and the loop scheduling policy selected, will generate the tasks that will execute the algorithm in parallel.

Since the OpenMP programming model assumes a shared address space, communication is implicitly done using variables. Programmers do have not to include any explicit directive or function to access the data used by multiple tasks.

In OpenMP, scheduling decisions are taken once per parallel loop or parallel Section. This fact gives OpenMP applications the characteristic that they can adapt the parallelism that they spawn to the number of processors available (malleability). Of course, to do that they need support from the run-time library, but in MPI this characteristic is not possible because it only takes scheduling decisions at the initial of the application.

OpenMP works by inserting directives in the sequential source code. In fact, the same code can generate a sequential or parallel version just activating or disactivating the OpenMP directives. Another advantage of this programming model is that parallelization can be done incrementally. Users can perform a profile of their sequential program and just parallelize the most time consuming loops of their code. The program can then be tested and evaluated. If the program achieves enough speedup, the parallelization can be finished at this point. Otherwise, users can parallelize additional loops.

Figure 2.16 shows an OpenMP example. The code in the left side shows a simple program with two loops. The *omp parallel for* directive specifies that the following loop is parallel.

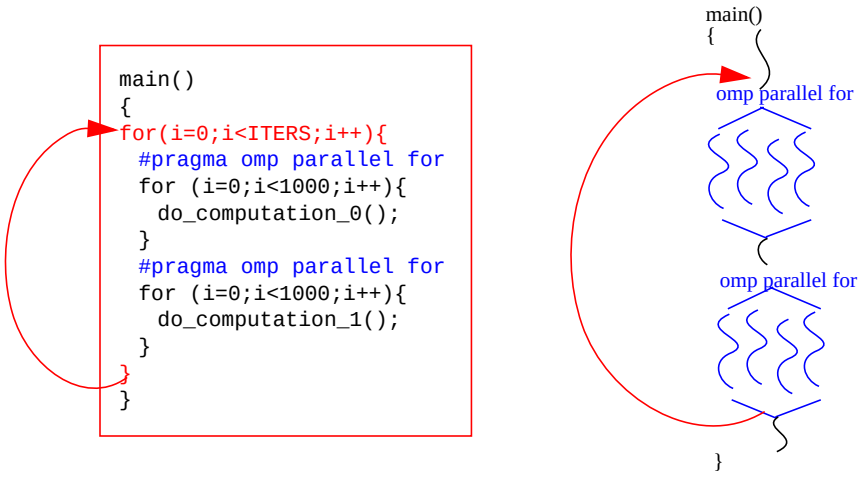


Figure 2.16: OpenMP example

The figure in the right side corresponds to the parallelism that will be generated in this example. The program is executed in sequential until the *omp parallel for* directive is found. The *omp parallel for* directive creates a *team*[4] of threads that will collaborate to execute the parallel loop.

2.7.3 OpenMP directives

The OpenMP programming model provides directives that specify parallel regions, work-sharing constructs, synchronization, and data environment constructs.

Parallel regions constructs

A parallel region is a block of code that is going to be executed by multiple threads in parallel. This is the fundamental parallel construct in OpenMP that starts a parallel execution. When a thread encounters a parallel region, it creates a team of threads, and it becomes the master of the team.

Figure 2.17 shows the OpenMP code to specify a parallel region. To give more information about the valid clauses see [77]. The number of physical processors actually hosting the threads at any given time is implementation-dependent. Once created, the number of threads in the team remains constant for the duration of that parallel region. It can be changed either explicitly by the user or automatically by the runtime system from one parallel region to another. The OMP_SET_DYNAMIC primitive and the

```
!$OMP PARALLEL [clause[[,] clause]...]
Block
!$OMP END PARALLEL
```

Figure 2.17: Parallel region directive

4. Threads may be physically created at this point and waiting for work in an idle loop

OMP_DYNAMIC environment variables can be used to enable and disable the automatic adjustment of the number of threads. Within the dynamic extent of a parallel region, thread numbers uniquely identify each thread. Threads numbers are consecutive and range from zero, for the master thread, up to one less than the number of threads within the team. *Block* denotes a structured *block* of Fortran statement. It is noncompliant to branch into or out of the block. The code contained in *block* is executed by each thread. The END PARALLEL directive is an implicit barrier at this point. Only the master thread of the team continues the execution past the end of a parallel region.

Work-Sharing constructs

A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it. A work-sharing construct must be enclosed dynamically within a parallel region, if not, it is treated as whether the thread that encounters it were a team of size one. The main work-sharing construct directives are DO and SECTION.

Figure 2.18 shows the DO directive format. It specifies that the iterations of the immediately following DO loop must be executed in parallel. Iterations of the DO loop are distributed across threads that already exists. One of the clauses that users can specify defines the loop scheduling to use. It can be STATIC, DYNAMIC, GUIDED or RUNTIME.

```
!$OMP DO [clause [[,]...]
do_loop
[!$OMP END DO [NOWAIT]]
```

Figure 2.18: DO directive

Figure 2.19 presents the SECTIONS directive which is a non-iterative work-sharing construct. It specifies that the enclosed Sections of code are to be divided among threads in the team. Each Section is executed once by a thread in the team.

```
!$OMP SECTIONS [CLAUSE [[,]CLAUSE]...]
[!$OMP SECTION]
block
[!$OMP SECTION
block]
...
!$OMP END SECTIONS [NOWAIT]
```

Figure 2.19: SECTION directive

Synchronization constructs

The OpenMP programming model provides directives to synchronize threads. The MASTER directive specifies that the code enclosed within the MASTER and END MASTER directives is executed by the master of the team. The CRITICAL and END CRITICAL directives restrict access to the enclosed code to only one thread at a time. The BARRIER directive synchronizes all the threads in a team. Other synchronization directives are the ATOMIC, FLUSH and ORDERED.

Data environment constructs

The data environment constructs controls the data environment during the execution of parallel constructs. The THREADPRIVATE directive makes named common blocks and named variables private to a thread but global within the thread. There are several data attribute clauses that specify, for instance, if a particular variable is PRIVATE or SHARED among threads in a team.

2.7.4 MPI+OpenMP

Last years, clusters of SMP's have appeared as a possible architectural structure that combines the benefits of the two approaches: the scalability of distributed-memory machines and the easiness of use of shared-memory architectures. The programming model proposed for these new architectures is an hybrid programming model: MPI+OpenMP. This programming model is also useful in DSM's such as the SGI Origin2000. Figure 2.20 shows the behavior of a MPI+OpenMP application. It tries to exploit the advantage provided by both programming models.

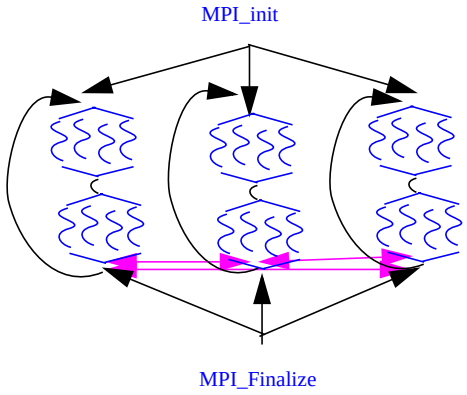


Figure 2.20: MPI+OpenMP

2.8 Summary

In this Chapter, we have briefly described the main multiprocessor architectures focusing in the SGI Origin 2000. The SGI Origin 2000 has been the target architecture of this Thesis because the availability and because it is representative of modern shared-memory multiprocessors architectures.

We have also presented an overview of the main approaches in scheduling policies. We classify scheduling policies in job scheduling, processor scheduling, and loop scheduling. This Thesis focuses on job scheduling and mainly in processor scheduling policies, then loop scheduling policies are not presented.

Finally, we have presented the two main programming models used to parallelize sequential applications: MPI and OpenMP. MPI is based on the idea that the application will be executed in a system with separated address spaces. Then, it partitions the problem in different task that will be executed in separated address spaces. On the other hand, OpenMP is based on the idea that the application will be executed in a system with a global shared address space. OpenMP parallelizes applications by specifying which parts of the code (mainly loops) can be executed in parallel. It delegates the decision of the parallelization to the run-time library, who takes this decision at the run-time based on the loop scheduling policy, application parameters such as the data size, and the number of processors available. We have also briefly presented a new programming model resulting from the combination of MPI and OpenMP. It is based on the idea of exploiting characteristics of systems that combines separated address spaces and global address spaces. This programming model incorporates the benefits and problems of the two previously presented programming models.

CHAPTER 3

Execution Environment

Abstract

In a multiprocessor environment with parallel applications concurrently running, the Operating System is responsible for optimizing the system utilization and the individual application execution. The system utilization depends on several factors such as the number of processors assigned to each application, or which particular processors are assigned to each application.

In this work, we present the particular characteristics of the three elements that constitute our execution environment: the long-term scheduler or queuing system (Launcher), the medium-term scheduler or simply the scheduler (CPUManager), and the runtime parallel library (NthLib). These three elements provide us a total control of how applications are scheduled. Having the control of these elements, we have a powerful tool to analyze the effect of the different scheduling issues of both the performance of the applications and the system utilization.

3.1 Introduction

In this Thesis, we use a practical approach. To do that, we have created our execution environment to implement the coordinated scheduling and the processor scheduling policies proposed in this Thesis. Figure 3.1 shows the elements that compound our execution environment: the queuing system (Launcher), the processor scheduler (CPUManager), and the run-time parallel library (NthLib [63][64]). In this Chapter, we give a lot of details about the implementation of these elements. If the reader is not interested in these details, the reading can be continued in Chapter 4.

The Launcher is the queueing system. It implements the job scheduling policy that decides which particular application must be executed at any moment. The Launcher controls the multiprogramming level, that can be defined by the administrator (if the Launcher works uncoordinated with the CPUManager), or by the processor scheduling policy (if it works coordinated with the CPUManager). The Launcher has been implemented to introduce repeatability in the submission of workloads of parallel applications with the aim of evaluating them under different execution environment configurations.

The CPUManager is the user-level processor scheduler. Once the Launcher starts the execution of a queued application, it enters under the CPUManager control. It implements the processor scheduling policy, which (1) decides how many processors to allocate to each application, and (2) enforces the processor scheduling policy decisions. The CPUManager uses the native operating system interface to manage processes and processors and provides the interface to communicate with the Launcher, see Figure 3.1.

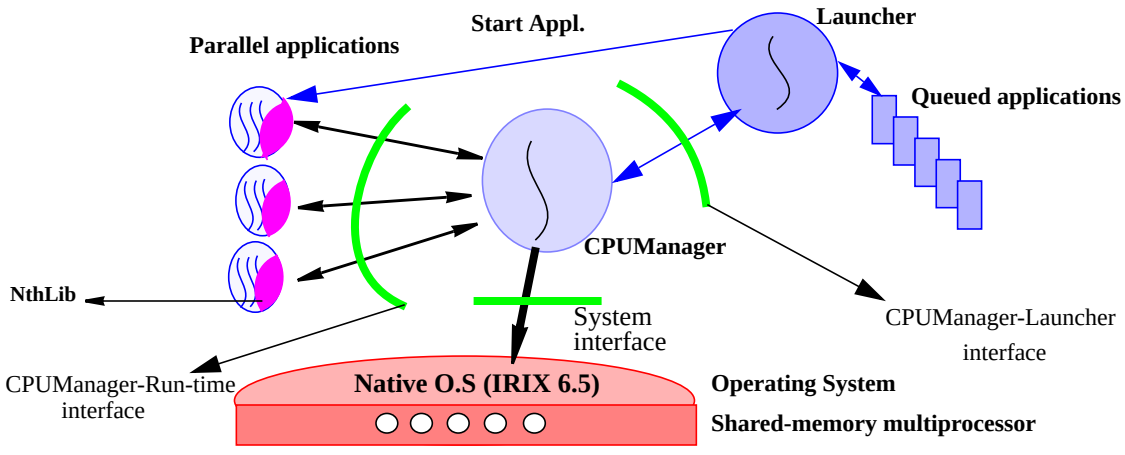


Figure 3.1: Execution environment

The NthLib [63][64] is the run-time library used in this Thesis to implement the policies and mechanism needed at the loop scheduling level. The NthLib was developed in the NANOS project [62][63][64] and modified in this Thesis. The NthLib supports the parallelism specified by users through OpenMP directives. It requests for processors using the CPUManager interface and reacts to changes in the number of processors allocated to the application.

The remainder of this Chapter is organized as follows: Section 3.3 presents the queueing system implemented in this work, the Launcher. Section 3.4 describes the internal structure of the CPUManager and the execution environment offered to parallel applications. Section 3.5 presents the characteristics that a parallel library should have to cooperate with the CPUManager, and a particular implementation of these features in NthLib. Finally, Section 3.6 presents the summary of this Chapter.

3.2 Related work: Resource Managers

The motivation to implement our own resource manager is clear, we need to have a complete control of this element. However, we comment in this Section some of the commercial resource managers that we can found.

Several computing industries are releasing operating system-based resource managers (RM). These RM have been designed and implemented under different operating systems and with different goals but with common features. The main characteristic of a RM is that it is designed to provide the administrator or the final user a major control over the architectural resources: CPUS's, virtual memory, I/O bandwidth, etc.

RM typically provide users or administrators an API to specify user requirements. User requirements can be as simple as *"to set aside specific CPUs to specific applications"*[101], or to *"guarantee a minimum entitlement of CPU, memory, and disk bandwidth available to a group of processes."*[100].

RM are usually implemented at user-level and they use the native operating system tools to allocate and manage the resources. Most of them allow a great configurability but they require a high knowledge about the system or the particular process to specify either the resource allocation to a workload or the resource requirements of a process. For instance, in the HP-UX Workload Manager [100], the administrator should specify performance goals and priorities for the workloads, and assign a performance monitor to the workload to measure its performance. The AIX WLM [45] allows the administrator to define different classes of jobs and assign different level of resources to each one. The Solaris RM [101] works in a similar way. These RM require that an administrator defines the different classes of jobs, priorities among them and the amount of resources that each one needs. They are oriented to guarantee a certain resource reservation from the point of view of the user, to ensure a certain medium-term resource distribution.

Other RM attack the problem of the resource reservation from the point of view of the thread. They are mainly RM oriented to real-time and multimedia applications [21][71][72]. In this case, the resources are specified as a quality of service. This kind of RM allow the co-existence of real-time and multimedia with time-sharing applications ensuring the reservation of the resource.

The CPUManager differs from the previously proposed RM in two main points. First, the CPUManager is application oriented. The unit of resource allocation and management is the parallel application. Other RM (like the Solaris RM or the AIX WLM) work at a workload or user granularity. Alternatively, RM oriented to real-time and multimedia applications work at thread granularity. Second, the CPUManager works at a different level than the rest of proposed RM. The CPUManager works at a similar level than a traditional operating system does. The CPUManager not only reserve CPUS to a particular application, but also performs the mapping between processes and processors and controls the initial memory placement.

3.3 The queueing system: The Launcher

The Launcher is the user-level queuing system used in our execution environment. It implements the job scheduling policy, that decides which application to execute from a list of queued applications. The aim of the Launcher is to be able to execute a workload of parallel applications several times under different system conditions and processor scheduling policies.

The Launcher executes a workload of parallel applications specified through a workload trace file. It receives as parameters the workload trace file, the maximum multiprogramming level, and a file with a list of applications. Using the workload trace file, we are able to (1) execute the same set of applications under specified conditions (submit time, initial request, etc.), (2) measure the system performance, and (3) compare results achieved under the different conditions.

The workload trace file follows the *Standard Workload Format* (SWF) proposed by Feitelson in [102]. Figure 3.2 shows a portion of a workload trace file. Columns different from -1 are *job_number*, *submit_time*, and *application_number*. In our case, the application number refers to the list of applications received by the Launcher. First application is application 0, second application is application 1, and so on.

```
#job_number submit_time application_number
0 12 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 3 -1 -1 -1 -1
1 20 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 2 -1 -1 -1 -1
2 21 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1-1 -1 -1 -1
```

Figure 3.2: Standard Workload Format

If the CPUManager is running, the Launcher will detect it and will work coordinated with it. In this case, the multiprogramming level received as parameter is not used at all, and it is decided by the multiprogramming level policy implemented by the CPUManager. If the CPUManager implements a processor scheduling policy that does not include a multiprogramming level policy, such as the Equipartition, the CPUManager implements a policy that decides a fixed multiprogramming level.

If the CPUManager is not running, such as when executing the native SGI-MP scheduling policy, the Launcher uses the multiprogramming level received as parameter.

3.3.1 Workloads used in this Thesis

We have used workloads that represent the execution of a system where applications arrive following a Poisson inter-arrival process. Figure 3.3 shows the equation used to compute the inter-arrival rate of applications. P is the number of processors in the system

(64 in our case), U is the load of the system that we want to generate. It ranges from 0 to 1. T^1_i is the execution time of the application *i* in sequential. And FRAC is the maximum fraction of machine that we want this application uses (from 0 to 1).

For instance, if we execute four different applications and we want each one to demand about the same amount of cpu time, FRAC will be equal to 0.25. With the last equation in Figure 3.3, we calculate the inter-arrival frequency per application.

$$\lambda_i \times T_i^1 = P \times U_i \longrightarrow \lambda_i = \frac{P \times U_i}{T_i^1} \longrightarrow \lambda_i = \frac{P \times U_i \times FRAC}{T^1_i}$$

Figure 3.3: Equation used to generate the inter-arrival rate of application i.

To generate the workload trace file, we have implemented an application that receives a list of applications, a $1/\lambda_i$ per application, and a maximum time per workload, *max_time*. This application generates a workload trace file that represents a system where each $application_i$ is submitted following a $1/\lambda_i$ and during *max_time* seconds. Workloads generated in this Thesis have been limited to 300 seconds. However, it is important to note that *max_time* only limits the maximum submission time. The Launcher waits for the complete finalization of all the jobs submitted, and all the jobs submitted are considered in our evaluation.

We have generated five workloads using four applications. We have used the swim, hydro2d, and apsi from the SPECFp95 [99] and the bt from the NASPB [48]. Table 3.1 shows the sequential execution time and the speedups of each one with 8, 16, and 32 processors. We have selected these applications because they have different speedup characteristics. Swim has a super-linear speedup, bt has a high speedup, hydro2d has low speedup, and apsi has very bad speedup. The complete performance analysis of these applications and their speedup curves can be found in Chapter 4.

Table 3.1: Parallel applications

Characteristic/Application(input)	swim(ref)	bt.A	hydro2d(train)	apsi(ref)
Exec.Time. in Sequential	212.2 sec.	1066.21 sec.	223.7 sec.	99 sec.
Speedup with 8/16/32/48 cpu´s.	14.5/26.5/32.7/26.2	5.9/12.1/20.5/24.1	6.7/7.4/5.5/3.6	0.93/0.93/0.92/0.9

Workload 1 is composed by swim's and bt's. Each one fills the 50% of the system. Workload 2 is composed by bt's (50%) and hydro2d's (50%). Workload 3 is composed by bt's (50%) and apsi's (50%). Workload 4 is composed by swim's (25%), bt's (25%), hydro2d's (25%), and apsi's (25%). Workload 5 is composed by only bt´s.

We have selected these workloads because each one is composed by applications with different speedup characteristics. In workload 1, 100% of the applications are scalable, swim´s with super-linear speedup and bt´s with good scalability. Workload 2 has a 50% of scalable applications (bt´s) and a 50% of applications with a medium speedup (hydro2d´s). Workload 3 is composed by a 50% of applications with good scalability and a 50% of applications with very bad scalability (apsi´s). Workload 4 is a mix of 25% of applications of each type: 25% of super-linear applications, 25% of scalable applications, 25% of applications with medium speedup, and 25% of no scalable applications. In workload 5, 100% of the applications are scalable but they are not super-linear.

3.3.2 Interface between the CPUManager and the Launcher

In this Thesis, we propose to establish an interface between the processor scheduler and the queueing system, coordinating the two levels. This coordination consists of sharing information and to consider this information to take scheduling decisions. The portion of the interface implemented by the CPUManager is presented in the CPUManager Section.

The Launcher informs the processor scheduler when a new application starts and when an application finishes. And the processor scheduler informs the queueing system when it can start a new application. Other information could be added such as the number of free processors, or the expected execution time of applications.

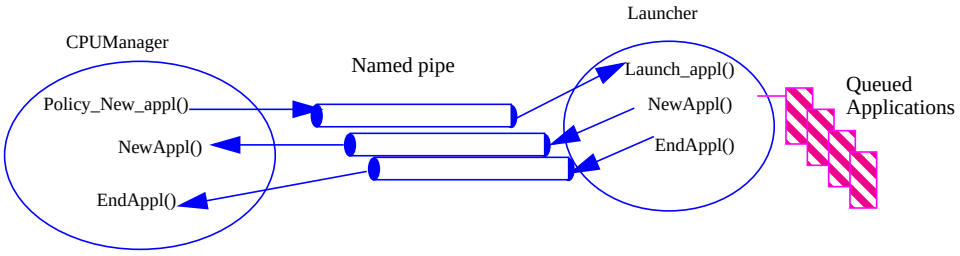


Figure 3.4: CPUManager-Launcher interface

In the current implementation, the CPUManager and the Launcher communicate through named pipes. A more complete version of the interface could be implemented using shared-memory but in this case it is efficient enough.

When the Launcher starts a new application, it sends the process identifier of the main thread of the application to the CPUManager. When the application finishes, the Launcher sends the process identifier of the main kernel thread to he application by other named pipe.

The CPUManager sends a byte to the Launcher each time the multiprogramming level policy decides that a new application can be started. Figure 3.4 shows the mechanism that implements the CPUManager interface.

3.4 The processor scheduler: The CPUManager

The CPUManager is a user-level scheduler. It implements the processor scheduling policy and enforces its decisions. It also implements the interface to coordinate with the Launcher and with the run-time library. The CPUManager uses the native operating system interface to enforce the processor scheduling decisions. In particular, it has been implemented on top of IRIX 6.5.

In order to implement a coordinated scheduler, the CPUManager and the other scheduling levels that communicate with it must agree in several rules related to the scheduling. The rules between CPUManager and the run-time library are the following:

- The run-time library has a list of work queues numbered from 0 to (maximum parallelism-1). However, the run-time library generates only work in the first P queues, where P is the number of processors available.
- The run-time library creates as many kernel threads as work queues. Therefore, it associates each kernel thread with a single work queue. The kernel thread executes the work inserted in the queue.
- When the CPUManager assigns one new processor to an application, the run-time library associates this processor with the first unallocated work queue.

Based on these three points, the CPUManager takes scheduling decisions such as deciding which kernel thread to associate to each processor, or deciding which kernel thread is more convenient to suspend when the CPUManager reduces the processor allocation of a running application. In the next subsections, we describe the CPUManager implementation.

The CPUManager was initially designed to implement space-sharing policies. In this Section, we explain the CPUManager under this kind of policies. Gang scheduling policies were implemented adding a time-sharing mechanism among jobs. Particular characteristics of Gang scheduling implementation are presented in Chapter 7.

3.4.1 CPUManager internal structure

The CPUManager wakes up periodically, at each quantum[1] expiration, and applies the processor allocation policy:

- It decides the **processor allocation** for the next quantum.
  - It decides **how many** processors to provide to each application.
  - It decides **which** processors to assign to each application.
  - It decides **which** kernel threads will run from each application.
  - It **maps** kernel threads with physical processors.
- It **communicates** its decisions to the applications.
- It **enforces** the processor allocation.

1. A typical quantum value is 100ms

Each one of these phases have several possibilities. The decisions concerning of which physical processors and which kernel threads will run are quite related, but we will consider them as independent phases.

Details about data used by the CPUManager are explained in Section 3.4.6. However, to understand the different phases we present the main data used by the CPUManager in Figure 3.5. The CPUManager has a *physical processor table*, with one entry per physical processor. Each entry records the application to which the physical processor is allocated (or NULL if it is free), and the last application to which it was allocated. The CPUManager also has a *job table*, with one entry per job. Some of the data associated to each job are the number of processors allocated, the number of processors requested, and a *kernel thread table*. This *kernel thread table* corresponds with the work queues managed by the run-time: the first entry corresponds to the first work queue, the second entry corresponds to the second work queue, and so on. The CPUManager records, per kernel thread, the physical processor identifier (if it is currently running), the last processor where it ran, the kernel thread status (RUNNING, PREEMPTED, etc), and the (process identifier / thread identifier). The per job table is allocated in a memory region shared by the run-time library and the CPUManager, and used to implement the interface between the CPUManager and the run-time library.

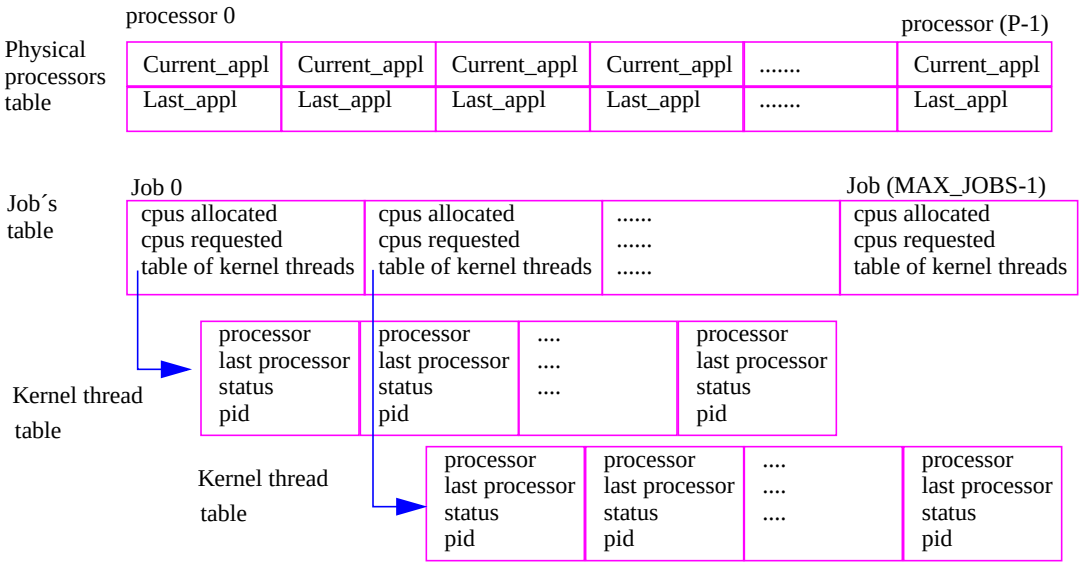


Figure 3.5: Main data structures managed by the CPUManager

3.4.2 Processor allocation

Processor scheduling policy decisions (how many processors)

The CPUManager implements the processor scheduling policy which is in charge of deciding how many processors will be allocated to each job in the next quantum.

The processor scheduling policy uses the *job table*. The processor allocation policy generates a new processor allocation, a temporal table, with an entry per job. This table has the number of processors allocated to each job.

Figure 3.6 shows a possible processor allocation generated by a processor allocation policy in a machine with 16 processors. The only information that this phase generates is a number of processors that the job will receive during the next quantum.

Job identifier	1	2	3	4	5
allocation	2	1	8	1	4

Processors=16

Figure 3.6: List of processor allocation

The only condition that the CPUManager imposes to the processor allocation policy is that the total number of processors allocated must be less or equal than the number of physical processors of the machine.

This is the only phase of the CPUManager that we will modify and evaluate in this Thesis. The particular processor scheduling policies implemented and evaluated are presented in Chapters 5, 6, and 7. Other aspects of the CPUManager have been fixed because they are out of the focus of this Thesis. However, to have a complete processor scheduler it has been necessary to implement all the CPUManager phases. In next sections we describe the different choices made at each one.

Allocating processors to jobs (which processors)

Once the scheduling policy has decided how many processors will assign to each application for the next quantum, the following step is to decide which physical processors will be allocated to each job. We will refer to the algorithm followed to decide which processors are assigned to each application as the processor placement policy, see Figure 3.7. This phase receives the processor allocation table generated in the previous phase (*alloc*), the *job table (jobs)*, and the *physical processors table (phys_proc)*. Based on this information, it generates a new *physical processor table (next_phys_proc)*. Figure 3.7 presents the algorithm used for this purpose. In this phase, the *physical processor table* is not still modified, only a temporal version is generated. The *physical processor table* is effectively updated in the last phase of the CPUManager.

```
input: physical_processor_table (phys_proc), job_table (jobs), table with number of
processors per job (alloc)
output: temporal physical processor table (next_phys_proc)
placement_policy()
{
  for(cpu=0;cpu<MAX_CPUS;cpu++)next_phys_proc[cpu]=NULL;
  for (current_job=0;current_job<active_jobs;current_job++){
    if (jobs[current_job].current<=alloc[current_job]){
      maintain_cpus=jobs[current_job].current; // receives equal or more processors
    }else{
      maintain_cpus=alloc[current_job]; // receives less processors
    }
    maintain_n_first_processors(current_job,maintain_cpus);
  }
  for (current_job=0;current_job<active_jobs;current_job++){
    if (alloc[current_job]>jobs[current_job].current){ // The application receives more
processors
      alloc_last_n_processors(current_job,alloc[current_job]-jobs[current_job].current);
    }
  }
}

input: physical_processor_table (phys_proc), job_table (jobs), table with number of
processors per job (alloc)
output: temporal physical processor table (next_phys_proc)
maintain_n_first_processors(int job, int cpus)
{
  for (kthread=0;kthread<cpus;kthread++){
    curr_cpu=job_table[job].kernel_threads[kthread].cpu;
    next_phys_proc[curr_cpu]=job;
  }
}
input: physical_processor_table (phys_proc), job_table (jobs), table with number of
processors per job (alloc)
output: temporal physical processor table (next_phys_proc)
alloc_last_n_processors(int job, int cpus)
{
  for(kthread=job_table[cpu].current;kthread<job_table[cpu].current+cpus;kthread++){
    curr_cpu=job_table[job].kernel_threads[kthread].last_cpu;
    // if the job has never run previously, we look for a free cpu near the rest of cpus
    // allocated to the job
    if(curr_cpu==NULL)curr_cpu=select_new_cpu(job);
    next_phys_proc[curr_cpu]=job;
  }
}
```

Figure 3.7: CPUManager processor placement algorithm

In a CC-NUMA machine, like the Origin 2000, the placement of processors has a significant influence in the execution time of parallel applications. For instance, if we assign separated processors to a parallel application, it will pay the cost of accessing remote pages. The aim of this phase is to select the more *convenient* set of processors per application.

With the aim of exploiting, as much as possible, the data locality, the CPUManager implements a placement policy oriented to maintain the processor affinity. It tries to execute the job in the same set of processors that in the last quantum.

The *placement_policy* function calculates how many processors from the previous quantum are kept for the next quantum per application. The *maintain_n_first_processors* function looks into the kernel thread table of the job and selects the cpus allocated to the first P work queues (from 0 to *alloc*[job]-1).

To those applications that will receive more processors, the *alloc_last_n_processors* function selects those cpus where kernel threads, from current to the new allocation, ran the last time. In the case that these kernel threads had never run, the function *select_new_cpu* selects new cpus to run them. This function tries to allocate a new cpu following three criteria: (1) a free cpu where the job has run previously, (2) a free cpu near the rest of cpus of the job, and (3) any free cpu.

After this phase, The CPUManager will have a *physical_processor_table* with the previous quantum distribution and a temporal *physical_processor_table* with the new processor distribution. Next phases will suspend and resume kernel threads to enforce the new processor distribution.

Selecting the set of kernel threads to execute each job (which kernel threads)

In the previous phase, the CPUManager has selected the set of physical processors that will run on each application. In this phase, it selects the set of kernel threads that will run from each application.

We have decided that, at any moment, each application will have as many kernel threads running as physical processors assigned, trying to execute in a efficient *operating point*, this is the "*process control*" approach proposed by Tucker and Gupta in [105]. Following the criteria commented in the introduction, the CPUManager will select the kernel threads associated with the first N work queues.

Figure 3.8 shows the algorithm used by the CPUManager to decide which kernel threads will run in the next quantum.

```
input: job_table (jobs), temporal physical processor table (next_phys_proc), table with
number of processors per job (alloc)
output: job_table (jobs)
select_kernel_threads()
{
  for(current_job=0;current_job<active_jobs;current_job++){
    for(kthread=0;kthreads<alloc[job];kthreads++)
      jobs[current_job].kernel_threads[kthread].tmp_status=SELECTED_TO_RUN;
    for(kthread=alloc[job];kthreads<MAX_KTHREADS;kthreads++)
      jobs[current_job].kernel_threads[kthread].tmp_status=SELECTED_TO_SUSPEND
  }
}
```

Figure 3.8: Kernel thread selection algorithm

Mapping the kernel threads to physical processors (map kernel threads)

Once decided which physical processors and which kernel threads will run in the next quantum, the CPUManager must establish the mapping among them. This phase receives the temporal physical processors table and the *job_table*, and modifies the kernel thread table of each job to decide the mapping between each kernel thread and the physical processor. The difference from the second phase is that in the second phase we select a set of processors to run the application and in this phase we assign one processor to each kernel thread. The two phases are very related but we have implemented it separately.

Figure 3.9 shows the algorithm used by the CPUManager to map kernel threads to physical processors. The CPUManager will maintain in the same cpu those kernel threads that were currently running. If the kernel thread is not currently running, The CPUManager selects the last cpu where the kernel thread ran. In the last case (it is a new kernel thread), the *select_new_cpu* function selects a cpu from the temporal *physical_processors_table* not yet allocated to any kernel thread.

Once finished this phase, the CPUManager has completely decided the new processor distribution for the next quantum, but not yet enforced it. In the next Section, we will describe the different CPUManager options to decide the moment at which the processor allocation decisions are enforced.

```
input:job_table (jobs), temporal physical processors (next_phys_proc), table with number of
processors per job (alloc)
output: job_table (jobs)
map_kernel_threads()
{
  for (current_job=0;current_job<active_jobs;current_job++){
    for(kthread=0;kthread<alloc[current_job];current_job++){
      cpu=jobs[current_job].kernel_threads[kthread].cpu;
      if (cpu==NULL){
        cpu=jobs[current_job].kernel_threads[kthread].last_cpu;
        if (cpu==NULL) cpu=select_new_cpu();
      }
      jobs[current_job].kernel_threads[kthread].cpu=cpu;
      }
    }
  }
}
```

Figure 3.9: CPUManager mapping algorithm

3.4.3 Enforcing the CPUManager decisions

Before the end of each activation, the CPUManager should enforce the processor allocation, which may involve suspending some *running* threads and resuming some *suspended* threads. This enforcement is done by using the native operating system calls. Table 3.2 shows the main system calls [46] provided by IRIX to manage processes. They are part of the system interface shown in Figure 3.1.

Table 3.2: IRIX system calls used in the CPUManager to manage processes

Functionality	System call
Resume kernel thread	unblockproc(pid)
Suspend kernel thread	blockproc(pid)
Bind kernel thread to processor	sysmp (MP_MUSTRUN_PID, cpu, pid)
Unbind kernel thread	sysmp (MP_RUNANYWHERE_PID, pid)

When the CPUManager suspends a thread, it could be executing the application code or the run-time code. If the kernel thread is executing application code, the CPUManager may suspend a kernel thread when holding a lock or just before ending its work. Suspending a kernel thread in one of these points could be critical, since it may cause the remaining threads to be delayed.

In order to avoid such problems, the CPUManager can work in two different modes, according to the moment when its decisions are enforced:

•*Immediate* mode: the allocation is effective before the end of the current activation of the CPUManager. CPUManager decisions are enforced in a synchronous way before sleeping for the next quantum.
•*Deferred* mode: the allocation is effective, at least, at the beginning of the next CPUManager activation. Changes have to be carried out by the applications themselves. At the beginning of each activation, the CPUManager checks that all their decisions made at the previous activation have been accomplished. This mode corresponds to the *two_minute_warning* mechanism proposed by Markatos et al. in [61].

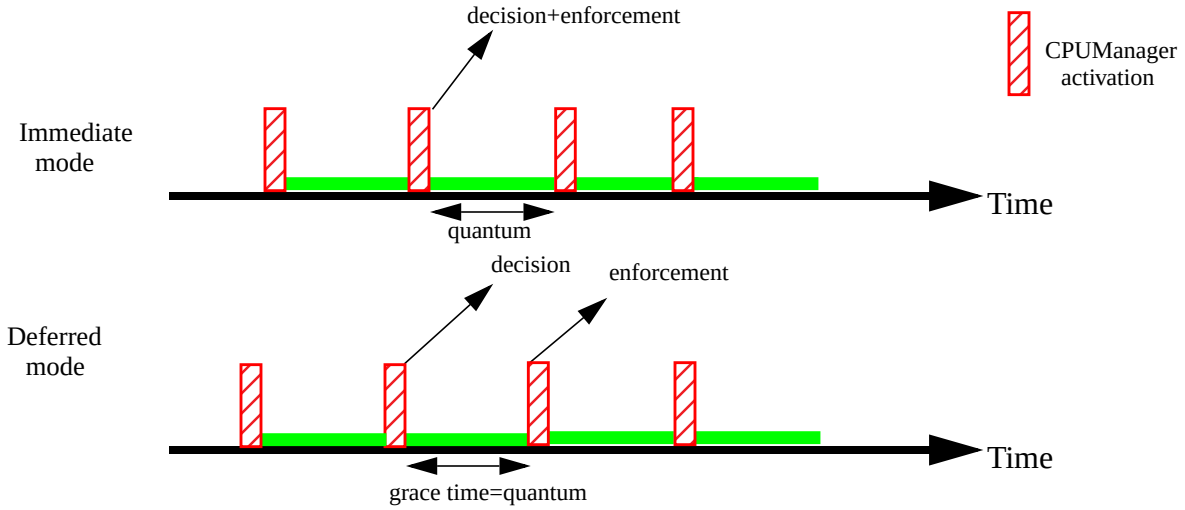


Figure 3.10: Immediate mode vs. Deferred mode

Figure 3.10 shows the different points at which the CPUManager decisions are enforced in the immediate mode and in the deferred mode.

Immediate mode

When the CPUManager works in immediate mode, it enforces the allocation decided just before the end of the current activation. When the CPUManager decides to take out some processors from an application, it changes the status of the selected threads from *running* to *suspended (*we refer to this situation as a *preemption*, and to these threads as *preempted*). Preempted threads must be recovered by the parallel library because they could be executing application code at the time of the preemption. To recover a preempted thread means that a *running* thread from the same application has to *handoff* its processor to the preempted thread until it finishes the work that it was executing. Figure 3.11 shows the function that enforces the CPUManager when executing in immediate mode.

```
input:      jobs_table(jobs),      physical_processor_table      (phys_procs),      temporal
physical_processor_table (next_phys_procs)
output: jobs_table(jobs), physical_processor_table (phys_procs)
enforce()
{
  for(cpu=0;cpu<MAX_CPUS;cpu++){
    phys_procs[cpu].last_appl=phys_procs[cpu].current_appl;
    phys_procs[cpu].current_appl=next_phys_procs[cpu];
  }
  for (current_job=0;current_job<active_jobs;current_job++){
    for(kthread=0;kthread<MAX_KTHREADS;kthread++){
      cpu=jobs[current_job].kernel_threads[kthread].cpu;
      pid=jobs[current_job].kernel_threads[kthread].pid;
      // One more processor
      if ((jobs[current_job].kernel_threads[kthread].tmp_status==SELECTED_TO_RUN) &&
          (jobs[current_job].kernel_threads[kthread].status==PREEMPTED)){
        jobs[current_job].kernel_threads[kthread].status=RUNNING;
        jobs[current_job].current++;
        sysmp (MP_MUSTRUN_PID, cpu,pid); //Binds the kernel thread to the cpu
        unblockproc(pid); // Resumes the kernel thread
      }
      // One less processor
      if ((jobs[current_job].kernel_threads[kthread].tmp_status==SELECTED_TO_SUSPEND) &&
          (jobs[current_job].kernel_threads[kthread].status==RUNNING)){
        jobs[current_job].kernel_threads[kthread].last_cpu=cpu;
        jobs[current_job].kernel_threads[kthread].cpu=NULL;
        jobs[current_job].current--;
        jobs[current_job].kernel_threads[kthread].status=PREEMPTED;
        sysmp (MP_RUNANYWHERE_PID, cpu,pid); //Unbinds the kernel threads
        blockproc(pid); // Suspends the kernel thread
      }
    }
  }
}
```

Figure 3.11: CPUManager enforcement function

If the number of processors assigned to an application changes frequently[2], the immediate mode can cause a significant number of thread migrations in order to recover preempted threads. To minimize these inopportune preemptions, we have implemented the deferred mode.

Deferred mode

The *deferred* mode is implemented through the *two_minute_warning* technique. It is a technique to minimize undesirable preemptions. The idea of this technique is to inform parallel applications that they have to release some processors, and give them the opportunity to release the processors while running in a safe state. The CPUManager informs applications setting a flag indicating that they have to release some of their processors, and applications have to check this flag.

2. This is not a desirable situation but depends on the scheduling policy

Applications are granted a *grace-time* by the CPUManager to release the processors. If this *grace-time* expires, and any application has not released its processors, the CPUManager will preempt its processors and will enforce the re-allocation.

Deciding which processors must be released is not a trivial task. The first option is that the CPUManager decide the set of processors to be released. The second option is that the parallel application selects the processors itself.

In our particular implementation, we have implemented the first approach. We have taken this decision because the CPUManager has a more global view of the system and its capacity to decide which processors have to migrate is better than the local view of the application.

The CPUManager provides applications with a list specifying the processors that they have to release, and a list with the applications that will receive each processor. With this information, applications can release a processor and allocate it to the new owner. In order to make easy the implementation, we have considered that the *grace-time* be equal to a quantum. We also assume that all the applications behave *friendly* and they will respond to the CPUManager request as soon as possible.

This technique involves cooperation among the CPUManager and parallel applications. The communication between the CPUManager and the run-time library has been implemented using shared-memory.

Since the *two_minute_warning* does not entirely avoid the inopportune preemptions, a recovery mechanism is still needed.

The *two_minute_warning* reduces the number of preemptions, which means less preempted threads and less thread migrations to recover the preempted threads. On the other hand, this technique works asynchronously. That means that the recovery mechanism must be very accurate. If the recovery mechanism is not very accurate, there is a risk that it enters in a cyclic phase, all the threads recovering all the threads. The part of the recovery mechanism implemented in the NthLib is explained in Section 3.5.1 .

We have selected the *two_minute_warning* mechanism because it is an accepted proposal in the bibliography. However, we have observed that in our execution environment, and with the processor scheduling policies implemented, the two-minutes warning mechanism does not introduce significant benefits. We have observed the behavior of some workload executions to see why this mechanism does not introduce significant benefits. The two-minutes warning mechanism was designed for an execution environment without coordination and where processor re-allocations where frequent. In our execution environment, processor scheduling policies try to maintain stable, as much as possible, the processor allocation and to coordinate the different scheduling levels. This observation demonstrate us, with a real example, that in an execution environment with coordination between levels and inside levels, some techniques that shown very

important in other environments, in our case are not needed. This result, enforces our Thesis about the necessity of a coordinated execution environment and a global design of scheduling policies.

For these reasons, even we have implemented it, we have not used the *two_minute_warning* mechanism in the evaluation of this Thesis.

3.4.4 Interface between the CPUManager and the Launcher

The CPUManager implements five functions to control the multiprogramming level: *Init_Multiprogramming_Level*(), *Dynamic_Multiprogramming_Level*(), *StartAppl*(), *NewAppl*(), and *EndAppl*().

Figure 3.12 shows the functions implemented by the CPUManager. The *Init_Multiprogramming_level*() function initializes the multiprogramming level value (*ML*) to a *Default* value, defined by the administrator, sets the number of running applications to 0 (*TotalAppls*), and initializes the number of pending applications to *Default*. This function is executed only once. The *Dynamic_multiprogramming_level*() function is executed at each activation of the CPUManager. This function executes the multiprogramming level policy if there are not pending applications.

In this case, we have shown the example of a fixed multiprogramming level: *Fixed_New_appl*(). This function returns TRUE if the multiprogramming level is less than a given *Default* multiprogramming level. In that case, the *ML* is increased and the queueing system is notified. The interface also includes two functions *NewAppl*(), executed each time a new application is started, and *EndAppl*(), executed each time an application finishes. These functions maintains consistent the values of *ML* and *TotalAppls*.

This multiprogramming level policy, *fixed*, has been used with the Equipartition and the Equal_efficiency evaluated in this Thesis.

```
Init_Multiprogramming_level(int Default)
{
  ML=Default; pendings=Default;
  TotalAppls=0
}
Dynamic_multiprogramming_level()
{
  if (pending==0)
    if (Fixed_New_appl()) StartAppl();
}
StartAppl()
{
  pending++;
  ML++;
  NotifyLauncher(); /* writes in the pipe */
}
```

```
NewAppl()
{
  pending--
  TotalAppls++;
}
EndAppl()
{
  TotalAppls--;
  ML--;
}
int Fixed_New_Appl()
{
  if (ML<Default) return TRUE;
  else return FALSE;
}
```

Figure 3.12: CPUManager-Launcher interface

The *Fixed_New_appl*() function will be substituted by the particular multiprogramming level policy implemented by the processor scheduling policy.

3.4.5 Interface between the CPUManager and the Run-Time library

The CPUManager basically implements the interface specified in the NANOS project to communicate the run-time parallel library with the processor scheduler. The basic set of functions that this interface defines is the following [63]:

Table 3.3: CPUManager-NthLib interface

Function	Description
int cpus_request(int ncpus)	Sets the number of processors the application would like to run on
int cpus_requested()	Informs about the number of requested processors
int cpus_current()	Informs about the number of currently assigned physical processors
int cpus_askedfor()	Returns TRUE if the calling thread is marked to be released
int cpus_release_self()	Releases the current physical processor.
int cpus_preempted_work()	Informs about the number of preempted kernel threads
work_t cpus_get_preempted_work()	Returns the identifier of a previously preempted work
int cpus_processor_handoff(work_t work)	Attempts to transfer the current physical processor to the previously preempted work
int cpus_future()	Returns the number of processors the application will have at the end of the current quantum

This interface is offered by the CPUManager to the run-time library. It is implemented through shared-memory between the CPUManager and the run-time library. In the next sub-section, we present the data structures managed by the CPUManager that implement this interface. Functions written in bold font have been implemented or modified in this Thesis[3].

We have introduced two modifications respect to the original NANOS specification. The first change is the functionality of the *cpus_askedfor*(). In the original NANOS specification *cpus_askedfor()* returns true if the application has to release some thread. In the current implementation, it only refers to the calling thread. It returns true if the calling thread is assigned to a physical processor selected by the CPUManager to be re-allocated to another application. The second difference is that the CPUManager implements a new function: *cpus_future()*, which informs about the number of processors that the application will have at the end of the current quantum.

3.4.6 Shared Data Structures

The CPUManager manages three data structures: jobs, kernel threads, and physical processors. The main fields associated to these data have been commented previously. In this Section, we detail data associated with each one. Jobs and kernel threads are shared between CPUManager and run-time. Physical processors are private to the CPUManager.

Jobs

The job is the unit of processor allocation. The CPUManager has a table of jobs, and each job has the following information:

- Job identifier
- maximum parallelism
- requested number of processors, *request*
- number of *running* threads, *current*
- processors that the job will have at the end of the next quantum, *future*
- number of preempted[4] threads, *preempted*
- list of processors to be released by the application
- speedup and execution time tables
- kernel threads table

Each job has a unique identifier automatically generated by the CPUManager. When the application starts, it communicates its maximum parallelism to the CPUManager. The maximum parallelism is the maximum number of processors that the application will require. Therefore, the application can dynamically change its processor request from 1 to its maximum parallelism. Its dynamic processor request is set in the *request* field, and it is one of the parameters that the processor allocation policy takes into account when it distributes processors. For instance, the application can set to 1 the *request* when it enters in a sequential phase or when it starts an I/O or it can reduce its *request* when it executes a loop with only a few iterations.

Current is the number of assigned/running processors that the application has at any moment. *Future* is the number of processors that the application will have assigned/running, by the end of the current quantum. When the CPUManager runs in *immediate* mode, values of *current* and *future* are always equal because re-allocations are

3. Some of them had only been specified but not implemented.
4. A preempted thread is a thread that has been suspended by the CPUManager

synchronous. On the other hand, when the CPUManager runs in *deferred* mode, the number of assigned and running processors can be different during a *grace-time* after the execution of the CPUManager. This is because the processors are asynchronously re-allocated.

The CPUManager maintains a counter of preempted threads by application, set in *preempted*. It does not maintain a list of preempted threads since this information is available checking the status of each kernel thread. When the CPUManager preempts a thread, its work has to be recovered by the application. The applications use the *preempted* value to check whether they have any preempted thread. The CPUManager preempts a thread either when it works in *immediate* mode and an application will receive less processors in the next quantum, or when the CPUManager works in *deferred* mode and the application has not reacted to the CPUManager requirements to release a processor. Each application has associated a list of processors to be released. This list is empty when working in *immediate* mode.

The CPUManager has the speedup and execution times tables in the shared memory. This information is used by the processor allocation policies that consider performance information. This table is modified by the run-time that measures the application performance and read by the processor scheduling policy.

Finally, each application has a kernel threads table. As we pointed out in the introduction, each kernel thread is associated to a work queue, numbered from 0 to (maximum parallelism -1).

Kernel threads

A kernel thread is the unit of scheduling to which a CPU can be assigned to execute it. The CPUManager associates the following information to each kernel thread:

- pid
- status
- *cpu_id*, if RUNNING, identifier of the physical processor, otherwise a NULL value.
- last cpu where it run

Each kernel thread has a unique identifier. In our case, it is the identifier that the operating system assigns to the kernel thread (pid). If the kernel thread is RUNNING, the CPUManager updates *cpu_id* with the identifier of the physical processors where it is running.

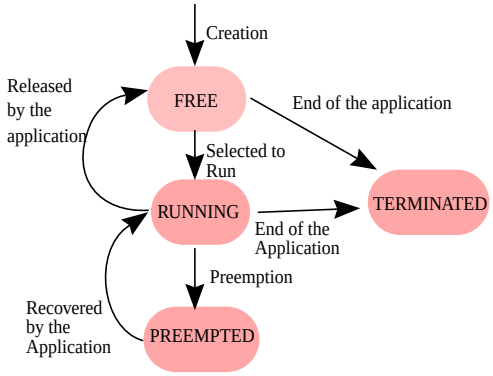


Figure 3.13: State diagram of a kernel thread

Kernel threads can be in four logical states: FREE[5], RUNNING, PREEMPTED or TERMINATED. Figure 3.13 shows the state diagram of a kernel thread. Kernel threads are created FREE when the application starts. When the CPUManager selects a kernel thread for execution, its status changes to RUNNING. If the application voluntarily releases the physical processor associated to a kernel thread and suspends it, its status changes to FREE. On the other hand, if the CPUManager suspends the thread, it becomes PREEMPTED. When an application ends its execution all its kernel threads become TERMINATED. A PREEMPTED thread must be recovered by a RUNNING thread from the same application because it could be executing application code when preempted. Finally, each kernel thread has a field that indicates the last cpu where it run.

Physical processors

The information associated to each physical processor is:

- current job, *current_appl*
- Identifier of the last application that runs in this processor, *last_appl*

If the physical processor is allocated to an application, the CPUManager will set in *current_appl* the identifier of the application. Otherwise, it will have a NULL value. The *last_appl* field contains the identifier of the last job that ran in this processor.

5. In both, *free* and *preempted* states, the kernel thread is suspended

3.5 Run-time library features

Parallel applications interact with the CPUManager through a run-time library. In this Section, we present the main features that a parallel library must include to coordinate with the CPUManager, and the main improvements introduced in this Thesis to NthLib [63][64], the run-time library used in this Thesis.

The parallel library communicates with the CPUManager in order to provide scheduling information: processor requirements, etc. And the CPUManager informs the parallel library about the number of processors assigned to it, and about the thread preemption.

The run-time library follows the rules specified in the introduction of Section 3.4.

Since the CPUManager can preempt application kernel threads, the run-time library has to provide a work recovery mechanism. The work recovery mechanism implies that some of the remaining threads must finish the pending work. This event also implies that the parallel library has to periodically check whether it has some preempted threads in order to recover it.

3.5.1 NthLib modifications

The nano-threads library, NthLib, is a user-level thread package mainly designed to support efficient parallel execution and good adaptability to the varying system conditions. It is designed primarily to support fine-grain parallelization and multiple levels of parallelism in shared-memory multiprocessors. It is further described in [63][62].

In this Thesis, we have modified the work recovery mechanism, implemented the two-minutes warning mechanism, the dynamic kernel thread creation/destruction, and activated the dynamic memory migration mechanism.

3.5.2 Work recovery mechanism

The work recovery mechanism is needed when a *running* thread is suspended by the CPUManager, becoming *preempted*. The work recovery mechanism has to guarantee that the pending work of the preempted thread is finished by another processor. This mechanism must be implemented "*with care*", avoiding entering in recovering cycles. In this Section, when we refer to kernel thread 0 (kt0), we mean the kernel thread associated to work queue 0, and so on.

,

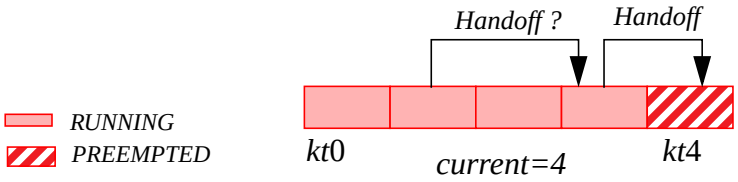


Figure 3.14: kt0 will handoff its processor to kt4 to finish its pending work.

Figure 3.14 shows an example about a possible problem related with the work-recovery mechanism. In this example, kt3 has detected that kt4 is preempted. It handoffs its processor to kt4 to finish its work. If kt1 detects that kt3 does not have processor (because it is recovering kt4), it could conclude that kt3 needs help to finish its work, when this is not the case.

This set of possibilities implies that the work recovery mechanism must specify:

- which thread can recover a preempted thread
- what recovering a thread means
- which threads need to be recovered

To specify the work recovery mechanism, we classify the kernel threads into two groups: those that are associated to work queues with identifier less than the *current* number of processors allocated, and those that are associated to work queues with identifier greater or equal than *current*. We will refer to the first set as the "*allocated zone*" and to the rest as the "*unallocated zone*".

The work recovery mechanism also introduces two new thread states: Recovering another thread and Being Recovered. The list of thread states then is the following: RUNNING, FREE, PREEMPTED, Recovering another thread (RECOVERING) and Being Recovered (BEING_RECOVERED).

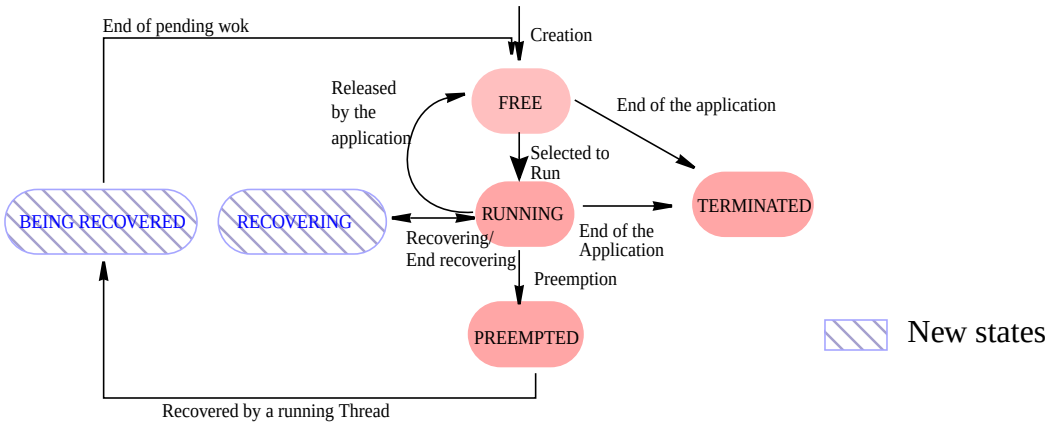


Figure 3.15: State diagram of a kernel thread with recovering mechanism modifications

Figure 3.15 shows the state diagram of a kernel thread after incorporating the new states resulting of the work recovering mechanism. We specify that only threads in the allocated zone and in state RUNNING can recover another thread. And the only threads that need to be recovered are threads in the unallocated zone in state PREEMPTED.

When a thread finishes its pending work, it executes a piece of code called *idle loop*. The *idle loop* continuously checks whether there is pending work and whether there are preempted threads.

When the kernel thread enters in the idle loop, it can be in four different situations (each kernel thread must check these situations and in this order):

- Marked as BEING_RECOVERED
- RUNNING, and there are PREEMPTED threads that need help
- RUNNING, and there are not PREEMPTED threads, (the normal/stable situation)

If the kernel thread is BEING_RECOVERED, it must handoff its processor to the kernel thread that previously helped it. To know that, the run-time has a per thread field (*return_to*) that specifies to which kernel thread it has to return.

If the kernel thread is RUNNING, it has to check if there are PREEMPTED threads that need help. In that case, if the work queue of the thread is in the allocated zone, it will get the first PREEMPTED thread, it will mark itself as RECOVERING, the PREEMPTED thread as BEING_RECOVERED (setting the *return_to* field), and finally, it will handoff its processor to the PREEMPTED thread.

If the thread is RUNNING and there is not PREEMPTED work, the thread has to iterate in the idle loop until it receives new work to execute.

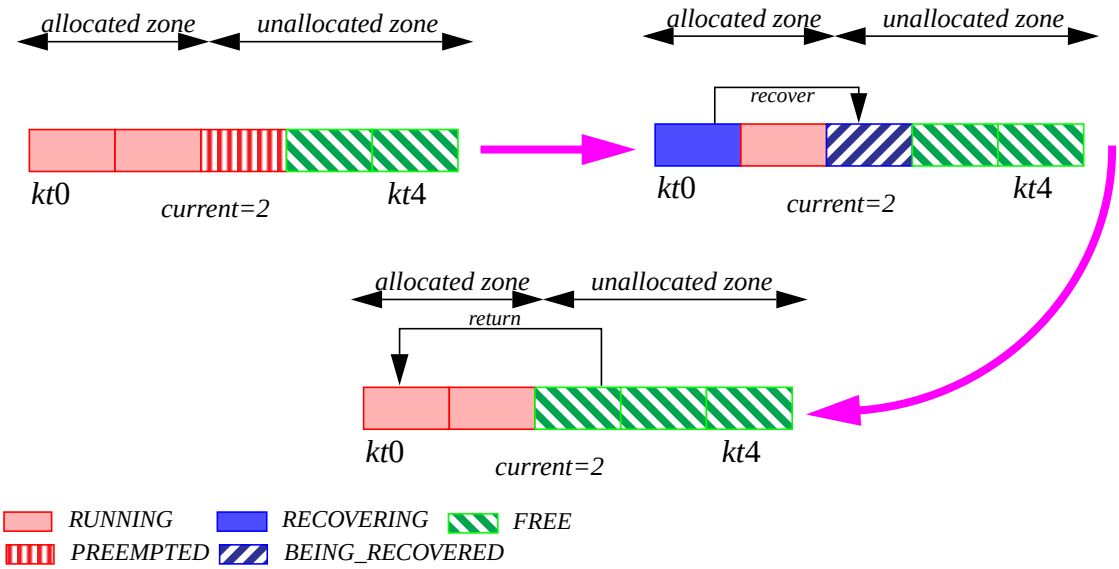


Figure 3.16: Recovery mechanism, kt0 recovers the work of the kt4 and returns to its queue.

Figure 3.16 shows an example about how the work recovery mechanism works. When kt0 detects that kt2 is preempted, it handoffs its processor and marks kt4 to return to work queue 0. When kt2 finishes its pending work and enters in the idle loop, it detects that it has to return to work queue 0, and then it handoffs its processor to kt0.

Figure 3.17 shows the work recovery algorithm: the *idle loop*, and the *cpus_get_preempted_work*() function.

```
idle_code()
{
  ...
  // If I was being_recovered I must return to the original work queue
  if (myself->status==BEING_RECOVERED){
    myself->status=FREE;
    my_job->kernel_threads[myself->return_to]->status=RUNNING;
    cpus_handoff(myself,my_job->kernel_threads[myself->return_to]);
  }
  // If there are preempted threads I must recover it,
  if ((cpus_preempted()>0) && (myself->wq_id<cpus_current())){
    // Only stable threads can recover preempted threads
      work_preempted=cpus_get_preempted_work();
      myself->status=RECOVERING;
      // The kernel thread must return to the original work queue
      work_preempted->return_to=myself->wq_id;
      work_preempted->status=BEING_RECOVERED;
      my_job->preempteds--
      // The kernel thread is lended
      cpus_handoff(myself,work_preempted);
  }
  ....
}
kthread * cpus_get_preempted_work()
{
  // The algorithm look for a thread in the unallocated zone and PREEMPTED
  for (kthread=cpus_current();kthread<MAX_THREADS;kthread++){
    if (my_job->kernel_threads[kthread]->status==PREEMPTED)
      return my_job->kernel_threads[kthread]
  }
}
```

Figure 3.17: Work recovery mechanism

3.5.3 Two_minute_warning mechanism

The two_minute_warning mechanism implies modifications in the CPUManager and in the run-time library: Decisions that in the immediate mode are executed synchronously by the CPUManager, in the deferred mode are executed asynchronously by the run-time library. These modifications introduce two main problems: in one hand the asynchronous behavior, and in the other hand the fact of giving responsibility to the run-time.

The deferred mode modifies the idle loop. In this mode, kernel threads have also to check if they are marked to be released. They will check this condition before checking if there are PREEMPTED threads.

The fact of releasing the processor to another application is much more complicated that it initially seems. There are several possible implementations of this technique, but we have decided that the run-time library will implement the same policy than the CPUManager. To do that, the run-time library needs access to the information of the rest of applications to have access to their kernel thread tables (pid, status, etc). The run-time also needs a new information: the CPUManager informs it about which specific kernel threads must be released and to which job it must be allocated. Since this is a research

environment, we have not taken into account security issues and we have made accessible all the CPUManager data structures to the run-time library. Another approach could consist of allocating the application kernel thread to an idle thread of the CPUManager and that this thread performs the context switch to the new job. However, we have adopted the first approach: allocation decisions have been moved from the CPUManager to the run-time library.

```
while (application not finished){
...
  if (cpus_asked_for())
    cpus_release_self()
....
  if (pending_work())execute(work)
}
```

Figure 3.18: Modifications in the idle loop

If the physical processor is marked to be released, the executing thread has to allocate its processor to the new owner and deallocate itself, see Figure 3.18. The *cpus_asked_for*() function returns TRUE if the kernel thread has been marked to be released. In that case, it executes the *cpus_release_self*() function. This function looks into the temporal processor allocation table to see the target application associated with it and applies the same processor placement algorithm implemented by the CPUManager presented in Section 3.4.2.

The two_minute_warning mechanism also affects the work recovery mechanism. When working in deferred mode, the criteria to differentiate between the allocated and unallocated zone can not be the value of *cpus_current()* because there is a fraction of time in which this value changes frequently (when the application is releasing threads).

The solution is to use the value of the *cpus_future*() function to differentiate the allocated zone and the unallocated zone. The *cpus_future*() function returns the number of processors that will be allocated to the application at the end of the current quantum. In this case, threads that can recover PREEMPTED threads are those that are allocated to work queues with identifier less than *cpus_future*().

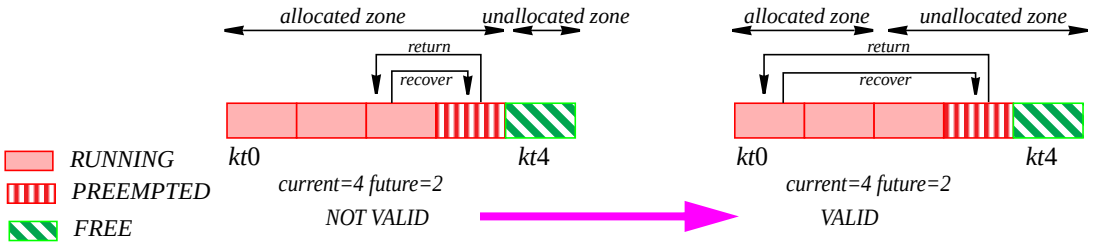


Figure 3.19: Modifications in the work recovery mechanism introduced by the deferred mode.

Figure 3.19 shows an example about the necessity of modifying the work recovery mechanism. In the left side, there is an example about a possible situation when executing in deferred mode and directly applying the work recovering mechanism explained in the previous Section. In this case, kt3 is in the allocated zone and it detects that there is a PREEMPTED thread. However, kt2 is a kernel thread that has to release its processor, and it can not recover PREEMPTED kernel threads.

On the right side of Figure 3.19, there is an example about how the work recovery mechanism works when executing in deferred mode with the modification introduced in this Section. The allocated zone is determined by cpus_future(), not cpus_current(). The cpus_future() value is fixed during the complete quantum. Using this value, the only kernel threads that can recover PREEMPTED threads are kt0 and kt1.

3.5.4 Demand based thread creation

We have also incorporated a mechanism to dynamically create and destroy the kernel threads in NthLib. The aim of this technique is to adjust, as much as possible, the number of kernel threads to the number of processors allocated.

This technique is motivated by the fact that it is a common practice to statically generate kernel threads. Usually, run-time libraries create as many kernel threads as the maximum parallelism specified by the user at the submission time. This implementation is efficient enough if the total number of requested processor by active applications is not very high. The problem is that in the case of dynamic processor scheduling policies, it is possible to reach situations such as a job executing in 4 processors and with 64 kernel threads created. In this kind of extreme situations, it can result in a system saturation due to the lack of resources, for instance filling the complete process table.

Our proposal is to dynamically create and destroy the kernel threads. We propose to create a few more kernel threads than processors allocated to the job, limited by the maximum parallelism specified by the job.

The CPUManager implementation is affected by this modification because we decided that the CPUManager performs the allocation of processors to processes. With this new mechanism, the CPUManager can try to allocate a processor to a work queue and it can find that the associated kernel thread is not yet created. To manage this new situation, we have modified the status diagram associated to kernel threads, and we have included more information in the interface between the CPUManager and the run-time library.

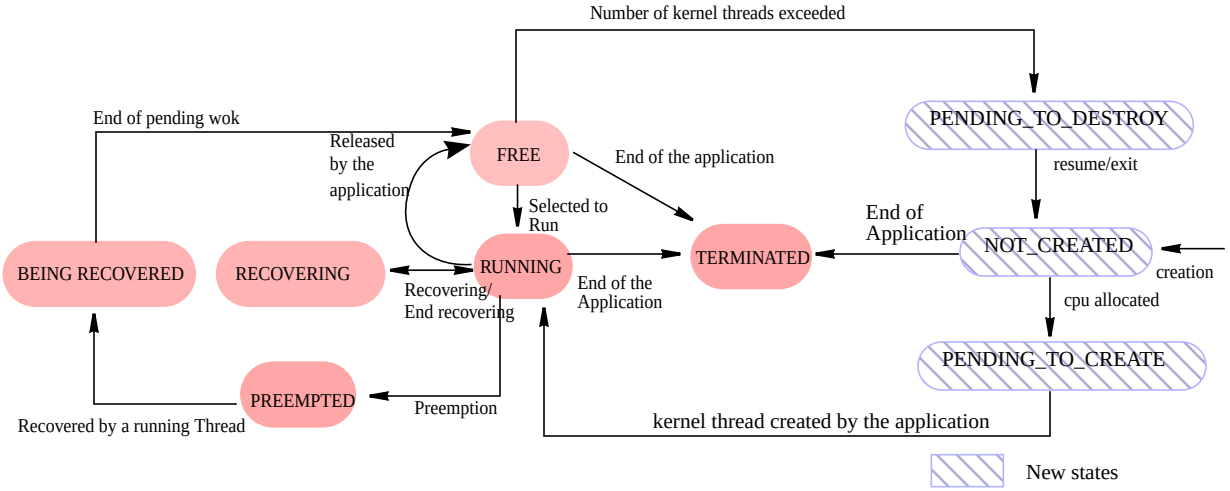


Figure 3.20: Kernel thread state diagram with dynamic thread creation

CPUManager modifications

We have included three new states: NOT_CREATED, PENDING_TO_CREATE, and PENDING_TO_DESTROY, see Figure 3.20.

We have also included two new fields per job in the job table, *cpus_pending* and *cpus_pending_to_release*. The *cpus_pending* is the number of kernel threads pending to be created by the application. This value will be consulted by the run-time through a new function in the CPUManager-run-time interface: *cpus_pending*(). The *cpus_pending_to_release* is a boolean that indicates that the application has an excessive number of kernel threads, compared with the number of cpus allocated, and that it must destroy some kernel threads. This value is consulted by the run-time library through the *cpus_pending_to_release*() function.

When the application starts, NthLib initializes the data associated with the first work queue with the information of the main process. The rest of work queues, do not have kernel threads associated to them. These work queues are initialized as NOT_CREATED.

When the CPUManager allocates more processors to the application, it follows the algorithm presented in Section 3.4.2. However, it checks, per work queue, if the associated kernel thread is created or not. If the kernel thread is not created, the CPUManager marks the kernel thread as PENDING_TO_CREATE and increments the *cpus_pending* value.

The last state, PENDING_TO_DESTROY, is set by the NthLib. When the CPUManager considers that the application has an excessive number of kernel threads (compared with the number of cpus allocated). In that case, the CPUManager sets the *cpus_pending_to_release* field. Then, when the NthLib detects that it has to release some kernel threads, it selects some of them and marks them as PENDING_TO_DESTROY.

We have implemented the decision of what is an excessive number of kernel threads using two different thresholds: a global and a local threshold. If the ratio between the total number of processes created and the number of physical processors is greater than the global threshold, the CPUManager will consider that the system is heavily loaded. In that case, it calculates the ratio of (processes created / processors allocated) per application. If this second ratio is greater than the local threshold, the CPUManager sets the *cpus_pending_to_release* of this application. In the current implementation, the global threshold has been set to 2 and the local threshold to 1.25.

Moreover, we use a third level of system load control. The CPUManager implements a *security threshold* to avoid a system crash due to lack of resources (kernel threads in this case). The CPUManager calculates the relationship between number of kernel threads and number of cpus. If this ratio is greater than the *security threshold*, no new applications are allowed to execute. This *security threshold* works coordinated with the multiprogramming level and modifies any multiprogramming level decision. In the current implementation, the *security threshold* has been set to 6.

Run-time library modifications

We have also modified the run-time library to periodically check if the job has kernel threads pending to create, or if it has to destroy kernel threads. This code has been introduced in the idle loop. If *cpus_pending*() returns a value greater than zero, the run-time will look for all the kernel threads in PENDING_TO_CREATE state, and it will create one kernel thread per work queue. Once created, the run-time decrements the counter *cpus_pending*, changes the thread state from PENDING_TO_CREATE to RUNNING, and resumes the kernel thread.

More complicated is the dynamic destruction of kernel threads. As in the previous case, we have introduced some code in the idle loop to detect if the job has to reduce the number of kernel threads (checking the *cpus_pending_to_release*() function). In that case, kernel threads associated with work queues identified with the highest numbers are selected to be destroyed. The number of kernel threads selected are calculated using the local threshold. Each kernel must destroy itself: the run-time marks selected threads as PENDING_TO_DESTROY, and resumes it. When kernel threads enter in the idle loop, they detect that they must exit.

Table 3.4 shows the IRIX system calls used to create and destroy kernel threads.

Table 3.4: IRIX system calls

Functionality	System call
Create kernel thread	sproc(...)
Delete kernel thread	exit(..)

3.5.5 Memory management

In CC-NUMA architectures, processes should be scheduled in processors near their memory pages to achieve a good system performance. There are two ways to enforce that: doing a good initial memory mapping, and using memory page migrations. The first choice is only valid if the application memory accesses follow a static pattern. This technique needs an individual analysis of each parallel application, and the processor allocation must be fixed during the complete execution of the application. The second choice, memory migrations, will allow us to work without any knowledge about the application memory behavior.

The native operating system used in this Thesis, IRIX 6.5, has a *dynamic page migration* mechanism. Dynamic page migration is a mechanism that provides adaptive memory locality to applications that execute in a NUMA machine. The migration mechanism checks the memory pages and decides whether a page must be migrated, depending on a migration policy. In this Section, we present the modification that we have introduced in the run-time to activate the dynamic page migration mechanism.

Policy Modules

Users are allowed to select a policy from a set of available policies for each virtual memory operation. Virtual memory operations are: Initial allocation (Placement policy, page size policy, and fallback policy), Dynamic relocation (migration policy and replication policy), and paging policy. Any portion of a virtual address space, down to the level of a page, may be connected to a specific policy via a Policy Module.

When the operating system needs to execute an operation to manage a Section of a process address space, it uses the methods specified by the Policy Module connected (attached) to that Section.

In this Thesis, we have modified the placement policy and the migration policy associated with all the code, data, and stacks of parallel applications.

Table 3.5: Policy modules functions

Function name	Description
pm_filldefault(...)	Fills a policy_set with predefined default values
pm_create(...)	Creates a policy module
pm_setdefault(...)	Selects a new default policy for stack, text, or heap.
pm_attach(...)	Connects a policy module to a virtual address space range

Table 3.5 shows the Policy Module functions used in the run-time library to modify the placement policy and the migration policy. The *pm_filldefault*() function fills a policy_set_t structure with the predefined values in the system. A policy_set_t structure has fields to

specify the following policies: placement policy name and arguments, fallback policy name and arguments, replication policy name and arguments, migration policy name and arguments, paging policy name and arguments, and page size.

Once we have a default Policy Module we can modify it. In this Thesis, we have modified the placement policy fields and the migration policy fields. We have set the placement policy to *PlacementFirstTouch* and the migration policy to *MigrationControl*. *PlacementFirstTouch* indicates that memory will be allocated in the node where creation happened. *MigrationControl* indicates that users can specify different migration parameters. The *MigrationControl* policy receives as argument a data structure that defines these user parameters. Once modified the Policy Module, we have to create the handle that we will use to associate to the memory regions. The Policy Module creation is done using the *pm_create*(..) system call, that receives as a parameter the Policy Module previously filled up.

Once created, it only remains to attach the Policy Module with the memory regions. In this case, we have created only one Policy Module because we want to apply the same policy to all the application memory regions. The association of a Policy Module with a memory region is done through the *pm_attach*( ) function. This function receives as parameter the Policy Module, the address of the memory region, and the size of the memory region.

The *pm_setdefault*() function associates a policy module to the stack, text, or heap memory regions. This function has been used to modify the default policy associated to these memory regions.

Migration parameters

The SGI Origin2000 hardware implements a competitive algorithm based on comparing the remote memory access counters to the local memory access counters. When the difference between remote and local accesses is greater than a tunable threshold, an interrupt is generated to inform the Operating System than the physical memory page is suffering an excessive number of remote references. The interrupt handler decides whether the page has to be migrated or not. The final decision depends on several controls that can limit the page migration.

Figure 3.21 shows the sequence of code used in the NthLib to create a PM with the dynamic memory migration mechanism activated. We first fill the Policy Module with the default values, and modify the placement and migration policies. Once filled up, we create it with the *pm_create*() function. The Policy Module created is used to modify the policy associated to code, data, and stacks. This default policy is inherited at *fork* or *sproc* time, and a new one is created at exec time.

The last memory regions that we have to modify are user stacks. These stacks are not allocated from the heap, then we have to attach them explicitly using the *pm_attach*() function. The *pm_attach*() function must be executed once per allocated user stack.

```
migr_policy_uparms migr_args;
policy_set polset;
pmo_handle_t PM;
pm_filldefault (&polset);

migr_args.migr_base_enabled = 1; // The rest of fields are default values
migr_args.migr_base_threshold = 50;
migr_args.migr_freeze_enabled = 0x0;
migr_args.migr_freeze_threshold = 0x14;
migr_args.migr_melt_enabled = 0;
migr_args.migr_melt_threshold = 0x32;
migr_args.migr_enqonfail_enabled = 0;
migr_args.migr_dampening_enabled = 0;
migr_args.migr_dampening_factor = 0x5A;
migr_args.migr_refcnt_enabled = 0;
polset.placement_policy_name="PlacementFirstTouch";
polset.placement_policy_args =NULL;
polset.migration_policy_name = "MigrationControl";
polset.migration_policy_args = &migr_args;
PM = pm_create (&polset);
pm_setdefault(PM,MEM_STACK);
pm_setdefault(PM,MEM_TEXT);
pm_setdefault(PM,MEM_DATA);
....
pm_attach(PM,stack_address,page_size);
....
```

Figure 3.21: Policy Module creation and migration mechanism activation

Activating the dynamic page migration mechanism, memory pages can be automatically migrated by the system. Experiments done in this Thesis have been performed activating this mechanism because it has been shown useful. Our experience also shows that this mechanism is useful in those environments where the process reallocation frequency is not very high. Otherwise, the mechanism is not able to migrate memory pages to follow the movements of the processors.

3.6 Summary

In this Chapter, we have described the three main elements in our execution environment: The queueing system (Launcher), the scheduler (CPUManager), and the run-time library (NthLib).

The Launcher controls the arrival of parallel applications and implements the job scheduling policy. The job scheduling policy decides the order of application execution. The Launcher has allowed us to execute workloads in a controlled way, that means under certain load conditions and with a specified job scheduling policy. Using the Launcher we have evaluated the processor scheduling policies presented in this Thesis under the same execution conditions.

The CPUManager is a user-level scheduler that implements the processor allocation policy and enforces its decisions. In this Chapter, we described the different phases that implements the CPUManager and the decisions adopted on each one.

Finally, we presented characteristics that a run-time library must accomplish to interact with the CPUManager. We present the particular case of the NthLib, the run-time library used in this Thesis. We have introduced several modification such as the implementation of the two_minute_warning mechanism, the work recovering mechanism, the memory page migration, or the dynamic thread creation. We have give a lot of details about our processor scheduler implementation and about issues that usually are not described, such as the processor placement function, but that in fact, have a great influence in the performance of the scheduling policies.

CHAPTER 4

Dynamic Performance Analysis: SelfAnalyzer

Abstract

In this Chapter, we present the first contribution of this Thesis: the SelfAnalyzer. The SelfAnalyzer is a run-time library that dynamically calculates the speedup of parallel applications and predicts their execution time.

The speedup is the relationship between the sequential and the parallel execution times of an application. It is typically calculated by executing jobs several times with different number of processors and measuring it statically. However, the speedup depends on run-time parameters such as the input data, the number of processes migrations, the distance between processes and memory, or the interference with other running applications. In this Thesis, we defend that the system must have its own criteria to evaluate the performance of applications, in addition to the user provided information, which can be considered as a first hint.

For these reasons, our first goal in this Thesis is to evaluate the system hability to dynamically measure the application performance, in particular, to measure the speedup of parallel applications.

Results show that we are able to measure the performance of parallel applications without significant interferences in the execution time of the applications.

4.1 Introduction

Many researchers have considered the use of application characteristics such as the speedup to improve the performance of the operating system scheduler [13][76]. Other application characteristics used are, for instance, the fraction of sequential code (*f*) [1], or the average parallelism (*A*) [30]. These application characteristics are mainly used to statically determine limits in the maximum speedup that a parallel application can reach, and not to request for more processors to the system. The use of these characteristics sometimes implies a new degree of difficulty to users, and does not provide as much information as the speedup. For these reasons, we consider more useful to directly work with the speedup.

The speedup is the relationship between the sequential and the parallel execution times of an application, see Figure 4.1. Traditional approaches compute the speedup statically by executing the sequential and the parallel versions several times with different number of processors, in order to provide this information to the scheduler as an *a priori* input. However, the speedup obtained by a parallel application depends on several factors such as its input data, the architecture, or the placement of the processors and some of these factors are only available at run-time.

$$\text{Speedup}(p) = \frac{T(1)}{T(p)}$$

Figure 4.1: Speedup equation

In this Chapter, we present the *SelfAnalyzer*, a new approach to measure at run time the speedup of parallel applications, avoiding the necessity of *a priori* information. The SelfAnalyzer also estimates the application execution time.

We have executed several parallel applications with the machine in dedicated mode, and calculated their speedups. Results show that the speedup calculated by our approach matches with the speedup calculated with the traditional approach. We have also analyzed the overhead introduced by our mechanism and we have found that it is acceptable.

The rest of this Chapter is organized as follows: Section 4.2 presents the related work. Section 4.3 describes our approach to dynamically compute the speedup in parallel applications and to estimate the execution time of parallel regions. Section 4.4 describes how to insert the SelfAnalyzer in parallel applications. Section 4.5 describes changes in the execution environment to incorporate the measurements taken by the SelfAnalyzer. Section 4.6 presents the evaluation of our proposal, including both a validation of the dynamically calculated speedup and an analysis of the overhead. Finally, in Section 4.7 we summarize the main conclusions of this Chapter.

4.2 Related Work

To obtain the characteristics of an application, previous systems have adopted approaches such as the use of hardware counters provided by the architecture, or monitoring the execution time of the different phases of the application. Weissman [108] uses the performance counters provided by modern architectures to improve the thread locality. McCann *et al* [65] monitor the idle time consumed by processors. Nguyen *et al* [74][75] combined both, the use of hardware counters and the measurement of idle periods of the applications.

The most studied characteristic of parallel applications has been the speedup. Several theoretical studies have analyzed the relation between the speedup and other characteristics such as the efficiency. Speedup is defined for each number of processors *P* as the ratio between the execution time with one processor and with *P* processors. Efficiency is defined as the average utilization of the *P* allocated processors. The relationship between efficiency and speedup is shown in Figure 4.2.

$$S(P) = \frac{T(1)}{T(P)} \longrightarrow E(P) = \frac{S(P)}{P}$$

Figure 4.2: Speedup and efficiency definitions

Helmbold *et al* analyze in [43] the causes of loss of speedup and demonstrate that the super-linear speedups may appear basically due to memory cache effects.

Nguyen *et al* [74] propose *self-tuning,* to dynamically calculate the efficiency achieved by parallel applications. In order to calculate the efficiency, they measure the different sources of overhead that cause loss of efficiency, and subtract them from 1[1]. Figure 4.3 shows the formulation proposed by Nguyen. The sources of overhead considered are: the *system overhead*, the *idleness* and the *processor stall time*[2]. These components were obtained by using the hardware counters provided by the architecture, and by instrumenting the parallel library.

1. They assume that the efficiency of a parallel application ranges from 0 to 1
2. It appears when processors have to access data in a remote cache memory.

$$\text{Efficiency}(p) = 1 - \left(\frac{WT(p) - UT(p)}{WT(p)} - \frac{IT(p)}{WT(p)} - \frac{PST(p)}{WT(p)} \right) \longrightarrow \text{Speedup}(p) = \text{Efficiency}(p) \times p$$

WT(p)= elapsed execution time with p processors
UT(p)= accumulated user-mode execution time
IT(p)= accumulated idle time
PST(p)= accumulated processor stall time

Figure 4.3: Nguyen proposal to calculate the efficiency

However, one of the major limitations of this method is that it is closely dependent on the architecture. In some current multiprocessors systems, such as the SGI Origin 2000 [93][110], the *stall time* cannot be measured, since the corresponding hardware counter is not provided by the architecture. On the other hand, this formulation does not consider an interesting effect that may appear when running an application in parallel, the *super-linear* speedup (i.e. when the speedup achieved with *P* processors is greater than *P*) [43]. It may occur when the accumulated number of cache misses of all processors that are running a parallel application is lower than the number of cache misses of the sequential application.

We defend that issues such as the super-linear speedup can only be detected by calculating the speedup as the ratio between two measurements. Therefore, the main difference between this work and our proposal is that we compute the speedup as the ratio between two measurements whereas their approach calculates the speedup just using one measurement.

4.3 Dynamic Performance Analysis: The SelfAnalyzer

The *SelfAnalyzer* is a run-time library that dynamically calculates the speedup achieved by parallel applications and estimates the execution time of parallel applications[3].

The *SelfAnalyzer* exploits the iterative structure of a significant number of scientific applications. The main time-consuming code of these applications is composed by a set of parallel loops inside a sequential loop (*iterative parallel region),* see Figure 4.4. Iterations of the sequential loop have a similar behavior among them. Then, measurements for a particular iteration can be considered to predict the behavior of the next iterations.

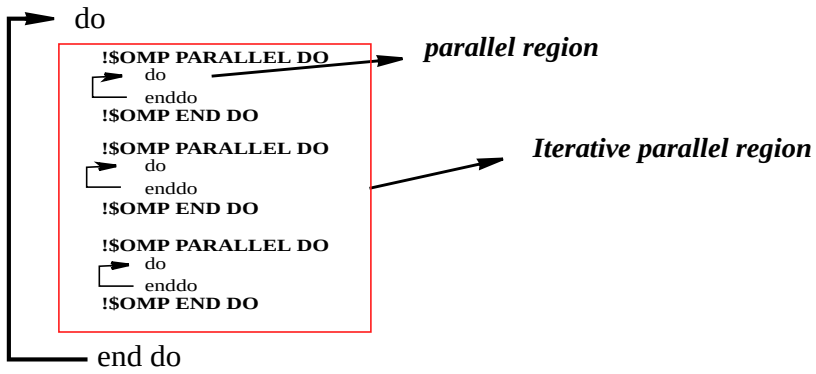


Figure 4.4: Structure of a *Iterative parallel region*

This characteristic is shared by a large number of scientific applications. In [76] it is shown that five out of ten SPLASH applications, the seven PerfectClub, and seven out of ten SpecFP95 applications are iterative. For instance, iterative applications are those that perform computational fluid dynamics, as for instance the *tomcatv* from the SpecFP95, or those that perform weather prediction like the *swim*, from the SpecFP95. Other applications that also follow this scheme are some of the NAS Benchmarks [47], such as *BT* or *SP* and real applications such as car crash simulations.

4.3.1 Dynamic speedup computation

We defend that the speedup must be calculated as the relationship between two measurements: the sequential, or *baseline* execution time, and the parallel execution time. Figure 4.5 shows the formulation used by the *SelfAnalyzer* to calculate the speedup.

3. When the application is mainly composed by one iterative parallel region, but this is a common case.

(1) $S(p) = \frac{T(Baseline)}{T(p)} \times AF(Baseline)$, where $AF(Baseline) = \frac{1}{\left(f + \frac{(1-f)}{Baseline}\right)}$ (2)

(3) if (baseline==1) AF(1)=1, $S(p) = \frac{T(1)}{T(p)} \times AF(1) = \frac{T(1)}{T(p)}$

Figure 4.5: Speedup calculation

The *SelfAnalyzer* measures the execution time of each outer iteration and also monitorizes the sequential and parallel regions inside the outer loop. The speedup is calculated as the relationship between the execution time of one iteration executed with a *baseline* number of processors and the execution time of one iteration with the number of processors allocated by the scheduler, multiplied by a factor that normalizes the speedup. This normalization factor is based on the Amdahl´s law and we refer to it as the Amdahl´s Factor (AF). This factor will allow us to compare speedups of two applications calculted with different baselines.

The SelfAnalyzer executes some initial iterations of the sequential loop with a predefined number of processors and measures the execution time. These measurements are averaged and used as the reference, T(*baseline*), for the speedup computation. The execution time of several iterations is averaged in order to achieve a more accurate measure. The execution of the parallel loop with *baseline* processors is managed by the NthLib and transparently to the scheduler. If the number of allocated processors (*P*) is lees than *baseline*, SelfAnalyzer will use *P* as *baseline*.

Once T(*baseline*) is computed, the application goes on measuring the execution time, with the number of processors allocated by the scheduler, T(p). In this case, the *SelfAnalyzer* also measures several iterations and calculates the average execution time. Note that, if T(*baseline)* is measured with one processor, that means *baseline*=1, the calculated speedup will correspond with the traditional speedup measurement, see (3) in Figure 4.5.

Amdahl's Factor

Using a baseline greater than one processor, and potentially different among applications, has the following problem: It does not allow us to directly compare speedups among applications. For instance, using a *baseline* of four processors, the speedup with four processors of an application that scales well will be one and the speedup with four processors of an application that scales poorly will be also one.

For these reasons we propose to normalize the calculated speedups using Amdahl's law [1]. Amdahl's law bounds the speedup that an application can achieve with P processors based on the fraction of sequential code, *f*.

Figure 4.6 shows the formulation used to calculate *f*. The SelfAnalyzer measures the execution time consumed by sequential code, (1) in the figure, and the execution time consumed by the parallel code, (2) in the figure. *f* is the fraction of not parallelized code. Since we do not have the execution time with 1 processor, we assume that (ParallelTime x CPUS) is an approximated value.

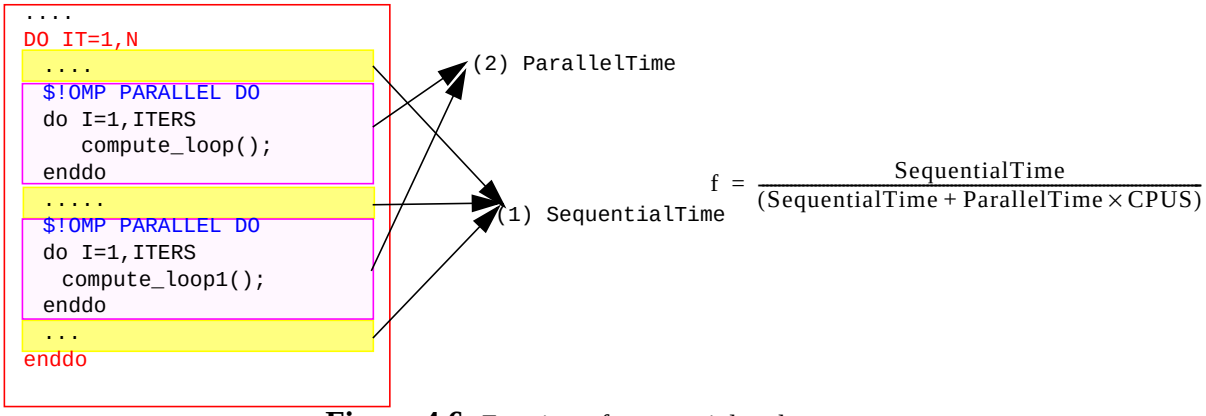


Figure 4.6: Fraction of sequential code

We call the function used to normalize the speedup the Amdahl's Factor (AF), see (2) in Figure 4.5. In this way, we calculate the AF of *baseline* and we use this value to normalize speedups calculated by the *SelfAnalyzer*. After normalizing speedups we can compare speedup values among applications. The AF is only useful in those applications that have their speedup limited by the Amdahl´s law, however, it is shown quite effective when using a *baseline* with few processors.

Implementation issues

The goal of using a baseline greater than one processor is to avoid the execution of some initial iterations with one processor becasue they could consume a lot of time. To define a *baseline* close to one processor has the advantage that provides a lot of information, the speedup calculated is similar to the traditional speedup (calculated with one processor). On the other hand, defining a baseline close to the number of processors allocated to the application reduces the time lost to measure the reference, but it reduces the measurement precision because we do not know what is the application behavior (respect to its scalability) from 1 to *baseline* processors. In the evaluation Section, we evaluate which is the more convenient value for *baseline*.

The speedup is continuously recalculated in order to detect both, variations in the behavior of the application performance, and variations in the number of allocated processors. Each time the *SelfAnalyzer* detects a variation in the number of processors it discards the execution time of the current iteration. If the *SelfAnalyzer* did not discard this iteration, the speedup would fall down due to the overhead introduced by data movements. These data movements are generated by the re-distribution of iterations.

Another issue concerning the implementation is that the *SelfAnalyzer* recalculates the speedup even when the number of processors does not change. We use a moving average function to recalculate it, see Figure 4.7. We assign a weight of 0.6 to the old speedup and a weight of 0.4 to the new speedup[4].

$$Speedup(P) = (SpeedupOld(p) \times 0.6) + (SpeedupNew(p) \times 0.4)$$

Figure 4.7: We calculate the speedup as a function of old and new speedup.

Using this function, small variations in the speedup values are directly filtered by the SelfAnalyzer, avoiding that these speedup fluctuations could be noted by the processor allocation policy resulting in an unstable processor re-distribution.

4.3.2 Execution time estimation

Considering characteristics of these parallel applications, and taking into account their iterative structure, we are able to estimate the execution time of the *iterative parallel region* (IPR). This is done by using the calculated speedup and the number of iterations that the application executes, see Figure 4.8. If the application is composed by only one IPR, a common case, the execution time estimation will correspond to the execution time of the complete application.

This estimation is calculated by adding the consumed execution time until the moment with the estimation of the remaining execution time. The remaining execution time is calculated as a function of the number of iterations of the IPR not yet executed and the speedup that the application is achieving on each iteration. The SelfAnalyzer knows the number of iterations of the sequential loop because the compiler provides it this information usign the SelfAnalyzer intreface.

$$ExTime(p) = ConsumedTime + \left(\frac{AF(Baseline) \times T(Baseline)}{S(p)} \times ItersRemaining\right)$$

Figure 4.8: Execution time estimation

The execution time of the parallel application can only be calculated if the total number of iterations is known. For this reason, the execution time of the application only can be calculated if the code has been statically instrumented. Application instrumentation is explained in the next Section.

Since this is a value that can not be always calculated, it is used in our scheduling policies just to tune the processor allocation, it is not a basic parameter.

4. We have tested other combinations and we have found that this provides the best accuracy

4.4 Application instrumentation

The SelfAnalyzer must instrument parallel applications in order to monitorize them and calculate their speedups. The SelfAnalyzer implements a set of functions that must be involved during the application execution. In this Section, we describe the SelfAnalyzer interface and the application instrumentation needed in order to be monitorized.

4.4.1 SelfAnalyzer interface

Table 4.1 shows the SelfAnalyzer interface.

Table 4.1: SelfAnalyzer API

Function	Description
init_parallel_region(ipr,length,iters)	Initialize data that should be initialized once per parallel region
end_parallel_region()	Marks the end of the parallel region
open_iteration()	Should be called at the start of each iteration
close_iteration()	Should be called at the end of each iteration
init_par()	Should be called before each parallel loop
end_par()	Should be called at the end of each parallel loop

- *init_parallel_region*. It initializes all the data structures that will be used by the SelfAnalyzer to monitor the execution of the parallel region and starts the time counters. The two parameters *ipr* and *length* identify the IPR. We will see that in the static instrumentation *ipr* is a number automatically generated by the compiler, and in the dynamic instrumentation is the address of the encapsulated loop. *Length* is the number of parallel loops that compound the IPR. The only condition that the SelfAnalyzer imposes is that each pair (*ipr, length*) must be unique during the execution of the application. The last parameter, *iters*, is the number of iterations of the application. If this parameter is not know, a negative value is required.
- *end_parallel_region*. It stops the time counters.
- *open_iteration*. It gets the current time and the number of processors. If the number of processors has changed since the last iteration, previous measurements are discarded. Otherwise, it continues the normal execution.
- *close_iteration*. It gets the current time and the number of processors available. If the number of processors has changed since the start of the iteration, the current measurement is discarded. Otherwise, if the number of measured iterations has reached a certain value, it calculates the speedup and informs the scheduler.
- *init_par* and *end_par*. They are called before and after each parallel loop inside the main sequential loop. These functions takes the time consumed by the parallel loops and sequential code to calculate the fraction of sequential code, *f*.

These functions should be called during the execution of the parallel application. They can be either inserted in the source code by the compiler or by the user (static instrumentation), or dynamically loaded at run-time (dynamic instrumentation).

4.4.2 Static instrumentation

Compilers that process OpenMP directives typically encapsulate code of parallel loops in functions. Figure 4.9 shows the resulting code after compiling the source code shown in Figure 4.4. Code in the left side, in Figure 4.9, shows the typical code generated by a compiler after processing the OpenMP directives.

The *omp_parallel_do* is part of the native run-time library interface. Threads execute that function and get a set of iterations of the parallel loop to be executed. Specific iterations depend of the thread identifier (*omp_get_thread_num*) and the loop scheduling policy applied (*static*, *dynamic*, *guided*, etc). In the NthLib there is an equivalent function to the *omp_parallel_do*.

The code in the right side shows the resulting code after inserting calls to the SelfAnalyzer. Note that (1) *init_parallel_region/end_parallel_region* are called only once per IPR, (2) *open_iteration/close_iteration* are called once per iteration, and (3) *init_par/end_par* are called after and before each parallel loop inside the IPR.

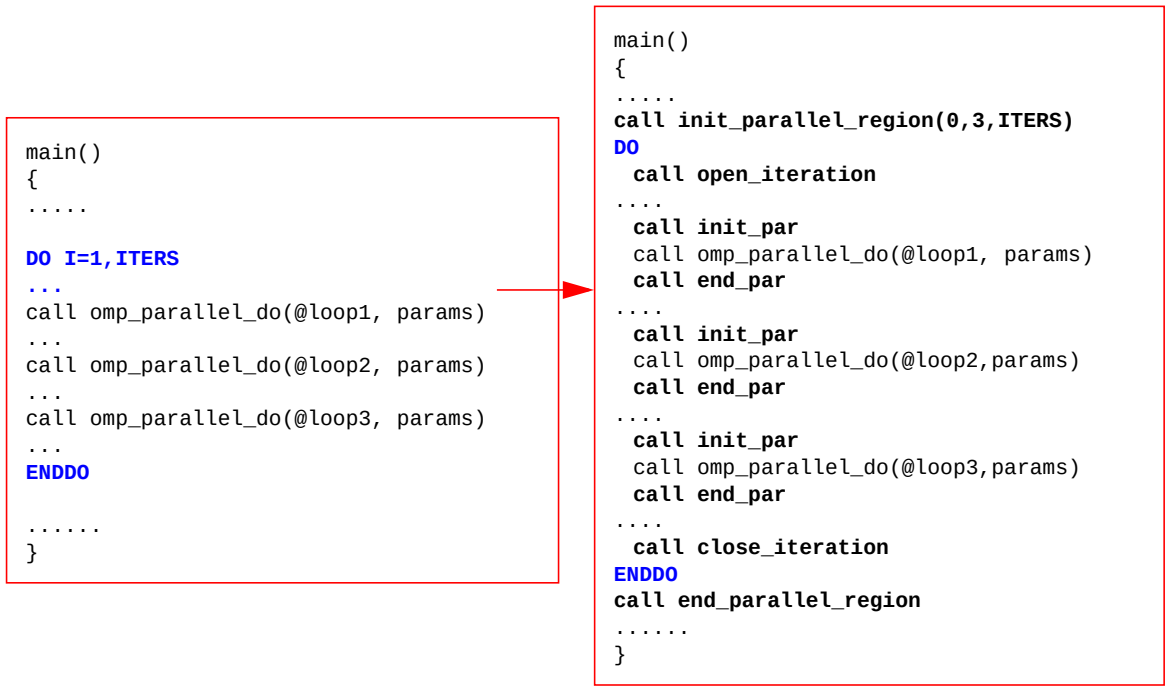


Figure 4.9: Static instrumentation

4.4.3 Dynamic instrumentation

If the source code of parallel applications is not available, we have two problems to instrument the parallel application. The first one is that we can not insert calls to the SelfAnalyzer interface in the code, and the second one is that we do not know the application structure, the iterative structure.

To solve these problems, we have used a dynamic interposition mechanism to be able to (1) insert calls to the SelfAnalyzer, and to (2) insert calls to a new run-time library to know the iterative structure of the application.

DITools is a *Dynamic Interposition Tool* proposed by Serra et al. in [87]. One of the DITools mechanisms allows us to define a list of functions that we want to dynamically intercept. It also allows to execute a code after and/or before these functions. Through DITools we can dynamically intercept calls to the *omp_parallel_do* function (or the equivalent function in the NthLib).

The second problem, how to know the iterative structure of the application, is solved by using a *Dynamic Periodicity Detector Tool*, the DPD Tool [39]. The DPD has been partially developed in this Thesis. This tool receives as input a sequence of values, a data stream (DPD does not interpret these values). The DPD processes the data stream and informs about its periodic behavior. Table 4.2 shows the DPD interface. The *DPD()* function receives a value of a sequence and returns true if this value starts an iterative pattern in the data stream. The *DPDWindowSize()* adjust the data size used by the DPD mechanism with the goal of reducing the overhead introduced in the application execution time.

Table 4.2: DPD interface.

Function	Description
int DPD (long sample, int *period)	Periodicity detection and segmentation
void DPDWindowSize (int size)	Adjust data window size

Using these two tools we can (1) dynamically intercept calls to encapsulated parallel loops (DITools), and (2) dynamically detect the iterative parallel structure of the application (DPD).

Figure 4.10 shows the iterative behavior of Hydro2d from the SpecFP95. Hydro2d is a parallel application with nested iterative regions. The graph shows in the *x* axis the dynamic sequence of parallel loops and the *y* axis shows the addresses of these loops. We identify parallel loops by the address of the function that encapsulates loop.

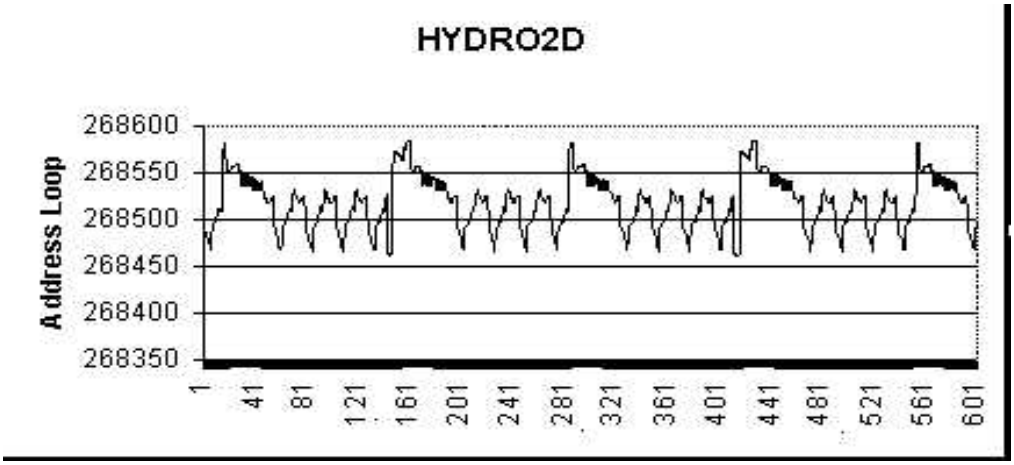


Figure 4.10: Iterative behavior of hydro2d (SpecFP 95)

Figure 4.11 shows how the SelfAnalyzer uses DITools and the DPD Tool to achieve the same behavior with a dynamic instrumentation than with a static instrumentation. In (1), the *omp_parallel_do* function call is intercepted by DITools. The *DI_event_pre* function receives as parameter the address of the intercepted function, in this case the *omp_parallel_do*, and its parameters. The first parameter of the *omp_parallel_do* function is the address of the encapsulated loop. This is the value that we pass to the DPD to detect the periodic patterns, (2) in Figure 4.11. We identify an IPR with the address of the first parallel loop and its period size.

Each time DPD detects the start of a IPR we check if we already are in an IPR or not. There are three possibilities:

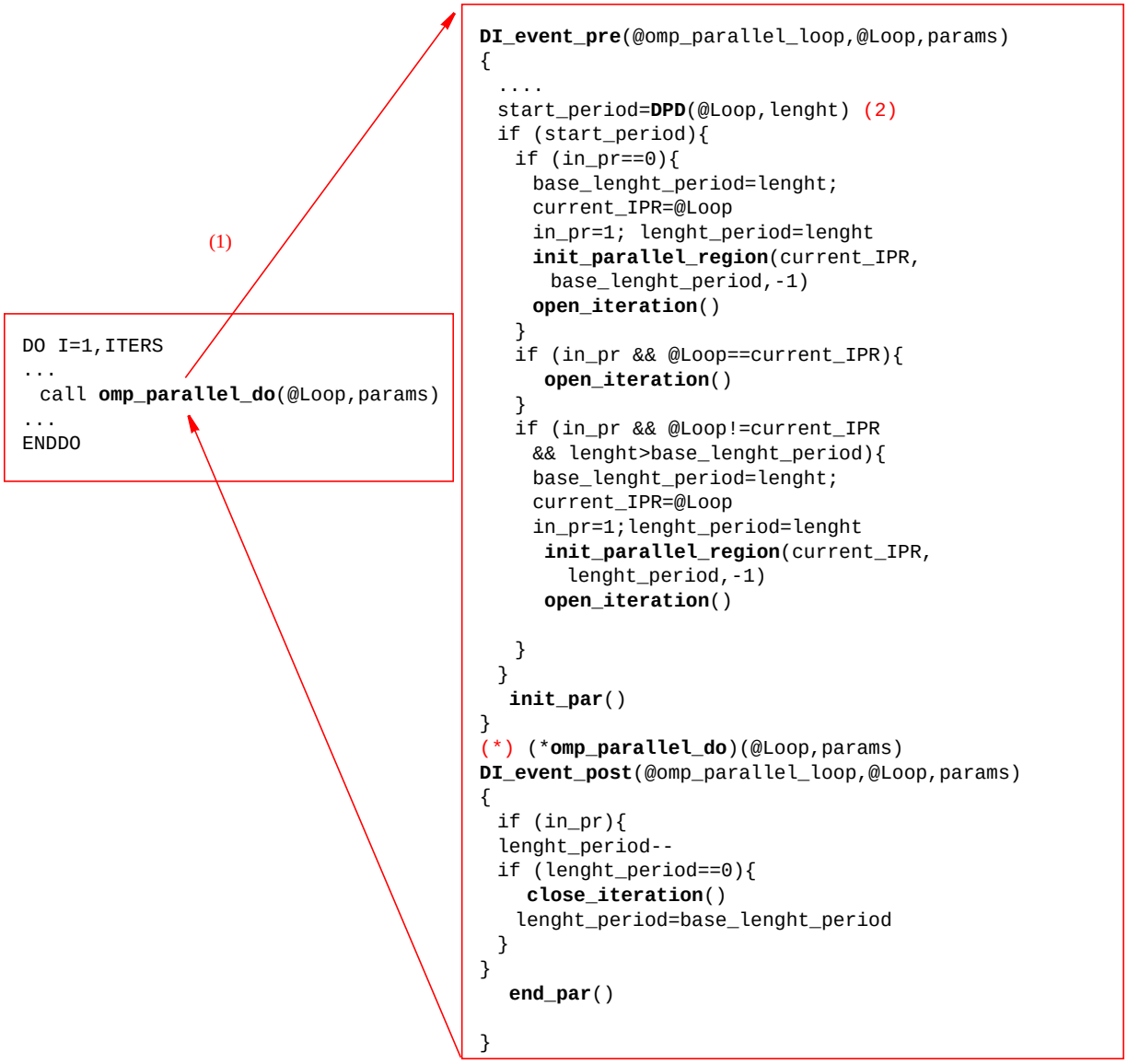
1.It is the first time we detect this IPR. We initialize the IPR by calling *init_parallel_region* and *open_iteration* functions.

2.We already were in the same IPR, that means that this is just a new iteration of the same IPR, then we simply call the *open_iteration* function.

3.It is a new IPR. If the length of the new IPR is greater than the old IPR we set the active IPR as the new IPR detected.

In any case, before calling the *omp_parallel_do* function we call the *init_par* function.

The *DI_event_post* is activated after executing the *omp_parallel_do* function. It receives the loop address function and its parameters. If the loop address corresponds with the end of an iteration of the main loop, we will call the *close_iteration* function. In any case, we call the *end_par* function.



(*) *omp_parallel_do* automatically called by the DITools mechanism

Figure 4.11: Dynamic Instrumentation (DITools + DPD)

4.5 Integration in the execution environment

The SelfAnalyzer must communicate the calculated speedup to the CPUManager. The SelfAnalyzer interacts with the CPUManager through the NthLib. We have modified the NthLib interface to include communication SelfAnalyzer/CPUManager. The CPUManager implements four new function calls, and the NthLib implements one new function (transparent to the CPUManager).

Table 4.3: New functionality

New function call	Description
nth_set_active_threads(P)	**Uses P threads out of the P' available**
appl_stopped()	Returns true if the application has been stopped since the last call
cpus_reference_time(t)	Sets the reference time
cpus_prediction_time(P,t)	Sets the estimated execution time with P processors
cpus_speedup(P,sp)	Sets the speedup with P processors

Table 4.3 shows the five functions implemented. Signalled row has been totally implemented in the NthLib. The *nth_set_active_threads* limits the number of processors that the application uses, rather than the number of processors allocated. P must be less or equal than the number of processors available. Using this function, the SelfAnalyzer sets the number of active threads to *baseline* processors to measure the reference execution time in a transparent way to the scheduler.

The *appl_stopped()* returns TRUE if the application has been stopeed since the last call to this function. This function is only used in systems where applications can be preempted, such as gang scheduling execution environments. It is called in the open_iteration and close_iteration. If the function returns true, measurements for this iteration will be discarded.

The *cpus_reference_time*, *cpus_prediction_time*, and *cpus_speedup* functions are only used to send the values calculated by the SelfAnalzyer to the scheduler. In that case, communication is implemented using shared-memory.

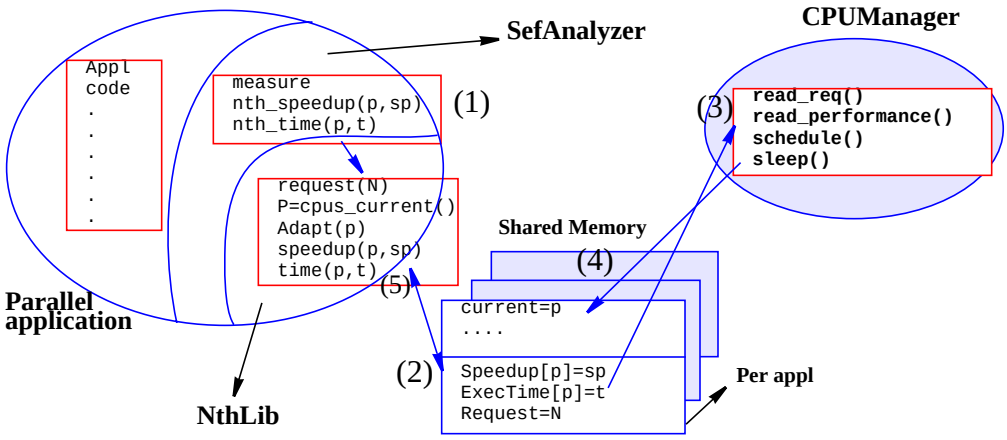


Figure 4.12: SelfAnalyzer integration in the execution environment

Figure 4.12 shows the behavior of the complete system. The SelfAnalyzer measures the speedup and the execution time of the application (1). Then, it informs the CPUManager using the NthLib interface (2). In (3), the CPUManager reads all this information and it schedules taking into account these data. In (4), it decides the processor allocation for the next quantum and the CPUManager sleeps until the next quantum. Finally, in (5) the application checks its current allocation and adapts its parallelism to its current allocation. As we described in Chapter 3, the CPUManager-NthLib interface is implemented through shared-memory.

Based on the speedup calculation, the CPUManager classifies each pair (application, P) in two different states:

- Performance Calculated (*PC*), the application has calculated its speedup with P processors.
- Performance Not Calculated (*PNC*), the application does not have calculated its speedup with P processors.

These two states are used by the scheduler to know if the speedup for a certain value of *P* is valid or not, and useful for scheduling.

4.6 Evaluation

The goal of the *SelfAnalyzer* is to dynamically calculate the speedup achieved by a parallel region. In order to calculate it, the *SelfAnalyzer* executes in sequential some iterations of the *iterative parallel region* to obtain the baseline measure. In this Section, we evaluate whether the dynamically computed speedup corresponds to the actual speedup of the application (i.e. that achieved by executing independently the parallel and the sequential version) and whether the *SelfAnalyzer* introduces overhead in the execution time of parallel applications.

The estimation of the execution time has not been explicitly evaluated because it is directly proportional to the speedup calculation. Moreover, this information can only be calculated if we have applied an static instrumentation.

It is delicate to evaluate whether the SelfAnalyzer works correctly because the execution time and the speedup of applications are very influenced by factors such as the execution in a multiprogrammed environment. We have fixed as much as possible those elements that can affect the speedup and execution time of applications to provide a measure about the SelfAnalyzer precision. To do that, we have executed several applications under controlled execution environment conditions. Each application has been executed in standalone mode with the machine dedicated, and using the CPUManager because it provides an execution environment more stable than the native execution environment.

We have used five applications: swim, tomcatv, hydro2d, and apsi from the SPECFp95, and bt from the NASPB. We have executed each application with different number of processors, from 4 to 48.

In this Section, we compare the speedup achieved by the applications executed just with the parallel library, with the speedup calculated by the SelfAnalyzer. Each point of the speedup curves has been calculated by averaging the speedup achieved by several executions. We also present the execution time of each application with and without the SelfAnalyzer in order to analyze the overhead introduced by it.

We have executed seven different configurations:

- *original*: the application is executed without the SelfAnalyzer. The speedup is calculated as the ratio between the execution time with 1 processor and the execution time with P processors.
- *static(1)*: the application is executed with the SelfAnalyzer. We have used a static instrumentation and a baseline of 1 processor.
- *dynamic(1)*: the application is executed with the SelfAnalyzer. We have used a dynamic instrumentation and a baseline of 1 processor.
- *static(4)*: the application is executed with the SelfAnalyzer. We have used a static instrumentation and a baseline of 4 processors.

- *dynamic(4)*: the application is executed with the SelfAnalyzer. We have used a dynamic instrumentation and a baseline of 4 processors.
- *static(half)*: the application is executed with the SelfAnalyzer. We have used a static instrumentation and a baseline of half of the number of available processors (4,8,16,32,48).
- *dynamic(half)*: the application is executed with the SelfAnalyzer. We have used a dynamic instrumentation and a baseline of half of the number of available processors (4,8,16,32,48).

With these configurations we want to evaluate the effect of using different baselines and instrumentation methods in the speedup calculation and in the execution time of applications. In the experiments executed in this Chapter, the number of requested processors is equal to the number of available processors because applications have been executed in alone. In all the execution the memory page migration mechanism was activated.

4.6.1 Tomcatv

Figure 4.13 shows, in the left side, the speedups calculated for the tomcatv application, and in the right side, the execution time of tomcatv with each different configuration. Tomcatv has a single parallel region. It is an application that reaches a super-linear speedup in its parallel region.

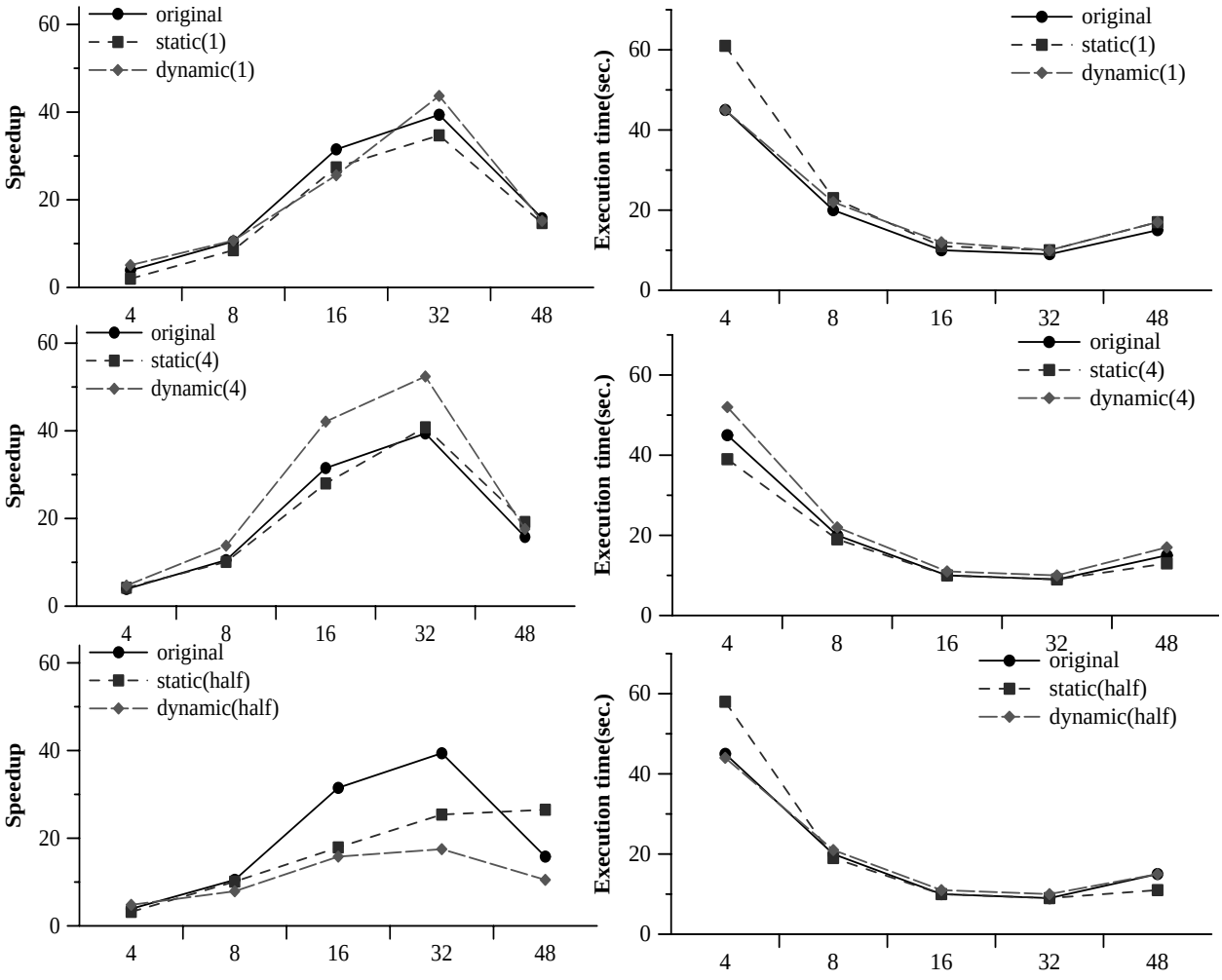


Figure 4.13: Tomcatv evaluation

In the case of the speedup curves, we can see that some of the curves calculated using the SelfAnalyzer are very similar to the curve calculated with the traditional formulation (more important that the specific values are the shape of the speedup curves).

Speedups achieved with static versions and baseline one or four processors are very precise. Dynamic versions are not so precise, they introduce a maximum difference of 30% in the case of baseline=4 and a 10% in the case of baseline=1 (both in the worst case with 32 processors). The differences introduced in the speedup calculations by dynamic versions are due to the effect in the data locality introduced by the SelfAnalyzer. For instance, in the case of the execution with 32 processors (and dynamic instrumentation), the application starts using 32 processors. This generates a data distribution among the different memory nodes. When the SelfAnalyzer detects that the application is iterative, it measures the reference value with the baseline number of processors. This measurement generates a data re-distribution, resulting in a worse execution time than in the static version (because in this case data has not even been distributed). The fact that

the reference measure is worse in the dynamic version than in the static version, generates that the speedup calculated is greater in the dynamic version than in the static version. However, we believe that differences are not so important because in multiprogrammed execution environments, there will be much more elements that will influence the execution time and speedup of the applications.

If we analyze measurements with a baseline of half of the number of available processors, the speedup curves do not have the same shape than the *original* curve. This difference is more important because the ratio between the different number of processors is not maintained. We will see that this conclusion will be the same with the rest of applications.

The problem of using a baseline set to half of the number of available processors appears when the application uses a high number of processors. In this case, the baseline is also very high. To use a high baseline introduces less overhead than using a small baseline. On the other hand, using a high baseline we loose the relationship with the traditional speedup equation that measures the relationship between the sequential and the parallel execution times. For this reason, if we use a high baseline, we have a hint about the application scalability, but not the speedup in its traditional meaning. In this case, even using the Amdahl's Factor, we are not able to detect the speedup.

As we can see, the best results are achieved with a baseline of four processors. In particular, it achieves the best results, both in the speedup calculation and in the overhead introduced. In addition, we can observe a curious effect: the execution time with SelfAnalyzer, baseline=4, and static instrumentation, has result in a better execution time than the version without SelfAnalyzer. This enforces our theory that the speedup of an application must be calculated at run-time. In this case, the difference is probably related with a positive influence of the SelfAnalyzer in the memory behavior.

4.6.2 Hydro2d

Figure 4.14 shows results for the Hydro2d application. Hydro2d is an application with a medium-low scalability. It also has the characteristic that it has several nested parallel regions. This is a bad case when using a dynamic instrumentation because measurements has to be restarted each time we detect a new parallel region.

If we observe the measured speedups, graphs in the left side, we can see that the achieved speedup precision is worse than the achieved precision in the case of tomcatv. The best results are achieved with a baseline of four processors and with a static instrumentation. One important thing is, as in the case of the tomcatv, measurements taken with baseline of one and four processors, correctly detect the shape of the speedup curve. In both cases, the speedup curves calculated have the same maximum and the same behavior. In the case of using a baseline of half, speedups calculated are very different to the traditional ones (the *original* curve).

Analyzing the overhead introduced by SelfAnalyzer, we can see that in this case it is much more significant than in the case of the tomcatv. However, the configuration with 4 processors and static instrumentation introduces an average overhead around the 10%. As in the previous case this is the best configuration. On the other hand, we can see that the execution time of static instrumented versions is better than the execution time of not instrumented version when executing with 48 processors. This enforces our theory that the execution time of a parallel application depends on a lot of factors such as the memory accesses that can be modified by just adding some data or some code lines.

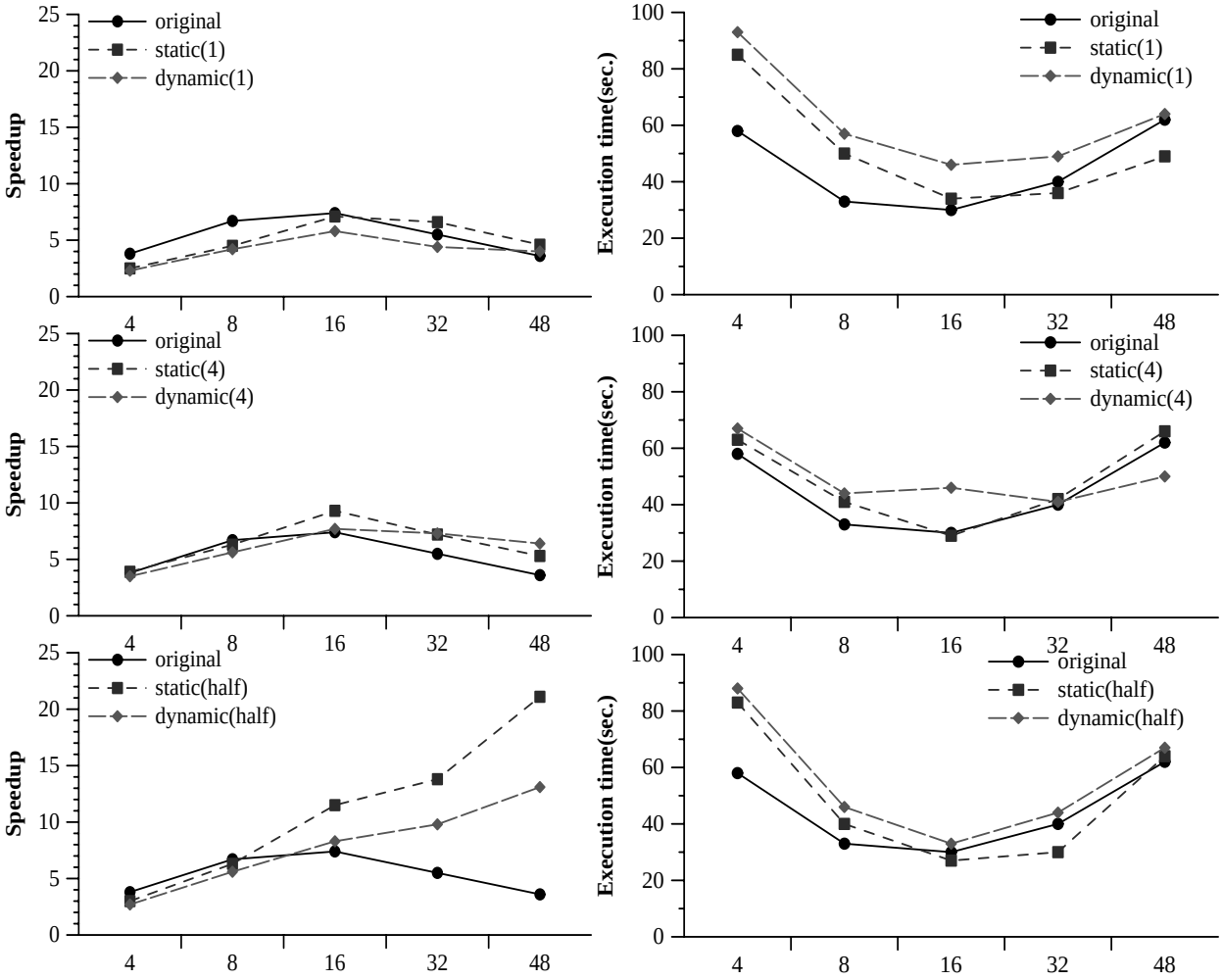


Figure 4.14: Hydro2d evaluation

4.6.3 Bt

Figure 4.15 shows results for the bt application. Bt is a parallel application with a high scalability and a single parallel region. The bt has the characteristic that its iterations consume more cpu time than the rest of evaluated applications. This fact could introduce some overhead when taking the reference measure.

In this case, we can appreciate that the overhead introduced is only significant in the case of using a baseline of one processor. In the rest of configurations the overhead is not significant.

In the case of the measured speedup, all the curves follow the speedup measured with the traditional method. The maximum difference, comparing absolute values, is around the 12% in the case of baseline=1, 16% if baseline=4, and 40% if baseline=half, all of them with 32 processors.

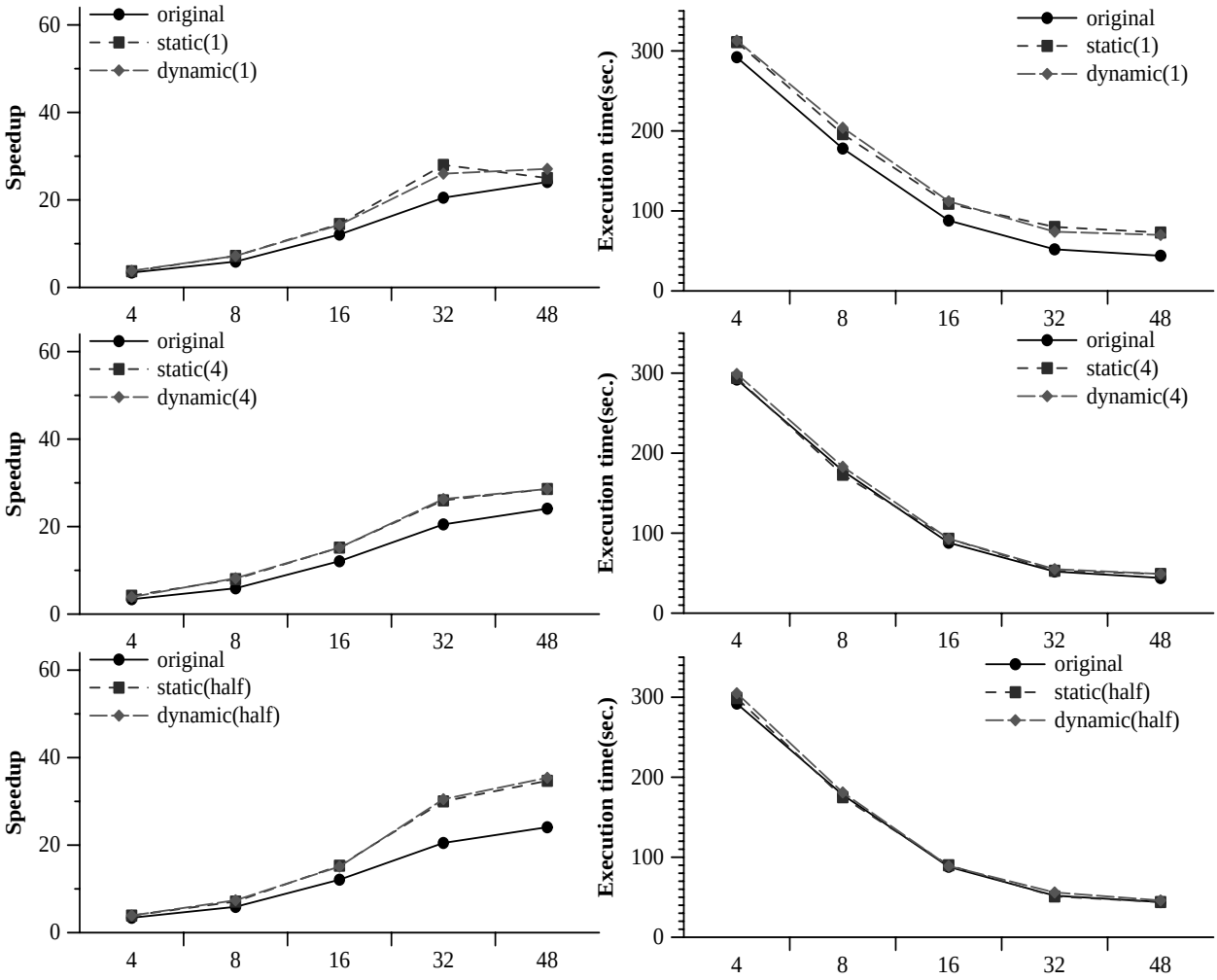


Figure 4.15: Bt evaluation

4.6.4 Swim

Figure 4.16 shows the results for the swim application. Swim is a super-linear application, with a single iterative parallel region. Swim has the characteristic that it has some parallel loops before the iterative parallel region. As a consequence, when the iterative region

starts, even with static instrumentation, data has been already distributed. This previous data distribution results in a different memory behavior when using SelfAnalyzer. It has the same problem that we commented with the tomcatv. In the case of baseline=1, the execution time of the baseline is very affected by this initial data distribution. For this reason, speedups calculated with respect to this baseline are greater than the calculated with the traditional method (*original*).

In the case of baseline 4, we can see that the speedup values are less than the calculated with the traditional method. This is because the swim reaches super-linear speedups in the first range of processors (4-8-16), but specially on four processors (6.5) because it is the number of processors where the data fit in the cache. If we take the reference measure with four processors, we are not able to detect this super-linear speedup, then the speedup curve will be displaced downwards to the graph (note that they have similar shapes but values are displaced).

In any case, we can see that the speedup curves calculated with baseline=1 and baseline=4 have the same shapes than curves calculated with the traditional method.

Observing the overhead introduced by SelfAnalyzer in this case we can see that using a baseline of four processors the overhead is not significant.

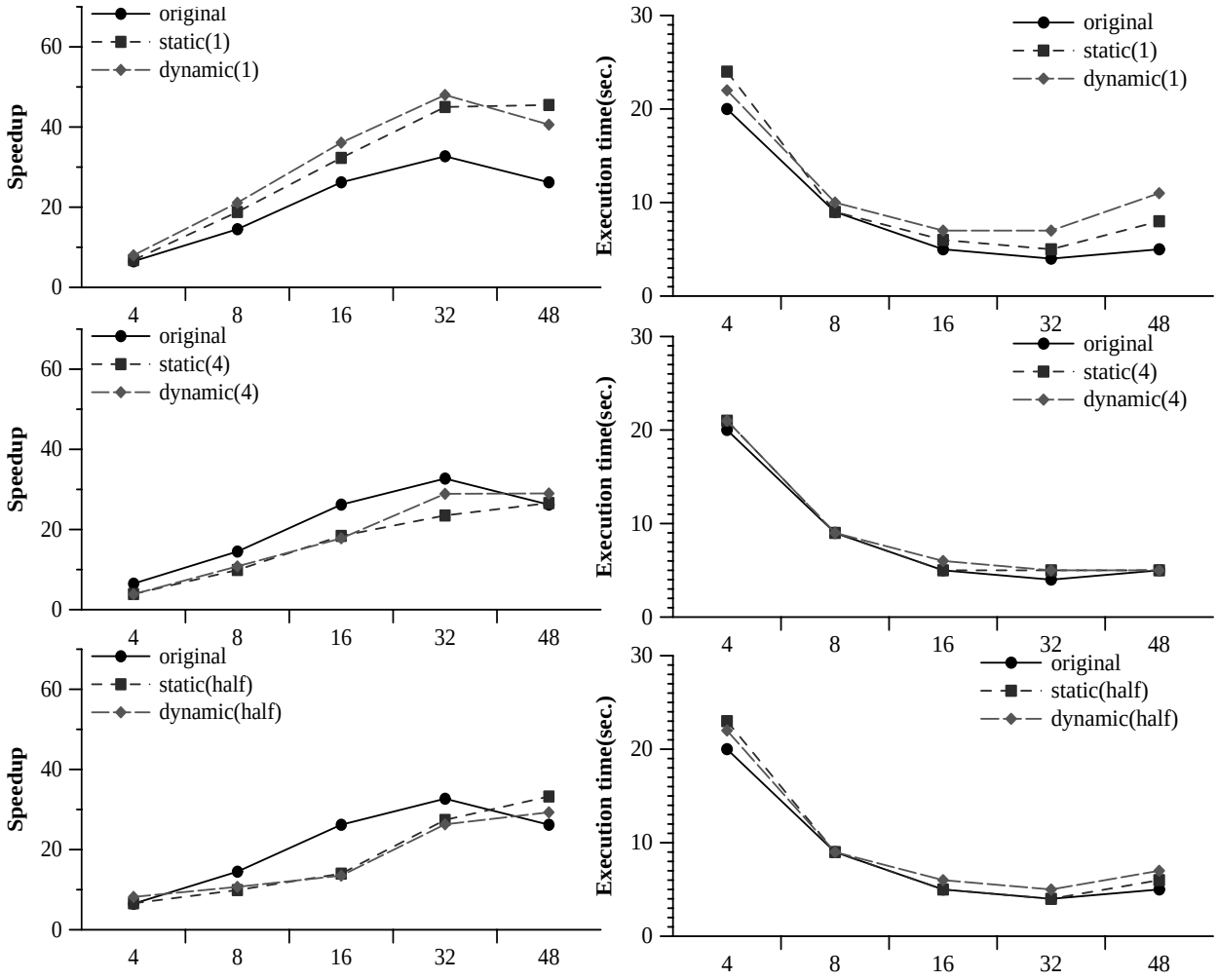


Figure 4.16: Swim evaluation

4.6.5 Apsi

Figure 4.17 shows the results for the apsi application. Apsi is a parallel application that does not scale at all. We have even observed that its execution time can be increased when executing in some multiprogrammed workloads (depending on the memory influence).

In this case, we can observe that SelfAnalyzer does not introduce overhead in the execution time and that the speedup values are equal to those calculated with the traditional method.

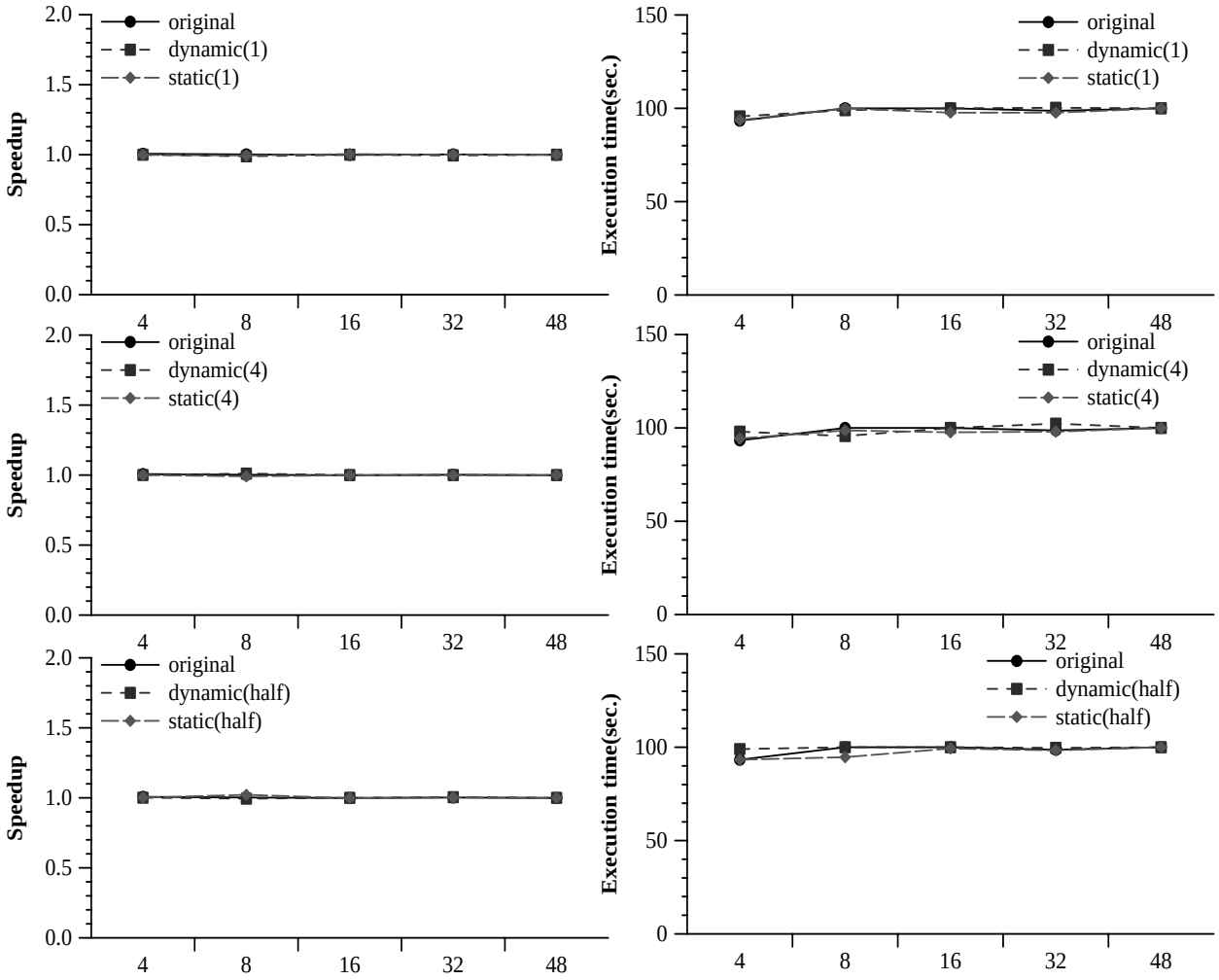


Figure 4.17: Apsi evaluation

4.7 Summary

In this work, we have presented the *SelfAnalyzer*, a new approach to dynamically calculate the speedup achieved by parallel applications. The *SelfAnalyzer* is a complete approach that works in executions environments with both static and dynamic scheduling policies.

Even though the SelfAnalyzer is designed to reach the maximum precision with the lowest overhead with the static instrumentation (when the source code of the application is available). SelfAnalyzer can also dynamically instrument parallel applications (when the source code is not available).

We have evaluated the SelfAnalyzer precision when dynamically measuring the speedup of parallel applications and the overhead introduced by it. We have evaluated several configurations with static and dynamic instrumentation, and with different baselines.

Results show that the best choice is to use a baseline of four processors, both with static and dynamic instrumentation. With this particular configuration, the speedup is well calculated in most of the evaluated applications. But the most important point is that with this configuration the shape of the speedup curves calculated by the SelfAnalyzer is the same than the calculated with the traditional method. Results show that the best results are achieved with a static instrumentation but that a dynamic instrumentation does not introduce a large overhead. Experiments have been done in a dedicated machine, executing only one application at each time to compare the traditional speedup with the measured by SelfAnalyzer.

Finally, we can also see that the overhead introduced by SelfAnalyzer is acceptable. However, we have seen that this overhead depends on the particular application characteristics.

CHAPTER 5

Performance-Driven Processor Allocation

Abstract

To consider the performance of parallel applications is critical to decide an efficient processor allocation. In this Chapter, we present the Performance-Driven Processor Allocation policy (PDPA). PDPA is a new coordinated scheduling policy. It implements a processor allocation policy and a multiprogramming level policy.

With respect to the processor allocation, PDPA is a dynamic policy that tries to allocate the maximum number of processors that reaches a target efficiency to running applications.

With respect to the multiprogramming level, PDPA allows the execution of a new application when there are free processors and the allocation of all the running applications is stable (PDPA has allocated the maximum number of processors that reach the target efficiency), or if they show a bad performance (they do not need more processors at all).

Results show that PDPA dynamically adjusts the processor allocation of parallel applications to reach the target efficiency, and that it adjust the multiprogramming level to the workload characteristics. PDPA improves the system utilization resulting in a better individual application response time.

5.1 Introduction

In this Chapter, we present our proposal for a coordinated scheduler. The processor scheduler will be coordinated with the run-time library, and with the queueing system.

Performance-Driven Processor Allocation (PDPA) is a processor scheduling policy that decides the processor allocation and the multiprogramming level in such a way that is coordinated with the loop scheduling level (run-time library), and with the job scheduling level (queueing system). Coordination means that PDPA informs about its decisions, and is informed about decisions related to it: On one hand, PDPA informs the run-time library about the number of processors available, preempted threads, etc, and, on the other hand, it informs the queueing system about when it is possible to start a new application. Coordination also means that PDPA takes into account the received information to take its decisions.

Moreover, in this Thesis we also propose that the processor allocation policy must consider the real performance of parallel applications and impose a target efficiency to avoid the inefficient use of processors. The performance of parallel applications must be calculated at run-time mainly because it depends on input data, influence of concurrently running applications, and because users are usually non-expert users and the system can not rely only on the information they provide.

With respect to the processor allocation policy, PDPA tries to find a processor allocation per application that achieves an acceptable efficiency. PDPA considers that an efficiency is acceptable if it is greater or equal than a given target efficiency.

With respect to the multiprogramming level, PDPA decides to start a new application when all the running applications have an acceptable efficiency or they have a "*bad*" efficiency. PDPA considers that the efficiency of an application is bad if it does not reach the target efficiency.

If we average results for the five workloads evaluated, we will find that PDPA outperforms the execution time of the evaluated workloads in a 245% compared with the native IRIX, a 75% compared with the Equipartition, and a 238% compared with the Equal_efficiency. If the workload reaches an efficient processor allocation with a simple Equipartition, results show that PDPA, in the worst cases, introduces an slowdown around the 10%. PDPA outperforms the evaluated policies because it dynamically adjust the processor allocation of running applications to reach a target efficiency, and the multiprogramming level to improve the system performance. Results also show that imposing a target efficiency to applications, the application execution time is sometimes increased, but the application response time is significantly improved.

The rest of this Chapter is organized as follows: Section 5.2 presents some related work. Section 5.3 describes the Performance-Driven Processor Allocation policy. Section 5.4 presents details about some implementation issues. Section 5.5 evaluates PDPA compared with some dynamic processor allocation policies. And finally, Section 5.6 summarizes this Chapter.

5.2 Related Work

Many researchers have studied the use of application characteristics to perform processor scheduling. Majumdar *et al.*[59], Parsons *et al.*[80], Sevcik [89][90], Chiang *et al.*[20] and Leutenegger *et al.*[55] have studied the usefulness of using application characteristics in processor scheduling. They have demonstrated that parallel applications have very different characteristics such as the speedup or the average of parallelism that must be taken into account by the scheduler. All these works have been carried out using simulations, not through the execution of real applications, and assuming *a priori* information.

Some researchers propose that applications should monitor themselves and tune their parallelism, based on their performance. Voss *et al.*[107] propose to dynamically detect parallel loops dominated by overheads and to serialize them. Nguyen *et al.*[75][76] propose *SelfTuning*, to dynamically measure the efficiency achieved in iterative parallel regions and select the best number of processors to execute them considering the efficiency. SelfTuning is applied at the run-time level.

Other authors propose to communicate these application characteristics to the a centralized scheduler and let it to perform the processor allocation using this information. Hamidzadeh [42] proposes to dynamically optimize the processor allocation by dedicating a processor to search the optimal allocation. This proposal does not consider application characteristics, only the system performance (throughput). Nguyen *et al.*[75][76] also use the efficiency of the applications, calculated at run-time, to achieve an *Equal_efficiency* in all the processors. The Equal_efficiency does not impose a target efficiency to running applications and does not coordinate the different scheduling levels. We will compare the Equal_efficiency with PDPA in the evaluation Section. Brecht *et al.*[13] use parallel program characteristics in dynamic processor allocation policies, (assuming *a priori* information). McCann *et al.*[65] propose *Dynamic*, a processor allocation policy that dynamically adjusts the number of processors allocated to parallel applications to improve the processor utilization. Their approach considers the application-provided idleness to allocate processors, resulting in a large number of re-allocations.

Our work has several characteristics that are different from the previously mentioned proposals:

1. With respect to the parameters used by the scheduling policy, our proposal considers two application characteristics: the speedup and the execution time, like Eager et al. in [30]. However, they propose to work with *a priori* calculated values.

2. We impose a target efficiency to running applications to maintain the processor allocation. This target efficiency has been shown very useful to ensure the efficient use of resources.

3. We propose to consider the speedup variation compared to the variation in the number of allocated processors, the *relative speedup* presented in Section 5.3.2.

4. We present a practical approach. We have implemented and evaluated our proposal using real applications and a real-commercial architecture, the SGI Origin2000. In this way, simulations do not consider important issues of the architecture such as the data locality. Most of the previous proposals are based on simulations and, in addition, they consider *a priori* information. We consider that this is not a desirable pre-condition because (1) we can not assume that users will provide this information, and (2) we can not assume that the information will be correct. The use of synthetic loads and an evaluation based on simulations a lot of times generates doubts about the validity of the results.

5. We propose a coordinated scheduler, where processor allocation decisions are coordinated with job scheduling decisions. This coordination has been shown as one of the most important sources of system improvement.

5.3 Performance-Driven Processor Allocation (PDPA)

PDPA is a scheduling policy that coordinates decisions related to the processor allocation with decisions related to the multiprogramming level. PDPA is executed periodically, (at each *quantum*[1] expiration). In this Section, we describe the PDPA processor allocation policy and the PDPA multiprogramming level policy.

5.3.1 Processor allocation policy

PDPA is a dynamic space-sharing policy. This kind of policies partition the machine and applications run in these partitions as in a dedicated machine.

PDPA allocates a minimum of one processor to each running application (run-to-completion). It mainly applies a search algorithm to each parallel application looking for the maximum processor allocation that achieves an *acceptable* efficiency. PDPA considers that the efficiency of a parallel application is acceptable, if it is greater than a given *target efficiency*. The goal of PDPA is to minimize the response time, while guaranteeing that the allocated processors are achieving a good efficiency.

PDPA is activated each time a new application arrives to the system, an application finishes, or a running application informs about its performance.

To apply the search algorithm, PDPA manages information related to the recent past of the application. It remembers the last processor allocation different from the current allocation and the efficiency achieved with it. PDPA uses this information to compare with the actual allocation and performance.

5.3.2 Application state diagram

PDPA considers each application to be in one of the states shown in Figure 5.1. These states correspond with the behavior of the application performance. These states and the transitions among them are determined by the performance achieved by the application and by some policy parameters.

The *PDPA* parameters are: (1) efficiency considered very good (*high_eff*), (2) target efficiency (*low_eff*), and (3) number of processors that will be used to increment/decrement the application processor allocation (*step*). In Section 5.3.3, we will present the solution adopted in the current implementation to define these parameters.

PDPA can assign applications to four different states: *NO_REF(*initial state*)*, *DEC*, *INC*, and *STABLE* (Figure 5.1). Each different state means the knowledge that PDPA has about the performance that each application had the last time PDPA evaluated it. Each time PDPA is activated, PDPA evaluates the performance of each application and decides the next state and the next allocation. Modifications in the processor allocation are associated to state transitions (even if the next state is the same).

1. A typical quantum value is 100 ms.

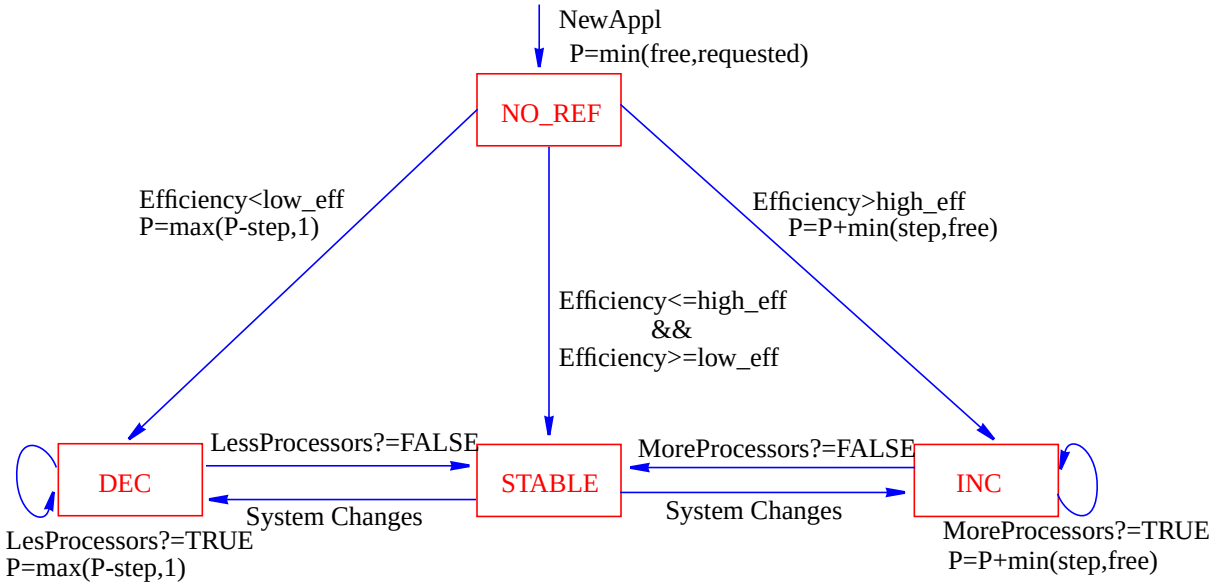


Figure 5.1: *PDPA*: Application state diagram

NO_REF state

Applications start in the *NO_REF* state. This state means that *PDPA* has no performance knowledge about this application (it is in its starting point). PDPA initially allocates the minimum between the number of processors requested and the number of free processors.

Once the application informs about its speedup, *PDPA* compares the achieved efficiency[2] with *high_eff* and *low_eff*. If the efficiency is greater than *high_eff*, the next state will be *INC, (*PDPA considers that the application performs well). If the efficiency is lower than *low_eff*, the next state will be *DEC (*PDPA considers that the application performs poorly). If the efficiency is between *high_eff* and *low_eff*, the next state will be *STABLE* (PDPA considers that the application has an acceptable performance).

If the next state is *INC*, the application will receive more processors in the next *quantum*. The number of additional processors will be the minimum between *step* and the number of free processors. If the next state is *DEC*, the application will receive step less processors in the next quantum. The application will receive a minimum of one processor. If the next state is *STABLE*, the processor allocation will be maintained.

2. Calculated as the ratio between the speedup with *P* processors and *P*.

INC state

Being in the *INC* state means that the application performed well the last time PDPA evaluated it. In this state, PDPA has to evaluate the performance achieved with the decision taken in the last quantum.

The *MoreProcessors*() function Figure 5.3, evaluates if the application performs better with the actual allocation than with the last allocation (the last processor allocation less than the actual). To decide that, PDPA evaluates (1) if the achieved efficiency is greater than high_eff, (2) that the achieved speedup is greater than the last speedup, and (3) if the *RelativeSpeedup* is greater than the percentage of additional processor multiplied by *high_eff*.

The RelativeSpeedup measures if the scalability of the application has been maintained with the last additional processors received. It is measured as the relationship between the execution time with the last allocation and the actual processor allocation. If the execution time is not available, the RelativeSpeedup is calculated as the relationship between the speedup with the current allocation and the speedup with the last allocation. With this formulation we detect and avoid situations where the speedup is super-linear within a range of processors (that means a very high efficiency) but later the speedup progression is not maintained.

```
Application 1 characteristics

ExecTime(1)=100 sec.
Speedup(16)=28 -> Efficiency(16)=1.75 -> ExecTime(16)=1.5
Speedup(32)=30 -> Efficiency(32)=0.937 -> ExecTime(32)=1.125
RelativeSpeedup=1.5/1.12=1.071
IncrementProcessors=32/16=2
```

Figure 5.2: Relative speedup example

For instance, consider the case of application 1 presented in Figure 5.2, which is just an extreme case to illustrate how the *RelativeSpeedup* filter works. With a *high_eff*=0.9, and without considering the *RelativeSpeedup*, PDPA will decide to allocate 32 processors to application 1 because speedup(32) is greater than speedup(16), and efficiency(32) is greater than *high_eff*. However, the *RelativeSpeedup* of Application 1 is only 1.02, even receiving 2 times more processors. For this reason, PDPA decides that it is more efficient for the system to limit the allocation of application 1 to 16 processors knowing that the multiprogramming level policy will decide to increase the multiprogramming level. In that case, assuming that there are queued applications, the queueing system will start a new application. This decision does not significantly negatively affects the execution time to application 1, and can significantly improve the performance of the system and the response time of applications.

If *MoreProcessors*() returns TRUE, the next state will be *INC*. Otherwise, the next state will be *STABLE*.

If the next state is *INC*, the application will receive *step* additional processors in the next quantum. If the next state is *STABLE*, the application will loose the *step* additional processors (received in the last transition) and it will continue its execution.

```
Uses last allocation per job (Last)
MoreProcessors(job)
{
  current=jobs[job].current;
  current_eff=jobs[job].Speedup[current]/current;
  current_speedup=jobs[job].Speedup[current];
  if (current>Last[job]){
     RelativeSpeedup=jobs[job].ExcTime[Last[job]]/jobs[job].ExcTime[current];
     IncrementProcessors=current/Last[job];
    if (eff>=high_eff) && current_speedup>jobs[job].Speedup[Last[job]] &&
      RelativeSpeedup>=(IncrementProcessors*high_eff)) return TRUE
    else                                          return FALSE
  }else{
    if (current_eff>=high_eff)                    return TRUE
    else                                          return FALSE
}
```

Figure 5.3: MoreProcessors() function

DEC state

The *DEC* state means that the application has not reached the target efficiency the last time PDPA evaluated it. The *LessProcessors*() function, presented in Figure 5.4, evaluates whether the performance of the application is acceptable with the current allocation.

If *LessProcessors*() returns TRUE, the application has not still reached an acceptable performance. In that case, the next state will be *DEC*. If *LessProcessors*() returns FALSE, the next state will be *STABLE*. In this case, the only condition is that the application reaches the target efficiency (*low_eff*).

```
LessProcessors(job)
{
  eff=jobs[job].Speedup[jobs[job].current]/jobs[job].current
  if (eff<low_eff)     return TRUE
  else                 return FALSE
}
```

Figure 5.4: LessProcessors() function

If the next state is *DEC*, the application will receive the maximum between (P-step) and 1 processor. If the next state is *STABLE*, the application will keep the current allocation.

STABLE state

The *STABLE* state means that the application has the maximum number of processors that PDPA considers acceptable. The processor allocation in this state is maintained.

If the policy parameters are dynamically defined, PDPA could change the state of an application from *STABLE* to either *INC* or *DEC*. If *low_eff* is increased, the efficiency achieved with the current allocation could be not acceptable. In that case, the next state will be *DEC* and application will loose *step* processors. In a symmetric way, if *high_eff* is decreased, next state will be *INC* and the application will receive *step* additional processors. In the same way, if the application performance changes, the next state and processor allocation are modified.

5.3.3 *PDPA* parameters

As we have commented in the introduction of this Section, there are three parameters that determine the PDPA aggressiveness. These parameters can be either statically or dynamically defined. Statically defined, for instance by the system administrator, or dynamically defined, for instance as a function of the number of running or queued applications. In the current *PDPA* implementation, the three parameters are dynamically defined as a function of the running applications. PDPA calculates values of *high_eff* and *low_eff* at the start of each quantum, before processing applications. If the machine is heavy loaded, *high_eff* is set to 0.9 and *low_eff* to 0.7. If the machine is low loaded, we will set *high_eff* to 0.7 and *low_eff* to 0.5.

Step is a parameter that defines variations in the processor allocation. This parameter is used to limit the number of re-allocations that are suffered by applications. Setting *step* to a small value, we achieve more accuracy in the number of allocated processors, but the overhead introduced by re-allocations could be significant. In the current implementation, applications in the INC state defines step to four processors.

Applications in the DEC state use also a step of four processors except in some cases. PDPA uses a different value of step in those cases where it detects that the achieved speedup is significantly bad. For instance, if an application reaches an speedup of 2 with 32 processors, PDPA assumes that the speedup with 28 processors wont achieve an acceptable efficiency. In that cases, we calculate the next allocation applying the equation presented in Figure 5.5. With this formula, we allocate the maximum number of processors that PDPA will consider acceptable to achieve this speedup.

$$\text{NewAlloc} = \frac{\text{jobs[job]speedup[jobs[job]current]}}{\text{higheff}}, \text{DynStep} = \text{jobs[job]current} - \text{NewAlloc}$$

Figure 5.5: Allocation decided in the case of a very bad speedup

5.3.4 Multiprogramming level policy

The multiprogramming level is the number of applications concurrently running in the system. As we commented in Chapter 2, traditional approaches execute parallel workloads (1) limiting the multiprogramming level, having the problem of the fragmentation, or (2) without controlling it, having the problem of the overloading.

We want to control at any moment the load of the system. For this reason, in this Thesis we discard the option of executing parallel workloads without controlling the multiprogramming level. The alternative, suffers from fragmentation in (1) systems where applications are rigid and can only be executed with the number of processors requested, and (2) when the total number of processors requested does not fit the complete machine.

However, in dynamic space-sharing policies, we have the advantage that we can execute an application without having to wait until as many processors be free as the application request.

Based on this consideration, we propose to coordinate the two scheduling levels and leave the decision about when to start a new application to the processor scheduling policy, and the decision about which application to the queueing system. This decision could also be taken by the queueing system by observing the number of processors idle. However, the queueing system does not know the application status. It only could base its decisions based on a limited information. In an execution environment such as our case, where applications start requesting for one processor, and request for P processors when they enter in the first parallel region, the queuing system will probably take incorrect decisions.

Figure 5.6 shows the PDPA_New_appl() function. The conditions that must be accomplished to allow to start a new application are: (1) it must be not allocated processors, and (2) the phase of all the applications must be STABLE or DEC. This is because in that cases no application will need more processors (assuming that application performance will not change).

```
int PDPA_New_appl()
{
  if (free_processors() && || AllApplication_STABLE_OR_DEC())) return TRUE;
  else                                                        return FALSE;
}
```

Figure 5.6: PDPA_New_appl() policy

5.4 Implementation issues

Figure 5.7 shows the main functions of the processor scheduler process. It is mainly composed by an infinite loop that activates the processor allocation policy, evaluates whether the multiprogramming level must be modified, and finally enforces the processor allocation policy decisions. This function implements the different phases described in Chapter 3. Once enforced, the CPUManager sleeps for one quantum.

Even PDPA only needs be activated each time a new application arrive, finish, or informs about its performance, we have implemented the PDPA activation by sampling. At each quantum, PDPA checks if any of these conditions are true, and in that case it will actuate. Since the quantum is quite fine, this sampling is quite enough to work in the same way that an event-driven implementation.

```
Processor_scheduler()
{
  ....
  Init_multiprogramming_level()
  while(1){
    PDPA_processor_allocation()
    Dynamic_multiprogramming_level()
    Enforce_processor_allocation()
    sleep(quantum)
  }
}
```

Figure 5.7: Processor scheduler main loop

Figure 5.8 shows the pseudocode that implements the PDPA policy. PDPA initially checks the internal status of applications and maintains the processor allocation to those applications that are in the *PNC*[3] (*Performance Not Calculated*) state. Transitions in the state diagram are only allowed either when applications are in the *PC (Performance Calculated)* state. The aim of this decision is to maintain the allocation of those applications that are calculating their speedup. If we modify the speedup of an application in *PNC* state as a consequence of the processing of another application, it could result in inaccurate allocations.

3. The application can be in Performance Calculated or Performance Not Calculated state, see Section 4.5.

```
input: jobs_table(jobs),Last allocation per job (Last), Phase per job (Phase)
output: table with number of processors per job (alloc), Last allocation per job (Last)
void PDPA_Processor_allocation()
{
  if (All_appl_in_PC_state()){
    Calculate_target_efficiency();
    Allocate_one_processor_per_application();
    for(current_job=0;current_job=active_jobs;current_job++){
      current=jobs[current_job].current; current_speedup=jobs[current_job].Speedup[current];
      last_speedup=jobs[current_job].Speedup[Last[current_job]];
      current_eff=current_speedup/current;
      switch(Phase[current_job]){
      case NO_REF:
        switch(Next_phase(current_job)){
          case INC:alloc[current_job]=current;
             Phase[current_job]=INC;
          case DEC:
             step=current-DynSTEP(current_job); alloc[current_job]=max(1,step);
             Phase[current_job]=DEC;Last[current_job]=current;
          case STABLE:alloc[current_job]=current;
             Phase[current_job]=STABLE;
        }
        break;
      case INC:
        if (MoreProcessors(current_job)){
          alloc[current_job]=current;
        }else{
          if ((current_speedup>last_speedup) && (current_eff>high_eff)){
                   alloc[current_job]=current;
          }else    alloc[current_job]=Last[current_job];
          Phase[current_job]=STABLE;
        }
        break;
      case DEC:
        if (LessProcessors(current_job)){
          alloc[current_job]=max(1,current-DynSTEP(current_job)); Last[current_job]=current;
        }else{ alloc[current_job]=current; Phase[current_job]=STABLE;}
        break;
      case STABLE:
        if (SystemChanges()){
          if (MoreProcessors(current_job){
            alloc[current_job]=current;Phase[current_job]=INC
          }else if (LessProcessors(current_job)){
            alloc[current_job]=max(1,current-DynSTEP(current_job));
            Phase[current_job]=DEC;Last[current_job]=current;
          }
        }else alloc[current_job]=current;
      }
    }
  }else{
     Maintain_allocation_of_PC_applications();
  }
  if (free_processors()) {
    Allocate_processors_to_new_appls(); // Equipartitioned
    Allocate_more_processors_to_INC_appl(); // Equipartitioned
  }
}
```

Figure 5.8: PDPA processor allocation code

Applications in *PC* state are processed and their next phase and allocation are calculated. PDPA maintains as much as possible stable allocations. For this reason, it initially fixes the allocation to those applications that will be in NO_REF, INC, and STABLE states. Processor allocation of applications that are, or will be, in DEC state is also calculated at this point, because they do not need more processors.

Applications in INC state are post-processed after processing all the applications. If there are free processors, they will receive more processors. In this post-process, applications are sorted by speedup to give a certain priority to those applications that perform better. PDPA distributes free processors equally among jobs in the INC state, sorted in that way.

The *SystemChanges*() function checks if application performance or PDPA parameters have changed. In that case, we check if the application must change its state and allocation. To avoid possible ping-pong effects, we have limited the number of changes per parallel region to three times. In fact, we have checked experimentally that the behavior of a parallel region is usually stable. Note that small changes in application performance are directly filtered by the SelfAnalyzer and they are not detected by the scheduler.

5.5 Evaluation

To evaluate the proposals of this Thesis, we have used the five workloads presented in Chapter 3. These workloads differ in the percentage of each type of applications: super-linear, highly scalable, medium scalable, and not scalable.

The scheduling policies that we have evaluated in this Chapter are: the native IRIX scheduling (IRIX), the Equipartition (Equip), which is a good approach if no performance information is used, the Equal_efficiency (Equal_eff), which is the only scheduling policy found in the literature that uses performance information calculated at run-time, and PDPA. The queuing system used has been the Launcher, and Equipartition, Equal_efficiency, and PDPA are implemented in the CPUManager.

In the case of IRIX, the CPUManager has not been executed and we have used the native SGI-MP library. This run-time library uses some environment variables that define its behavior, such as the MP_SET_NUMTHREADS, OMP_DYNAMIC, the MP_BLOCKTIME, or the _DSM_MIGRATION. The MP_SET_NUMTHREADS defines the number of kernel threads to be created by the application. The OMP_DYNAMIC enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. We have set OMP_DYNAMIC to TRUE. We have also set the MP_BLOCKTIME environment variable. MP_BLOCKTIME controls the amount of time a slave thread waits for work before giving up. The value of MP_BLOCKTIME specifies the number of times to spin in the wait loop. This value was tuned experimentally and set to 200.000 in [62]. We have used the same value for this variable. The _DSM_MIGRATION environment variable which specifies aspects of automatic page migration, set to ALL_ON enables migration for all data.

IRIX, Equipartition, and Equal_efficiency have been executed with a multiprogramming level fixed set to four applications. PDPA uses a default multiprogramming level of four applications. Table 5.1 shows the characteristics of the four different execution environments evaluated in this Chapter.

Table 5.1: Execution environments evaluated

Policy	Queueing system	Processor scheduler	Run-time library	Multiprog. Level
IRIX	Launcher	IRIX	SGI-MP	Fixed = 4
Equip	Launcher	CPUManager	NthLib	Fixed = 4
Equal_eff	Launcher	CPUManager	NthLib	Fixed = 4
PDPA	Launcher	CPUManager	NthLib	Dynamic, Def.=4

All the workloads are executed as in a open system, that is, a system where applications are submitted to the system following a Poison inter-arrival function. We have generated workloads to simulate systems with an estimated demand of 60%, 80%, and 100% of the total capacity of the system.

As we commented in Chapter 3, we have used workload trace files that specify the arrival sequence of applications to the system, then all the scheduling policies evaluated have executed the same set of applications and with the same submission times.

We have calculated the average response time and the average execution time per scheduling policy, workload, and application class. The response time of each application is calculated as the difference between the finalization time and the submission time, the number of seconds the application is in the system, time queued + time executing. The execution time is calculated as the difference between the finalization time and the starting time, the number of seconds the application is executing.

One important thing is that we have submitted applications only during 300 seconds, but we have considered all the applications submitted to calculate the response time and the execution time, not only those that have finished during this initial period. This implies that, for instance, in workloads with a 100% of load, in fact the 100% of load is only real during the first 300 seconds. After this time there is a queue of remaining applications that are also considered but that are not executed under a 100% of load.

We have also measured the total execution time per workload. These results are shown at the end of the evaluation, in Section 5.5.6.

Table 5.2 summarizes the main characteristics of the five workloads used in this Thesis. In the next sections, we detail these characteristics and present the results for each workload. Three of the workloads are composed by two types of applications, one is composed by the four types of applications, and one by only one type of applications. The *% of cpu* column is relative to the system load that generates the workload.

Table 5.2: Workload characteristics

	Application 1			Application 2			Application 3			Application 4		
	appl.	req.	% of cpu	type	req.	% of cpu	type	req.	% of cpu	type	req.	% of cpu
w1	swim	30	50%	bt	30	50%	-	-	-	-	-	-
w2	bt	30	50%	hydro	30	50%	-	-	-	-	-	-
w3	bt	30	50%	apsi	2	50%	-	-	-	-	-	-
w4	swim	30	25%	bt	30	25%	hydro	30	25%	apsi	2	25%
w5	bt	30	100%	-	-	-	-	-	-	-	-	-

We have evaluated all these workloads in a SGI Origin2000 like the one described in Chapter 2. It is a CC-NUMA machine with 64 processors. However, we have only used 60 processors to evaluate our workloads. We have used one of the idle processors to execute a tracing tool, *scpus*, to monitorize the execution of the workloads. This tool generates a trace file that can be visualized with the Paraver Tool [52].

5.5.1 Workload 1

Figure 5.9 shows results from workload 1. Graphs in the top of the figure show the average response time (in seconds) of swim's (super-linear), and bt's (highly scalable). In the x axis we represent the different loads of the system generated. Graphs in the bottom of the figure show the average execution time (in seconds) of swim's and bt's. In this workload, the request of all the applications has been set to 30 processors.

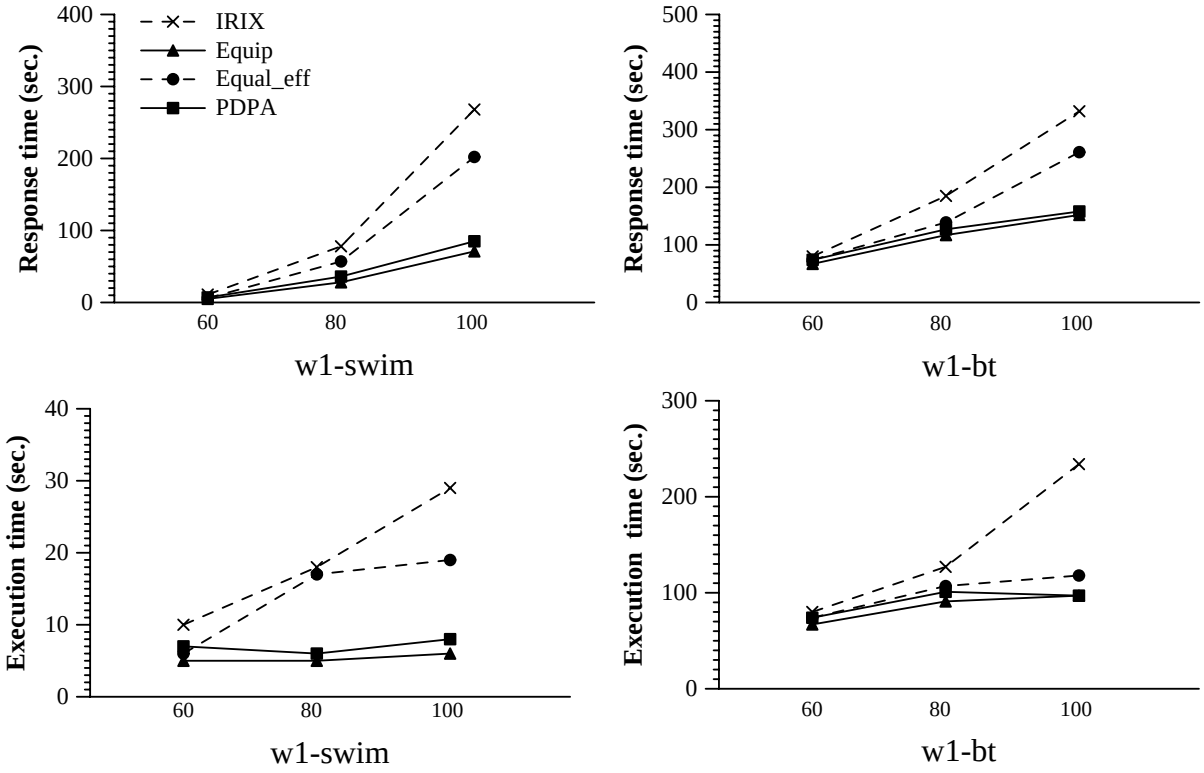


Figure 5.9: Results of workload 1, M.L.=4

This workload has the characteristic that (1) applications are scalable, (2) they have been previously tuned to select the number of processors that reaches the maximum speedup, and (3) the multiprogramming level set to four is a good value for this workload. The multiprogramming level set to four applications generates that applications under the Equipartition execute with 15 processors (we have a total of 60 processors) when the machine is high loaded and with 30 processors if the machine is low loaded. In the first case, 15 processors is the number of processors that achieves the best ratio speedup / efficiency, and the second case, 30 processors, is the number of processors that achieves the best speedup for the two applications. For all these reasons, this workload could be a bad case for PDPA because there is "*nothing*" to improve.

Results show that both the response time and the execution time of PDPA (line with boxes) outperform the ones achieved by IRIX and Equal_efficiency, and it is slightly worse than the performance achieved by Equipartition. Comparing the response time achieved by PDPA and Equipartition, PDPA is a 10% worse than Equipartition in the case of bt´s, and around a 30% worse than Equipartition in the case of swim´s.

The Equal_efficiency has the problem that it has a high sensitivity to small changes in the efficiency measurements. Small variations in the efficiency generates high variances in the processor allocation, resulting in a high number of processor reallocations. As we commented in the introduction of this Thesis, the system has to be conscious that applications are malleable but that reallocations are not free, and it is something that must be done "*with care*". Another problem related with the Equal_efficiency is that the formulation used to extrapolate the values sometimes generates a lot of differences between applications that have the same performance. For instance, in the case of the load=100%, we have measured the processor allocation received by swim's, and we have found that the Equal_efficiency has allocated from 2 until 28 processors. This is an unfair allocation because two applications with the same performance and requesting for the same number of processors should received the same amount of processors.

Observe that, in this type of workloads, it could be beneficial to reduce the multiprogramming level to improve the execution time of running applications. However, in this Thesis we give priority to the overall system performance rather than to the individual application performance. To reduce the multiprogramming level to improve the individual application speedup is something that it remains as a future work, but we believe that this kind on modification could be easily introduced in PDPA.

We have executed this workload by varying the multiprogramming level and we have found that PDPA always set it to four applications. This is the reason because PDPA does not improves the execution of this workload, because the static configuration is the same than PDPA dynamically decides.

Multiprogramming level set to three applications

We have executed the same workload varying the baseline multiprogramming level to compare the behavior of PDPA with the Equipartition behavior, which is the policy that reaches the best results.

Figure 5.10 shows the response time and execution time for swim's and bt's under Equipartition and PDPA when using a default multiprogramming level of three. Graphs on the top of the figure show the average response time of swim's and bt's, and graphs on the bottom of the figure show the average execution time of swim's and bt's.

PDPA reaches the same performance than Equipartition comparing the response time. If we compare the execution time, the Equipartition outperforms PDPA in the case of swim's and in the case of bt's it seems that PDPA slightly outperforms the Equipartition.

We have calculated the multiprogramming decided by PDPA. It has been set to four applications, the same that the one defined statically in the previous experiment.

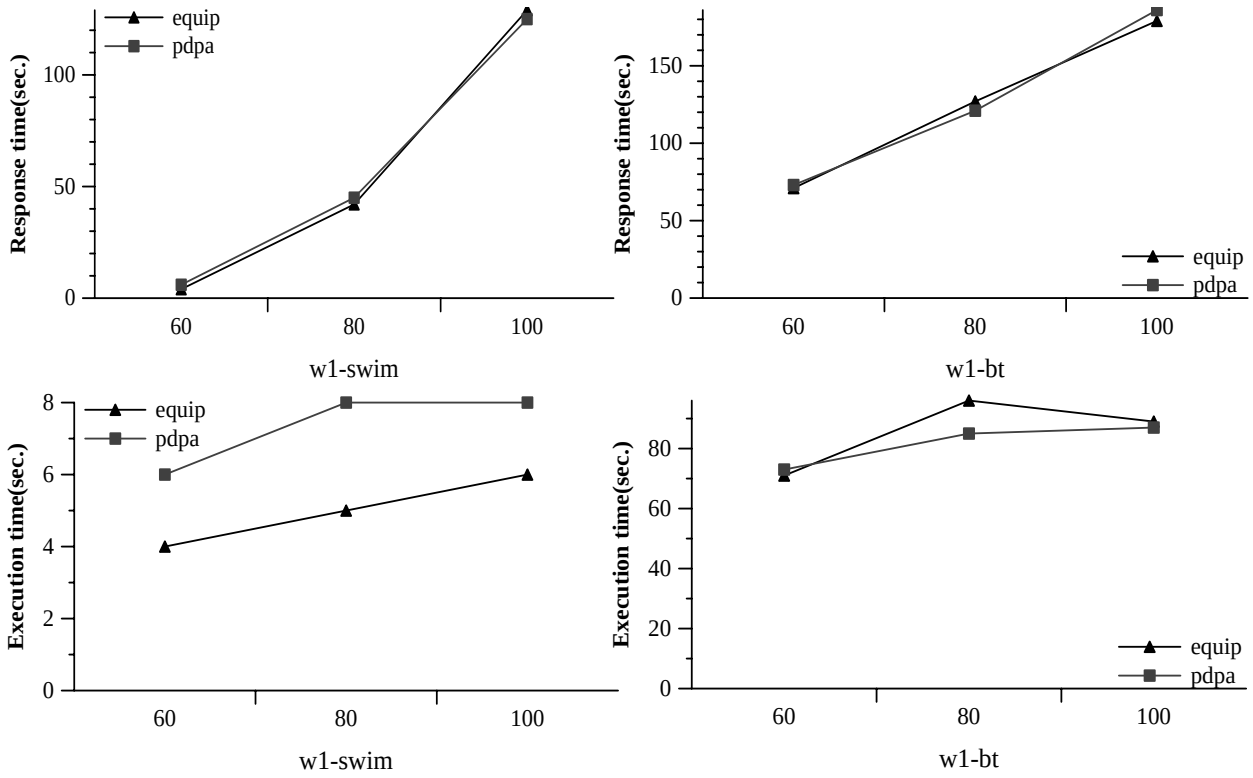


Figure 5.10: Results of workload 1, M.L.=3

Multiprogramming level set to two applications

Figure 5.11 shows the same comparison when setting the multiprogramming level to two applications. In this case results are very similar to the previously presented, but in this case it seems that the Equipartition slightly outperforms PDPA in the average execution time (not in the response time).

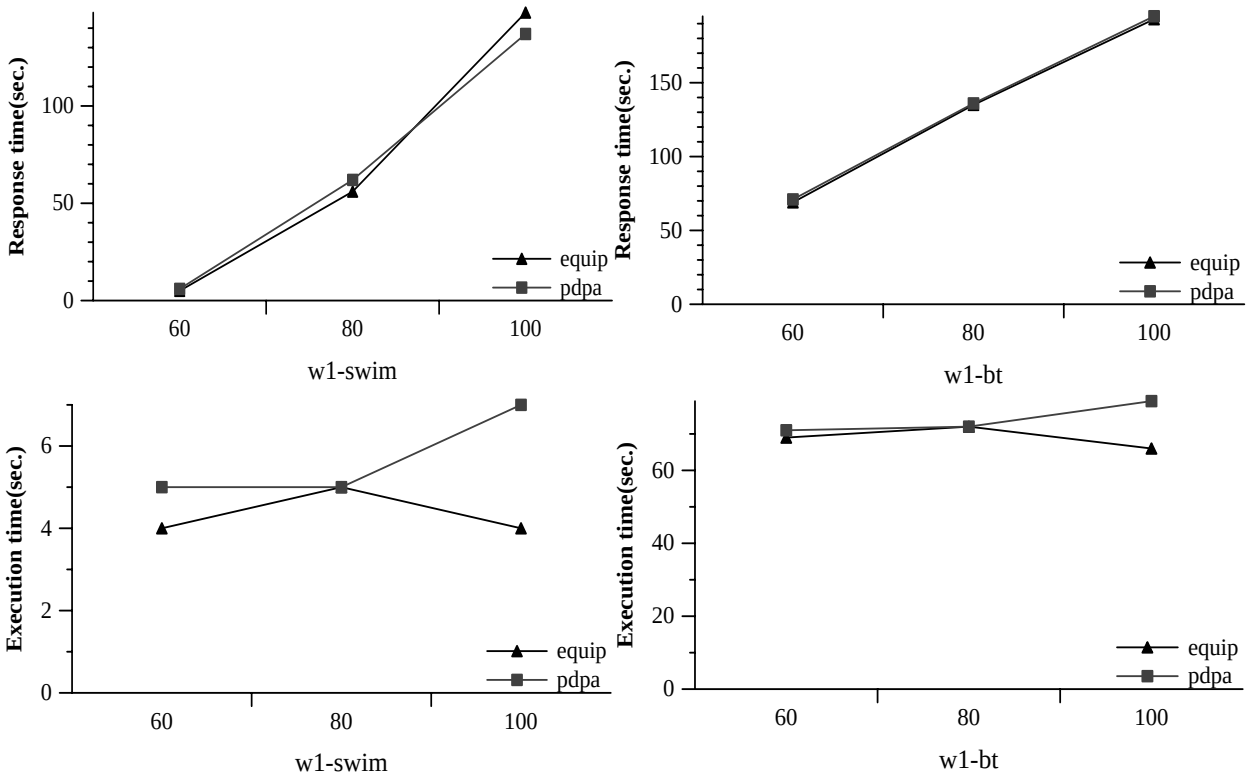


Figure 5.11: Results from workload 1, M.L.=2

Figure 5.12 shows the workload execution time when executed under Equipartition and PDPA with multiprogramming levels set to four, three, and two applications.

When the load is equal to the 60%, Equipartition and PDPA reach the same performance and the M.L. does not influence on the total execution time. In the case of the load set to the 80%, Equipartition slightly outperforms PDPA, but PDPA is slightly more stable considering changes in the multiprogramming level. The execution time of the workload under Equipartition is a 20% slower with M.L.=2 than with M.L.=4, and only a 10% with PDPA. When the load is equal to the 100%, the percentage is 13% slower with M.L.=2 than with M.L.=4, in the case of PDPA, and 28% in the case of Equipartition.

We can see in this graph that the best choice for the system is to use a higher multiprograming level because processors are more efficiently used. With PDPA we can set the default M.L. to a small value and let the policy to adjust it automatically.

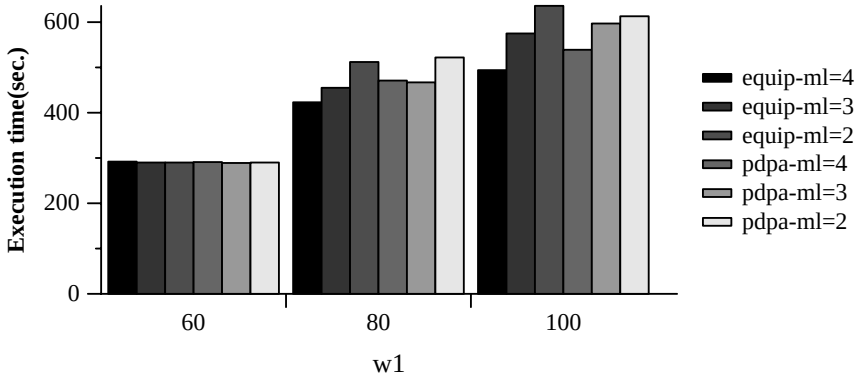


Figure 5.12: Workload execution time: Equipartition vs. PDPA, (workload 1)

Related issues that affect the system performance

As we have previously commented, there are several issues that affect the performance of parallel applications, not only the processor allocation.

We have observed that both Equipartition and PDPA significantly improve results achieved by IRIX and the Equal_efficiency. In the case of IRIX, the main reasons are the unresponsiveness of the native run-time to changes in the system load, and the coordination with the Launcher. But it also has a significant influence the placement policy used by IRIX. This placement policy is based on maintaining the processor affinity as much as possible. However, sometimes it generates that two kernel threads can be allocated to the same processor, degrading the application performance and generating a lot of process migrations.

Figure 5.13 shows the trace file visualization for the workload 1 executed under IRIX, PDPA, and the Equipartition (load=100%, M.L.=4). This graph has been generated with the Paraver Tool [52]. We have used Paraver to study the behavior of multiprocessor multiprogrammed environments, to debug our execution environment once implemented, and to evaluate the different system configurations. Each line represents the activity of a CPU and each color represents a different application. The x axis represents time, and we have set the same x scale to compare the three trace files.

We can appreciate that the look of the execution under the native IRIX scheduler is chaotic. The other two traces show that the respective executions are quite stable. We can clearly differentiate the execution of the different applications on them. We will show that this stability is very important to help the rest of mechanisms of the operating system (such as the memory migration) to do their work efficiently.

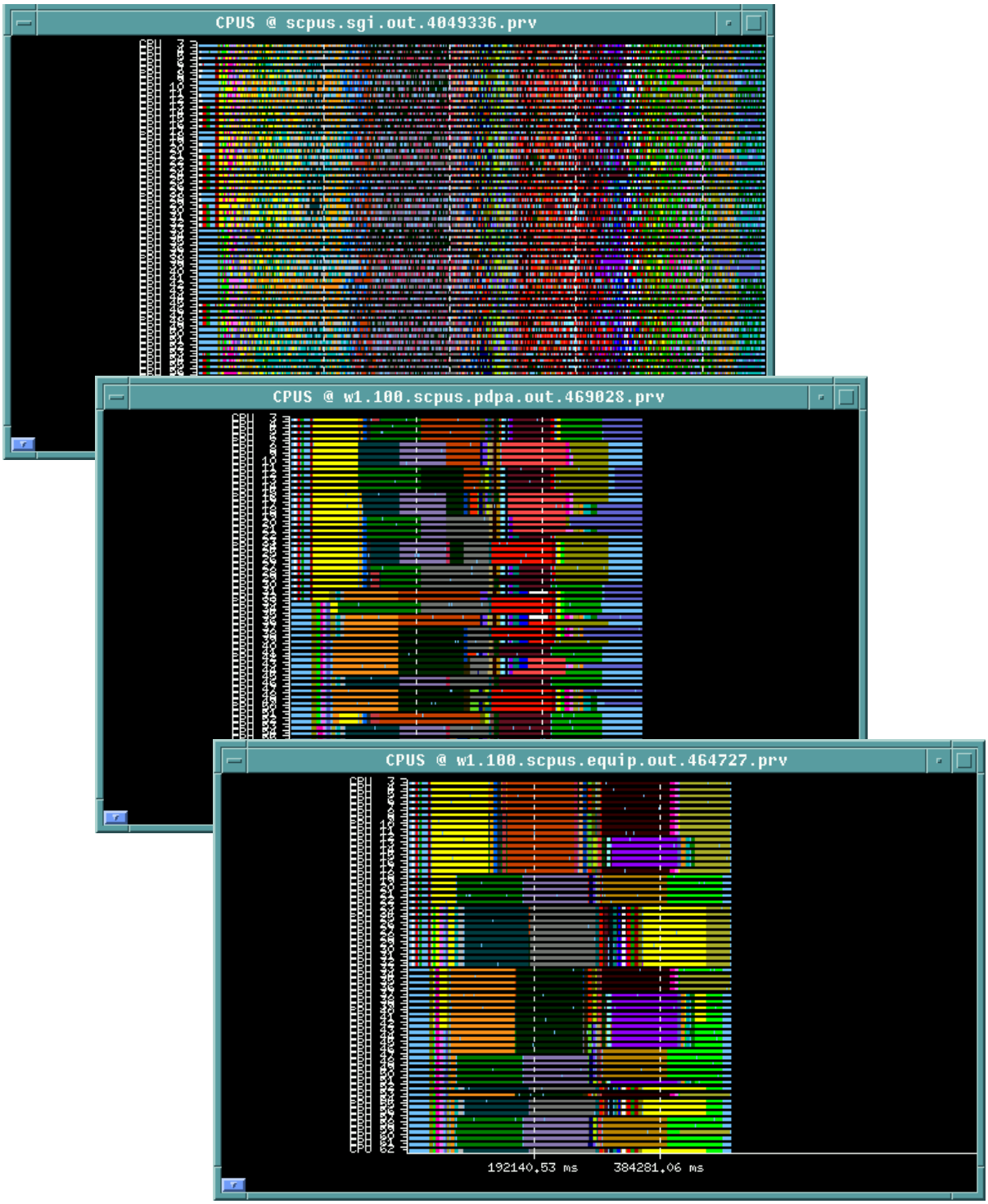


Figure 5.13: Execution views for workload 1 under IRIX, PDPA, and Equipartition (load=100%)

Also using Paraver, we have measured the total number of processes migrations, the duration of the bursts executed by each cpu, and the number of bursts executed per cpu. Table 5.3 shows the results obtained from the three policies. As we can see, the native IRIX scheduler generates much more kernel threads migrations. The time each cpu is executing the same application under IRIX is around 50 times less than under PDPA or Equipartition.

This behavior is not resulting from the processor allocation policy, it is generated by the rest of phases of the IRIX processor scheduler and by the IRIX run-time library characteristics. In this particular workload, we have measured the number of cpus allocated under IRIX and under Equipartition and in both cases they are around 15 processors.

Table 5.3: IRIX vs. PDPA and Equipartition, workload 1 (load=100%)

	Migrations	Average exec. time burst per cpu	Average number of bursts per cpu
IRIX	159.865	243 ms.	2882
PDPA	66	10.782 ms.	41
Equipartition	325	11.375 ms.	43

We have performed a second experiment to measure the effect of the processor scheduler quality in the system performance. We have not activated the memory page migrations to see how this mechanism influences in the execution time of both applications and the workload.

Figure 5.15 shows the execution time of workload 1 with and without memory migrations under the native IRIX scheduler, Equipartition, and PDPA (load=100%). Comparing Figure 5.15 with Figure 5.13, we can see that memory migrations has a significant and positive influence in the execution time of the workload. We can also observe the different effect depending on the policy. In the case of the native IRIX scheduler the memory migrations mechanism improves the execution time of the workload in only a 6%, whereas in the Equipartition the speedup has been 72%, and in PDPA 38%.

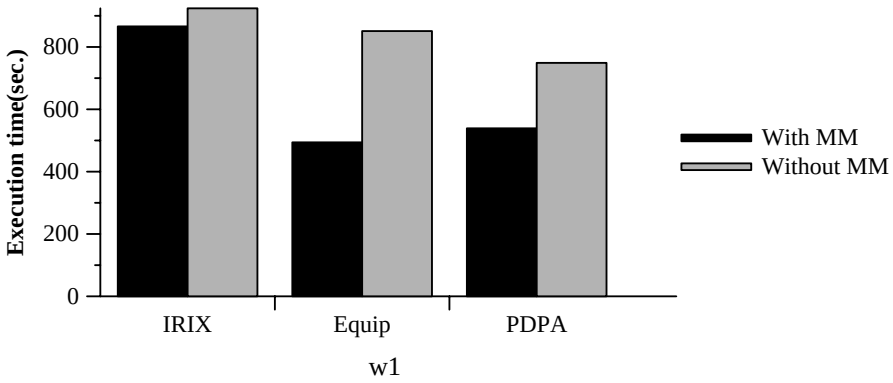


Figure 5.14: Execution time of workload 1 with and without memory migrations

This is because the memory migration mechanism can only improve those workloads that keep stable enough the processor allocation of the applications and, as we show in Table 5.3, this is not a characteristic of the native IRIX scheduler. Figure 5.15 shows the visualization of the execution of the workload 1 under the native IRIX scheduler, Equipartition, and PDPA, without memory migrations (load=100%). Although the execution view has a similar behavior to that shown in Figure 5.13, the execution times of the workload without memory page migrations are very different. With these measurements we want only to give a hint about the importance for the system of having a common goal in all the components (processor scheduler, memory management). We have executed the rest of workloads with and without memory migrations, and we have found that the dynamic memory migration mechanism improves the system performance in all the workloads and with all the policies evaluated.

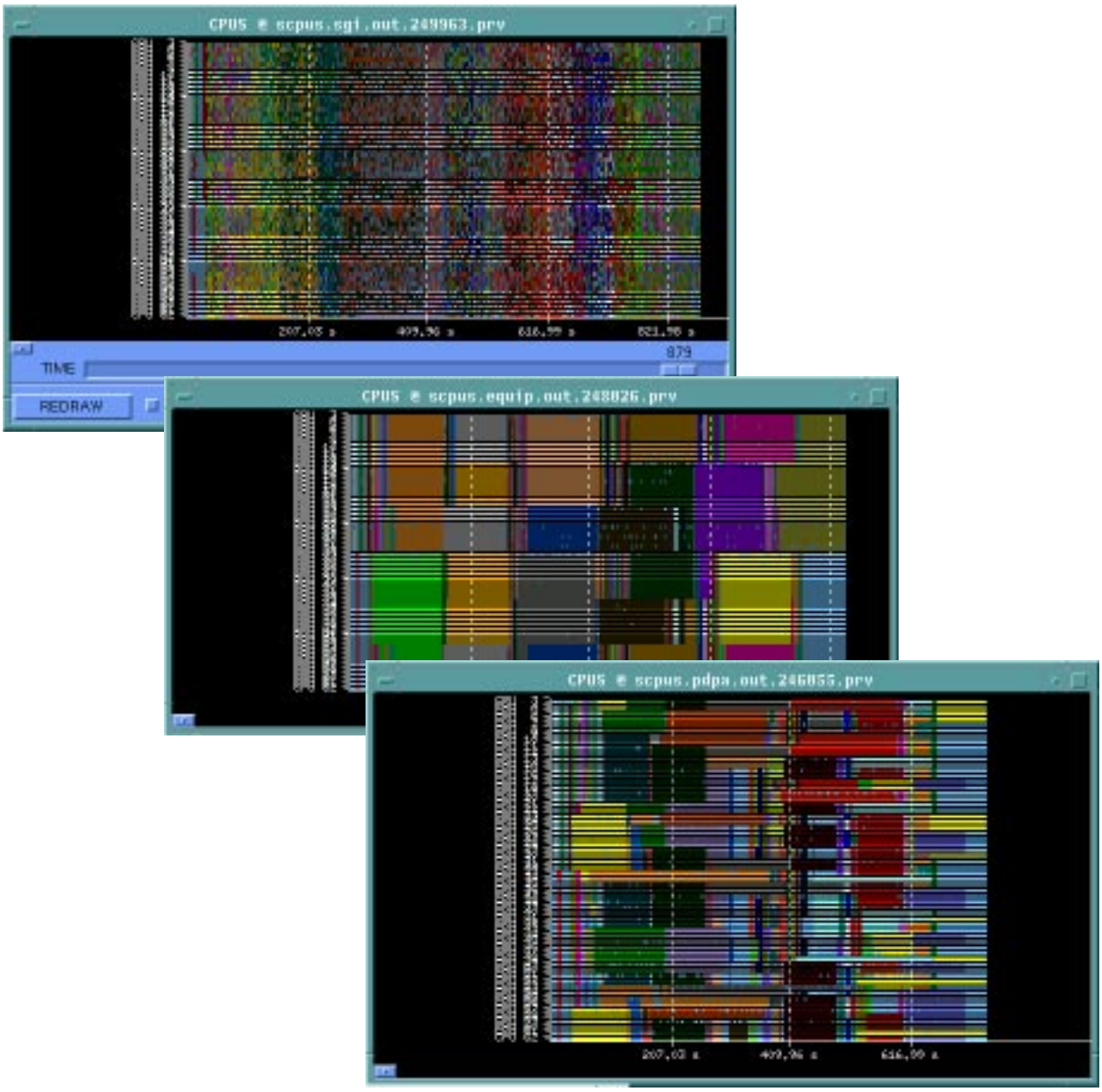


Figure 5.15: Execution of workload 1 without memory migrations under IRIX, Equipartition, and PDPA (load=100%)

5.5.2 Workload 2

Figure 5.16 shows results from workload 2. Graphs in the top of the figure show the average response time (in seconds) of bt's applications (highly scalable), and hydro2d's (medium scalability). In the x axis, we represent the different loads of the system generated. Graphs in the bottom of the figure show the average execution time (in seconds) of bt's and hydrod2d's. In this workload, the request of all the applications has been set to 30 processors.

This workload has been designed to evaluate the behavior of the evaluated policies when executing a workload where the 50% of the load consists of applications with high scalability, and the rest have a medium scalability.

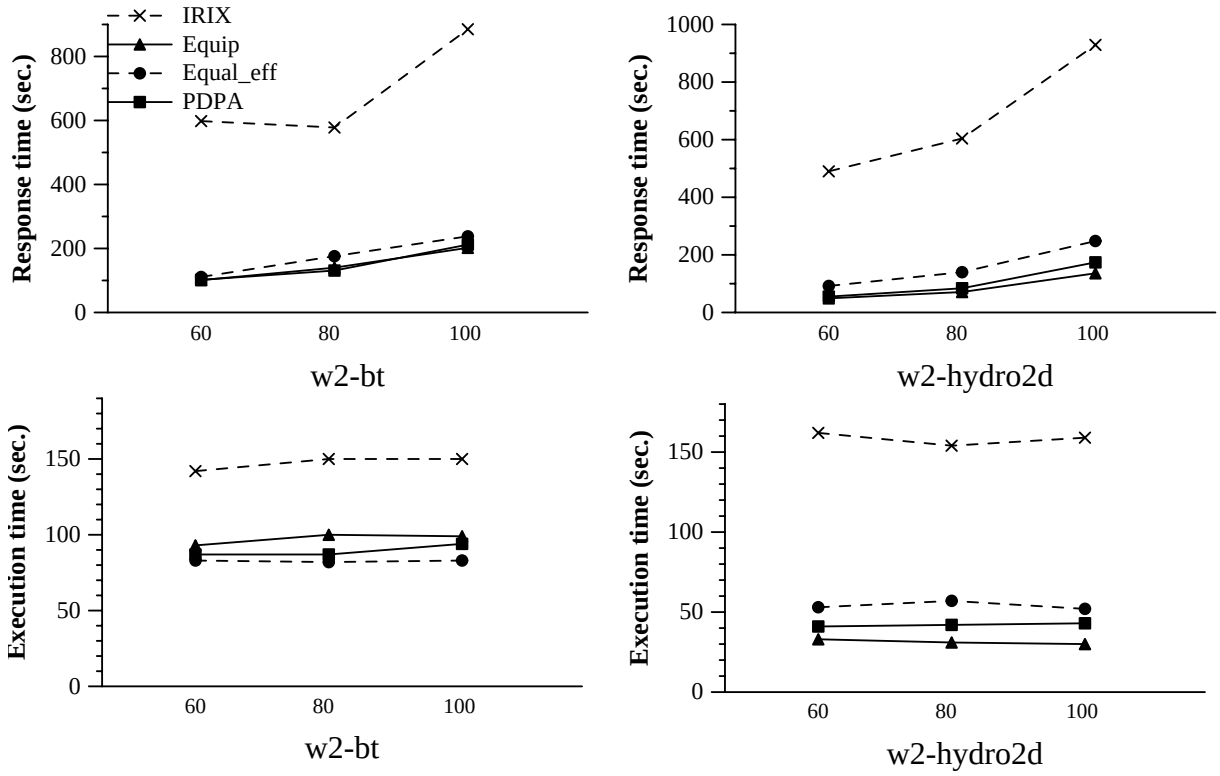


Figure 5.16: Results from workload 2, M.L.=4

Results show a behavior similar to workload 1. Equipartition and PDPA significantly improve IRIX and Equal_efficiency, and the two policies show a very smooth increment in the response time when increasing the system load.

To see the benefits provided by PDPA in this workload we have to analyze the workload execution in more detail. The percentage of cpu's that, in average, receives each type of application is 20 cpus to bt's and 9 cpus to Hydro2d's. The allocation decided by Equipartition is the same to both applications (around 15). This better distribution results in a better execution time of bt's executed under PDPA compared with bt's executed under Equip. In this workload, PDPA outperforms Equipartition by 10% in the response time and execution time of bt's, but in the case of hydro2d's, Equipartition outperforms PDPA between 23% and 30%.

In the case of the hydro2d's, even if the response time is quite the same under PDPA and Equipartition, the execution time is slightly worse with PDPA due to two reasons, the small number of processors allocated to them, and that the hydro2d is an application that suffers overhead due to the measurement process.

Comparing results achieved by PDPA with the Equal_efficiency we can see that PDPA outperforms the Equal_efficiency both in the response time and in the execution time. However, in this workload, the difference between PDPA and Equal_efficiency is less significant than in the previous workload. PDPA outperforms Equal_efficiency by 18% in the case of the response time of bt's and by 58% in the case of the response time of hydro2d's. In the case of load=100%, the Equal_efficiency has allocated 30 processors (in average) to bt's and 10 processors (in average) to hydro2d's.

Till now, it seems that PDPA does not provide significant benefits to the workload executions if a pervious tuning of both applications and system parameters have been performed previously. However, we will see that if we change any of these parameters, PDPA is able to maintain the system performance whereas the Equipartition is not. PDPA is quite robust to both changes of the application request (which depends on users), and on the system parameters.

Multiprogramming level set to three applications

As in the previous workload, we have also executed this workload with the multiprogramming level set to three and two. Figure 5.17 shows results for workload 2 executed with a multiprogramming level set to three applications. Graphs in the top of the figure show the response time achieved by bt's and hydro2d's executed under Equipartition and PDPA, and graphs in the bottom of the figure show the execution time for bt's and hydro2d's.

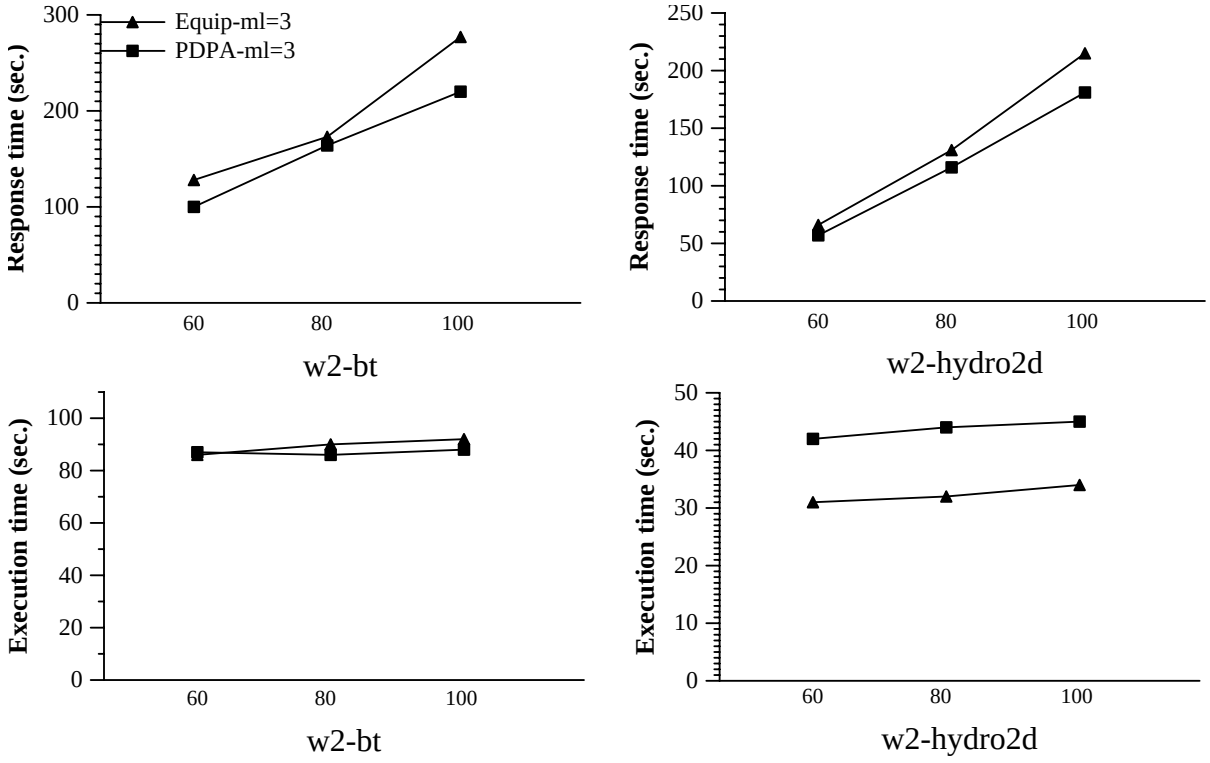


Figure 5.17: Results from workload 2, M.L.=3

If we compare the response time, PDPA outperforms Equipartition. This is because PDPA distributes processors proportionally to the application performance, not proportionally to the number of running applications (as the Equipartition does). Observing the execution time, the Equipartition outperforms PDPA in the case of hydro2d's, and this is because of two reasons: PDPA assigns less processors to hydro2d's than Equipartition, and hydro2d is the application with more overhead introduced by SelfAnalyzer.

Multiprogramming level set to two applications

Figure 5.18 shows results for workload 2 executed with a multiprogramming level set to two applications. In this case, applications under the Equipartition are receiving as many processors as they request, because the multiprogramming level is set to two, and they request for 30 processors. This is the reason why the execution times under Equipartition are better than under PDPA. However, PDPA significantly outperforms Equipartition when analyzing the response time achieved by applications. PDPA allocates less processor to applications than Equipartition, and increases the multiprogramming level, improving the efficient use of the system.

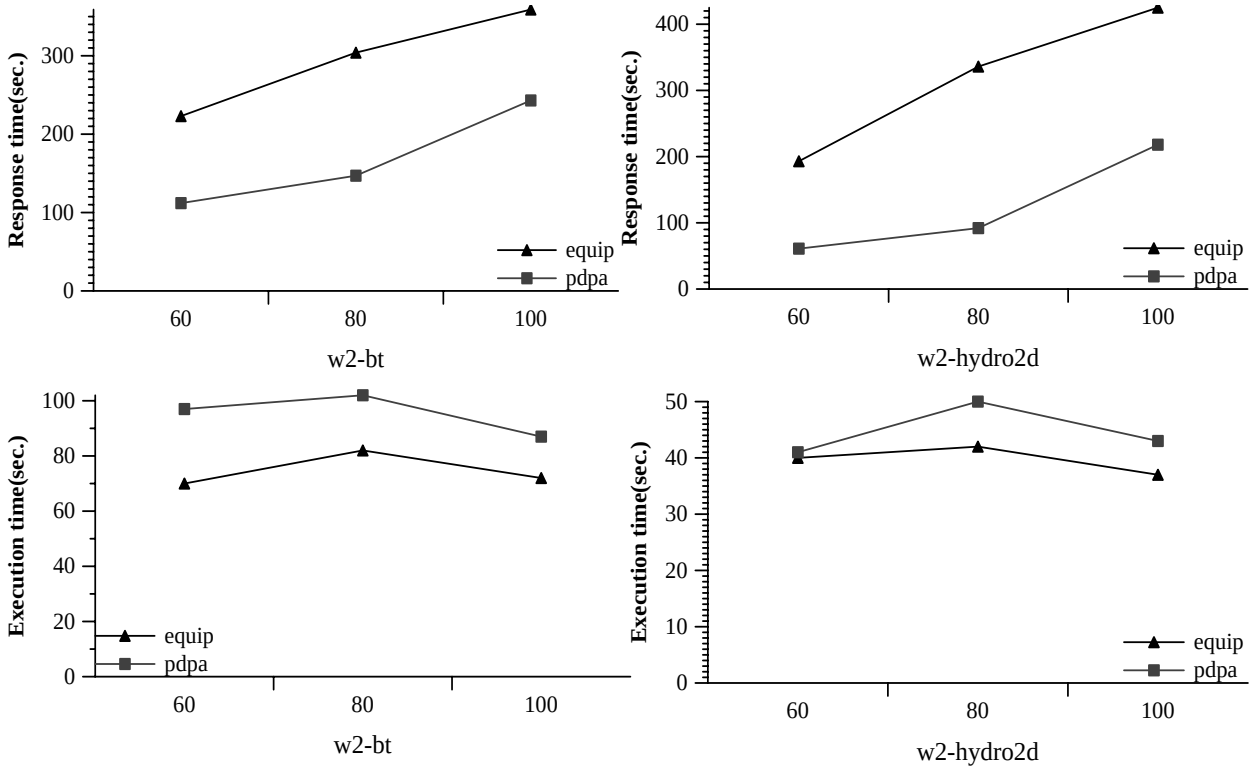


Figure 5.18: Results from workload 2, M.L.=2

Figure 5.19 shows the processor allocation dynamically decided by PDPA in a subset of applications. This figure only shows the task view of a portion of the workload. We can see that PDPA implements a search for the maximum number of processors that reaches the target efficiency, and also some moments where it tries to allocate more processors. In the case of the first application (hydro2d), PDPA allocates a number of processors that achieves an acceptable performance, then its allocation is maintained till the application finishes. In the case of the second application (bt), PDPA detects that it can use more processors and when it is possible, it increases its allocation, finally it receives 28 processors. The three last applications are hydro2d's. They initially receive a number of processors that PDPA considers not acceptable, then their allocation is reduced. Note that the reduction in the processor allocation has not been in four processors, this is because the dynamic step used by PDPA.

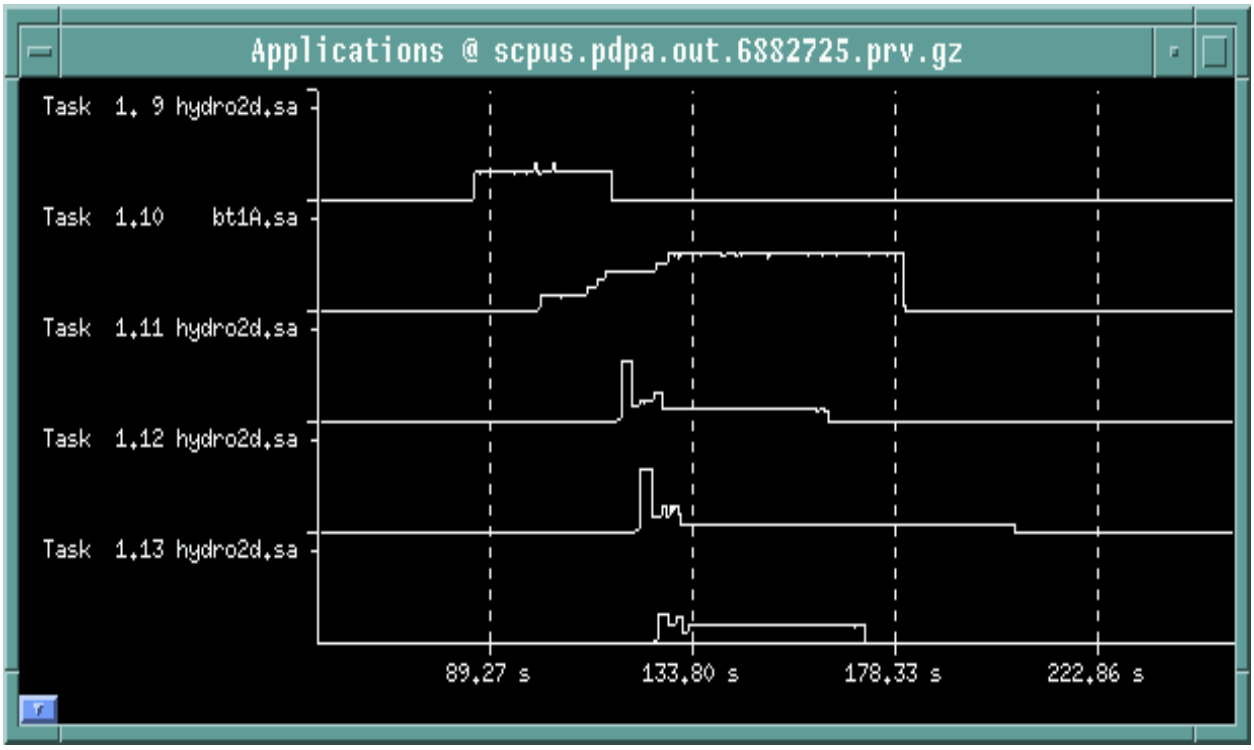


Figure 5.19: Processor allocation decided by PDPA (subset of applications)

We have also measured the standard deviation in the processor allocation and in the execution time of applications in the case of Equipartition and PDPA.

Table 5.4: Standard deviation

	bt		hydro2d	
	Allocation	Execution Time	Allocation	Execution time
Equip	2.09	7.18	3.09	9.04
PDPA	2.72	8.59	3.3	9.67

Table 5.4 shows the standard deviation generated in the workload of load=100%. We can see that PDPA generates roughly the same deviation than Equipartition.

Figure 5.20 shows the complete list of applications for Equipartition and PDPA. In this case we show the processor allocation as colors with different gradient: dark-blue colors mean many processors, and light-green colors mean few processors. We can see that Equipartition allocates more processors to applications than PDPA. However, we can also see that Equipartition does not differenciate between bt's and hydro2d's. PDPA detects that bt's scale better than hydro2d's and it allocates more processors than hydro2d's. We

have calculated the average processor allocation for the execution of this workload and we have found that PDPA has allocated (in average) 8 processors to hydro2d's, and 23 processors to bt's.

We have set the same time scale (x axis) in both graphs to compare them. We can observe that PDPA outperforms Equipartition because (1) PDPA decides a better processor distribution, and (2) since processor are efficiently used, more applications can be executed concurrently, improving the throughput of the system.

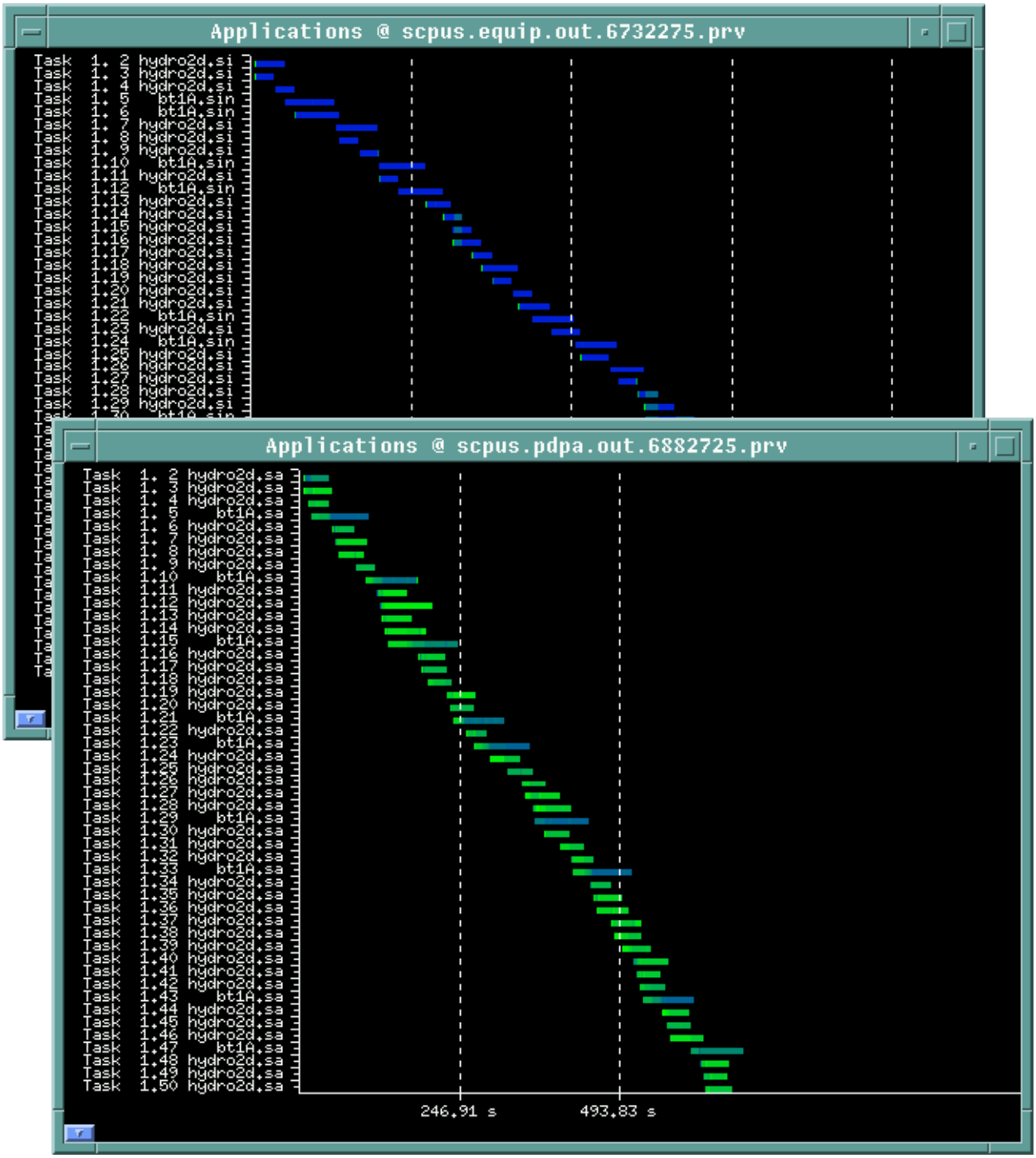


Figure 5.20: Workload 2, processor allocation decided by Equipartiton and PDPA (M.L=2, load=100%)

The multiprogramming level decided by PDPA has reached up to six applications during the workload execution (in the case of load=100%).

Figure 5.21 shows the dynamic multiprogramming level decided by PDPA in the case of load=100%. The x-axis is the time axis and the y-axis is the multiprogramming level value. As we can see, PDPA adapts the multiprogramming level to the characteristics of running applications, and it is not fixed during the complete execution of the workload.

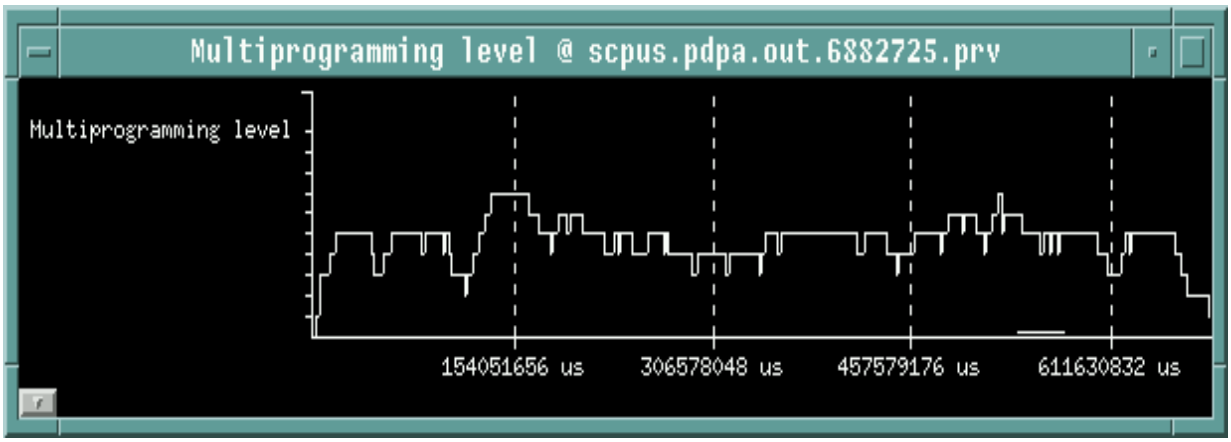


Figure 5.21: Multiprogramming level decided by PDPA

Figure 5.22 shows the workload execution time under Equipartition and PDPA when using different multiprogramming levels and with different system load (60%, 80%, and 100%). From this graph, we can extract two main conclusions. The first one is that PDPA is more robust than Equipartition to the multiprogramming level decided by the system administrator: PDPA dynamically detects the optimal value at each moment. In fact, the ideal decision in a system with PDPA is to set the multiprogramming level to a small value and let PDPA to dynamically adjust it. We have compared the execution time achieved by this workload when using a multiprogramming level of four vs. using a multiprograming level of two applications, in the cases of load=80% and load=100%.

Table 5.5: Slowdown introduced, ML=4 vs. ML=2

Policy/Load	80%	100%
Equipartition	97%	86%
PDPA	2%	15%

Table 5.5 shows the slowdown introduced by Equipartition and PDPA when executing with multiprogramming level set to two applications. It is calculated as the relationship between the execution time when the multiprogramming level was set to two and the execution time when the multiprogramming level was set to four. As we can see PDPA is more robust than Equipartition the value of the multiprogramming level.

The second conclusion that we can extract from this experiment is that PDPA is able to adjust the processor allocation of running applications taking into account their performance. As we can see in the case of the multiprogramming level set to two

applications, Equipartition allocates the number of processors requested, resulting in better execution times per application[4]. But this is not a good result neither for the system performance nor for the response time of applications. In fact, for users of a heavy loaded server, the response time is more important than the execution time because it includes the total time the application is in the system.

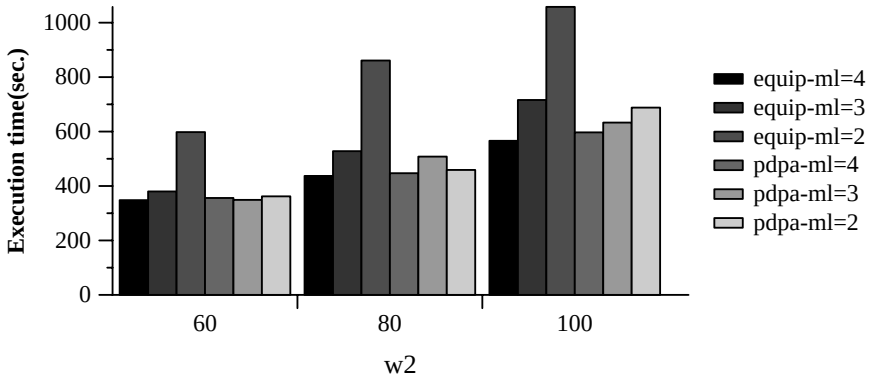


Figure 5.22: Workload 2, Equipartition vs. PDPA

5.5.3 Workload 3

Figure 5.23 shows results from workload 3. Graphs in the top of the figure show the average response time (in seconds) of bt applications (highly scalable), and apsi's (not scalable). In the x axis we represent the different loads of the system generated. Graphs in the bottom of the figure show the average execution time (in seconds) of bt's and apsi's. In this workload, the request of bt's has been set to 30 processors and the request for apsi's has been set to 2 processors.

This workload has been designed to evaluate the behavior of the evaluated policies when executing a workload where the 50% of the workload is composed by scalable applications, and the rest does not scale at all.

In this kind of workloads, PDPA can improve the processor scheduling by attacking two points: the first one is improving the processor allocation, and the second one is by coordinating with the queueing system. However, since we have performed a previous manual tuning of the processor request, the processor allocation of running applications can not be significantly improved (as maximum, apsi's could receive one processor rather than 2, but this is not a significant change). However, the system can be significantly improved if the processor scheduler and the queueing system are coordinated to better use the system in those moments that there are only one or zero bt's executing.

4. In this workload, because of the previous tuning.

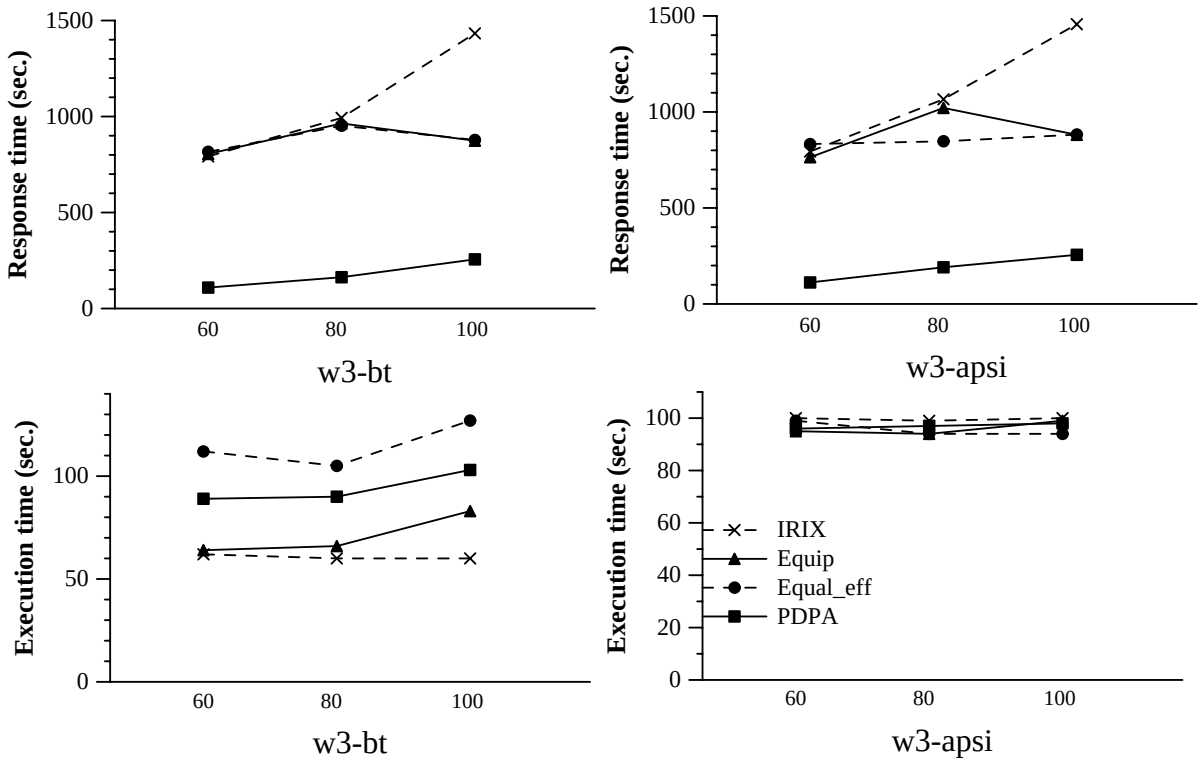


Figure 5.23: Results from workload 3

Results demonstrate our theory. If we observe the response time graphs, we can observe that PDPA significantly improves the rest of evaluated policies because both bt's and apsi's do not have to wait so many time queued before starting their execution. Those policies that do not coordinate with the processor scheduler are not able to see that, in some moments, the system is under utilized and that it could be started a new application. We have analyzed results from PDPA and we have found that the multiprogramming level has been set in some moments to 34 jobs (load=100%), see Figure 5.24.

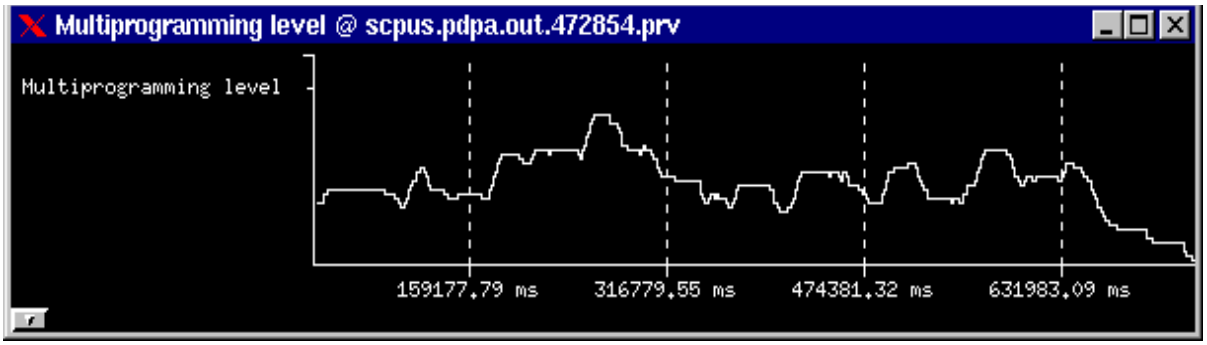


Figure 5.24: Multiprogramming level decided by PDPA, W3 M.L.=4, load=100%

Analyzing results in detail, we can see that in this workload PDPA outperforms Equipartition in a 600% in both the response time of bt's and apsi's, at the expense of only the 30% of slowdown in the case of bt's, and not slowdown in the case of the execution time of apsi's.

In this workload, the main problem for the Equal_efficiency is also the multiprogramming level. In this workload, the Equal_efficiency has allocated 30 processors in average to bt's and two processors to apsi's. However, we can see that even that bt's have received more processors than under PDPA, the execution time of bt's under PDPA is better than under Equal_efficiency. In average, PDPA outperforms the execution time of bt's in a 20%. Comparing the response time, PDPA outperforms the response time achieved by bt's under the Equal_efficiency in a 558%.

We have also executed some experiments modifying the processor request of apsi's to evaluate a case where apsi's were submitted without any previous tuning. The experiment consists of executing the same workload, with the same submission times but setting the apsi's request to 30 processors.

We have only executed the first case, with the load set to the 60%, because results were significant enough. Table 5.6 shows the results achieved in this case. We can see that PDPA significantly improves the Equipartition performance.

Table 5.6: Results from w3, apsi's requesting for 30 processors (not tuned) load=60%

	Bt		Apsi		Workload	ML
	Resp. time	Exec. time	Resp. time	Exec. time	Exec. time	
Equip	949 sec.	102 sec.	890 sec.	107 sec.	33 min. 13 sec.	4
PDPA	95 sec.	88 sec.	107 sec.	98 sec.	7 min. 7 sec.	29
PDPA speedup	998%	15%	831%	9%	466%	

Figure 5.25 compares the execution time of workload 3 when tuning the application request and without tuning it. A very important point is that the behavior of applications under PDPA is not affected if users have or do not have previously tuned their applications request. PDPA allow non-experts users to request for a high number of processors without fear that they use an excessive number of processors. The processor scheduling policy will be in charge of deciding the number of processors that they must use. Note that results with a tuned and with a non tuned workload are more or less equal. It does not happen the same with the Equipartition, which is not able to solve this situation.

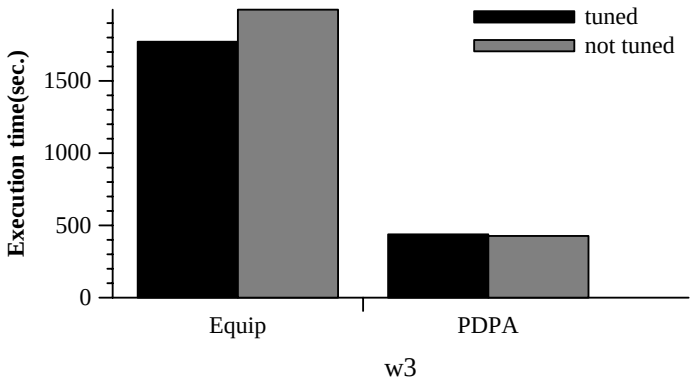


Figure 5.25: Workload 3, tuned vs. not tuned (load=60%)

5.5.4 Workload 4

Figure 5.26 shows results from workload 4. We show the average response time for swim's, bt's, hydro2d's, and apsi's, and the average execution time for the four applications.

This workload is a mix of the four type of applications, and each type receives the same amount of cpu percentage. The request of swim's, bt's, and hydro2d's has been set to 30 processors, and the request of apsi's has been set to two processors. As in the previous workloads, this is not the best case for PDPA because the processor allocation has been tuned and the load is quite enough to fill the complete machine. However, also in this case we can observe how the response time achieved by applications when executing under PDPA significantly improves results achieved by other policies, without significantly increase the execution time.

We have analyzed the processor allocation decided in the case of load=80%, and we have found that swim's have received (in average), 17 processors, bt's have received 20 processors, hydro2d's have received 10 processors, and apsi's 2 processors. Moreover, we have observed that the maximum multiprogramming level has been set up to 14 applications in some moments of the workload execution.

It can surprise that swim's receive less processors than bt's, having better speedup. This is because swim achieves the super-linear speedup in the first range of processors (between 8 and 16 processors). With the rest of processors the achieved speedup is also super-linear, but the relative speedup is not so high. On the other hand, bt's have a more progressive scalability, and their relative speedup is better. For this reason it is more beneficial for the system to allocate more processors to bt's than to swim's.

Analyzing results achieved by applications under the Equal_efficiency, we can see that they are similar to those achieved by the Equipartition and the native IRIX scheduling policy. We have calculated that, in average, Equal_efficiency has allocated 26 processors to swim's, 28 to bt's, 27 to hydro2d's, and 2 to apsi's.

We have compared the response time and the execution time achieved by applications under PDPA and Equal_efficiency. PDPA outperforms Equal_efficiency by 1095%, 502%, and 442% in the response time of swim's, bt's, and hydro2d's respectively. And Equal_efficiency outperforms PDPA by 16%, 8%, and 1% in the execution time of swim's, bt's, and hydro2d's respectively. However, this is at the expense of 52% more processors in the case of swim's, 40% more processors in the case of bt's, and 270% in the case of hydro2d's. Analyzing these results we can conclude that PDPA outperforms Equal_efficiency in this workload because we have improved the response time of application by (1000%..500%) at only the expense of an slowdown in the execution time of the (16% .. 1%).

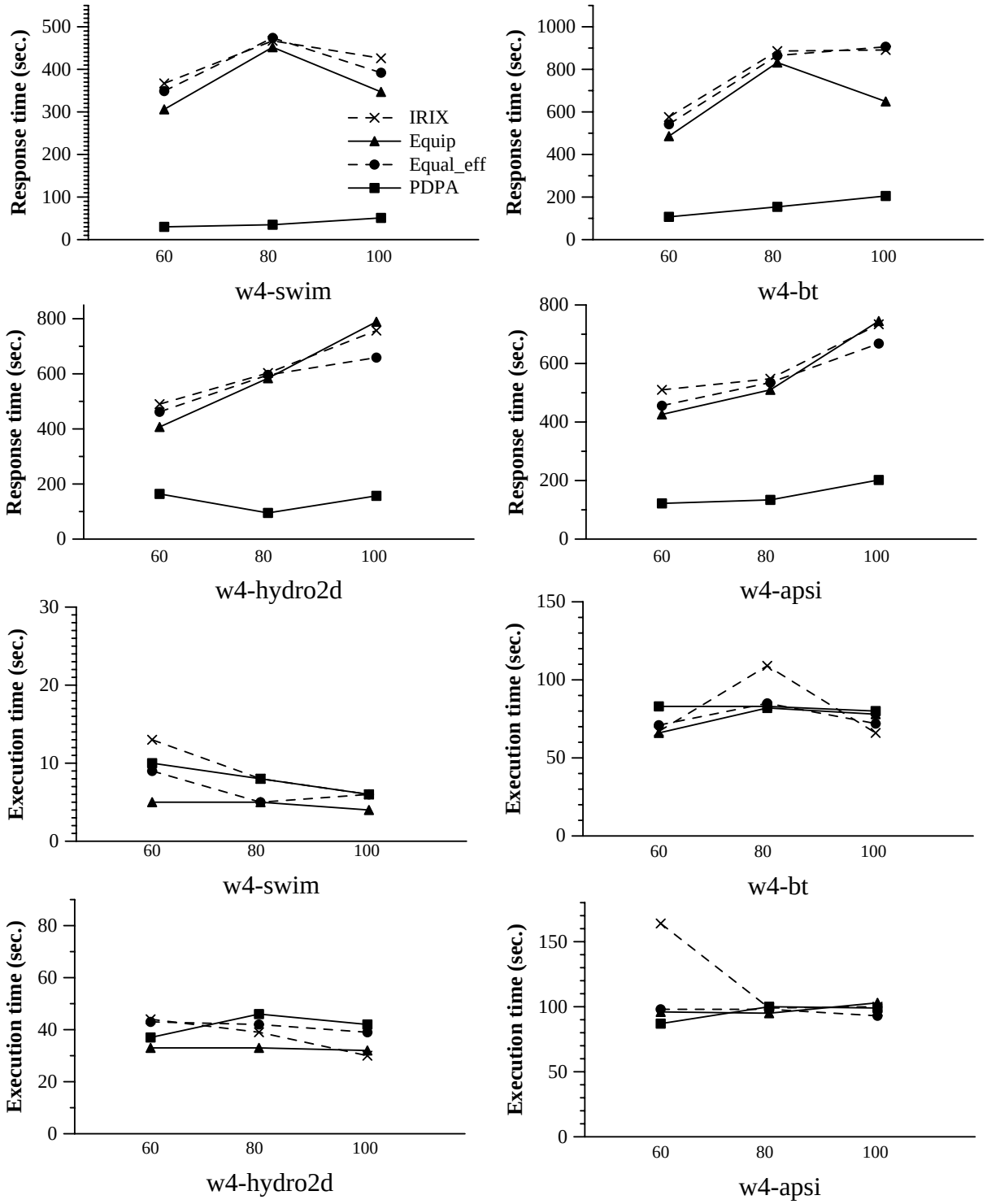


Figure 5.26: Results of workload 4

As in the previous workload, we have executed a different configuration of this workload, assuming that no previous tuning of applications has been performed and setting the request of all the applications to 30 processors (the only difference are apsi's).

Table 5.7 shows results for load=60%. The last row shows the ratio of PDPA vs. Equipartition. In those cases that Equipartition has improved PDPA, we have noted it as negative speedups. We can see how PDPA has outperformed the complete execution time of the workload by 282%, and the individual response time of applications from a 109% up to a 2830%. We can also see that all this benefits have been achieved by only sacrificing a maximum of 30% in the execution time of some applications.

Table 5.7: Results from workload 4 not tuned, load=60%

	swim, req=30		bt, req=30		hydro2d, req=30		apsi, req=30		Total
	exec.time	resp. time	exec.time	resp. time	exec.time	resp. time	exec.time	resp. time	exec.time
Equip	6sec.	368sec.	101sec.	568sec.	32sec.	453sec.	104sec.	773sec.	20min. 6sec.
PDPA	8sec.	13sec.	81sec.	92sec.	37sec.	45sec.	98sec.	109sec.	7min. 6sec.
%	-30%	2830%	-24%	617%	-15%	1006%	6%	109%	282%

Figure 5.27 shows the trace file visualization of the execution of the workload 4 without previous tuning when the load was set to the 80%. The x axis represesents time. We have set the same scale in both views to compare the cpu time consumed by the workload under the Equipartition and under PDPA. The dark blue color means cpu running and each row shows the cpu activity.

We can observe that the workload under PDPA has consumed less of half the cpu time to execute the same set of applications. This is a clear example that a *high cpu utilization* is not equal to a *good cpu utilization*. PDPA detects that some of the applications do not need so many processors, then it reduces their allocations and starts a new one. This policy significantly improves the system performance.

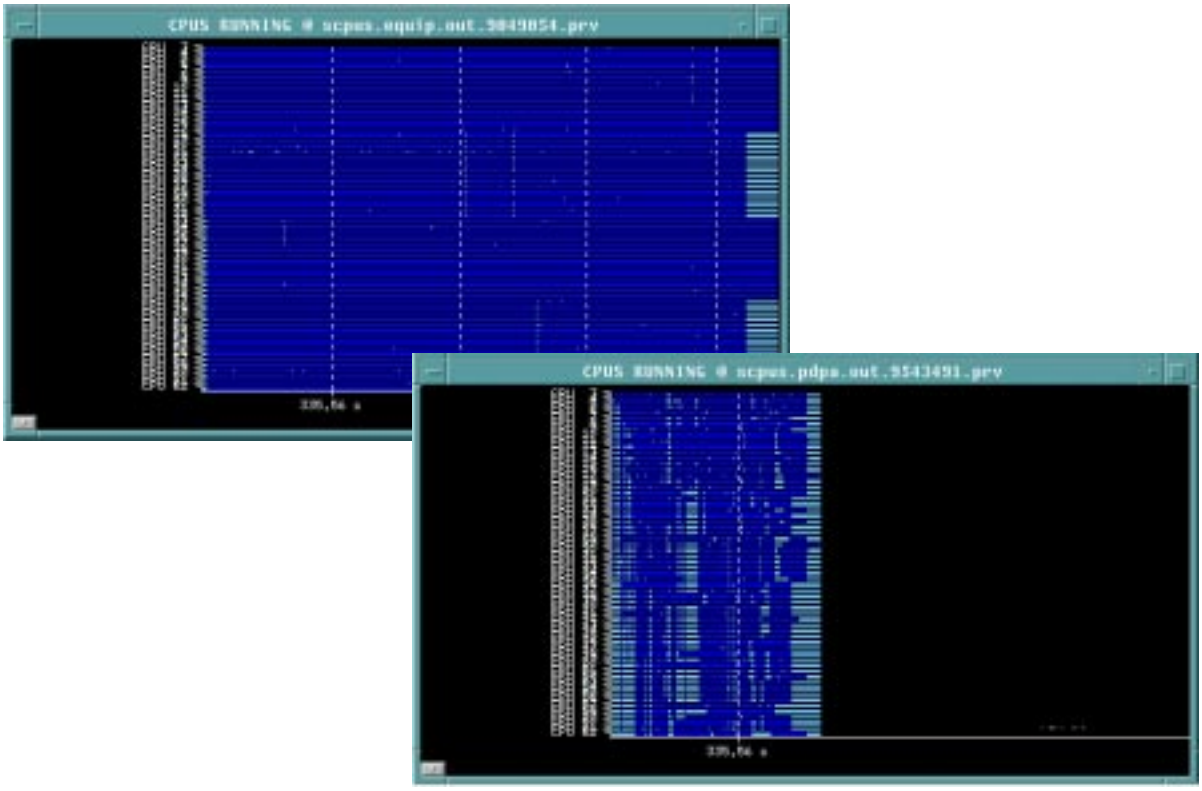


Figure 5.27: Workload 4. CPU utilization, Equipartition vs. PDPA, not tuned. Load=80%

5.5.5 Workload 5

Figure 5.28 shows results for workload 5, which is composed by applications that have a high scalability (bt's). This workload is quite similar to workload 1. We designed it with the goal of evaluate whether there was any difference if some of the applications were super-linear or not.

Results do not show significant differences compared with conclusions extracted from workload 1. We believe that the reason is that swim´s have an execution time small compared with the execution time of bt´s and their performance do not have a clear influence in the performance of neither bt´s nor in the workload performance.

However, since this workload have also been previously tuned in both the number of processors request per application and the multiprogramming level, the processor allocation generated by the Equipartition directly reaches a good efficiency. Then, Equipartition outperforms PDPA by 10% (in average) in both the execution time and the response time of bt's.

Compared with the Equal_efficiency, PDPA outperforms its results in both the response time and the execution time of bt's. In the case of the response time, PDPA outperforms Equal_efficiency by 40%. In the case of the execution time, PDPA

outperforms Equal_efficiency by 28%. The mean processor allocation when load=100% decided by Equal_efficiency is 18 processors. However, we have found that the processor allocation has a large standard deviation, ranging from 8 to 29 processors.

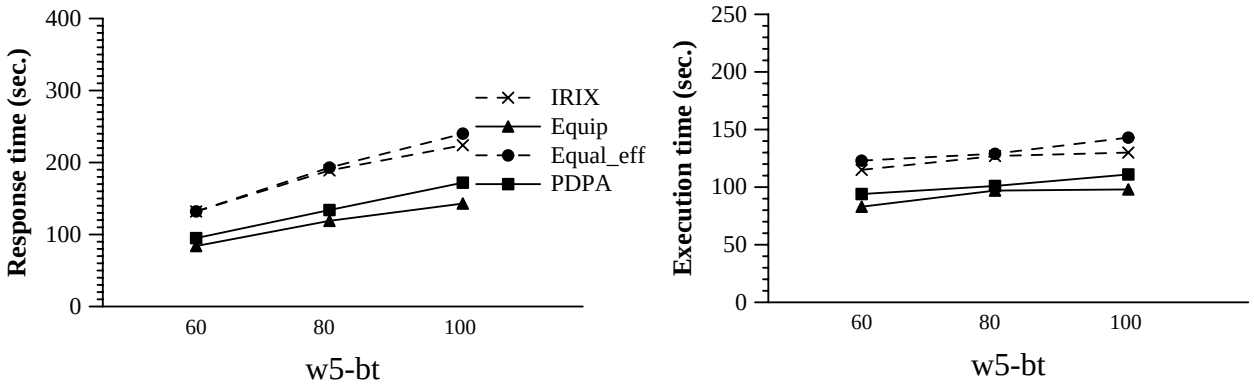


Figure 5.28: Results from workload 5

5.5.6 Workload execution times

Figure 5.29 shows execution times for the five workloads evaluated in this Thesis. As in the previous graphs, in the x axis we show the system load, and in the y axis we show the execution time of the complete workload. The execution time is calculated as the difference between the time the last application finishes its execution and the time the Launcher is started.We only show results for M.L.=4.

Workload execution times have a direct relationship with results observed in the individual response times of applications. In those workloads that PDPA achieves a similar performance, or slightly worse than Equipartition in the response time of applications, it also achieves a similar performance in the execution time of workloads. In particular, Equipartition outperforms PDPA by 10% in the first workload, workload 2 shows the same performance (in the case of M.L.=4). In workload 3 PDPA outperforms Equipartition in a 400% (at least), in workload 4 PDPA outperforms Equipartition by 240%, and in workload 5 Equipartition outperforms PDPS by 5%.

The speedup of PDPA with respect to the Equipartition in workload 3 is approximated because the execution of the workload was aborted. We decided to abort the execution of this workload because the more important thing was to demonstrate the validity of the ideas of the Thesis, not to know the execution time of the workload. Results presented in previous Sections have been calculated with applications that have finished before aborting the workload execution.

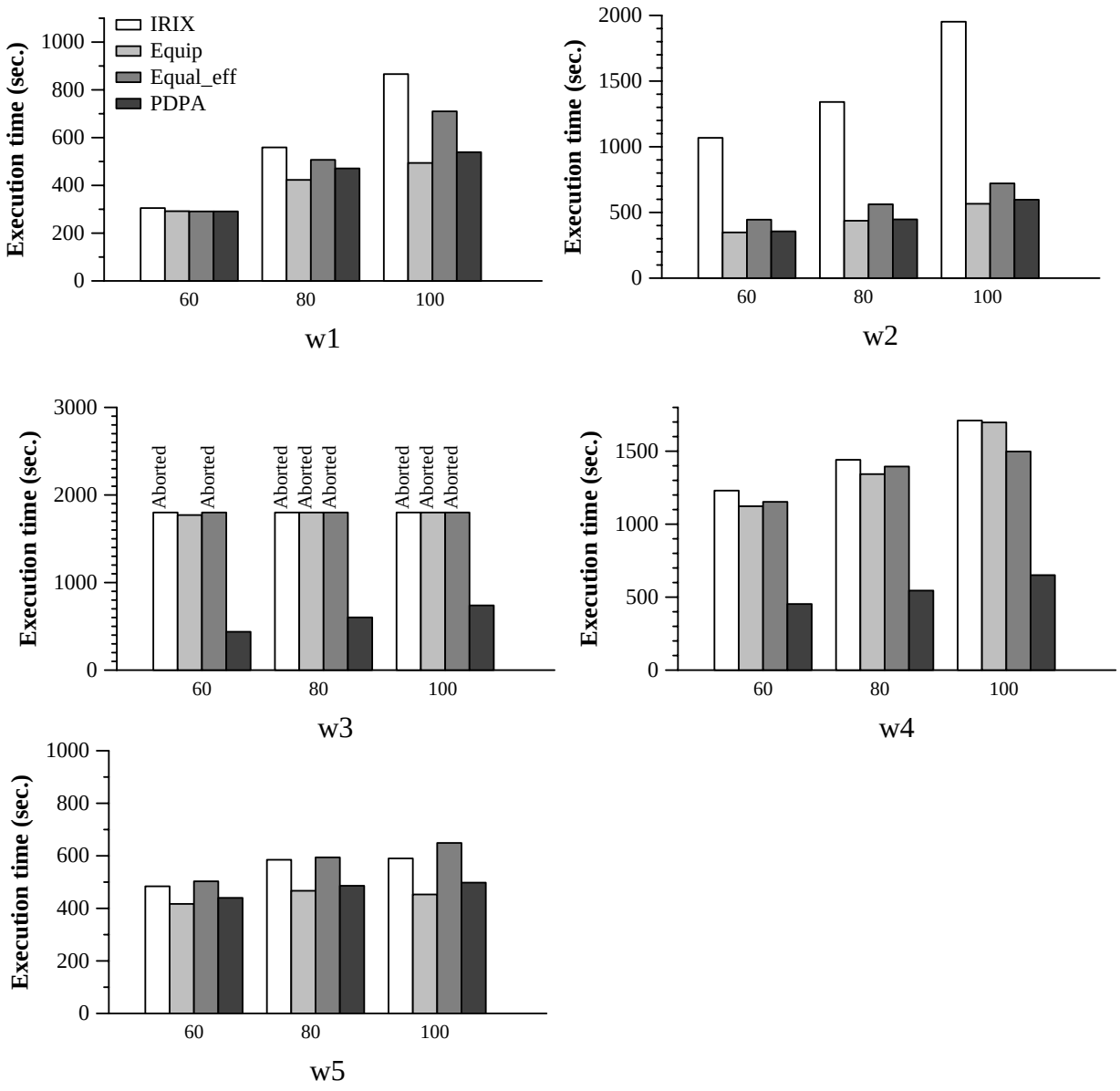


Figure 5.29: Execution time of the five workloads evaluated in this Thesis with M.L.=4

The expected performance of IRIX was equal or even better than the Equipartition (because it is the native parallel library and without the possible overhead introduced by the CPUManager). However, results show that Equipartition outperforms results achieved by the IRIX scheduler. This is also because of the coordination. We have experimentally observed that execution times of applications executed with the native parallel library, but executed in standalone mode, sometimes achieve slightly better results than the same experiment with the NthLib. However, when executing a multiprogrammed workload, the combination NthLib and CPUManager outperforms the native execution environment. All the scheduling policies executed under the CPUManager and the NthLib include coordination between the processor scheduler and

the run-time library. In addition, the NthLib includes specific mechanisms to provide an efficient execution of applications, such as the recovering mechanism, when executing in a multiprogrammed system.

As we have commented previously, the problem of the Equal_efficiency is not only the lack of coordination between the processor scheduler and the queueing system, but also the sensibility to changes in the performance values. We have observed that changes in the performance of one application can generate changes in the allocation of the complete workload, and also that two applications with the same speedup characteristics receive very different number of processors. Taking into account that the Equal_efficiency uses extrapolated values, this is a very usual situation.

5.6 Summary

In this Chapter, we have presented PDPA, a coordinated scheduling policy fully based on the performance of applications calculated at run-time. With respect to the processor allocation, PDPA tries to allocate to each application the maximum number of processors that reaches a target efficiency (low_eff). PDPA implements a multiprogramming level policy that decides to allow the execution of a new application if there are idle processors, and the allocation of all the running applications is stable or they do not need more processors.

Results show that in workloads composed by applications that scale well, previously tuned, and where the load is quite enough to fill the system, PDPA has improved results compared with IRIX and Equal_efficiency. Compared to the Equipartition, PDPA has introduced a maximum overhead of the 10% in the total execution time of the workload, and a maximum of a 30% in the individual response time of some applications.

In workloads that include not scalable applications, PDPA improves the system performance in two ways. The first one is by adjusting the processor allocation of applications to reach the target efficiency, ensuring the efficient use of processors. The second one is through dynamically adjusting the multiprogramming level, adapting it to the workload characteristics. We have executed these kind of workloads with and without previous tuning, and we have observed that benefits provided by the two points are orthogonal and complementary. In this kind of workloads, PDPA has outperformed Equipartition 400% in the total execution time. For these reasons, we can conclude that the fact of dynamically measuring the performance of applications and imposing a target efficiency gives PDPA a robustness that do not have the rest of evaluated policies.

Results also show that the first level of coordination between the processor scheduler and the run-time library, and the quality of the processor scheduler, are also very important, as demonstrates the differences between the Equipartition and the native IRIX scheduler. The processor allocation must be maintained, as much as possible, stable, because a high number of reallocations degrades the application and the system performance.

We have observed that it is very important that scheduling policies that use extrapolated values verify that these values correspond with the real ones.

CHAPTER 6

Performance-Driven Multiprograming Level

In the previous Chapter, we presented a specific processor allocation policy fully based on performance analysis. However, we believe that performance analysis is a point that is not exclusive from other criterion used in processor scheduling policies. It can be included in processors scheduling policies to self-evaluate processor scheduling decisions based on the performance achieved by running applications.

For this reason, we present a new methodology to improve scheduling policies by including job performance analysis and coordination with the queueing system. We call it Performance-Driven Multiprogramming Level (PDML).

PDML has been applied to Equipartition and Equal_efficiency, we have named the resulting policies Equip++ and Equal_eff++. Results show that PDML introduces significant benefits in Equipartition. In the case of Equal_eff++, it depends on the workload. Characteristics of the original Equal_efficiency generates that the benefits depends on the workload and the application.

6.1 Introduction

In the previous Chapter, we have shown that by means of a scheduling policy that coordinates the three different scheduling levels, and that considers the performance of parallel applications to distribute processors, we can significantly improve the system performance. The use of performance information ensures the efficient use of resources, processors in this case. And the coordination between levels ensures the efficient system utilization.

We have observed that the main points that constitute PDPA (use real performance information / ensure target efficiency / coordinated scheduler), are orthogonal to other criterion used in processor scheduling policies. We believe that the benefits provided by the ideas defended in this Thesis can be added to those concepts exploited in other scheduling policies.

To demonstrate it, we propose a new methodology that will transform processor scheduling policies to include the ideas defended in this Thesis, Performance-Driven Multiprogramming Level (PDML). PDML is a methodology that transforms policies in iterative algorithms that self-evaluate their decisions and modify them based on (1) their original criteria, and (2) the achieved application performance (efficiency). It also includes a multiprogramming level policy to implement the coordination with the queueing system.

We have applied PDML to two scheduling policies proposed so far, the Equipartition and the Equal_efficiency. We refer to the modified Equipartition and Equal_efficiency policies as Equip++ and Equal_eff++.

Results show that Equip++ is able to detect situations where the original Equipartition fails in its decisions, improving its performance. In fact, in some workloads the Equip++ even improves PDPA. This could be expected because the Equip++ incorporates the benefits of both PDPA and Equipartition.

On the other hand, the benefit that PDML introduces in the Equal_efficiency depends on the workload and the application. The Equal_efficiency takes scheduling decisions based on extrapolated values. These extrapolated values do not always correspond with the real ones. This fact, combined with the fact that the Equal_eff++ limits the processor allocation if the performance achieved does not reach the target efficiency, generates that some applications receive a small number of processors, even though they do not have a real bad scalability. This behavior is more frequent if applications extrapolate their efficiency values based on a efficiency calculated with few processors. In any case, in average, Equal_eff++ has improved the performance achieved by Equal_efficiency.

The remainder of this paper is organized as follows: Section 6.2 describes the methodology presented in this Chapter. Section 6.3 describes the two scheduling policies to which we have applied PDML, Equipartition and Equal_efficiency, and the resulting

scheduling policies, Equip++ and Equal_eff++. Section 6.4 presents the resulting processor scheduling policy taxonomy after including our policies. Section 6.5 presents the evaluation of this proposal, and finally Section 6.6 summarizes this Chapter.

6.2 Performance-Driven Multiprogramming Level

Performance-Driven Multiprogramming Level is a methodology that incorporates the concepts of:

- Use of real performance information
- Ensure a target efficiency
- Coordination between scheduling levels

to previously proposed processor scheduling policies.

To do that, PDML transforms the processor scheduling policy in an iterative algorithm that self-evaluates its decisions based on the performance achieved by running applications. It also includes a default multiprogramming level policy.

The processor allocation is initially decided based on the original policy criteria. If the performance achieved reaches the target efficiency, it will be considered acceptable. Otherwise it is considered not acceptable. If the performance is acceptable, the processor allocation will be kept up until the original re-allocation conditions given by the original policy indicate that it must be changed. If these conditions become true, or the performance is not acceptable, the processor allocation is adjusted based on both, the performance information, and the policy criteria.

With respect to the multiprogramming level policy, we have implemented a default policy that decides to allow the execution of a new application if there are free processors and the efficiency of all the running applications is greater than the target efficiency.

6.2.1 Processor scheduling policy scheme

Usually, a dynamic processor allocation policy decides the processor distribution based on a criteria and does not change it until the workload or the policy parameters change. For instance, Equipartition does not change the processor distribution until a new application arrives or a running application finishes. Figure 6.1 shows the general behavior of dynamic space-sharing policies. These policies decide the processor allocation and the scheduler, in our case the CPUManager, enforces it. Depending on the policy, there are several conditions, such as the arrival of a new application, that determine when the processor allocation policy must be re-applied.

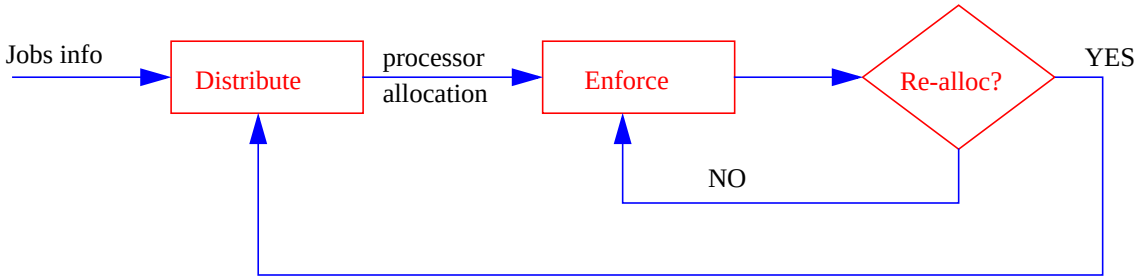


Figure 6.1: Typical processor scheduling policies behavior

Performance-Driven Multiprogramming Level: Processor Allocation

Figure 6.2 summarizes the Performance-Driven Multiprogramming Level (PDML) methodology, with respect to the processor allocation policy. We propose to periodically check if policy decisions are acceptable considering the performance achieved by running applications. When applications inform about their performance (speedup, efficiency), the scheduler evaluates if the performance of running jobs reaches the target efficiency. As in the case of PDPA, the target efficiency is a parameter and can be as simple as a static value, or more elaborated such as a function of the number of queued jobs or the memory utilization. The value of the target efficiency defines the aggressiveness of the policy. The higher the target efficiency, the higher the scalability of application must be to receive processors.

If any of the running applications does not reach the target efficiency, the processor allocation of these applications is reduced in *step* processors. *Step* is also a policy parameter that can be a static value such as one or four processors per turn. In some cases, it can be more efficient to reduce the allocation more aggressively to avoid excessive reallocations. It can be also proportional to the difference between the achieved efficiency and the target efficiency, reducing its value when the efficiency is closer to the target efficiency.

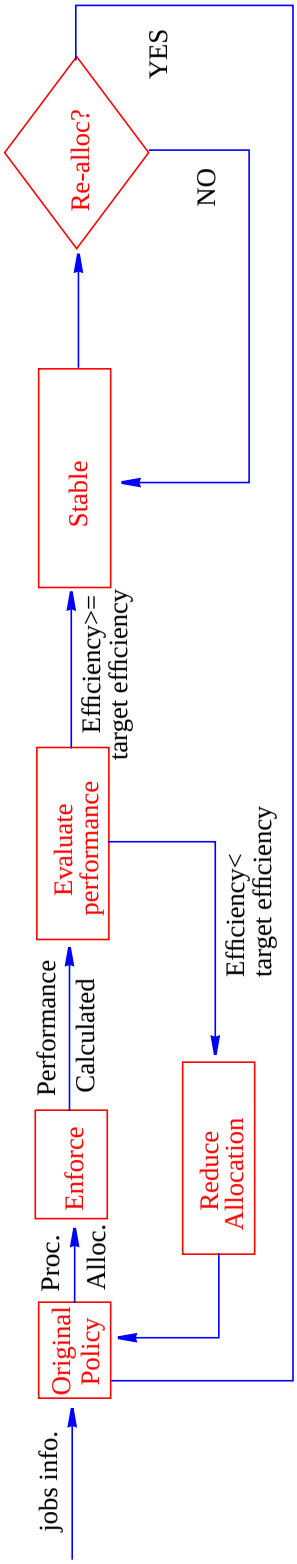


Figure 6.2: Performance-Driven Multiprogramming Level: processor allocation

Once reduced the processor allocation of those applications that do not reach the target efficiency, the processor scheduling policy is re-applied. The processors that are freed by those applications that do not reach the target efficiency are reallocated among the rest of applications. Once all the applications reach the target efficiency, the system becomes stable.

Performance-Driven Multiprogramming Level: Multiprogramming level policy

The multiprogramming level policy is applied when the system is stable. As we have commented, the system becomes stable when the efficiency of all the running applications reaches the target efficiency.

PDML includes a default multiprogramming level policy that returns true if the system is stable and there are free processors. What the processor scheduling policy must decide is what it considers a stable allocation.

The implementation of the dynamic multiprogramming level uses the same interface proposed in the previous Chapter, then we will only present the specific *Policy_New_appl*() function (it returns true when a new application can be started).

In next Sections, we present the resulting policies after applying PDML to the Equipartition and the Equal_efficiency: the Equip++ and the Equal_eff++.

6.3 Scheduling policies

We have applied PDML to two previously proposed processor scheduling policies: Equipartition and Equal_efficiency. In next sub-sections, we describe these two policies in detail and the resulting policies: Equip++ and Equal_eff++.

6.3.1 Equipartition

Equipartition is a dynamic space-sharing policy proposed by McCann *et al*. in [65]. Figure 6.3 shows the algorithm that implements the Equipartition algorithm in the CPUManager. The main goal of the Equipartition is to perform an equal allocation among running applications.

```
input: job_table(jobs)
output: table with number of processors per job (alloc)
void Equipartition()
{
cpus_available=MAX_CPUS
cpus_requested=Sum_individual_request()
Reset_cpus_allocated()/* Set to zero the alloc structure*/
current_job=0
while ((cpus_available>0) && (cpus_requested>0)){
(1) if (jobs[current_job].requested>alloc[current_job]){
     alloc[current_job]++
     cpus_available--
     cpus_requested--
   }
   current_job=(current_job+1)%active_jobs
}
}
```

Figure 6.3: Equipartition algorithm

Figure 6.4 shows the complete scheme that defines the Equipartition behavior. Once decided the processor allocation, it is maintained until a new application starts its execution or a running application finishes its execution. In that case, the Equipartition algorithm, is re-applied. The problem of Equipartition is that in most cases an equal allocation is not a synonym of neither *equal performance* nor *good performance*. If we apply PDML to the Equipartition we achieve the Equip++.

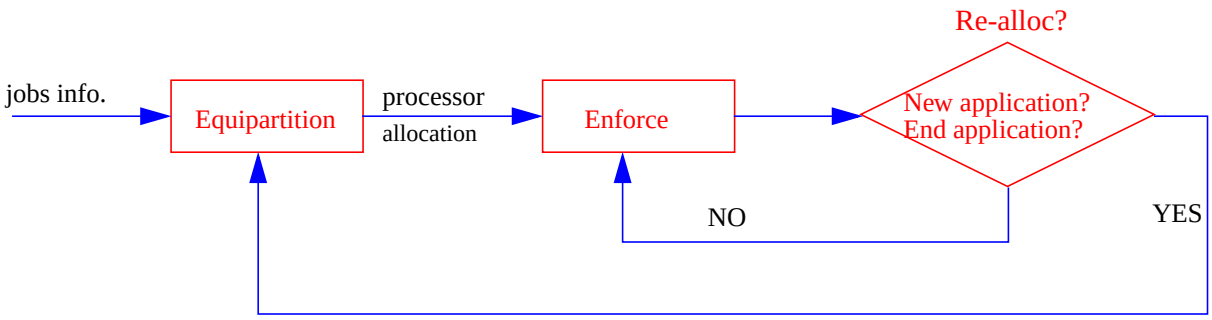


Figure 6.4: Equipartition scheme

6.3.2 Equip++

Figure 6.5 shows the resulting Equip++ algorithm: the processor allocation policy and the multiprogramming level policy.

Before allocating a new processor to an application, (1) in Figure 6.3, the Equip++ not only checks if the number of requested processors is greater than the number of allocated processors, but also checks if the efficiency with *(alloc[curr_job]+1)* has been measured. If it has been measured, the algorithm checks if the achieved efficiency is greater than target efficiency. In that case, the processor is allocated to the application, and the algorithm goes on with the next processor and application. Otherwise, the application will not receive more processors in the next quantum.

If the efficiency of *(alloc[curr_job]+1)* is not calculated, the processor is allocated to the application, being optimistic, and the process continues allocating processors.

```
input: job_table(jobs),target_efficiency
output: table with number of processors per job (alloc), STABLE
void Equip++_processor_allocation()
{
cpus_available=MAX_CPUS
cpus_requested=Sum_individual_request()
Reset_cpus_allocated()
current_=0; STABLE=1
while ((cpus_available>0) && (cpus_requested>0)){
   if (jobs[current_job].requested>alloc[current_job]){
     if (SpeedupCalculated(current_job,alloc[current_job]+1)){
       next_eff=jobs[current_job].Speedup[alloc[current_job]+1]/
        (alloc[current_job]+1)
       if (next_eff>=target_efficiency)
        one_more=1
       else{
        one_more=0; STABLE=0
        cpus_requested-=(jobs[current_job].requested-alloc[current_job])
       }
     }else  one_more=1
     if (one_more){
       alloc[current_job]++
       cpus_available--
       cpus_requested--
     }                                                  main modifications
   }
   current_job=(current_job+1)%active_jobs
}
}
int Equip++_New_appl()
{
if ((cpus_availables>0) && (Stable_allocation()) NewAppl=1
else NewAppl=0
return NewAppl
}
```

Figure 6.5: Equip++ algorithm

With this simple modification, we can ensure that the efficiency achieved by parallel applications reaches a minimum efficiency. Note that this algorithm also allocates a minimum of one processor to running applications because, by definition, efficiency(1) is 1, and the target efficiency will be always less than 1.0. In this Chapter we have used a target efficiency of 0.7.

The behavior of Equip++ is different to the behavior of Equipartition just in those cases where applications do not reach the target efficiency. In these cases, the Equip++ moves processors from applications that do not scale well to applications that scale well while they reach the target efficiency. Once distributed, if there are free processors, Equip++ will increase the multiprogramming level.

6.3.3 Equal_efficiency

Equal_efficiency is a processor scheduling policy proposed by Nguyen *et al.* in [76]. The goal of the Equal_efficiency [76] is to maximize the system efficiency. The idea is to allocate more processors to those applications that have better efficiency and less processors to applications with worse efficiency.

The Equal_efficiency assumes that all the applications have the same efficiency, then it allocates the same number of processors to all of them during a quantum (an equal allocation). In this quantum, applications measure their efficiency and inform the scheduler. Once informed about application's efficiency, the scheduler moves processors from applications with low efficiency to applications with high efficiency, and repeat the process.

$$\text{Efficiency}(p) = \frac{(1+\beta)}{(p+\beta)}$$

Figure 6.6: Extrapolated efficiency formulation

In order to avoid a great number of re-allocation, that will imply a great overhead, the Equal_efficiency extrapolates the efficiency curve, see Figure 6.6, from the most recently measured efficiency. This formulation was proposed by Dowdy in [25], and calculates the complete efficiency curve based on one measurement. It assumes that all the applications have a similar behavior but with different slope, see Figure 6.7.

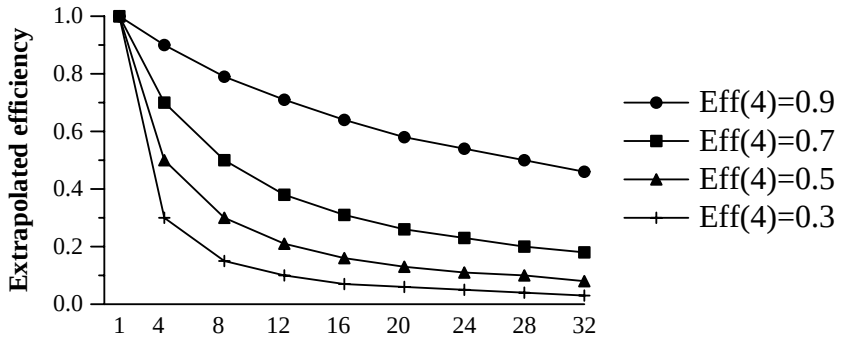


Figure 6.7: Extrapolated efficiency curves

Figure 6.7 shows the resulting efficiency curves calculated with the formulation presented in Figure 6.6 for different cases. We have calculated the curves assuming that the efficiency with four processors is known and that it is 0.9, 0.7, 0.5, and 0.3 respectively. The x axis shows processors, and the y axis shows the extrapolated efficiency values.

Looking at the graph shown in Figure 6.7, it seems that this formulation works quite well. However, we have found several problems when using it. The two main problems related with its use is that it does not accept efficiency values greater or equal to 1.0, and that it does not accept the efficiency of one processor as initial value to extrapolate the curve. For instance, if we introduce as initial value efficiency(4)=1.0, the efficiency values calculated are invalid floating points values (*nan's*). If we introduce as initial value the efficiency of one processor[1], the extrapolated curve calculates all the values equal to 0.0. Finally, if we introduce the initial value of (for instance) four processors set to 1.1, the extrapolated efficiency with 32 processors is 16.5.

Once extrapolated, the Equal_efficiency works in the following way: it initially assigns a single processor to each application, and then it assigns the remaining processors, one by one, to the application with the currently highest (extrapolated) efficiency. The Equal_efficiency has been implemented following the algorithm of Figure 6.8.

```
input: job_table(jobs),target_efficiency
output: table with number of processors per job (alloc)
void Equal_efficiency()
cpus_available=MAX_CPUS
cpus_requested=Sum_individual_request()
Allocate_one_processor_per_application()
cpus_availables=MAX_CPUS-active_jobs;
Extrapolate_efficiency_curves()
while ((cpus_available>0) && (cpus_requested>0) {
(1) current_job=Seach_appl_higher_eff() (*)
   cpus_allocated[current_job]++
   cpus_available--
   cpus_requested--
}
}
(*) Search the appl. that have the higher efficiency with (cpus_allocated[appl]+1)
processors,        among        those        applications        that        have
(cpus_allocated[appl]<cpus_requested[appl]).
```

Figure 6.8: Equal_efficiency algorithm

The *Extrapolate_efficiency_curves*() function has been implemented in such a way that only not calculated values are extrapolated, and that we have treated as special cases the commented previously. In those cases where the efficiency achieved by the application is super-linear, we have substituted this value by 0.999.

Note that the Equal_efficiency initially allocates a minimum of one processor per application. This is because it assumes that the efficiency of any application with one processor is 1.0. This is a common approach in processor scheduling policies that

1. In this case, the efficiency value does not matter

considers application performance since by definition the efficiency of a parallel application with one processor is one, and this is always an acceptable value. Figure 6.9 shows the complete Equal_efficiency scheme.

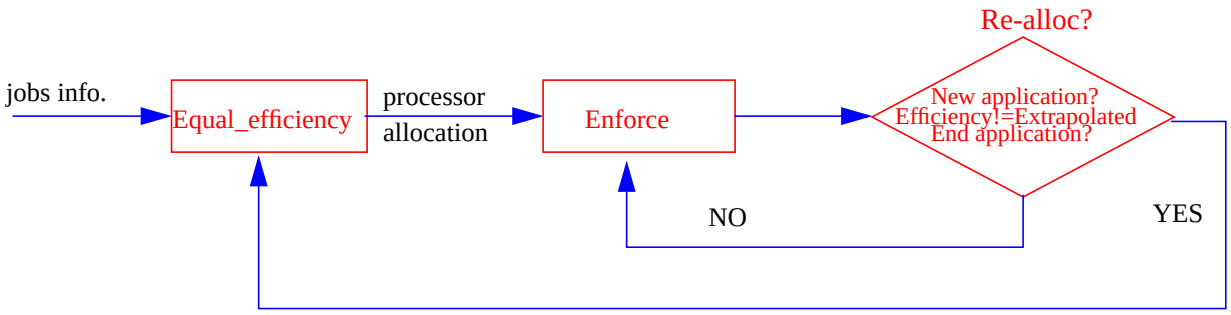


Figure 6.9: Equal_efficiency scheme

The Equal_efficiency has the problem that it considers that an equal efficiency is equal to a *good efficiency*, it does not impose a minimum efficiency. Applying PDML to the Equal_efficiency it becomes the Equal_eff++, ensuring the efficient use of processors.

6.3.4 Equal_eff++

If all the running applications achieve an acceptable efficiency, the Equal_eff++ will have the same behavior than the original policy. Otherwise, the algorithm will limit the processor allocation to those applications that do not reach the target efficiency.

Figure 6.10 shows the Equal_eff++ algorithm. Before allocating a processor to an application, (1) in Figure 6.8, the Equal_eff++ evaluates if the application efficiency reaches the target efficiency. In that case, the processor is allocated and the process goes on. Otherwise, no more processors will be allocated to any application because we know that this application has the higher (extrapolated or calculated) efficiency.

```
input: job_table(jobs),target_efficiency
output: table with number of processors per job (alloc), STABLE
void Equal_eff++_Processor_allocation()
{
cpus_available=MAX_CPUS
cpus_requested=Sum_individual_request()
Allocate_one_processor_per_application()                          main modifications
cpus_availables=MAX_CPUS-num_appls;
Extrapolate_efficicency_curves()
follow=1; STABLE=1
while ((cpus_available>0) && (cpus_requested>0) && (follow)){
  current_job=Seach_appl_higher_eff() (*)
  next_eff=jobs[current_job].Speedup[alloc[current_job]+1]/(alloc[current_job]+1)
  if (next_eff>=target_efficiency)
     one_more=1
  else{
    follow=0
    one_more=0; STABLE=0
  }
  if (one_more){
    alloc[current_job]++
    cpus_available--
    cpus_requested--
  }
}
}
int Equal_eff++_New_appl()
{
if ((cpus_available>0) && (Stable_allocation()) NewAppl=1
else NewAppl=0
return NewAppl
}

(*) Search the job that have the higher efficiency with (alloc[appl]+1) processors, among those
applications that have (alloc[appl]<cpus_requested[appl]).
```

Figure 6.10: Equal_eff++ algorithm

The *Equal_eff++_New_appl*() function implements the multiprogramming level policy. It returns true if a new application can be started. The *Stable_allocation*() function evaluates if all the running applications have informed about their efficiencies with the current distribution and all of them achieve the target efficiency. If *Stable_allocation*() returns true and there are free processors, a new application can be started.

6.4 Taxonomy

The taxonomy presented in Chapter 2 has been modified by the proposals made in this Thesis. Figure 6.11 shows the resulting taxonomy once included the new classification to differentiate between policies that do not impose a target efficiency and our policies, that impose a target efficiency.

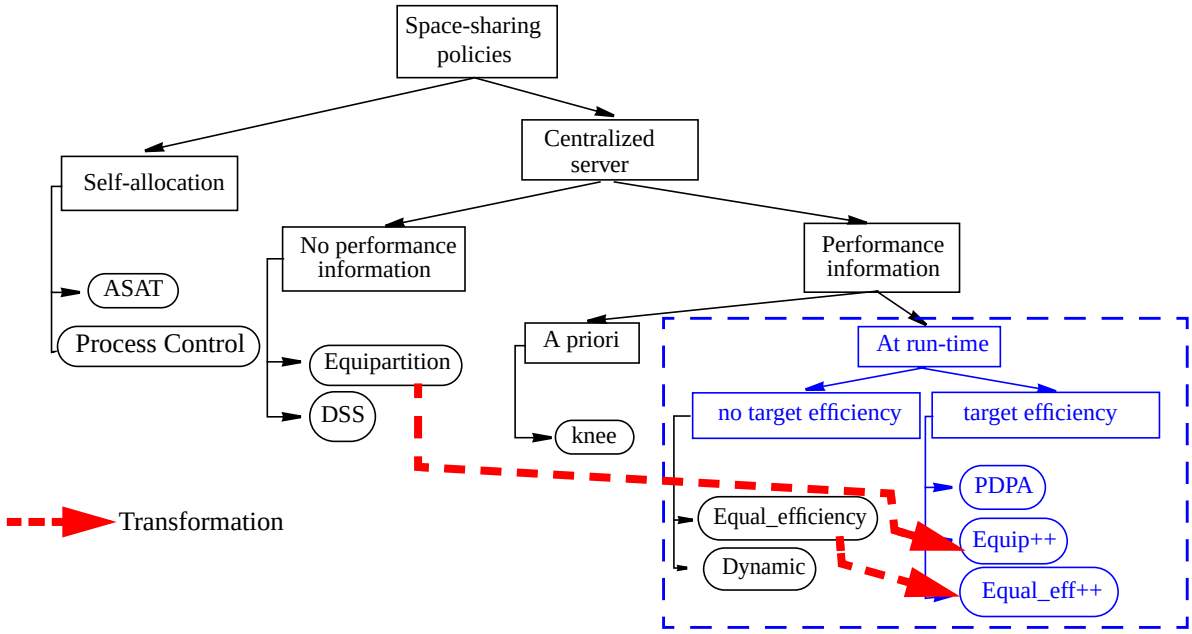


Figure 6.11: Processor scheduling taxonomy

Modifications introduced in the Equipartition includes the use of real performance information, the imposition of a target efficiency, and the coordination with the queueing system. Modification in the Equal_efficiency includes the imposition of a target efficiency and the coordination with the queueing system.

6.5 Evaluation

In this Chapter, we compare results achieved by Equipartition and Equal_efficiency with results achieved by Equip++ and Equal_eff++. As in the previous Chapter, we have used the five workloads presented in Chapter 3. Table 6.1 resumes their main characteristics.

Table 6.1: Workload characteristics

	swim, super-linear		bt, scalable		hydro2d, medium scalable		apsi, not scalable	
	req.	% of cpu	req.	% of cpu	req.	% of cpu	req.	% of cpu
w1	30	50%	30	50%	-	-	-	-
w2	30	50%	30	50%	-	-	-	-
w3	30	50%	2	50%	-	-	-	-
w4	30	25%	30	25%	30	25%	2	25%
w5			30	100%	-	-	-	-

Table 6.2 resumes characteristics of the different configurations evaluated in this Chapter.

Table 6.2: Configurations

Policy	Queueing system	Processor scheduler	Run-time library	Multiprog. Level
Equip	Launcher	CPUManager	NthLib	Fixed = 4
Equip++	Launcher	CPUManager	NthLib	Dynamic, default=4
Equal_eff	Launcher	CPUManager	NthLib	Fixed = 4
Equal_eff++	Launcher	CPUManager	NthLib	Dynamic, default=4

6.5.1 Workload 1

Figure 6.12 shows results for workload 1. Comparing the Equipartition and Equip++, we can observe in this workload that both processor scheduling policies achieve the same performance. This is because the same reasons presented in previous Chapter: the load generated, and the number of processors requested by swim's and bt's, generate that they receive a number of processors that reaches the target efficiency with a simple Equipartition. The conclusion that we can extract from this workload is that we can apply PDML without introducing significant overhead if the workload is directly well dimensioned. The overhead introduced by Equip++ respect Equipartition in this workload has been around 5%.

Comparing Equal_efficiency, line with circle marks, with Equal_eff++, line with box marks, we can see that Equal_eff++ has outperformed the Equal_efficiency in both the response time of swim's (44%) and the execution time of swim's (201%). On the other hand, the Equal_efficiency has outperformed the Equal_eff++ in the response time of bt's (22%) and the execution time of (bt's).

We have analyzed results for the two policies and we have found that the Equal_efficiency has allocated (in average) 14 processors to swim's and 22 processors to bt's. The Equal_eff++ has allocated 26 processors to swim's (in average), and 12 processors to bt's. Comparing these results we could conclude that the problem is that Equal_eff++ has decided a different allocation.

However, the real problem is intrinsic to the policy: it takes decisions only based on extrapolated values and it does not evaluate that these values correspond with real values. We have noted that, due to the dynamic multiprogramming level, the Equal_eff++ has more tendency than the Equal_efficiency to allocate a small number of processor to new applications, at least initially. As we have commented previously, the function used to extrapolate does not accept several values as input. However, there are other valid values that generate not realistic efficiency curves. For instance, if the measured efficiency of an application with 2 processors is 0.8, the function generates that the efficiency with 4 processors will be 0.57 (less than the target efficiency). However, if the input value is 0.9, such as in the case of the swim's (because they are super-linear in this range of processors), the extrapolated efficiency with 32 processors is 0.76 (greater than the target efficiency). This behavior, intrinsic to the original policy, implies that bt's are more affected to swim's. Since the Equal_efficiency does not limit the processor allocation based on the performance, bt's under it have more chances to receive processors than under Equal_eff++.

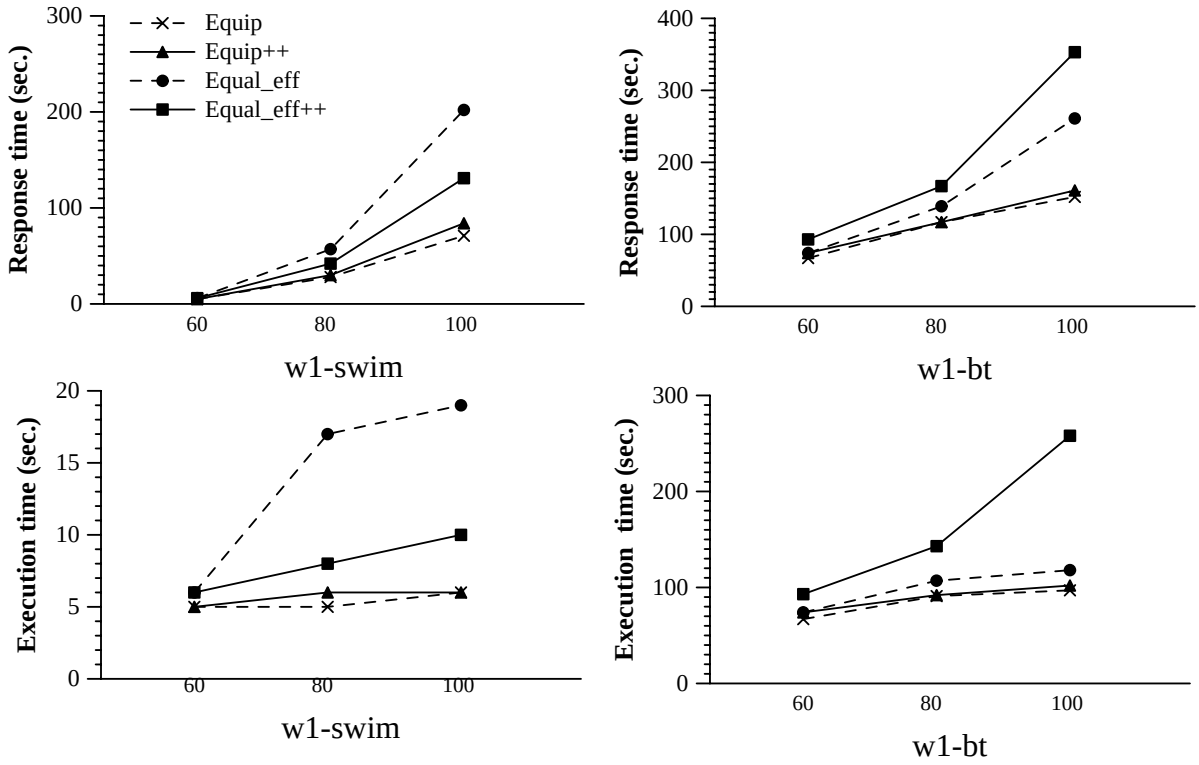


Figure 6.12: Results from workload 1, M.L.=4

6.5.2 Workload 2

Figure 6.13 shows results for workload 2. In this second workload, we can see how Equip++ also achieves a comparable performance to Equipartition. In fact, workloads under Equip++ have a similar behavior than under PDPA. Equip++ allocates more processors to bt´s than to hydro2d´s. However, this difference is not quite enough to result in a significant difference in the response time of bt´s and in the total execution time of the workload (see Section 6.5.6). This is mainly because of the difference in the execution time between bt´s and hydro2d´s.

The higher execution time shown by hydro2d´s is due to the combination of the overhead introduced by the SelfAnalyzer in this application and the reduction in the processor allocation decided by the Equip++. On the other hand, bt´s receive more processors because they scale better than hydro2d´s, resulting in a better execution time.

Comparing results achieved by the two policies, we can say that Equip++ outperforms the response time of bt's respect to Equipartition in a 4%, and the execution time by 14%. The behavior of hydro2d's, as we have commented, is quite different. Equip++ slowsdown by 9% the response time with respect to the Equipartition, and increases by 25% in the execution time. We have to note that results shown in Figure 6.13 corresponds with a multiprogramming level set to four processors.

These results are similar to those achieved in the previous Chapter by PDPA. As in the previous Chapter, the multiprogramming level used generates that the processor allocation decided by Equipartition is directly acceptable.

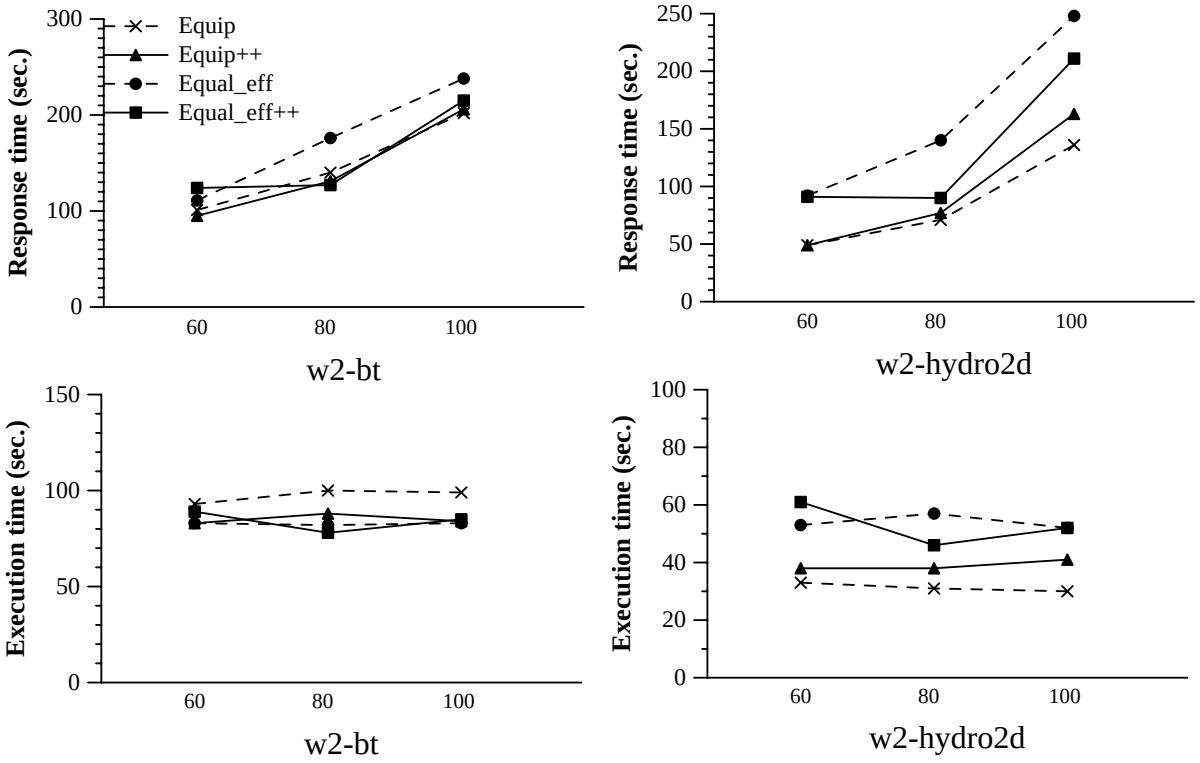


Figure 6.13: Results from workload 2, M.L.=4

Comparing results achieved by Equal_efficiency and Equal_eff++ in this workload, we can see that Equal_eff++ slightly outperforms Equal_efficiency. Equal_eff++ outperforms Equal_eff++ by 12% in the response time of bt's and 24% in the response time of hydro2d's. The execution time achieved by the two policies has been quite similar.

Equal_eff++ has limited the processor allocation and has increased the multiprogramming level. This workload is also affected by the effect commented in the previous workload related to the extrapolation function, then applications have received much less processors under Equal_eff++ than under Equal_efficiency. The mean allocation under Equal_eff++ is 12 processors to bt's and 8 processors to hydro2d's, and under Equal_efficiency is 30 processors to bt's and 10 processors to hydro2d's. However, note that the execution time has not been very affected by this reduction in the processor allocation.

The multiprogramming level has been increased up to 9 applications under Equal_eff++. This value is greater than the one decided by Equip++ because applications under Equal_eff++ receive less processors, and there are free processors more frequently.

Multiprogramming level set to two applications

Figure 6.14 shows results for workload 2 when executing with a multiprogramming level of two applications under Equipartition and Equip++. Comparing the response time of individual applications we can see how Equip++ clearly outperforms Equipartittion. This is because Equipartition is allocating to applications as many processors as they request. However, if we compare the execution time of applications under the two policies we can see that Equip++ does not introduce a significant overhead compared with Equipartition and the best utilization in the processor allocations results in an reduction around the 50% in the response time of applications.

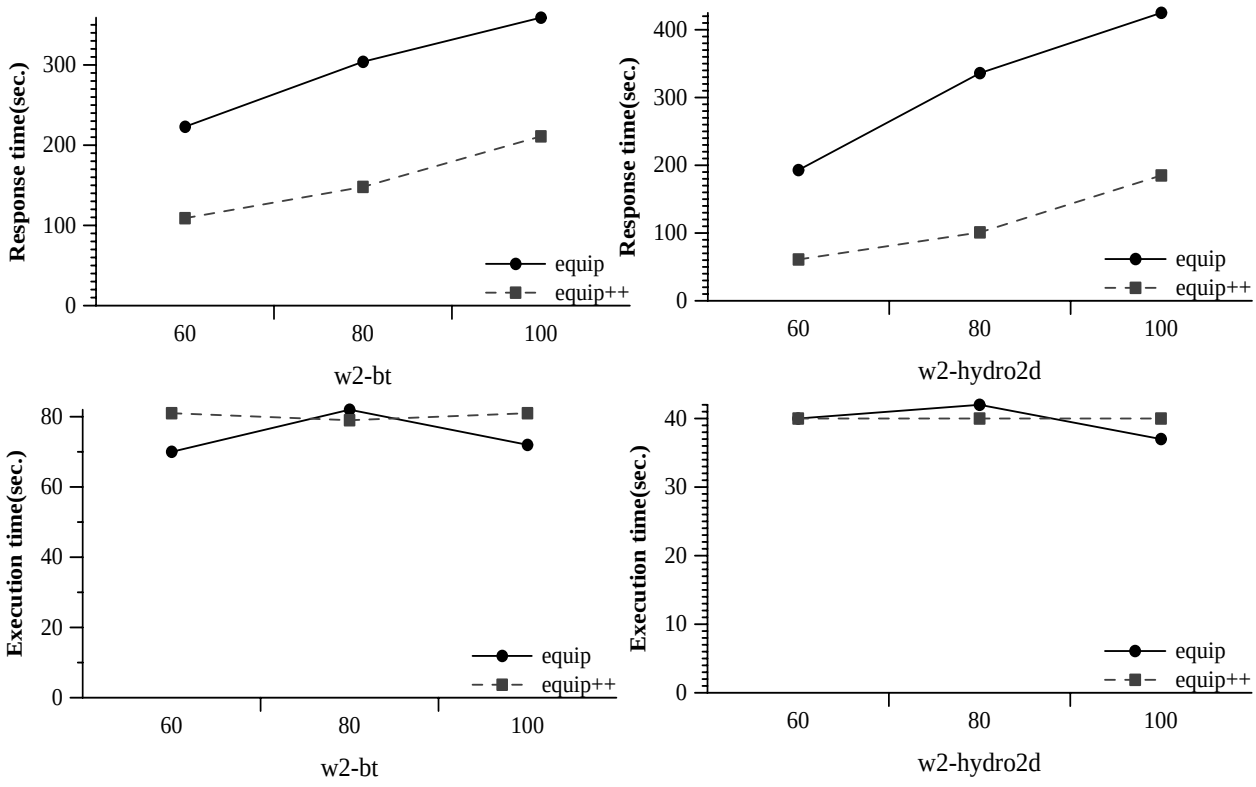


Figure 6.14: Workload 2, M.L.=2 Equipartition vs. Equip++

Figure 6.15 shows with different colors the application processor allocation decided by Equipartition and Equip++. Dark-blue colors means many processors and light-green colors means few processors. We have set the same time scale in both figures to compare them. We can see that Equipartition allocates more processors to applications than Equip++ and that all the applications receive the same number of processors. On the other hand, Equip++ allocates more processors to bt's than to hydro2d's because bt's scale better than hydro2d's. Moreover, Equip++ decides to increment the M.L.. This behavior improves both the response time of applications and the throughput of the system.

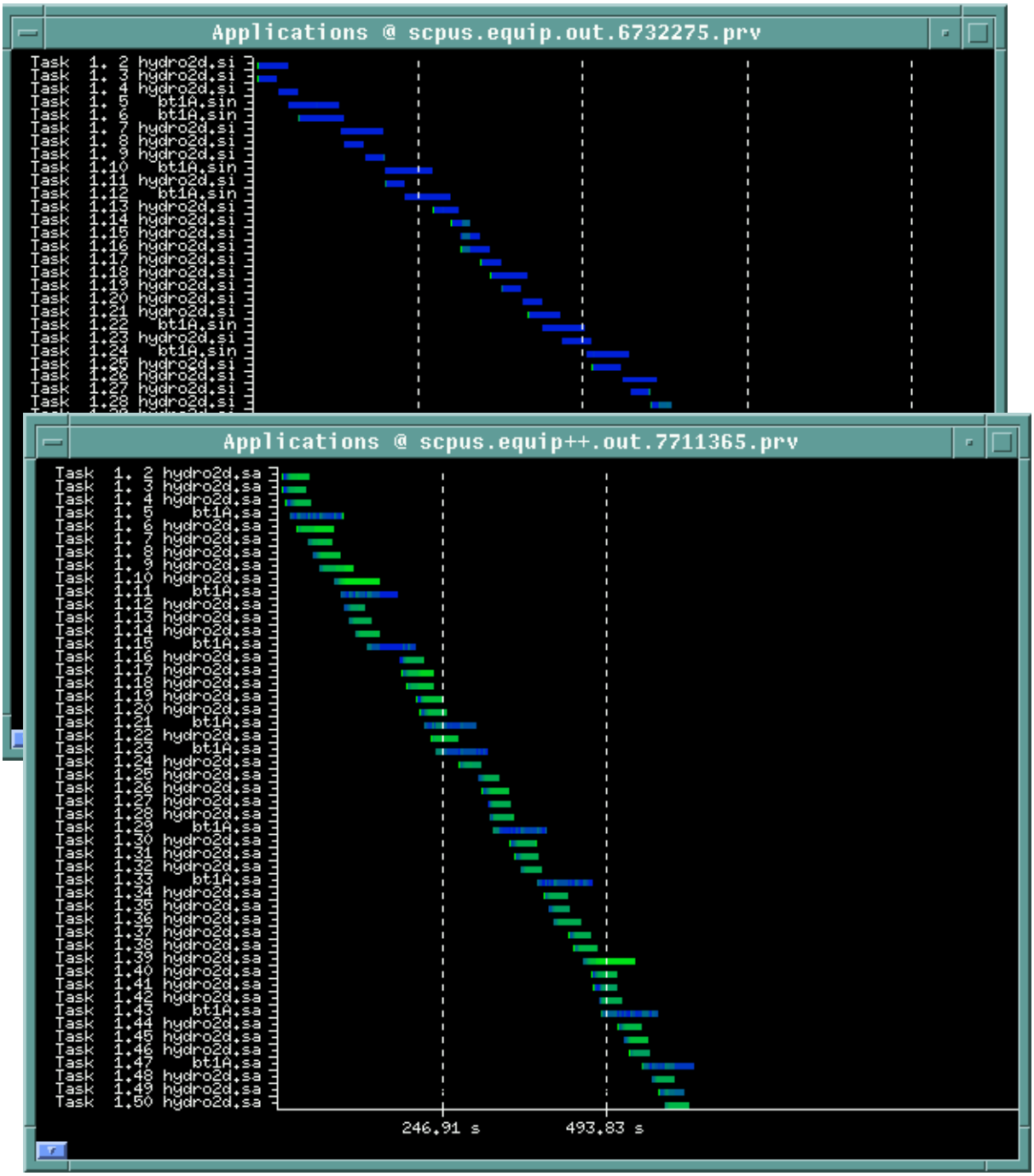


Figure 6.15: Processor allocation decided by Equipartition and Equip++, (M.L=2,load=100%)

Figure 6.16 shows the multiprogramming level decided by Equip++. We can see how Equip++ dynamically adjust the number of running applications to variations in the workload.

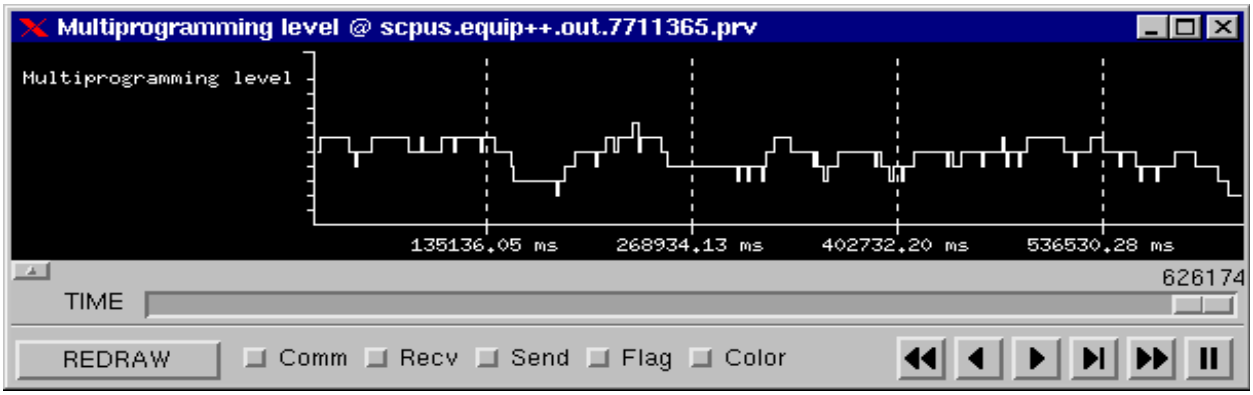


Figure 6.16: Dynamic multiprogramming level decided by Equip++, (M.L.=2, load=100%)

Figure 6.17 shows the effect of the multiprogramming level under Equipartition and under Equip++. In this case we have only executed the configuration with multiprogramming level set to four and two applications. We can observe that rather than Equipartition, Equip++ is not affected by the multiprogramming level decided by the system administrator. Equip++ consumes around the 50% less cpu time than Equipartition when the multiprogramming level is initially set to two applications. In the case of the multiprogramming level set to four applications the cpu time consumed is the same by the two policies.

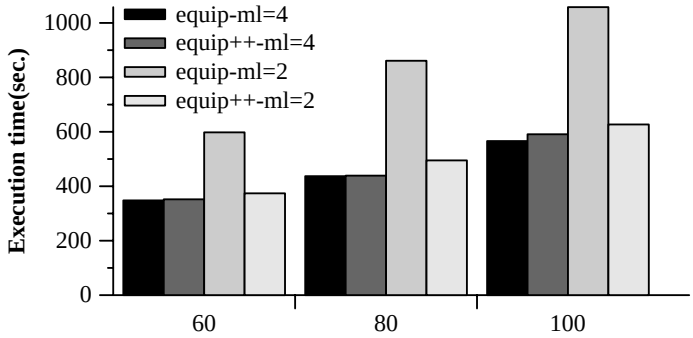


Figure 6.17: Workload execution time, Equipartition vs. Equip++

As in the previous Chapter, results show that the best choice for those applications that can dynamically adjust the multiprogramming level is to set the default multiprogramming level to a small value and let the policy to modify it.

6.5.3 Workload 3

Figure 6.18 shows results for workload 3. Results achieved by workload 3 are quite different from previous workloads. In this case, we can observe that Equip++ significantly improves results achieved by Equipartition. Equip++ has improved the response time of

Equipartition by 997% (in average) in the case of bt's, and by 616% (in average) in the case of apsi's. With respect to the execution time, Equip++ has introduced a slowdown in apsi's around 2% and 15% in the case of bt's.

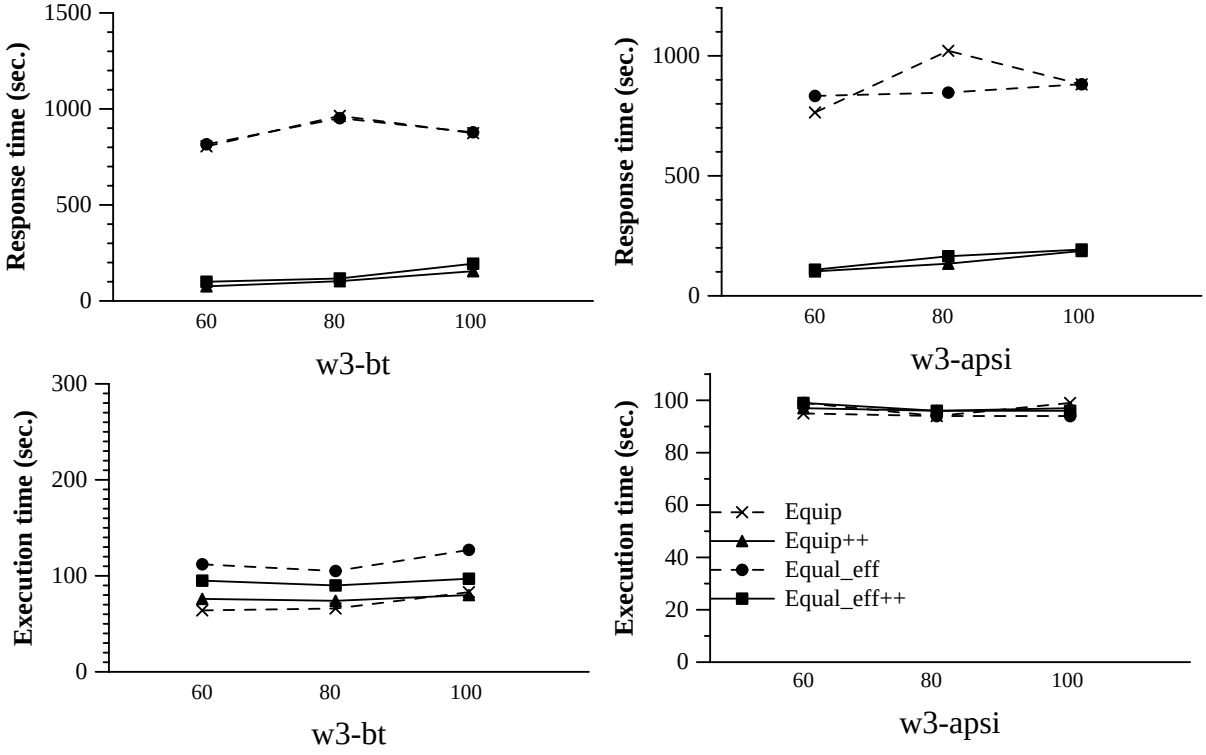


Figure 6.18: Results from workload 3

Workload 3 is the workload where the effect of using a dynamic multiprograming level is more significant. In this case, the modification in the processor allocation is not very high because bt's are scalable and apsi's request for only two processors.

Figure 6.19 shows the cpu utilization under Equipartition and Equip++. The dark blue color means cpu in use, and the light blue color means cpu idle. We have defined the same x scale to appreciate the difference in the execution time achieved by the two workloads. The Equipartition uses the 23% of the machine during the 1800 sec. that the workload takes. Moreover, bt's under Equipartition receive 30 cpus in average, and consume 2480 seconds per application. On the other hand, Equip++ uses the 77% of the machine and the workload consumes 596 seconds. Bt's under Equip++ receive 22 cpus in average, and consume 1786 seconds per application. We can see that processors are more efficiently used with Equip++ than with Equipartition.

In this workload, the main advantage is introduced by the fact of using a multiprogramming level policy that eliminates the idleness generated by the use of a fixed multiprogramming level. The multiprogramming level value that can generate

good results in some workloads, such as in the workload 1 and 2, can generate very bad results in other workloads, such as in this case. PDML introduces robustness to the Equipartition.

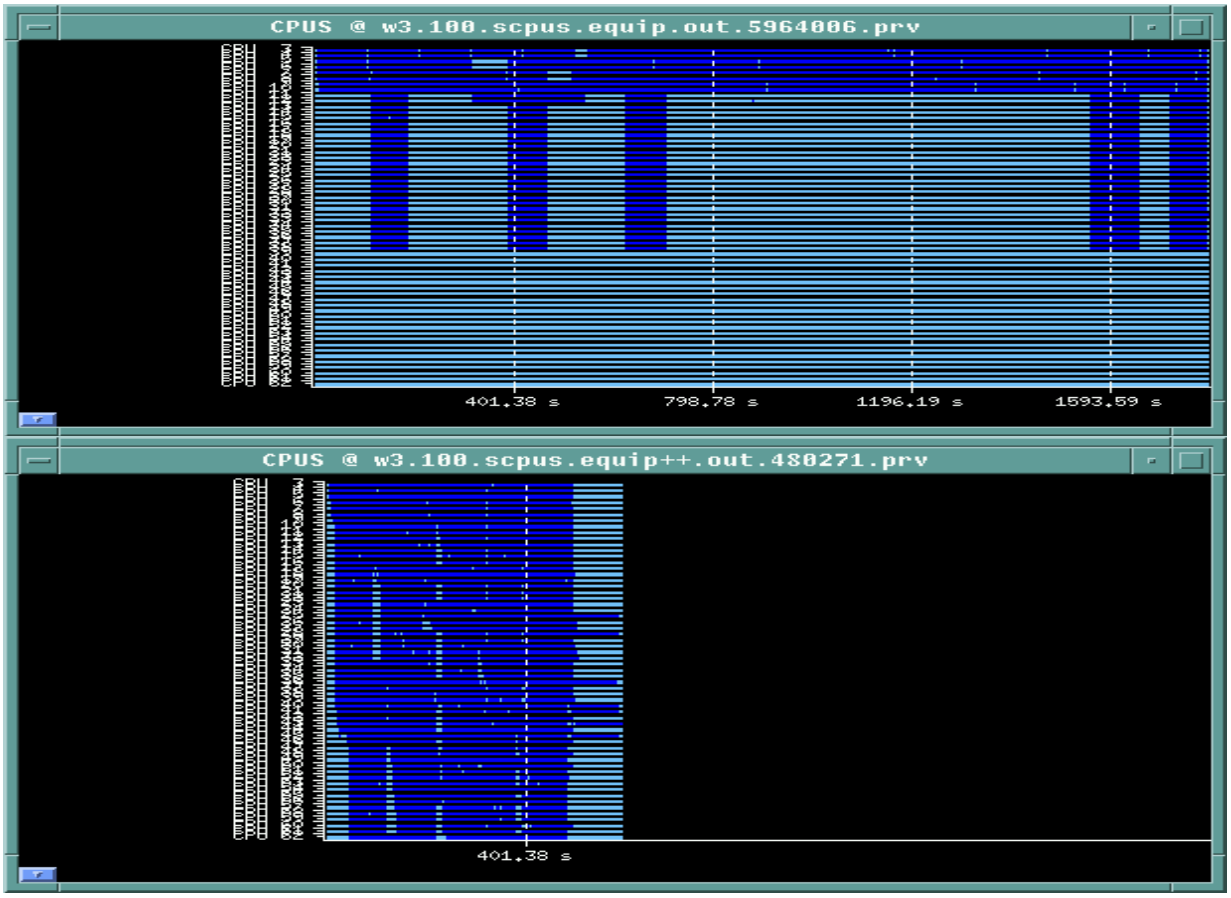


Figure 6.19: Cpu utilization in the case of Equipartition and Equip++, load=100%

In this workload, The Equal_eff++ has decided a mean allocation of 11 processors to bt's and a multiprogramming level of up to 37 applications. This is because of the same reason of previous workloads, the few number of processors allocated to bt's. However, even the Equal_eff++ has allocated less processors to bt's than Equal_efficiency, the execution time of bt's under Equal_eff++ has been better. Equal_eff++ has outperformed Equal_efficiency by 693% in the response time of bt's and a 21% in the execution time of bt's. This is because when the bt is executed with a lot of processors is more affected by the concurrent execution of other applications.

We have executed the same workload but without tuning the request of apsi's, only Equipartition and Equip++. Executing the not tuned version, we achieve the results shown in Table 6.3. We can see that in this workload, where the request of applications has not been tuned, Equip++ corrects the bad behavior shown by Equipartition.

In the total execution time of the workload, Equip++ outperforms Equipartition by 461%. We have also measured the percentage of cpu received by applications under the two policies. In the case of Equipartition, all the applications have received the same amount of processors, around 15 processors. In the case of Equip++, bt's have received around 24 processors and apsi's around 2 processors. Moreover, the maximum value of the multiprogramming level has been set to 24 applications.

This demonstrates that modifications introduced by PDML are able to solve incorrect allocations generated by policies that do not consider the application performance and to dynamically adjust the multiprogramming level to improve the system performance.

Table 6.3: Results from w3, apsi's requesting for 30 processors (not tuned) load=60%

	Bt		Apsi		Workload
	Resp. time	Exec. time	Resp. time	Exec. time	Exec. time
Equip	949 sec.	102 sec.	890 sec.	107 sec.	33 min. 13 sec.
Equip++	83sec.	75sec.	107sec.	97sec.	7min. 12 sec.
Equip++ speedup	1143%	36%	831%	10%	461%

6.5.4 Workload 4

Figure 6.20 shows results for workload 4. If we compare the execution time of applications executed under Equip++ and under Equipartition, we can see that in some cases they are better under Equipartition, a 20% in the case of swim's and hydro2d's. However, if we compare the response time of applications under the two policies, we can see the significant benefits provided by PDML: a 2500% in the case of swim's, a 600% in the case of bt's, a 800% in the case of hydro2d's, and a 400% in the case of apsi's. The number of processors allocated by Equip++, in the case of load=60%, has been: 18 processors to swim's, 23 processors to bt's, 12 processors to hydro2d's, and 1 processor to apsi's. The multiprogramming level has been increased up to 12 jobs in the same configuration.

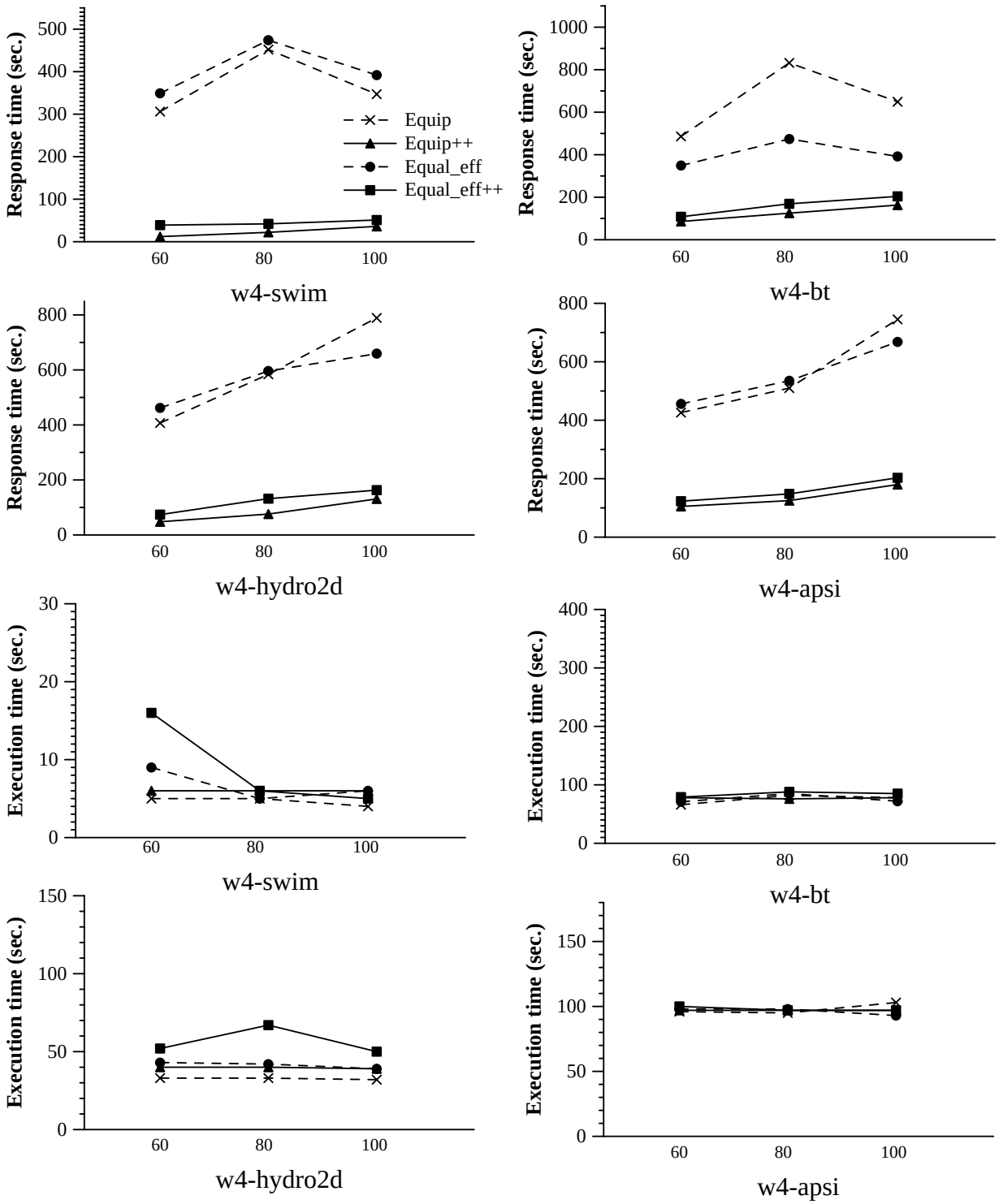


Figure 6.20: Results from workload 4

Comparing the Equal_eff++ with the Equal_efficiency, the Equal_eff++ has significantly improved the response time of the Equal_efficiency. Reasons are the same than in the case of the Equip++, the reduction in the processor allocation combined with the multiprogramming level policy. Comparing the execution times of applications, swim's and hydro2d's has been slightly increased. However, bt's and apsi's have achieved similar execution times under Equal_eff++ than under Equal_efficiency.

We have executed the same workload without tuning the number of processors requested by applications. Table 6.4 shows results achieved in this case. The last row, *speedup*, shows the Equip++ speedup with respect to Equipartition. Analyzing the execution time, Equipartition outperforms Equip++ in the execution time of hydro2d's by 18%, but Equip++ outperforms Equipartition in the execution time of bt's by 24% and apsi's by 7%.

Analyzing the response time achieved, the differences between Equipartition and Equip++ are more significant. Equip++ outperforms Equipartition by 66% in the case of apsi's, by 2830% in the case of swim's. In the case of the total execution time of the workload, Equip++ has improved the system performance by 283%. If we measure the number of processors allocated by each policy, we found that Equip++ has allocated in average 18 processors to swim's, 24 processors to bt's, 12 processors to hydro2d's, and 1 processor to apsi's. The maximum multiprogramming level has been 12 jobs.

As we can see, results achieved by Equip++ either with or without tuning the request of applications are quite the same (6min 49sec. without tuning). This is a very interesting conclusion because it shows that we can use Equip++ rather than Equipartition and reach the same performance independently of the way users submit their applications, only depending on their real characteristics and the load of the system.

Table 6.4: Results from workload 4, without tuning, load=60%

	swim, req=30		bt, req=30		hydro2d, req=30		apsi, req=30		Total
	exec.time	resp. time	exec.time	resp. time	exec.time	resp. time	exec.time	resp. time	exec.time
Equip	6sec.	368sec.	101sec.	568sec.	32sec.	453sec.	104sec.	773sec.	20min. 6sec.
Equip++	6sec.	13sec.	81sec.	92sec.	38sec.	57sec.	97sec.	116sec.	7min.6sec.
speedup	0%	2830%	24%	617%	-18%	794%	7%	66%	283%

6.5.5 Workload 5

Figure 6.21 shows results of workload 5. In this workload, Equipartition improves the execution time of Equip++ by 10% both in the response time and the execution time of bt's. In this workload, the number of processors allocated to bt's by the Equip++ is the same than the Equipartition. In addition, the multiprogramming level has been maintained. However, the measurement process introduces some overhead in running applications, resulting in this small slowdown.

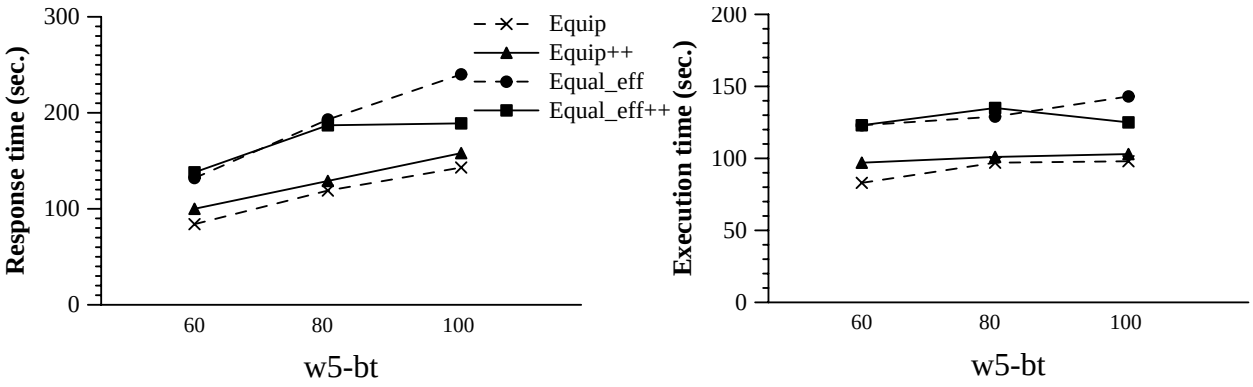


Figure 6.21: Results from workload 5

As in the previous workloads, Equal_eff++ has slightly improved the Equal_efficiency in both the execution time and the response time. Equal_eff++ outperforms Equal_efficiency in 4% in the execution time of applications and 8% in the response time.

The mean processor allocation decided by the Equal_eff++ has been 15 processors to bt's. Figure 6.22 shows the processor allocation decided by Equal_eff++ of some of the bt's under Equal_eff++ in the case of load=100%. We can see the re-allocations decided by Equal_eff++ that are generated by the use of the extrapolation function. As we have commented in previous workloads, the criteria used by the Equal_efficiency, and also by the Equal_eff++, to allocate processors to applications with the higher efficiency, one by one, can generate that small changes in the measurement of one application imply global re-allocations.

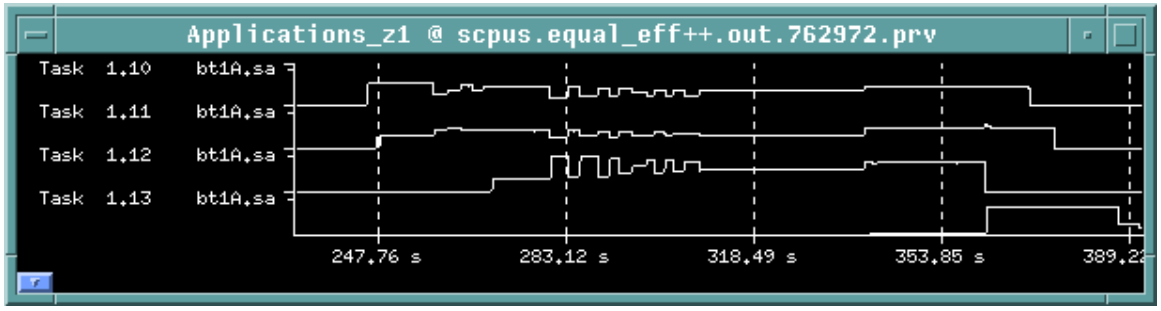


Figure 6.22: Re-allocations decided by Equal_eff++ because of the extrapolation function

6.5.6 Workload execution times

Figure 6.23 shows the execution time of the five workloads evaluated in this Thesis. We can see how in the case of Equip++, it reaches the same or better performance than with the original Equipartition. It is important to note that it is normal that in those configurations where the resulting allocation from Equipartition (due to the load, the

request, and the policy) is directly efficient with an Equipartition, PDML has "*nothing to improve*". This is the case of workloads 1, 2, and 5 (with M.L=4). In those cases, Equip++ has consumed the same time than Equipartition. However, in workload 3 and 4, Equip++ significantly outperforms the original Equipartition policy.

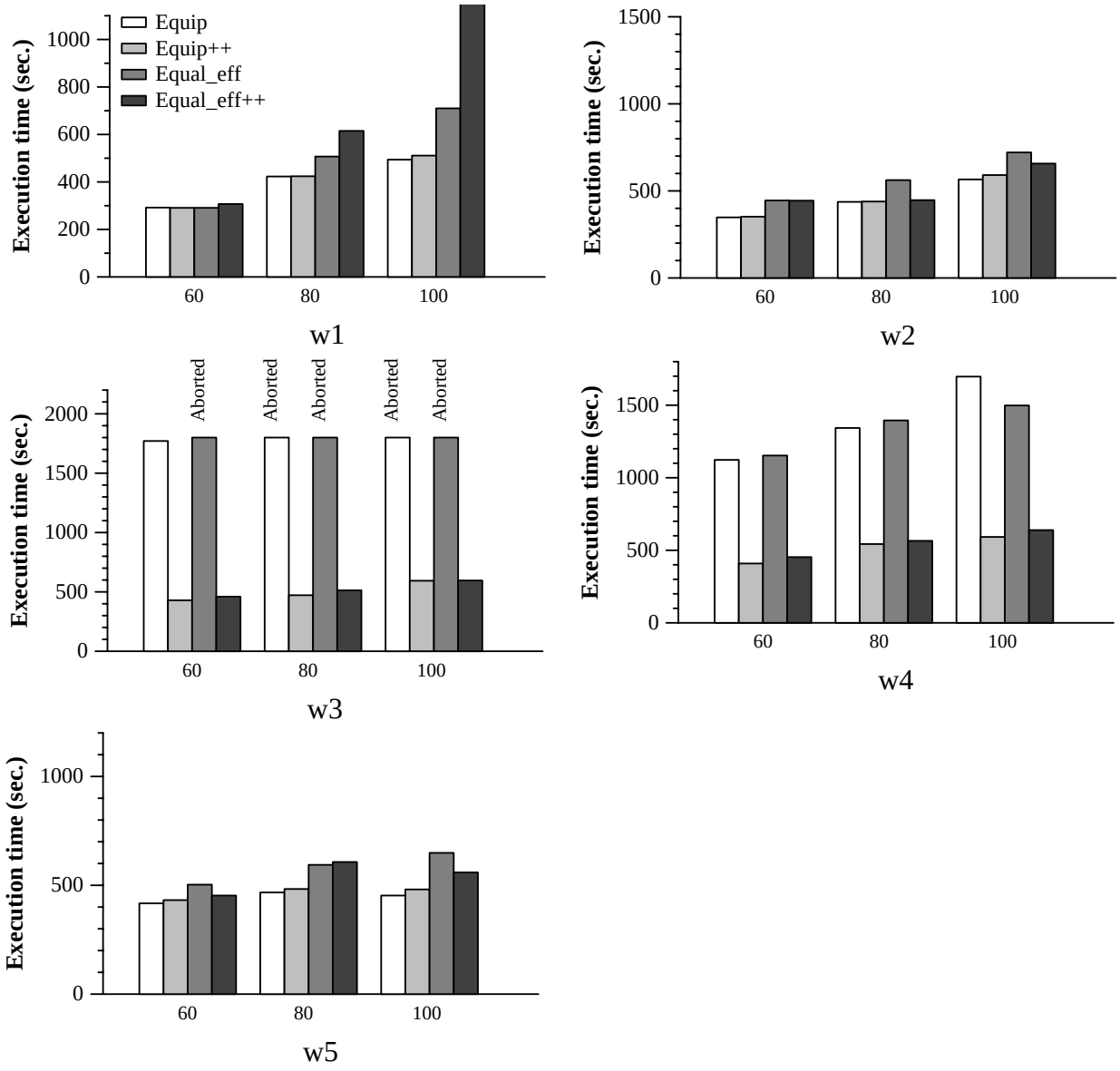


Figure 6.23: Execution time of the complete workloads

Comparing the Equal_efficiency and the Equal_eff++ we can see that in most of the workloads Equal_eff++ has improved the Equal_efficiency. Workload 1 has reached very different results than the rest of workloads. This is because, in the case of load=100%, the Equal_eff++ has allocated only one processor to one bt, then the execution time of the workload has been significantly increased. We do not have modified the values generated

by the extrapolation function in the case of initial value equal to one processor because we want to remark the negative influence that this kind of formulations can generate if they are not explicitly considered. Authors of the Equal_efficiency do not have the problem to work with super-linear applications because, even they exist, their mechanism does not consider them. Moreover, they do not comment any additional problem and then do not provide any solution.

This effect is more usual under the Equal_eff++ than under the Equal_efficiency because the multiprogramming level is higher under Equal_eff++ than under the Equal_efficiency. Then, the probability to receive less processors is higher with Equal_eff++.

6.6 Summary

In this Chapter, we have presented Performance-Driven Multiprogramming Level (PDML). PDML is a methodology that transforms processor scheduling policies to include feedback based on performance information with the aim of avoiding the inefficient use of processors. PDML modifies processor scheduling policies by periodically evaluating the performance achieved by running applications. If applications reach a given target efficiency, their allocation will be maintained, otherwise it will be adjusted.

PDML also includes a multiprogramming level policy based on the system stability. If all the applications are stable, that is, all of them reach the target efficiency, and there are free processors, the multiprogramming level will be increased. PDML has been applied to the Equipartition and the Equal_efficiency, resulting in the Equip++ and the Equal_eff++.

Results show that the performance of a policy not only depends on the number of processors allocated to each application, but also in the stability of the system, and in the number of reallocations that applications suffer. In the case of Equip++, it reaches the same performance that Equipartition in those workloads that directly perform well in Equipartition, and outperforms Equipartition in those workloads in which Equipartition does not decide an efficient processor allocation.

In the case of Equal_eff++, we can conclude that the main goal of PDML is achieved: to ensure the efficient use of processors and to coordinate the different scheduling levels. However, in some cases the behavior of the original policy generates that some pathological situations are more frequent under Equal_eff++ than under Equal_efficiency, resulting in some incorrect allocations. This situation mainly appears when the performance of parallel applications are initially measured with a small number of processors. In some of the workloads, the benefit is mainly generated by the stability that PDML introduces in the Equal_efficiency. This stability does not have the same impact in the case of the Equipartition because it is much more stable than the Equal_efficiency. In some other workloads the benefit is also due to the multiprogramming level policy. We could conclude that the use of formulations to extrapolate values is possible, but it must be done *with care* and always being conscious that they are not real values.

A difference that we have found between PDPA and these policies is that PDPA has more control about the behavior of the applications and that PDPA is more conscious that the behavior of the application can change. PDPA gives more chances to running applications to receive more processors. This behavior, in some workloads could introduce some overhead, but in other cases could solve incorrect allocations due a punctual bad measurement or medium scalability in a certain range of processors.

CHAPTER 7

Contributions to Gang Scheduling

In this Chapter, we present two techniques to improve Gang Scheduling policies by adopting the ideas of this Thesis. The first one, Performance-Driven Gang Scheduling, is the result of applying PDML to a traditional gang scheduling policy. The second one is the Compress&Join algorithm, a new re-packing algorithm that exploits the job malleability of OpenMP applications and the use of real performance information.

Performance-Driven Gang Scheduling mainly attacks the problem of the inefficient use of resources by parallel applications, that indirectly results in an excessive number of time slots.

Compress&Join is totally oriented to reduce the number of time slots by reducing the number of processors used by application proportionally to their performance.

These two techniques are orthogonal among them and can be used with any previously proposed scheme.

7.1 Introduction

Gang Scheduling [33][34] is a combination of time-sharing and space-sharing approaches. Characteristics of Gang Scheduling are: (1) threads are grouped into gangs, (2) threads in a gang are executed simultaneously, and (3) time sharing is used among gangs.

Gang scheduling appeared as the solution to the problems of job scheduling policies in those systems where the processor scheduling was a simple dispatch. In this kind of systems, the main problem seems to be the fragmentation, then reasons to use gang scheduling were presented as responsiveness and efficient use of resources. However, gang scheduling still has the problem of the fragmentation [115][33], and the excessive number of time slots [115]. These two problems result in an inefficient use of resources (processors and memory).

In this Chapter, we present two new approaches to improve gang scheduling by exploiting the ideas presented in this Thesis: use of real performance information, imposing a target efficiency, and coordination with the queueing system.

Our first contribution consists of applying PDML to the traditional gang scheduling scheme. This scheme has a scheduling phase that is a simple dispatch. We propose to self-evaluate the allocation decided by this dispatch, to impose a target efficiency, and to coordinate with the queueing system. We have called the resulting policy *Performance-Driven Gang Scheduling*, PDGS.

The main goal of PDGS is to ensure the efficient use of resource. However, one of the main problems of gang scheduling, the excessive number of time slots, is still a problem. Our second approach to improve gang scheduling consist of a new re-packing algorithm, *Compress&Join*, that combines two characteristics of our execution environment (the job malleability and the use of real performance information). The Compress&Join algorithm adjusts the processor allocation of applications based on their performance to fit the same number of applications in less time slots.

Results show that PDGS and the Compress&Join algorithm outperform the gang scheduling approach used as baseline. We will see that the ideas proposed in this Thesis of (1) measuring the performance of applications at run-time and (2) adjust the processor allocation based on this information, are also valid for gang scheduling policies. As in previous Chapters, results show that with our proposals, the execution time of applications is slightly increased because applications usually receive less processors than with other approaches. However, we will see that by adjusting the allocation, the system is efficiently used and that benefits both the system and the individual applications.

The next of this Chapter is organized as follows: Section 7.2 describes modifications introduced in the CPUManager to implement gang scheduling policies and the particular implementation we have done. Section 7.3 presents Performance-Driven Gang

Scheduling. Section 7.4 presents the Compress&Join algorithm. Section 7.5 evaluates PDGS and the Compress&Join algorithm compared with the gang scheduling baseline implemented. Finally, Section 7.6 summarizes this Chapter.

7.2 Gang Scheduling

In this Section, we present characteristics of gang scheduling policies and the particular implementation that we have used as baseline in this Thesis.

Figure 7.1 shows the behavior of generic gang scheduling policies. The different gangs are grouped in time slots following some re-packing algorithm. The total number of processors requested by gangs in a time slot must be less or equal than the total number of processors of the machine. Periodically, at each quantum expiration, the scheduler selects a new time slot to execute all of its gangs. If the workload has changed during the execution of the last quantum, the re-packing algorithm will be re-applied. In any case, the new slot selected is scheduled.

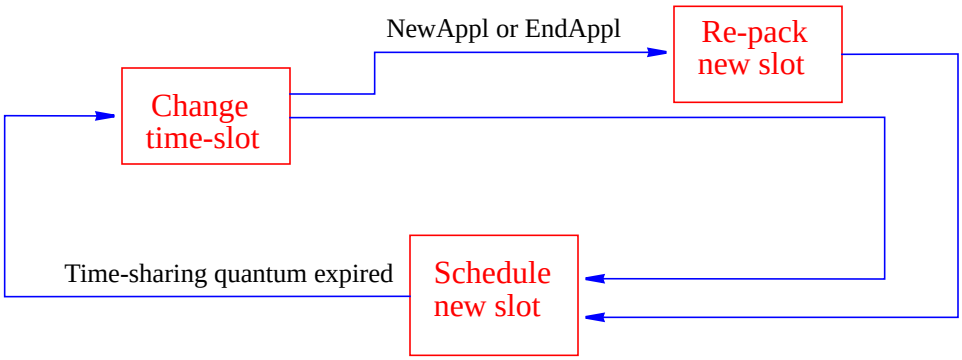


Figure 7.1: Gang scheduling generic scheme

7.2.1 Gang scheduling implementation

There are many versions of gang scheduling. The simplest version of gang scheduling always allocates a gang in the same set of processors. However, most flexible versions have been proposed [81]. One of them is *migratable* [32] preemptions, where jobs that are preempted in a set of processors can be resumed in a different set of processors. A more complex version is *malleable* [32] gang scheduling, where jobs can be resumed in a set of processors of different size, however, malleable preemptions has not been used till now because it is difficult to implement. For this reason, usually the scheduling of the slot is a simple dispatch, where applications receive as many processors as they request. In this Thesis, we will show that malleable gang scheduling is feasible and that to exploit this feature, combined with performance information, improves the system performance. Characteristics of our execution environment (OpenMP applications and space-sharing architecture) make malleable gang scheduling a valid approach.

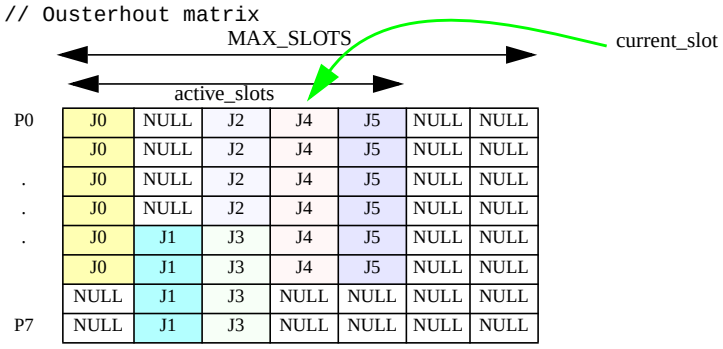
The main difference among gang scheduling policies is the job re-packing algorithm. Feitelson argues in [33] that the best option is to apply a buddy algorithm or to use migration to re-map the jobs (based on a first-fit algorithm). In [88] Setia shows that gang scheduling policies that support migration offer significant performance gains over policies that do not support it.

In the next Section, we present the characteristics of the gang scheduling implementation that we have used as baseline. We have tried to select the best configuration presented in the literature to compare with our proposals. For this reason, we have implemented migratable preemptions [88], that is, threads in a gang can be preempted in a set of processors and resumed in another set based on a first-fit algorithm.

Job organization: Ousterhout matrix

To implement Gang scheduling we have used the mechanism proposed by Ousterhout in [78], the Ousterhout matrix. The Ousterhout matrix defines a two dimensional matrix where one dimension represents processors and the other is time.

Figure 7.2 shows the main data used by the CPUManager to implement the gang scheduling mechanism. The table is an example of a possible configuration of the Ousterhout matrix in a system with eight processors. This table has as many rows as processors and a maximum of columns (MAX_SLOTS). Each column is a time slot, composed by a list of one or several gangs. In the example, slot 0 has one gang (Job_0), slot 1 has one gang (Job_1), slot 2 has two gangs (Job_2,Job_3), and so on. Time-sharing is performed between slots, and the sum of processors allocated to gangs in a slot must be less or equal than the number of processors in the machine. In this particular example there are five *active_slots* (with applications associated to them), and the *current_slot* is the slot 3. Applications in the current slot are the only ones that are running, the rest of applications are stopped.



P0	J0	NULL	J2	J4	J5	NULL	NULL
	J0	NULL	J2	J4	J5	NULL	NULL
.	J0	NULL	J2	J4	J5	NULL	NULL
.	J0	NULL	J2	J4	J5	NULL	NULL
.	J0	J1	J3	J4	J5	NULL	NULL
	J0	J1	J3	J4	J5	NULL	NULL
	NULL	J1	J3	NULL	NULL	NULL	NULL
P7	NULL	J1	J3	NULL	NULL	NULL	NULL

Figure 7.2: CPUManager data structures

In the CPUManager implementation we have introduced a small modification with respect to the traditional Ousterhout matrix. In our execution environment, we have the problem that applications start their execution requesting for one processor, and they request for *P* processors when they open their first parallel region.

We have treated this case in a similar way to have two applications, a small sequential application and a new parallel one. The problem is that it is possible that when the application spawns parallelism, it does not fit in the slot. For this reason, we have defined two limits in the Ousterhout matrix. The first one, set to 4 slots, determines the number of slots to which the system will perform the time-sharing. The second one, set to 50 slots, is similar to a buffer of applications that have been started but that do not fit in any of the first four slots when they spawn parallelism. These applications are not scheduled until any of the applications in the first slots finish their execution.

Figure 7.3 corresponds with the execution of the Ousterhout matrix presented in Figure 7.2. In the example, the matrix has five time slots, that means that each application will be executed once every five slots. The system will execute the following sequence of applications: (J0), (J1), (J2 / J3), (J4), (J5), (J0), (J1), (J2 / J3), (J4), (J5), etc. The example presented in Figure 7.3 does not include the job migration used in this Thesis because we want to present the gang scheduling behavior in a progressive way. The execution including job migration is shown in Figure 7.4, and the job migration algorithm is presented in the next Section.

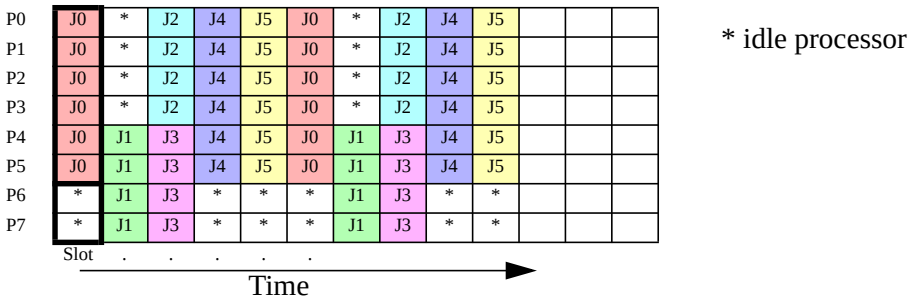


Figure 7.3: Gang scheduling behavior

Job re-packing algorithm: first-fit + migrations

The job re-packing policy implemented is a first-fit algorithm [33], (combined with job migrations) implemented in the following way:

- When a new job arrives to the system, it is placed in the first slot with a sufficient number of idle processors.
- A job completion does not imply a job re-packing.

At each time-sharing quantum, the scheduler performs the following steps:

- It stops the running gangs (this implies suspend all the threads)

- It advances the *current_slot* pointer and selects the next slot.
- If there are free processors in *current_slot*, and some application has finished in the last quantum, the job migration algorithm is applied. The job migration mechanism tries to move jobs from low loaded slots to high loaded slots. The load of the slot is computed as used processors divided by total number of processors.

At this point, we have selected a slot that can be dispatched by the scheduler. However, to improve the system utilization, we have introduced a mechanism to fill the remaining holes in the slot. To do that, we have introduced a phase of job replication. We generate a temporal slot, initialized with applications in *current_slot*, and we look for applications that can fit in the slot. The aim is to allow the execution of applications in more than one slot, but maintaining it physically allocated to only one slot in the Oustherout matrix. The Dispatch phase will receive this temporal slot to schedule.

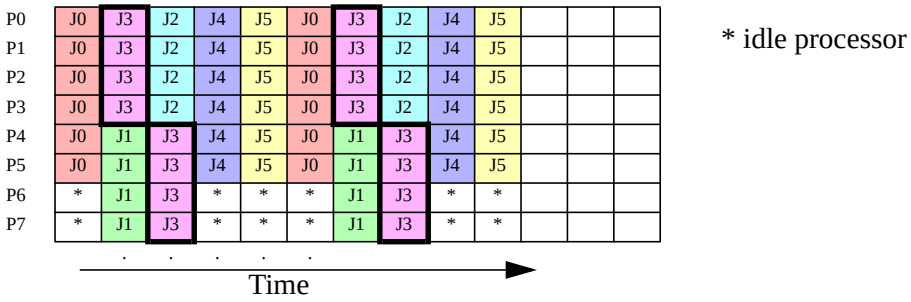


Figure 7.4: Migratable gang scheduling

Figure 7.4 shows the behavior of execution of the Oustherout matrix shown in Figure 7.2 with migratable preemptions. We will allow the execution of J3 in two time slots, using the processors more efficiently. Job 3 is only allocated to slot 1 but executed in slot 1 and 2. J3 is not migrated to J1 because the load of slot 1 (50%) is less than load of slot 2 (100%). If the quantum of the time slot is well dimensioned, the system performance can be quite acceptable.

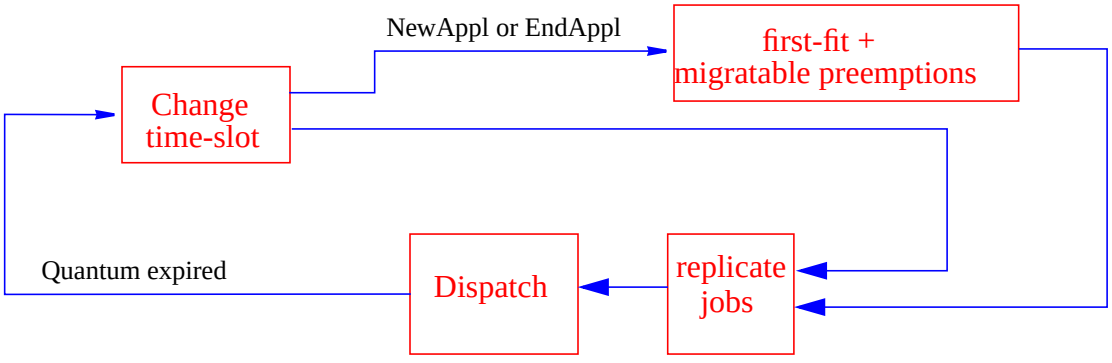


Figure 7.5: Gang scheduling baseline scheme

Figure 7.5 shows the gang scheduling scheme that represents our gang scheduling baseline: first-fit algorithm+migratable preemptions as job re-packing algorithm, and a simple dispatch as scheduling.

7.3 Performance-Driven Gang Scheduling

In this Section, we present *Performance-Driven Gang Scheduling* policy, PDGS. PDGS is the result of applying PDML to a traditional gang scheduling. Figure 7.6 shows the PDGS scheme. In PDGS, scheduling decisions are self-evaluated and corrected based on the performance achieved by running applications compared with the target efficiency. As in dynamic space-sharing policies, PDML introduces a space-sharing quantum to periodically evaluate the application performance. At each space-sharing quantum, the scheduler is activated and it evaluates the achieved performance. The processor allocation of those applications that do not reach the target efficiency is reduced as presented in Chapter 6. In fact, we have applied a processor allocation policy equal to the implemented in PDPA, a bit more complicated than to the one implemented in PDML.

PDGS also modifies the conditions to apply the re-packing algorithm. In this case, the re-packing algorithm must be also applied when the allocation has been adjusted because we consider changes in the processor allocation as a new application arrival.

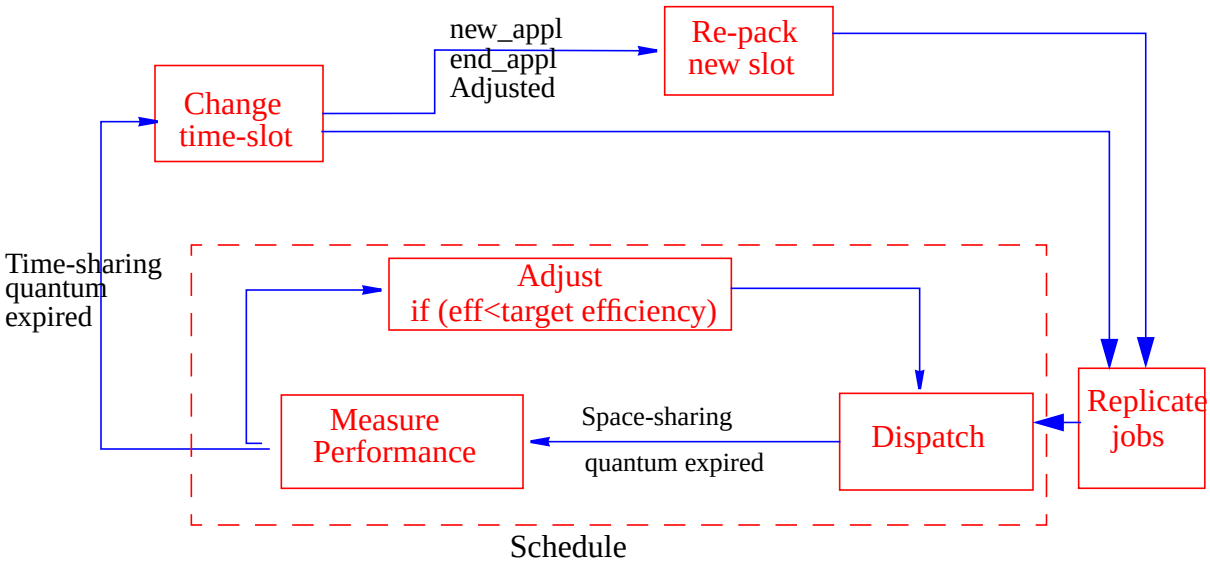


Figure 7.6: Performance-Driven Gang Scheduling scheme

To simplify the implementation, the time-sharing quantum is a multiple of the space-sharing quantum. In the current implementation we have set the space-sharing quantum to 100 ms, as in the previous chapters, and the time-sharing quantum in 6 sec. The space-sharing quantum set to 100 ms. does not imply a re-allocation at each 100 ms. It implies that, at each 100 ms., the processor allocation policy will be applied and, if the performance of running applications have been calculated, the algorithm will evaluate their performance and will adjust their processor allocation to reach the target efficiency (if needed).

Figure 7.7 shows an example about the different behavior of a system with a traditional gang scheduling and a system with PDGS. Initially, there are six jobs executing in a machine with eight processors. The red (dark) color means that the application does not reach the target efficiency and the yellow (light) color means that application reaches the target efficiency.

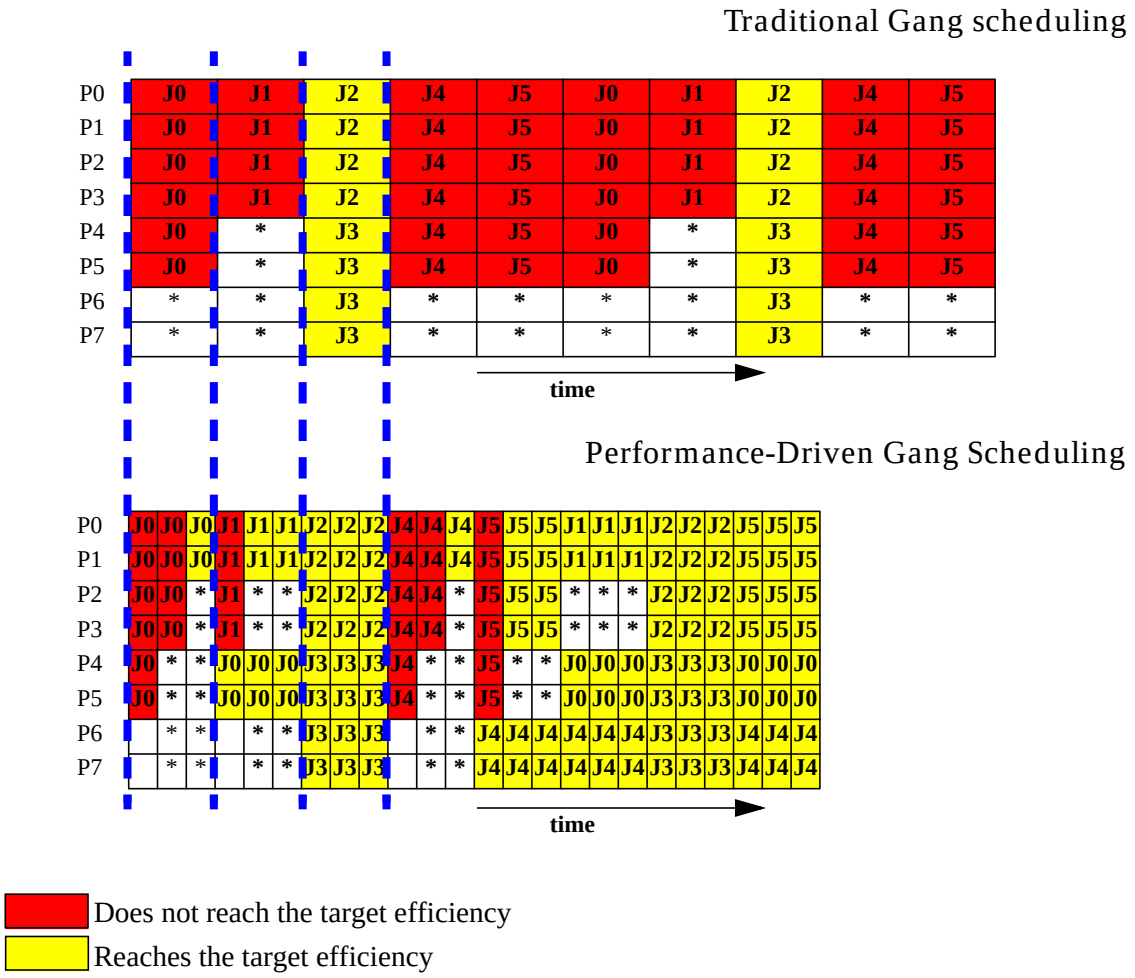


Figure 7.7: PDGS behavior

In a traditional gang scheduling policy, the scheduler is not conscious of the application performance. It executes each job with the number of processors requested. The number of active slots is five because job 2 and job 3 are executed in the same time slot. Then each job is executed one out of every five time slots. The utilization of the system is the 75% and processors of J0, J1, J4, and J5 are not efficiently used. The percentage of cpus that each job receives is: J0 (15%), J1 (10%), J2 (10%), J3(10%), J4 (15%), and J5(15%).

In this example, each time-sharing quantum has been divided into three space-sharing quanta and the *step* has been set to two processors. PDGS initially allocates to each job as many processors as they request. At each space-sharing quantum, PDGS evaluates the achieved performance. If it does not reach the target efficiency, the processor allocation will be reduced in *step* processors. Once finished the time sharing quantum, the slot is changed. The processor allocation of Job_2 and Job_3 is not changed because they reach the target efficiency. The reduction in the processor allocation indirectly favours the migration of jobs and the reduction in the number of time slots. In the example, when Job_1 is started, the re-packing algorithm notes that Job_0 now only uses 2 processors, and that it can be migrated to this slot. This process (evaluation / adjust / migration / re-packing) is continuously repeated. In this example, the stable configuration has only three time slots, the system utilization is 91%, and the utilization per job is: Job_0 (16%), Job_1(8%), Job_2 (16%), Job_3(16%), Job_4 (16%), and Job_5(16%). Even receiving less processors than in the traditional gang scheduling, most of the jobs receives a higher percentage of cpu. The system utilization is better, and cpus are more efficiently used. This is just an example to show the expected benefits due the PDGS. In the evaluation Section we will really demonstrate that PDGS improves gang scheduling approaches that do not consider the application performance.

7.3.1 Multiprogramming level policy

PDGS also includes a multiprogramming level policy. In this case, the policy adopted is the same than in PDPA. Inside each space-sharing quantum, the scheduler evaluates if all the applications assigned to this slot are stable and there are free processors in the current slot. PDGS also evaluates if the number of active slots does not exceed a maximum. If these two conditions are TRUE, a new application is allowed to join the slot. We have set the maximum number of time slots to 4.

We have experimentally observed that it is important to limit to a small number of slots the Ousterhout matrix because of two main reasons. The first one is that the amount of resources used by running applications can saturate the system. We have executed some experiments that show very bad results because the system does not have more processes to execute applications, even when there are empty slots in the Ousterhout matrix. This situation is not so strange because parallel libraries usually generate more processes than they really use. This was our main motivation to modify the NthLib to include the dynamic thread creation (to adjust the number of processes created to the number of allocated processors). The second problem is related to the amount of memory used . We have observed that an application executes faster when executed in standalone mode than when executed inside a multiprogrammed workload because of the interferences that the rest of applications generate, mainly in memory. In gang scheduling policies, memory interferences are generated by running and active applications, which are proportional to the number of time slots.

7.4 Compress&Join: Malleability based on performance information

With PDGS, the number of time slots can be still a problem because the reduction in the number of slots that PDGS generates depends on the performance of running applications. It is possible to get a resulting Ousterhout matrix equal to the original matrix if the performance of running application reaches the target efficiency with the number of requested processors. Moreover, PDGS does not modify the allocation to fit applications in the resulting holes.

In an execution environment with malleable applications, the system fragmentation has no sense because jobs can adapt their parallelism and fit holes in the Ousterhout matrix. Based on this consideration, we propose a new re-packing algorithm that "*compresses*" applications to fit them in a reduced number of slots. The "*application compression*" is made based on the achieved performance and to "*compress*" an application means to reduce its processor allocation.

Compress&Join re-generates the complete Ousterhout matrix. The goal of this algorithm is to minimize the delay introduced by the alternation between slots by reducing their number. We are assuming the benefit provided by reducing the number of slots is greater than the slowdown produced by the reduction in the number of processors allocated to each application.

For instance, consider a simple case when we have one time slot that runs a parallel application with 64 processors. In that case this application does not suffer slowdown because with one time slot there are not context switches. If a new application arrives requesting 64 processors, a normal packing algorithm opens a new time slot and performs time-sharing between the two applications. In that case each one receives 50% of the cpu time and suffers and slowdown of 2. The Compress&Join algorithm will adjust the processor allocation of each application to 32 processors, reducing the number of slots from two to one. After applying the Compress&Join each application will receive half of the number of processors that in a normal algorithm, but (1) they will suffer no context switches, and (2) their efficiency will be also greater with 32 processors than with 64 processors. We assume that the benefit generated by reducing the number of time slots is greater than the penalty by reducing the processor allocation. One important thing is that all the applications collaborate to reduce the number of slots, not only those applications that we try to fit in holes, the rest of applications in the slot are also compressed.

In a general case, this algorithm will significantly reduce the number of context switches, and will eliminate most of the holes in the Ousterhout matrix because the algorithm will try to compress applications in these holes.

Since theoretically a malleable application could be reduced until it only uses one processor, we define a limit in the compression that an application can suffer. This limit is based on the speedup that the application will reach compressed (this may be an extrapolated value), compared with the speedup achieved with the current allocation (which is a calculated value).

Figure 7.8 shows the scheme that results of including the Compress&Join algorithm in a gang scheduling policy. Note that Compress&Join substitutes the re-packing algorithm but that it is totally compatible with the fact of applying a gang scheduling policy such as PDGS.

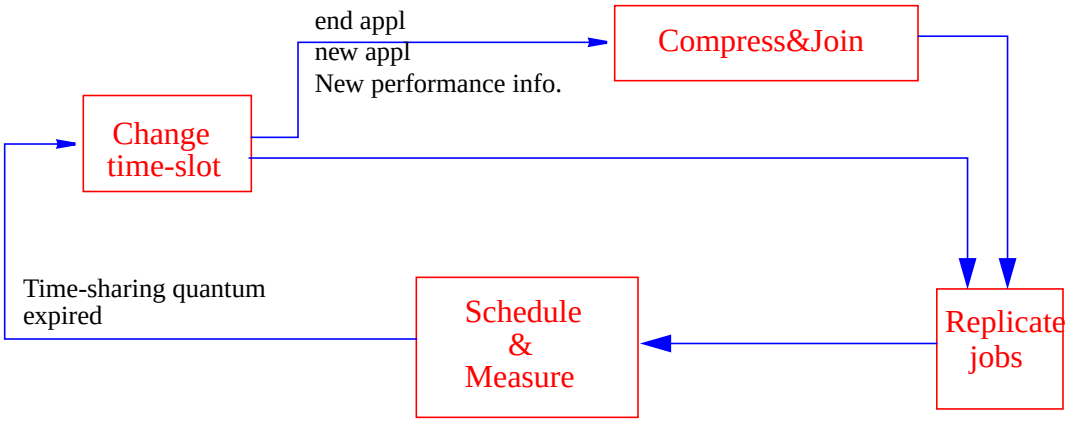


Figure 7.8: Compress&Join scheme

Figure 7.9 shows the main loop that re-generates the Ousterhout matrix in the Compress&Join algorithm. It starts by removing the complete matrix, then it tries to fit each application in some of the active slots. The number of active slots is initially set to zero since the matrix has been cleared. If the job can be compressed in the slot currently processed, the algorithm will go on with the next job. Otherwise, the algorithm tries to compress the job in the next active slot. Finally, if the job has not been compressed in any active slot, the algorithm will activate a new slot and will fit the job in this new slot. Each time the algorithm processes a new job it starts from the first slot. The idea is to first fit holes, rather than to compress jobs.

```
void Compress_and_join()
{
  last_slot=0
  RemoveTimeSlotTable()
  for(job=0;job<total_jobs;job++){
    for (slot=0;slot<last_slot;slot++){
      if (Compressed(job,slot)break;
  }
  if (slot==last_slot){
    last_slot++
    new_slot(appl,slot)
  }
}
```

Figure 7.9: Main Compress&Join loop

Figure 7.10 shows the algorithm that implements the compression of a particular job in a time slot, the *Compressed*(...) function in Figure 7.9. This function receives as parameter a slot identifier and a job identifier. A slot has a set of applications associated to it. This set always will have at least one job because new slots are initialized through the *new_slot*(...) function.

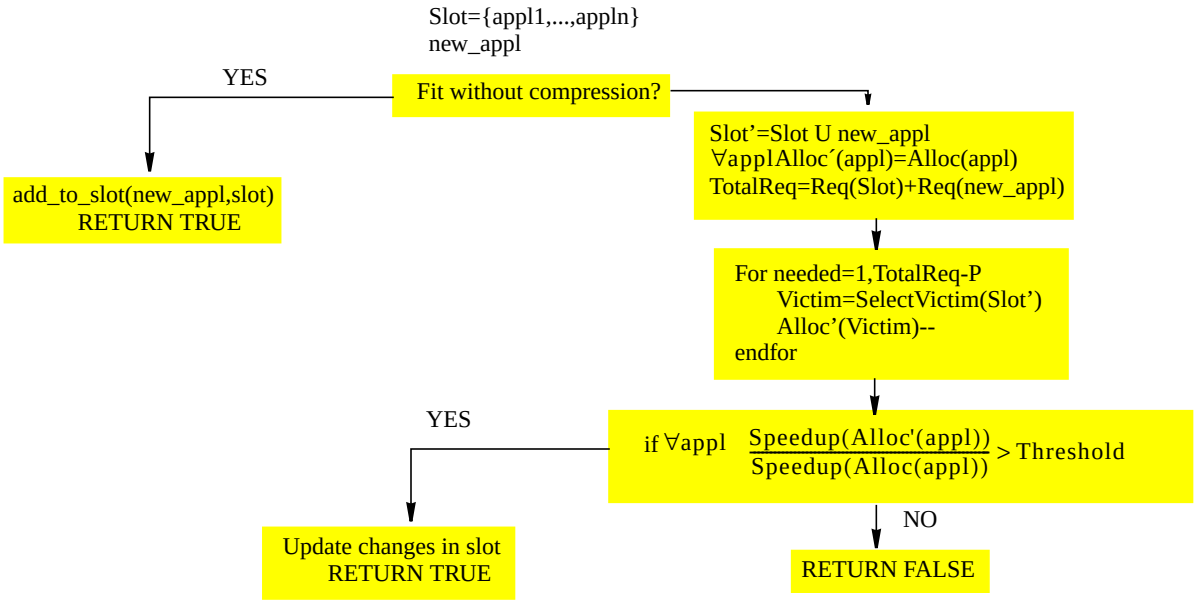


Figure 7.10: Algorithm to compress one application in one time slot

The algorithm initially tries to directly fit the application in the slot. If it fits, the slot will be updated with the new information and the *Compressed* function will return TRUE indicating that the job has been "compressed" in the slot. If the application requests for more processors than the currently available in the slot, the algorithm will try to fit it in the slot by compressing all the applications in the slot. This compression is proportional to the performance of applications in the slot. We compute a temporal slot, *Slot'*, composed by the new application and all the applications in *Slot*, and a temporal allocation, *Alloc'*, initialized to the current allocation of each application, *Alloc*. We also compute the total number of processors requested by *Slot´*, which is the sum of the processors requested by applications in *Slot* and the processors requested by *new_appl*. This value gives us an idea about how many processors we need, which is the difference between *TotalReq* and *P*. For instance, if we have eight processors in our system, six processors allocated to the processed slot, and the new application requests for four processors, we have a total request of ten processors, that means that we need to reclaim two processors to jobs in *Slot'* to have eight processors as maximum allocation. The question now is to check whether we can reduce the allocation of some applications in the slot, including the new one, to fit the new application in this slot.

The criteria used is to select, one by one, the application in *Slot*' with the small efficiency, and to reduce its processor allocation in one processors. The idea is that we will always increase the efficiency of an application if we reduce its allocation[1]. We repeat this process as many times as processors needed. Since the complete efficiency could be not known at this point, the algorithm extrapolates those values that have not been calculated by SelfAnalyzer.

Once computed the new (even temporal) allocation, we evaluate if the reduction in the speedups that this compression will generate is considered acceptable by the algorithm. To decide if a compression is acceptable the Compress&Join defines a threshold. If the ratio between the speedup achieved with the new allocation and the original allocation achieved by all the applications in *Slot*' is greater than this threshold, the compression will be considered acceptable. Otherwise, it is discarded. If it has been acceptable, the *Slot* is updated with the changes, the new application and the new allocation associated to applications in the *Slot*, and the algorithm returns TRUE. If changes has been discarded, the algorithm will return FALSE.

The threshold is a parameter of the algorithm. In our current implementation we have set it to a reduction in 50% in speedup.

The last question is how do we know the speedup achieved with the new allocation. A possible solution is to maintain the compressed allocation until the application informs about its new speedup. However, this solution has the problem that we have to apply the changes in any case, then measure, and then undo the compression if we found that it was an incorrect decision. To do something more "*intelligent*", we have used the formulation proposed by Dowdy in [25] and used in the Equal_efficiency policy. We are conscious that, in some cases, this function can not be representative of the real behavior of the application, but we use it just as a hint to take an initial decision. In any case, once the real performance information is available, we check our decisions. In addition, if the real performance information is available, we use this information, not the extrapolated value.

1. This is not always true but it is a good approximation

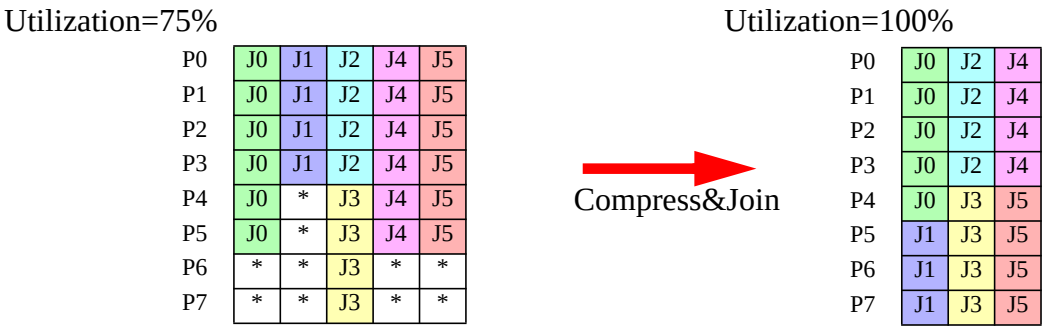


Figure 7.11: Ousterhout Matrix generated by the *Compress&Join*

Figure 7.11 shows the resulting matrix after applying the *Compress&Join* algorithm to the matrix presented in Figure 7.2. We can see how after applying the *Compress&Join* algorithm jobs have been proportionally reduced. In this example, system utilization goes from 75% to 100%, and the cpu percentage that each application receives is shown in Table 7.1. In this case, the different colors do not represent any efficiency values, they are only to better differentiate the different jobs.

Table 7.1: Cpu percentage per job

	J0	J1	J2	J3	J4	J5
Gang	15%	10%	10%	10%	15%	15%
Compress&Join	20.8%	12.5%	16.6%	16.6%	16.6%	16.6%

Observe that, even receiving less processors, each job has increased its percentage of cpu utilization. This is because the number of cpus that each application receives per unit of time is greater than in the Gang version with a simple first-fit algorithm.

Rather than the PDGS that is applied to any running application, the Compress&Join algorithm should be executed when the system conditions require it. These conditions are:

- A new application arrives to the system
- An application finishes its execution
- New real performance information is available and there are significant differences with some extrapolated values
- The allocation of some of the running applications has changed

The two firsts conditions are traditional conditions to re-apply any job re-packing algorithm. The two last are specific of the Compress&Join algorithm.

7.5 Evaluation

In this Section, we compare results achieved by the gang scheduling policy selected for baseline, with results achieved by PDGS and gang scheduling plus the Compress&Join algorithm. As in the previous Chapter, we have used the five workloads presented in Chapter 3. Table 7.2 resumes their main characteristics.

Table 7.2: Workload characteristics

	swim, super-linear		bt, scalable		hydro2d, medium scalable		apsi, not scalable	
	req.	% of cpu	req.	% of cpu	req.	% of cpu	req.	% of cpu
w1	30	50%	30	50%	-	-	-	-
w2	30	50%	30	50%	-	-	-	-
w3	30	50%	2	50%	-	-	-	-
w4	30	25%	30	25%	30	25%	2	25%
w5			30	100%	-	-	-	-

Table 7.3 resumes characteristics of the different configurations evaluated in this Chapter. The multiprogramming level is not fixed because gang scheduling always includes a variable multiprogramming level. The Compress&Join configuration includes the gang scheduling policy presented in this Chapter and used as baseline plus the Compress&Join algorithm.

Table 7.3: Configurations

Policy	Queueing system	Processor scheduler	Run-time library
Gang	Launcher	CPUManager	NthLib
PDGS	Launcher	CPUManager	NthLib
Compress&Join	Launcher	CPUManager	NthLib

7.5.1 Workload 1

Figure 7.12 shows results for workload 1. Workload 1 is a mix of a 50% of super-linear applications (swim's), and a 50% of highly scalable applications (bt's). The number of processors requested by these applications (30 each one of them), implies that the workload does not generate fragmentation with Gang scheduling. The two graphs in the top of the figure show the average response time of swim's and bt's. The two graphs in the bottom of the figure shows the average execution time of swim's and bt's.

PDGS ensures the efficient use of resources by imposing a target efficiency. The reduction in the resulting processor allocation, sometimes generates the indirect benefit of a reduction in the number of slots in the Ousterhout matrix. However, this is not the goal of PDGS. Taking into account that this workload does not generates fragmentation, and applications have been previously tuned, we do not expect significant benefits such as in the previous chapters. Results show that PDGS reaches the same performance than gang in the response time of applications, and slightly worse than gang comparing the execution time of applications. This is quite normal because if (1) the reduction in the allocation does not imply a reduction in the number of slots, and (2) does not imply an improvement in the execution time, the benefit will not be significant.

The Compress&Join algorithm introduces benefits in the system performance. Nevertheless, in this case, we have introduced some undesired fragmentation. After compressing applications and joining slots, some applications have not been reduced and have been executed using one slot for a single application, whereas in the baseline gang scheduling there is no fragmentation in this workload.

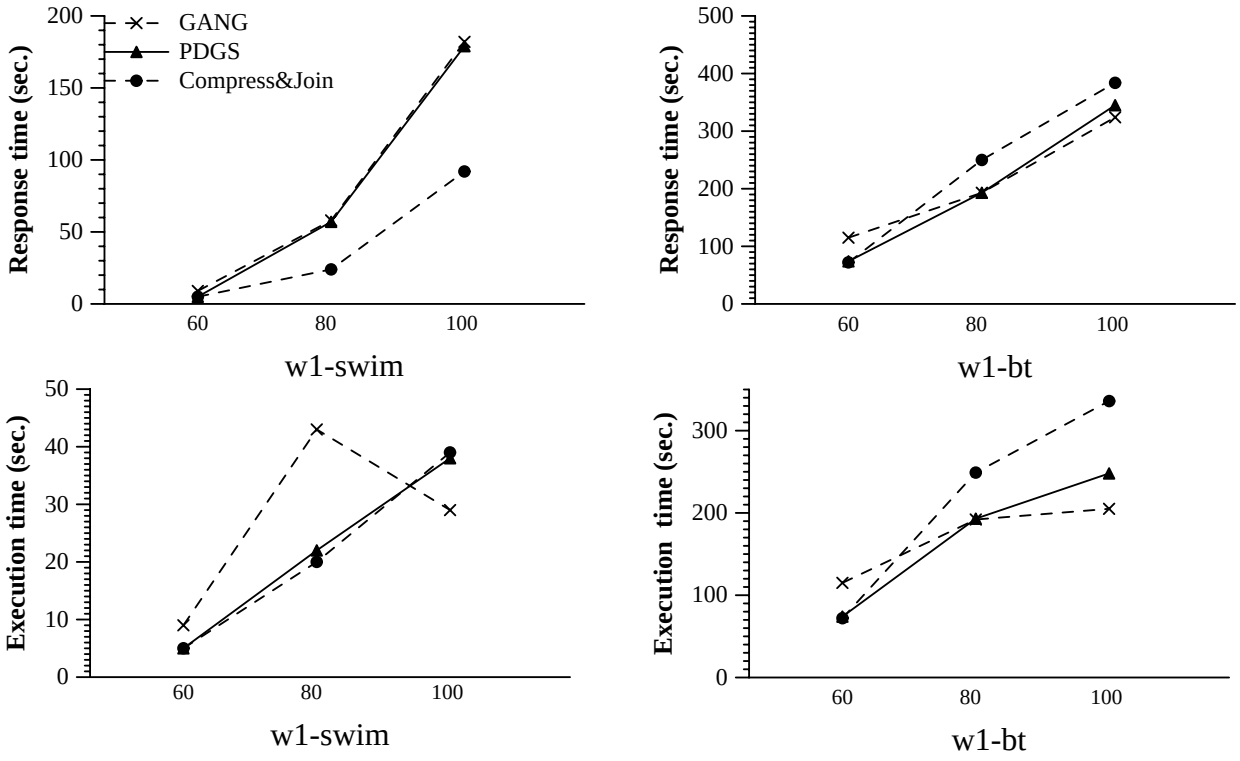


Figure 7.12: Results from workload 1

7.5.2 Workload 2

Figure 7.13 shows results for workload 2. Workload 2 is a mix of a 50% of highly scalable applications (bt's), and a 50% of applications with medium scalability (hydro2d's). In this workload, applications also request for 30 processors each one. As in the previous workload, this workload does not generate fragmentation during its execution. The reason is that all the applications request for 30 processors and the system has 60 processors, then any combination of applications fits in the system.

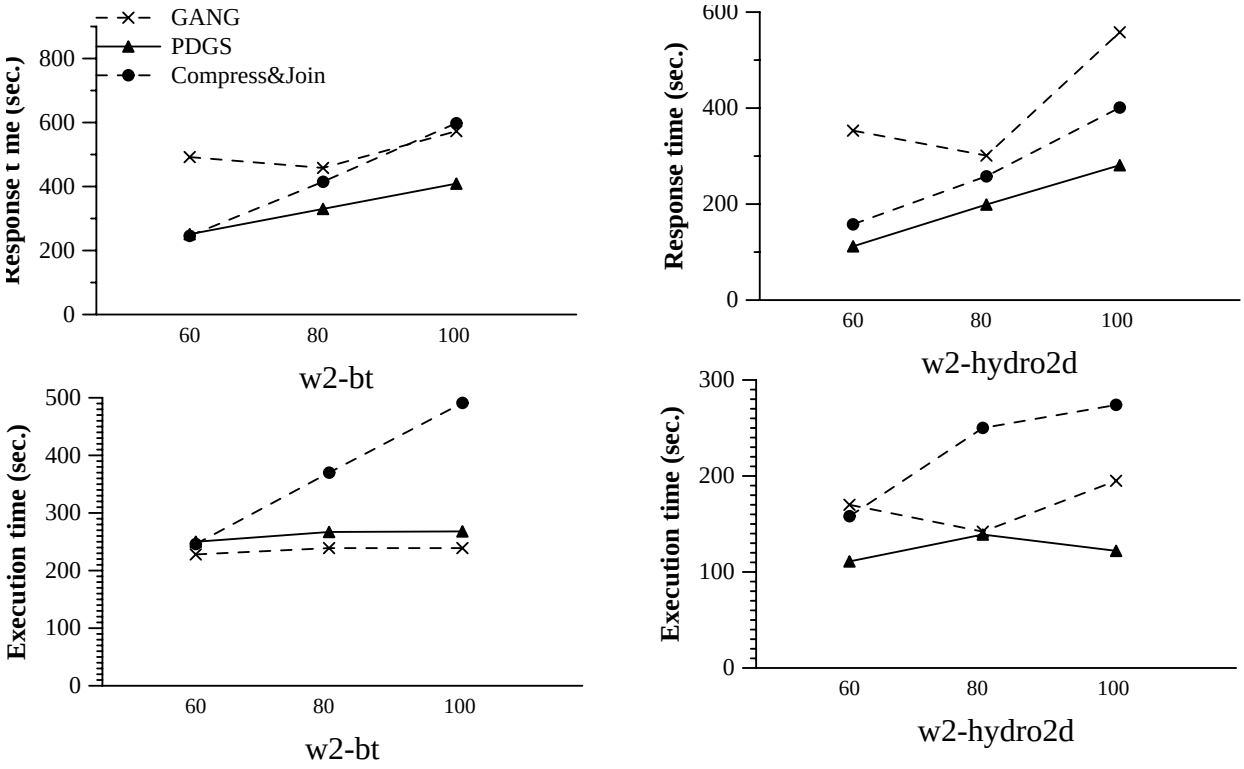


Figure 7.13: Results from workload 2

Comparing the results achieved by PDGS with gang scheduling, we can see that PDGS outperforms gang. In the response time of applications, PDGS outperforms gang by 58% (on average). Gang scheduling improves the execution time of bt's by 10% (on average) compared with PDGS. And PDGS improves the execution time of hydro2d's by 38% (on average) compared with gang scheduling.

In the case of the Compress&Join algorithm, we can observe that it has reduced the response time of applications but at the expense of increasing the execution time by 50% (on average). The response time of applications has been reduced also by 50% (on average).

Table 7.4: PDGS vs. Compress&Join (load=100%)

	PDGS		Compress&Join	
	cpus	time running	cpus	time running
bt's	24	91	17	160
hydro2d's	10	59	10	81

Table 7.4 compares PDGS with Compress&Join. The *cpus* column shows the number of cpus allocated on average by each policy to each application. The *time running* column shows the time the application has been executing, that is, it is not considered the time that the application has been active but not running. As we can see, PDGS allocates more

processors to bt's than gang+Compress&Join. This is because the goal of PDGS is to ensure that applications reach the target efficiency, and the goal of the Compress&Join is to reduce the number of slots, at the expense of reducing the allocation of running applications.

An interesting effect is that PDGS and Compress&Join has allocated the same number of cpus to hydro2d's, but hydro2d's under Compress&join has consumed a 37% more cpu time that under PDGS. This is because the influence of the number of simultaneously running applications. Figure 7.14 shows the multiprogramming level during the execution of the workload under gang+Compress&Join (top of the figure), and under PDGS (bottom of the figure). The x axis is time and the y axis is the number of applications concurrently running at each moment. We have fit the x scale to the duration of the workload but the y scale is the same in both graphs. We can appreciate how under PDGS there are less applications than under gang+Compress&Join.

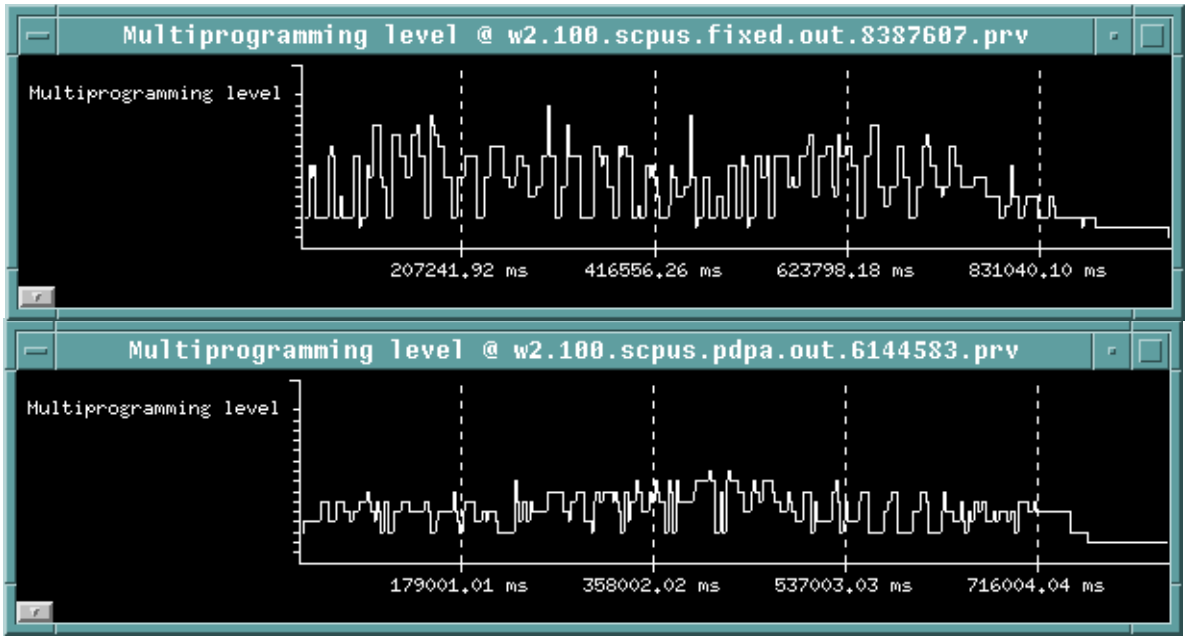


Figure 7.14: Multiprogramming level with Compress&Join and PDGS (load=100%)

Effect of the fragmentation in the workload execution

Workloads used in this Thesis do not generate fragmentation because in most of them applications request for 30 processors and the system uses 60 processors. To give an insight about the potential of exploiting the job malleability, we have executed the workload 2 in such a way that we have generated a bad case for a traditional gang scheduling policy: we have set the request of bt's and hydro2d's to 32 processors. With this modification, a traditional gang scheduling policy is not able to run more than one job per time slot. This situation is possible because in a real system, applications are submitted by different users, and they are not going to tune the request of their applications to fit with the rest of jobs. However, it is obvious that just by reducing in two

processors the allocation of each application we could run, at least, two jobs per time slot. We have performed this experiment with gang scheduling and with gang+Compress&Join.

Figure 7.15 shows the trace file visualization of the execution of the workloads with the two configurations. Each line shows the cpu activity, each color represents a different application. We have set the same x scale to compare them. The first trace file corresponds with the execution with gang scheduling. Blue light color means that the corresponding cpu is idle and each other color represents a different job. The system utilization under gang is the 52% and under Gang+Compress&Join is the 91%.

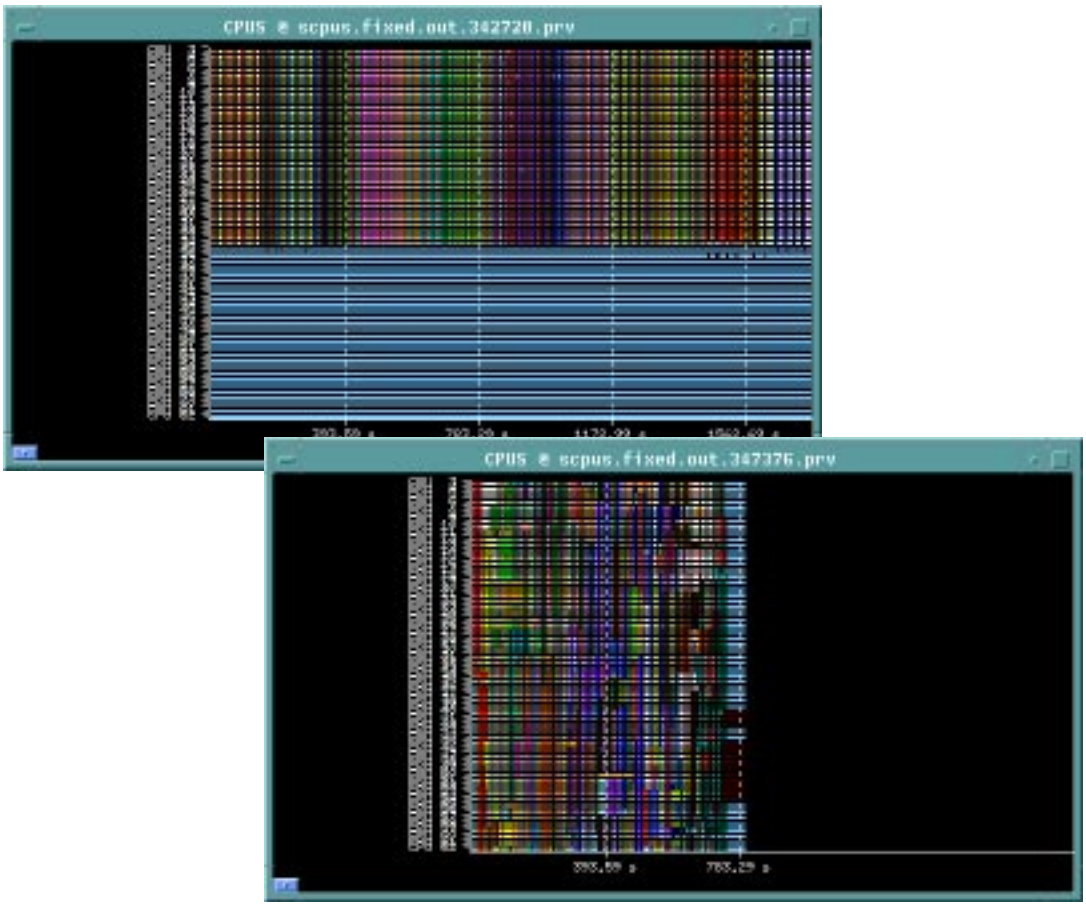


Figure 7.15: Workload 2 (req.=32 proc., load=100%) Gang vs. Gang+Compress&Join

We have also measured the number of cpus allocated by the gang+Compress&Join version. Bt's have received 19 processors on average and hydro2d's 11 processors. These values are very similar to those achieved by the workload execution where applications request for 30 processors. In the previous execution bt's receive 17 processors and hydro2d's receive 10 processors, see Table 7.4. We believe that the small difference is due

to the different application speedup when executing in alone or with other applications at the same time. As we have commented previously, the speedup of a parallel application not only depend on the number of processors received to run.

Figure 7.16 shows the execution time of workload 2 when setting the request of applications to 30 processors compared with the execution time when setting the request of applications to 30 processors. In gang scheduling the execution time has been increased by 33%. In the case of Compress&Join the execution time has been even reduced by 20%. Comparing gang with gang+Compress&join, gang+Compress&Join has speedup the execution of gang by 219%.

What is very important is that gang+Compress&Join is not significantly affected by the user request. This is a common characteristic of all our proposals: they are very robust to changes in the application request and to changes in the system parameters such as multiprogramming level.

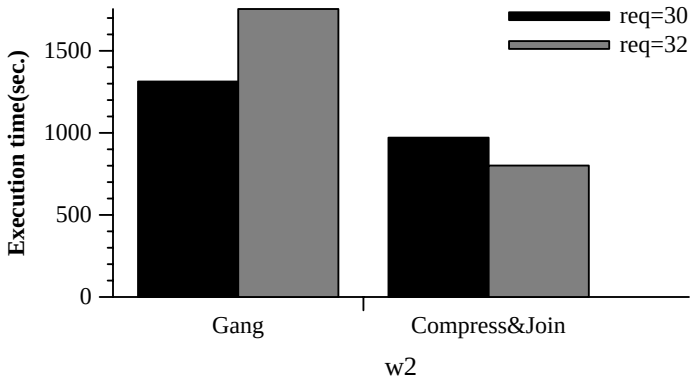


Figure 7.16: Workload 2, (req.=30, load=100%). Gang vs. Gang+Compress&Join

With Compress&Join we can not only improve the system by reducing the number of time slots, but also by reducing the fragmentation that can appear if applications are rigid or the processor allocation policy is not dynamic.

7.5.3 Workload 3

Figure 7.17 shows results for workload 3. Workload 3 is composed by a mix of a 50% of scalable applications (bt's), and a 50% of not scalable applications (apsi's).

Comparing PDGS with gang, we can see that PDGS reaches a similar performance to gang in the response time of bt's, but PDGS outperforms the response time of apsi's. This is because PDGS allocates less processors to bt's, improving the response time of the rest of applications because they can be started before. PDGS improves the response time of bt's by 10%, respect to gang, and the response time of apsi's by 27%.

Compress&Join has introduced an average slowdown of 1% in the response time of bt's, and has improved the response time of apsi's by 22%. In this workload, the fact of having applications requesting two processors generates that the multiprogramming level is very high, in some moments of the workload execution up to 32 applications.

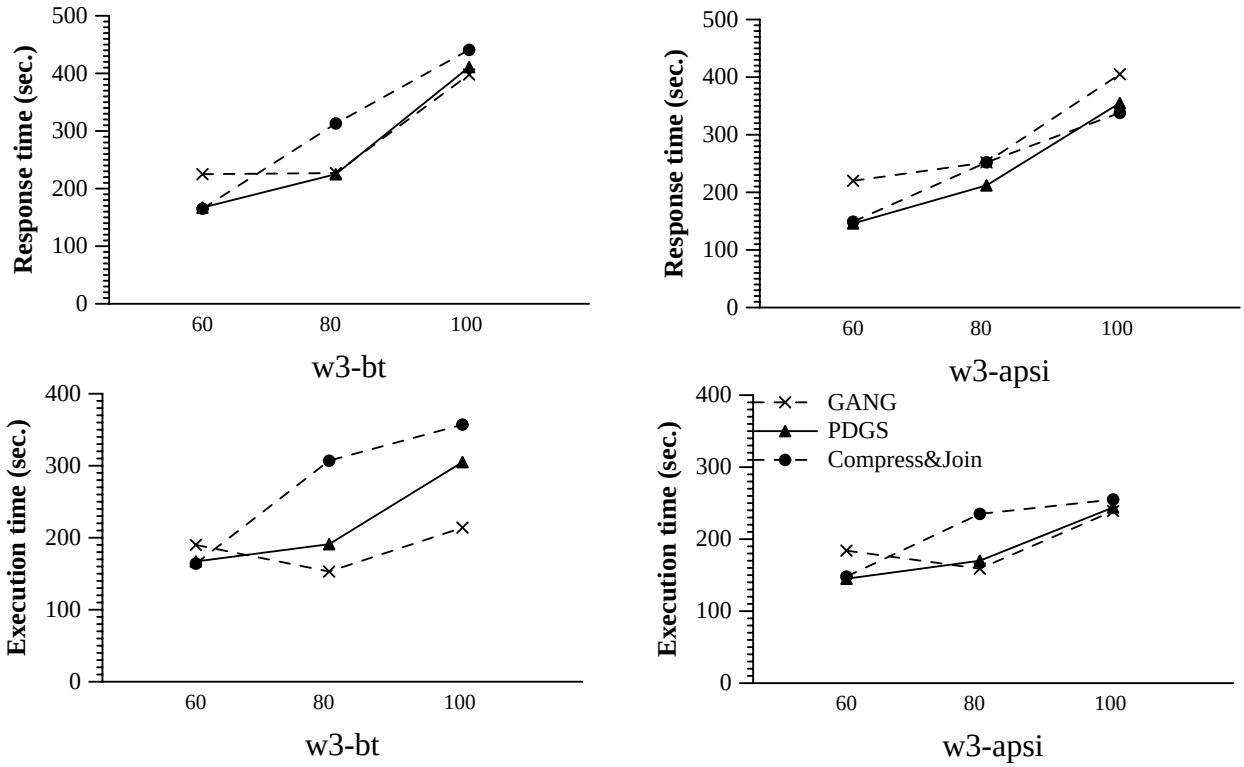


Figure 7.17: Results from workload 3

7.5.4 Workload 4

Figure 7.18 shows results for workload 4. Workload 4 is a mix of the four types of applications: 25% of super-linear applications, 25% of scalable applications, 25% of medium-scalable applications, and 25% of not scalable applications.

Results show that both approaches, PDGS and Compress&Join outperform the baseline gang scheduling. When the load is set to 80%, the particular concurrency of applications generates that results are slightly different than those achieved with the load set to the 60% and 100%. However, on average, PDGS outperforms gang by 248% and the Compress&Join algorithm outperforms gang by 188%. Comparing PDGS and Compress&Join, PDGS shows better results than gang+Compress&Join.

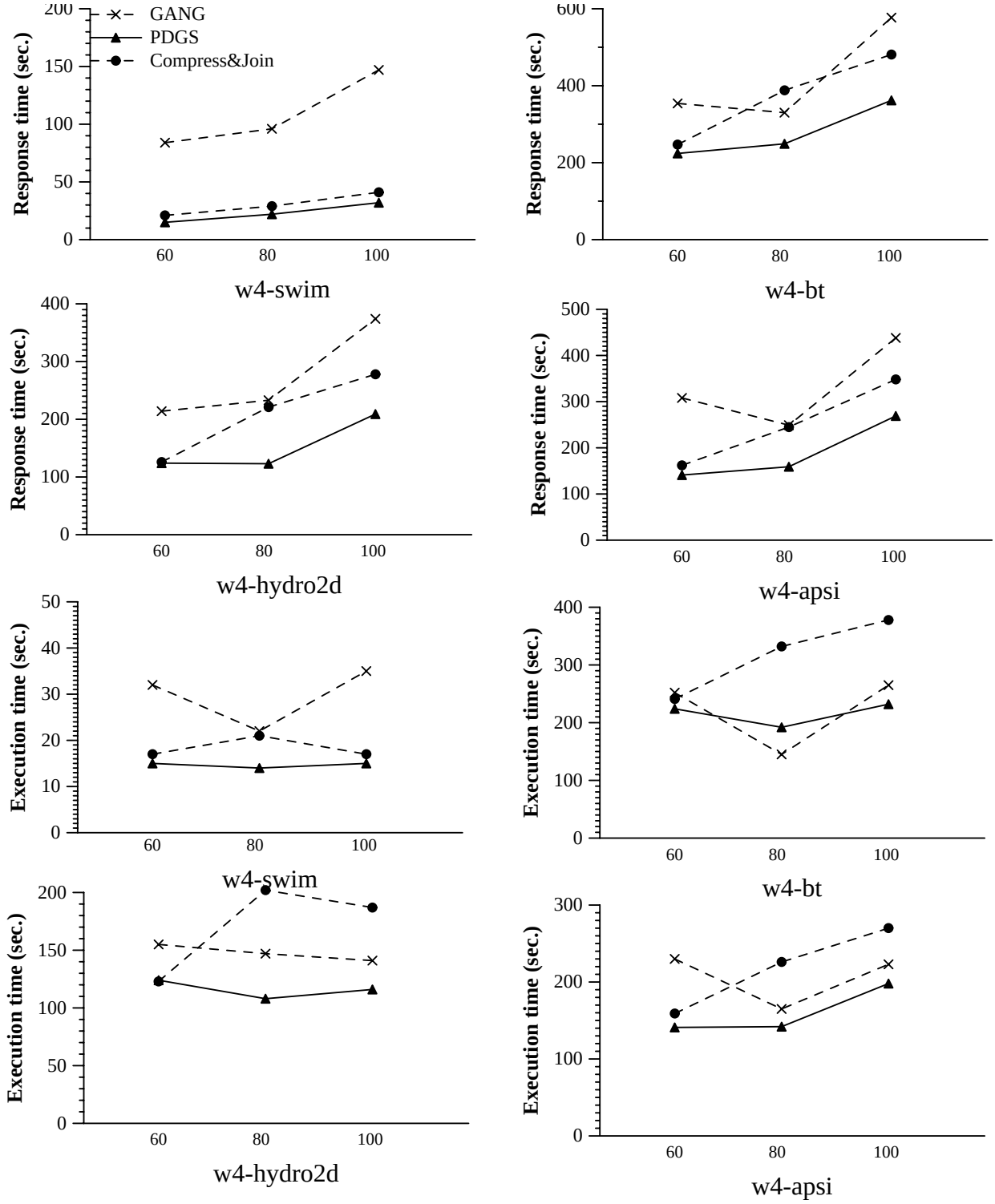


Figure 7.18: Results from workload 4

Table 7.5 shows the number of cpus allocated (on average) to each application per policy, the running time (on average), and the cpu time consumed by these applications when the load is set to the 100%. The cpu usage has been calculated as the number of cpus multiplied by the running time. We have marked in bold font the best result per application. We want to show that using a reduced number of processors, applications can be executed using less total cpu time with a similar execution time. To show that we have analyzed in detail results for workload 4 when the load is set to the 100%.

In the case of swim's, the three policies allocate a high number of processors, and the running times are similar. The policy that achieves the best cpu usage is the baseline, gang scheduling. Swim's under gang consume 11% less cpu time than under PDGS and 6% less than Compress&Join. However, if we analyze the response time achieved by swim's in this workload, Figure 7.18, we can see that both PDGS and Compress&Join significantly outperform gang. This is because PDGS and Compress&Join significantly improve the cpu usage of the rest of applications, resulting in a benefit in the response time of swims. We can see that in the other two policies PDGS and Compress&Join consume much less cpu time than gang, 266% in the case of PDGS and 92% in the case of Compress&Join, and that the running time is also better. We have not presented results for the apsi application because it requests for two processors and then there are no chances to adjust its allocation.

Table 7.5: Cpu usage and running time in workload 4, load=100%

	SWIM			HYDRO2D			BT		
	cpus	running time	cpu usage	cpus	running time	cpu usage	cpus	running time	cpu usage
CJOIN	28	5.65 sec.	158.2 sec.	12	61.3 sec.	735.6 sec.	18	114 sec.	**2052 sec**.
PDGS	27	5.6 sec.	151.2 sec.	11	48.22 sec.	**530.4 sec**.	23	91sec.	2093 sec.
GANG	27	5.29 sec.	**142.8 sec**.	27	52.36 sec.	1413.7 sec.	26	104 sec.	2704 sec.

Figure 7.19 shows the multiprogramming level generated by each policy when the load is set to the 100%. The x axis is the time and the y axis is the multiprogramming level. The graph in the top shows the multiprogramming level generated by gang. Its maximum value has been 20 applications. The second one shows the PDGS multiprogramming level. Its maximum value has been 26 applications. And the last one shows the Compress&Join multiprogramming level. Its maximum value has been 27 applications.

If we observe Table 7.5, the execution time achieved by applications under the different approaches, we will see that PDGS achieves the best results, and that gang reaches better execution times than gang+Compress&Join. As we have commented previously, this is because applications receive less processors when using the Compress&Join algorithm.

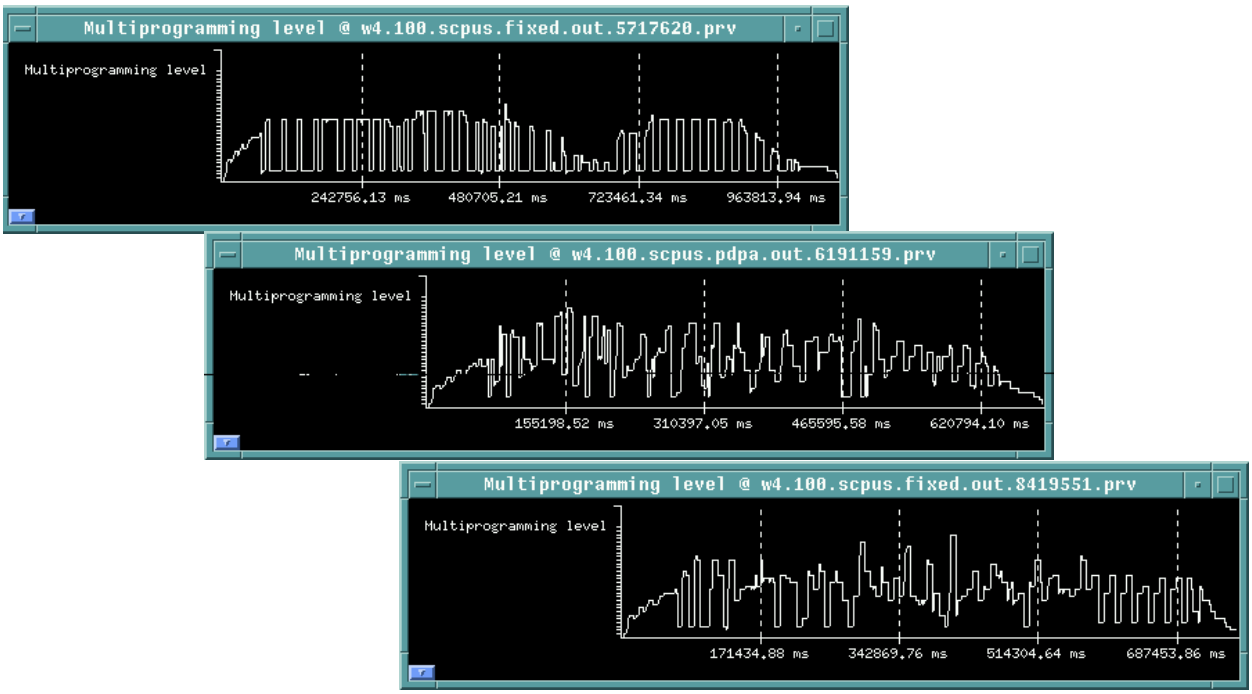


Figure 7.19: Multiprogramming level decided by Gang, PDGS, and Compress&Join, load=100%

7.5.5 Workload 5

Figure 7.20 shows the results for workload 5. Workload 5 is composed by only bt's. As in the previous workloads, the best results are achieved by PDGS. On average, gang+ Compress&Join outperforms gang by 32%, and PDGS outperforms gang by 19%.

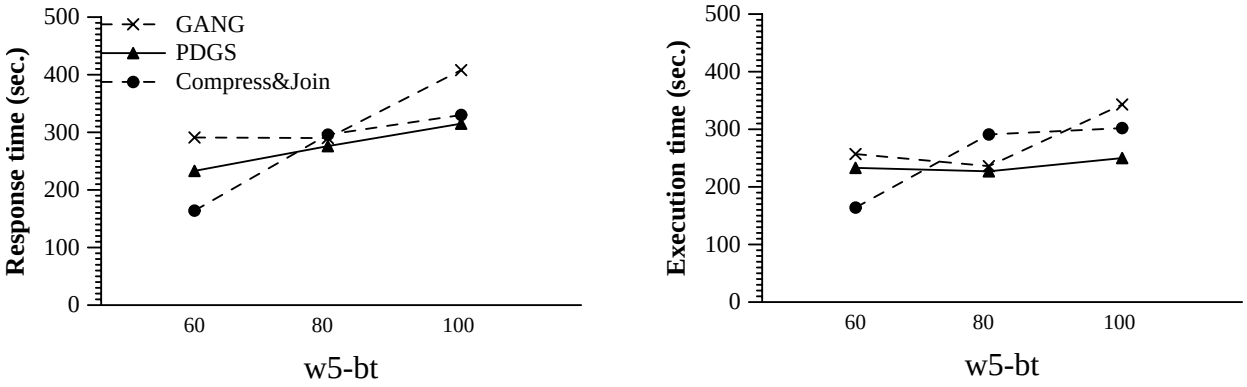


Figure 7.20: Results from workload 5

Table 7.6 shows the number of cpus allocated (on average) by each policy to bt's in the workload 5. As in the previous workloads, PDGS allocates less processors than Gang to applications but consumes less cpu time than Gang, and also consumes less cpu time than

Gang+Compress&Join. Under gang+Compress&Join, applications receive less processors than with the other approaches. If we compare the cpu usage we can see that the best result is achieved by Compress&Join. However, PDGS consumes a 1% more cpu time than Compress&Join, but PDGS outperforms Compress&Join by 11%.

Table 7.6: Cpu allocation and running time in workload 5, load=100%

	BT		
	cpus (avg.)	running time (sec.)	cpu usage
GANG	29	100.51 sec.	2900 sec.
PDGS	24	**94.8 sec.**	2256 sec.
Compress&Join	21	106 sec.	**2226 sec**.

7.5.6 Workload execution times

Figure 7.21 shows the execution time of the five workloads evaluated in this Thesis. We can see that both approaches, PDGS and the Compress&Join algorithm introduce benefits in the execution time of the workload, which is the main goal of this Thesis. We have shown that PDGS and Compress&Join improve the response time of applications, and this has a direct effect in the total workload execution time.

Table 7.7 compares the execution time under gang compared with the execution time under PDGS and Gang+Compress&Join. We show the ratio of the execution time with Gang scheduling and the execution time under each policy. We show the percentage of improvement of our approaches with respect to Gang scheduling. For instance, in the first row, PDGS executes the workload 1 a 14% faster than Gang with load=60%, and a 5% slower with load=80%.

Table 7.7: Percentage of improvement, PDGS and Compress&Join vs. Gang

	60%	80%	100%	AVG.
w1-Gang/PDGS	14%	-5%	-7%	0%
w1-Gang/Gang+CJoin	13%	-3%	-3%	**2%**
w2-Gang/PDGS	75%	36%	56%	**55%**
w2-Gang/Gang+CJoin	76%	25%	35%	45%
w3-Gang/PDGS	42%	17%	21%	**26%**
w3-Gang/Gang+CJoin	38%	7%	13%	19%
w4-Gang/PDGS	54%	38%	54%	**48%**
w4-Gang/Gang+CJoin	45%	21%	40%	35%
w5-Gang/PDGS	15%	0%	25%	13%
w5-Gang/Gang+CJoin	32%	-1%	27%	**19%**

As we can see, both approaches outperform the baseline Gang scheduling, showing the benefit that results from measuring the performance of running applications and adjusting the allocation based on this information in Gang scheduling policies. We have marked in bold type the approach the reaches the best performance (on average) per workload. In three of the five workloads the best performance is reached by PDGS and in two of them by gang+Compress&Join.

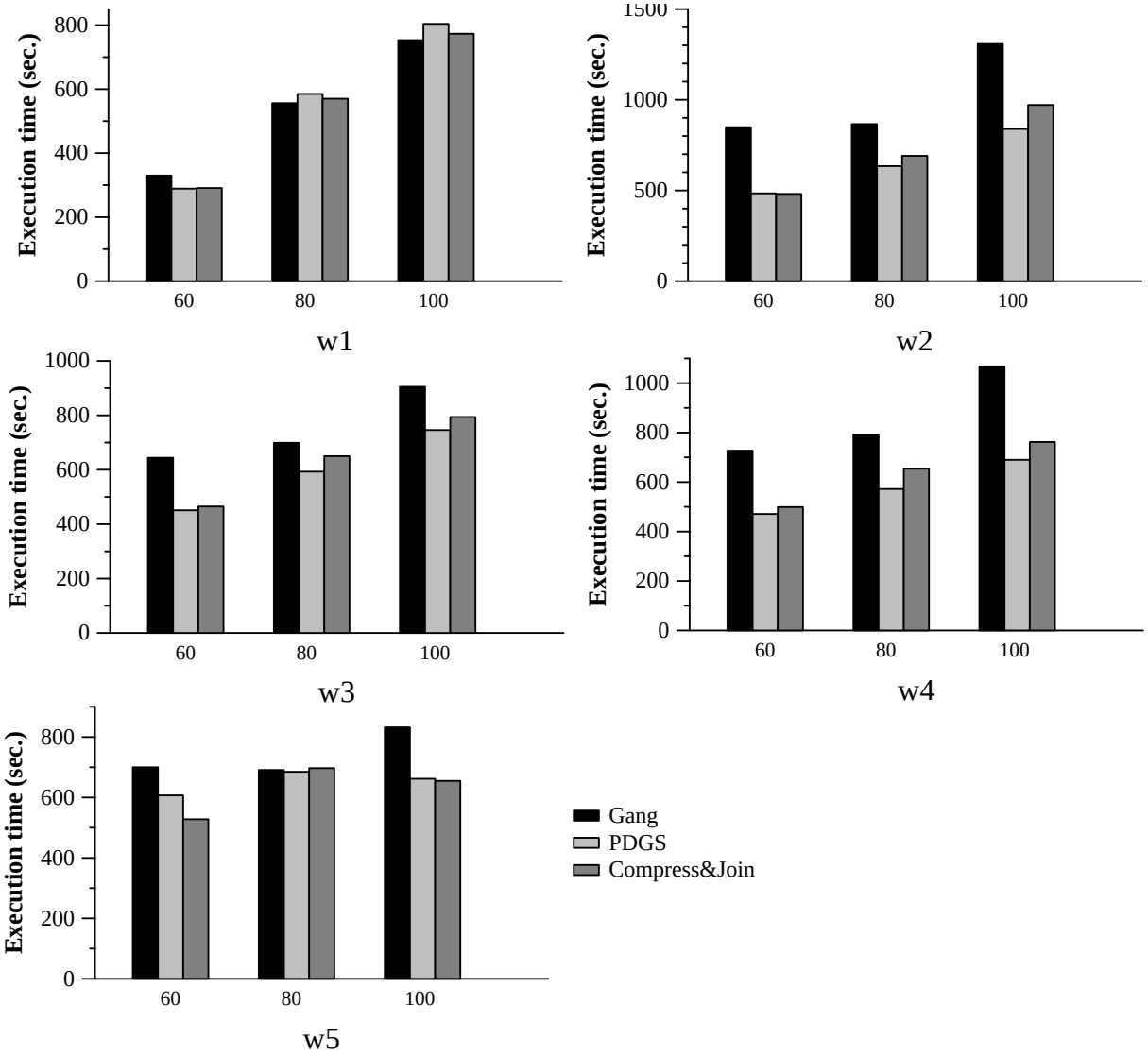


Figure 7.21: Execution time of the complete workloads

7.6 Summary

In this Chapter, we have presented two approaches to improve gang scheduling. Our first proposal consist of adjusting the allocation of running applications based on the performance achieved by them. Applications initially receive the number of processors requested and the system adjust their allocation until they achieve the target efficiency. Once the target efficiency is achieved, the application becomes stable and the system re-applies the re-packing algorithm with the new allocation. This approach is called Performance-Driven Gang Scheduling, PDGS.

The second approach is a job re-packing algorithm, Compress&Join, totally oriented to reduce the number of time-slots in the system. The Compress&Join algorithm reduces the application processor allocation based on the performance achieved. The algorithm imposes a maximum slowdown with respect to the speedup achieved with the requested number of processors. With this reduction, the same set of application can be placed in a reduced number of slots. The aim of this algorithm is both to reduce the fragmentation and to reduce the number of time slots.

We have compared both approaches with a gang scheduling used as baseline that includes a first-fit algorithm, job migration, and the execution of jobs in multiple slots. Results show that both approaches introduce benefits in the execution of the workloads. In some of the cases, the execution time of applications has been increased, but the response time of applications has been improved, resulting in a better workload execution time. We have also shown that the cpu usage is better under PDGS and Compress&Join than under the baseline gang scheduling. This is because processors are more efficiently used.

Finally, an observation based on our experience is that this kind of policies can solve some of the problems presented by space-sharing policies that have a fixed multiprogramming level, such as Equipartition, or Equal_efficiency. On the other hand, results achieved in previous Chapters show that, comparing gang scheduling with dynamic space-sharing policies that have a dynamic multiprogramming level such as PDPA, or Equip++, dynamic space-sharing policies reach better results and are easier to implement.

Our experience also demonstrates us that it is important to dimension the time-sharing quantum to a relatively high value because, otherwise applications can significantly degrade their performance. In the evaluation presented in this Chapter we have used a quantum of six seconds, showing good results. However, in previous evaluations we used a quantum of three seconds and most of them showed bad execution due to a system performance degradation. Reasons were that the overhead generated by the context switch of applications, mainly because of the loss of memory locality, was more significant than the benefit introduced by having a single time slot to execute.

CHAPTER 8

Conclusions and Future Work

This Chapter presents the main conclusions of this Thesis and the future work. In this Thesis, we have demonstrated that, to dynamically measure and take into account the performance of parallel applications to decide the processor allocation improves the system performance. Another important point is the coordination between levels. It is shown that coordinating the different scheduling levels, the system performance and the system utilization are also improved.

As a future work we plan to export the ideas presented in this Thesis to other execution environments, with different architectures and applications. We also plan to evaluate the utilization of these techniques to reduce the power consumption without loss of performance. And finally, we also plan to dynamically measure the utilization of resources that can limit the application scalability, such as the memory bandwidth, and use this information to modify the processor scheduling.

8.1 Goals and contributions of this Thesis

This Thesis had the main goal of improving the execution of workloads of parallel applications in shared-memory multiprocessor architectures. In particular, our research has been focused in the improvement of the processor allocation decisions based on these points:

- The measurement of the application performance at run-time.
- The coordination among the different scheduling levels.
- The imposition of a target efficiency to ensure the efficient use of processors.
- The use of real performance information to decide or to adjust the processor allocation.

To demonstrate the validity of our ideas, we have adopted a practical approach consisting of implementing a complete research execution environment where we have included and evaluated our contributions compared with some previous proposals to evaluate the effectiveness of our work.

We have proposed several scheduling mechanisms and policies that have been implemented and evaluated in this Thesis. The particular contributions and conclusions extracted from their evaluation are presented in the next Section.

8.2 Conclusions of this Thesis

8.2.1 Dynamic performance measurement

The first objective of this Thesis was to demonstrate that the performance of parallel applications could be measured at run-time.

We started from the idea that the system can not rely on users requirements. Based on this premise, we propose that the system considers user requirements as hints, but that it dynamically evaluates the real performance that applications are reaching.

We have proposed a run-time library, SelfAnalyzer, that measures the speedup of OpenMP applications at run-time. SelfAnalyzer exploits the iterative behavior of most of the scientific applications to predict the application behavior based on the speedup achieved by few iterations. We assume that the behavior of several iterations can be extrapolated to the behavior of the complete application. The speedup is measured as the ratio between the average execution time of several iterations with a baseline number of processors and the average execution time with the available number of processors.

Results show that the real application speedup actually has the same behavior than the iteration speedup. We have demonstrated that we can measure the speedup of OpenMP applications and that the overhead introduced is acceptable. We have shown that using a baseline of four processors we can measure the application speedup with a small overhead. Of course, both the overhead introduced and the speedup measurement

precision depends on the application characteristics. However, in all the evaluated policies the shape of the speedup curve was correctly detected and the overhead was acceptable.

8.2.2 Coordination between levels

In the initial planning of this Thesis, we planned to demonstrate the convenience of coordinating the different scheduling levels. The aim was to avoid such situations where decisions taken by a scheduling level negativelly affect the system performance because they are not compatible with decisions taken at the other levels.

In this Thesis we propose that the coordination must be extended to all the scheduling levels to achieve a good overall performance. Coordination means exchanging information between different scheduling levels in such a way that this information will be used to take future scheduling decisions. We have shown with the different policies and methodologies proposed that it is relatively easy to coordinate the different levels.

We propose to include a multiprogramming level policy in the processor scheduler to decide the number of applications that can be concurrently running in the system. With this simple modification, the number of applications can be adapted to the workload characteristics.

The coordination between levels has also shown us that some techniques that in the literature were accepted as quite effective, are not necessary if the scheduling levels are coordinated. This is the case of the two_minute_warning technique.

Moreover, once finished this work, we believe that the coordination should be extended to all the parts of the system. Our experience has shown us that one of the problems related to the coordination is the amount of information that the different levels must manage.

8.2.3 Imposing a target efficiency to ensure the efficient use of resources

In this Thesis, we wanted to demonstrate the benefits in both the application and the system performance of considering the real performance of parallel applications could introduce.

With this aim we have designed and implemented a processor allocation policy that establishes its processor allocation decisions on the speedup achieved by applications, Performance-Driven Processor Allocation policy.

Results show us that PDPA is able to dynamically detect the optimal processor allocation of running applications based on a main criterion: to allocate the maximum number of processors that reaches a given target efficiency. PDPA maintains the processor allocation of those applications that reach an acceptable performance with the number of processors requested, and adjusts the allocation of those applications that do not reach it.

In the worst cases, PDPA introduces a maximum slowdown around 10% in the execution time of some workload respect to best execution time achieved by the rest of policies, with an Equipartition. On the other hand, PDPA speedups the rest of evaluated policies in up to a 400% in extreme cases.

The main conclusion that we extract from PDPA is that the performance of parallel applications can be and must be considered to decide the processor allocation. Results also have shown that PDPA is a robust policy that is neither affected by incorrect user requests nor by incorrect system decisions (multiprogramming level in this case).

The point that has been shown as main in the consideration of performance information for processor scheduling is the imposition of a target efficiency. The use if this target efficiency ensures the efficient use of resources.

8.2.4 Using performance information in multiprocessor scheduling

We have shown that the consideration of the speedup is not exclusive from other criteria. Based on that, we have proposed a methodology that modifies processor scheduling policies to include feedback based on performance information. This methodology, Performance-Driven Multiprogramming Level, proposes to use the performance information to adjust the processor allocation decisions taken by the original policy to which PDML is applied. PDML also proposes to define what the system considers a stable allocation, and in this situation modify the multiprogramming level. PDML has been applied to the Equipartition and the Equal_efficiency resulting in the Equip++ and the Equal_eff++. Results show that with simple modifications, the feedback based on performance information, and the multiprogramming level policy, can be included in the processor scheduling policy. In the evaluation we have shown that Equip++ and Equal_eff++ improve the original policies in those cases where they fail in their processor allocation decisions or where the workload execution was affected by the fixed multiprogramming level used. Results also shown that the overhead introduced in those cases where the original policies perform well is not significant (around the 5%).

We have also observed that it is important to take into account the processor scheduling characteristics to decide parameters such as the default multiprogramming level. For instance, if the policy considers the default multiprogramming level as a minimum, it is better for the system performance to set it to a small value and let the policy to increase it when necessary.

We have also evaluated the ideas of this Thesis in a gang scheduling execution environment. We have tried to improve this kind of policies in two ways: by applying PDML to a baseline gang scheduling policy, developping the Performance-Driven Gang Scheduling, and by a new re-packing algorithm that reduces the number of slots based on the performance achieved by running applications. Results show that both approaches can be included in gang scheduling policies and that improve the execution time of

workloads. We have observed that in this kind of policies a critical point can be the time-sharing quantum used because it can result in a degradation on both the applications and system performance.

Another important point that we have observed in the evaluation of gang scheduling contributions is that, under this kind of policies, the pressure that applications introduce in the system resources is greater than under space-sharing policies because of the resource sharing. In this case, the number of applications concurrently loaded (not necessarily running) in the system is very high. We have observed that even receiving the same number of processors, applications do not perform equal if they are running concurrently with a small number of applications (2..6) than when they are running with a high number of applications (20..25).

Related with the use of real performance information, we want to remark that the use of extrapolated values must be done "*with care*" because some times this formulation can result in incorrect values. We believe that these values must be always considered as temporal and verified with real values. Another important point is the stability of the processor scheduling policy. We have detected that one of the main problems of the Equal_efficiency is that small variations in the performance of one application can result in a global processor distribution. If this situation is very frequent, to completely re-distribute processors is a bad approach for performance.

8.2.5 General remarks

The development of this Thesis has given us a valuable experience in the execution of parallel applications in multiprocessor multiprogrammed execution environments. This experience goes from the parallelization of applications, to the operating system design. We want to include some remarks based on this experience.

The first point that we want to remark is the convenience of using a programming model that provides malleability to applications. Malleability is a characteristic that gives flexibility to the system decisions, and to the applications. To the system because its decisions can be dynamically modified and not determined by the user requirements. And to the applications because the number of chances to be executed and to receive processors increases if the allocation of applications is not limited to a fixed number of processors. Malleability is basic to improve the system performance and we have shown that it can be exploited with any kind of policy: space-sharing and gang scheduling. All the applications used in this Thesis use the OpenMP programming model that gives applications this characteristic.

The second and third points are claims for Operating System vendors and compiler developers. We believe that the O.S. should provide mechanisms to have an easy and efficient access to the hardware counters that modern architectures provide. By the moment, information provided by the hardware is very limited, in one hand, and on the other hand, the cost to access to this information is very high (it is through operating system calls). If this information was easily accessible, researchers will have a way to

develop and evaluate their ideas based on this information. The third point is a claim for compilers developers. It will be very interesting that the compiler provides the run-time with information about the application structure such as the iterative structure and the total number of iterations the application executes. We have found that there are several information related with the sequential structure of the application that is very interesting for the run-time parallel libraries.

Finally, we also want to remark the convenience to define what is considered a workload of scientific application, with a well defined workload trace file. It is also important to provide the applications binary code. The problem is that the available workload trace files represent several months of the load of a real system, and are only valid to use in simulations. If we want to be able to reproduce the same load, we need reduced workload representative from real ones and also to have the application binary codes.

8.3 Future work

Once finished this Thesis, there are several issues that remain opened related with the use of performance information in multiprocessor scheduling, and new points that have appeared as a result of the experience achieved doing this Thesis.

New architectures and programming models

This Thesis has been based on shared-memory multiprocessor architectures and OpenMP applications. However, we believe that the ideas defended in this Thesis can be extrapolated to other environments and to other programming models.

We have planned to analyze the main differences between shared-memory architectures and clusters of SMP's when executing OpenMP applications. As we commented in Chapter 2, clusters of SMP's have the characteristic that they have hardware shared-memory inside each SMP node, and distributed memory between nodes. However, there are some proposals to implement Software Distributed Shared-Memory in clusters of SMP's.

We want to analyze the impact of modifying the allocation of OpenMP applications in this kind of architecture. Our policies and mechanisms already contemplate a step as parameter. This parameter must be extensively tuned in these architectures based on the size of the SMP node. We also have to evaluate the cost to expand the parallelism from one node to more than one node. Probably, the benefit provided by the fact of increasing the parallelism in this way will be lower than in a CC-NUMA architecture, and we have to find solutions to that.

Another point is to complete the SelfAnalyzer in an environment such as a cluster where the cost of modifying the allocation of an application could be significant. An indirect approach could consist of providing the system with a different measure rather than the speedup, and consider the currently number of available processors as the

baseline. A possible approach could be to measure the idleness of the application as a first hint for the system, and only measure the relationship between two execution time measures if the system decides to change the processor allocation of the application.

Computational power balancing based on run-time measurements

We have also planned to use some run-time measured information rather than the speedup to improve the load balancing in non-OpenMP applications. In particular, we want to evaluate the effectiveness of this approach in MPI+OpenMP applications, and a possible useful information coud be the amount of load unbalance between MPI processes.

The number of MPI processes is fixed because this programming model does not consider to dynamically modify the application parallelism. However, if MPI is combined with OpenMP, we can exploit the malleability of OpenMP applications to balance the computational power of each MPI process.

Resource usage distribution

One of the points that we also plan to evaluate is the dynamic measurement of other types of resources such as the memory bandwidth. If two applications that consume a lot of memory bandwidth are concurrently executed, both applications will result negativelly affected. If the memory bandwidth consumed is constant, maybe the best choice will be execute them in serial. However, if these applications have bursts of high memory bandwidth combined with bursts of low memory bandwidth, a good approach will be to phase out their execution and distribute them in the time the moments where each application accesses to memory.

We plan to dynamically measure the memory bandwidth consumed, and to propose an automatic mechanism to detect the behavior of the application respect to the usage of this resource. The goal is to modify the processor scheduling policy to take into account this information and to solve this problem avoiding the system saturation.

Reduction in the power consumption by limiting the use of processors

Proposals made in this Thesis have been oriented to improve the system utilization. The idea was to ensure the efficient use of resources. Using PDPA or applying PDML we can ensure that applications are efficiently using their processors. If there are queued applications, free processors are filled with new applications.

We have planned to evaluate the impact of our proposals in the power consumed by the architecture. The idea is to stop those processors that are not used or even to stop not efficiently used processors. We want to evaluate the impact of modifying the target efficiency in the power consumed by the architecture and the performance achieved by the system and the applications.

Bibliography

[1] G. M. Amdahl, “Validity of the single processor approach to achieving large-scale computing capabilities”, in Proc. AFIPS, vol. 30, pp. 483-485, 1967.

[2] T. E. Anderson, B. N. Bershad, E. D. LAzowska, and H. M. Levy, "Scheduler activations: effective kernel support for the user-level management of parallelism", ACM Trans. Comput. Syst. 10(1), pp. 53-79, Feb 1992.

[3] W.C. Athas and C.L. Seitz, "Multicomputers: messae-passing concurrent computers". Computer 21(8), pp. 9-24, Aug 1988.

[4] J.E. Bahr, S.B. Levenstein, L.A. McMahon, T.J. Mullins, and A.H. Wottreng, "Architecture, design, and performance of Application System/400 (AS/400) multiprocessors". IBM J. Res. Dev. 36(6), pp.1001-1014, Nov 1992.

[5] V. Bala and S. Kipnis. Process groups: a mechanism for the coordination of and communication among processes in the Venus collective communication library. Technical report, IBM T. J. Watson Research Center, October 1992.

[6] V. Bala, S. Kipnis, L. Rudolph, and Marc Snir. "Designing efficient, scalable, and portable collective communication libraries". Technical report, IBM T. J. Watson Research Center, October 1992.

[7] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Visualization and debugging in a heterogeneous environment. IEEE Computer, 26(6):88--95, June 1993.

[8] F. Bellosa, "Locality-information-based scheduling in shared-memory multiprocessors". In Job Scheduling Strategies for Parallel Processing, D.G. feitelson and L.Rudolph (eds.), pp. 271-289, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol 1162.

[9] M. Berry, D. Chen, P.Koss, D.Kuck, S.Lo, Y.Pang,L.Pointer, R. Roloff, A. Sameh, E. Clementi, S.Chin, D. Schneider, G. Fox. P. Messina, D. Walker, C. Hsiung, J. Scharzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum and J, Martin. “The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers”. *The International Journal of Supercomputer Applications*, 3(3):5-40,1989.

[10] G.E. Bier and M.K. Vernon, "Measurement and prediction of contention in multiprocessor operating systems with scientific application workloads". In Intl. Conf. Supercomputing, pp. 9-15, Jul 1988.

[11] D.L. Black, "Processors, priority, and policy: Mach scheduling for new environments". In Proc. Winter USENIX Technical Conf., pp. 1-12, Jan 1991.

[12] D.L. Black, "Scheduling support for concurrency and parallelism in the Mach operating system". Computer 23(5), pp. 35-43, May 1990.

[13] T. B. Brecht, K. Guha. ”Using Parallel Program characteristics in dynamic processor allocation”, Performance Evaluation, 27&28, pp. 519-539, 1996.

[14] J. Corbalan, X. Martorell, J. Labarta. "Performance-Driven Processor Allocation". In Symposium on Operating Systems Design and Implementation (OSDI 2000), pp. 59-71, October 2000.

[15] J. Corbalán and J. Labarta. "Improving Processor Allocation through Run-Time Measured Efficiency". In 15th International Parallel and Distributed Processing Symposium (IPDPS'2001), pp. 74-80, April 2001.

[16] J. Corbalán, X. Martorell and J. Labarta. "Improving Gang Scheduling through Job Performance Analysis and Malleability". In 15th ACM International Conference on Supercomputing, pp. 303-311, June 2001.

[17] J. Corbalán, J. Labarta, "Dynamic Speedup Calculation through Self-Analysis", Technical Report number UPC-DAC-1999-43, Dep. d'Arquitectura de Computadors, UPC, 1999.

[18] J. Corbalán, X. Martorell, J. Labarta, "A Processor Scheduler: The CpuManager ", Technical Report UPC-DAC-1999-69 Dep. d'Arquitectura de Computadors, UPC, 1999.

[19] Cray T3E. `http://www.psc.edu/machines/cray/t3e/t3e.html`

[20] S.-H. Chiang, R. K. Mansharamani, M. K. Vernon. "Use of Application Characteristics and Limited Preemption for Run-To-Completion Parallel Processor Scheduling Policies", In Proc. of the ACM SIGMETRICS Conference, pp. 33-44, May 1994.

[21] H. Chu, . Nahrstedt, 'A Soft Real-Time Scheduling Server in UNIX Operating System", University of Illinois at Urbana Champaign, UIUCDS-R-97-1990.

[22] D. De Paoli, A. Gonscinski, M. Hobbs, and P. Joyce, "Performance comparison of process migration with remote process creation mechanism in RHODOS". In 16th Intl. Conf. Distributed Comput. Syst., pp. 554-561, May 1996.

[23] M. Devarakonda and A. Mukherjee, "Issues in implementation of cache-affinity scheduling", In Proc. Winter USENIX technical Conf., pp. 345-357, Jan 1992.

[24] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated PVM framework supports heterogeneous network computing. Computers in Physics, 7(2):166--75, April1993.

[25] L. Dowdy. "On the Partitioning of Multiprocessor Systems". Technical Report, Vanderbilt University, June 1988.

[26] A.B. Downey, "Using queue time predictions for processor allocation", In Job Scheduling Strategies for Parallel Processing, D.G. Feitelson and L.Rudolph (eds), pp. 35-57, Springer Verlag, 1997. Lectures Notes in Computer Science Vo. 1291.

[27] D. L. Eager, R. B. Bunt, "Characterization of programs for scheduling in multiprogrammed parallel systems". *Performance evaluation* 13 (1991) pp. 109-130.

[28] D.L. Eager, E.D. Lazowska, and J. Zahorjan. "Adaptive load sharing in homogeneous distributed systems". IEEE Trans. Softw. Eng. SE-12(5), pp. 662-675, May 1986.

[29] D.L. Eager, E.D. Lazowska, and J. Zahorjan. " The limited performance benefits of migrating active processes for load sharing". In SIGMETRICS Conf. Measurement & Modeling of Comput. Syst., pp. 63-72, May 1988.

[30] D. L. Eager, J. Zahorjan, E. D. Lawoska. "Speedup Versus Efficiency in Parallel Systems", IEEE Trans. on Computers, Vol. 38,(3), pp. 408-423, March 1989.

[31] D. G. Feitelson, B. Nitzberg. "Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860", in JSSPP Springer-Verlag, Lectures Notes in Computer Science, vol. 949, pp. 337-360, 1995.

[32] D. G. Feitelson. “Job Scheduling in Multiprogrammed Parallel Systems”. IBM Research Report RC 19790 (87657), October 1994, rev. 2 1997.

[33] D. G. Feitelson, “Packing Schemes for Gang Scheduling”, Job Scheduling Strategies for Parallel Processing, pp. 89-110, Springer-Verlag, 1996. Lectures Notes in Computer Science, vol. 1162.

[34] D. G. Feitelson, M. A. Jette, “ Improved Utilization and Responsiveness with Gang Scheduling”, Job Scheduling Strategies for Parallel Processing, pp. 238-261. Springer-Verlag, 1997. Lectures Notes in Computer Science vol. 1291.

[35] D. G. Feitelson, L. Rudolph, “Distributed Hierarchical Control for Parallel Processing”,

[36] D. G. Feitelson, L. Rudolph, “Toward Convergence in Job Schedulers for Parallel Supercomputers”, Job Scheduling Strategies for Parallel Processing, pp. 1-26. Springer-Verlag 1996. Lectures Notes in Computer Science vol. 1162.

[37] D. G. Feitelson, L. Rudolph, “Evaluation of Design Choices for Gang Scheduling Using Distributed Hierarchical Control”, journal of Parallel and Distributed Computing 35, pp. 18-34 (1996)

[38] D. G. Feitelson, A. M. Weil, “Utilization and predictability in scheduling the IBM SP2 with backfilling”, In 12th International Parallel Processing Symposium, pp. 542-546, April 1998.

[39] F. Freitag, J. Corbalán and J. Labarta. "A Dynamic Periodicity Detector: Application to Speedup Computation". In 15th International Parallel and Distributed Processing Symposium (IPDPS’2001), pp. 2-8, April 2001.

[40] G. Ghare, S. Leutenegger, “The effect of Correlating Quantum Allocation and Job Size for Gang Scheduling”, Job Scheduling Strategies for Parallel Processing 1999.

[41] R. Gibbons, "A historical profiler for use by parallel schedulers". In Job Scheduling Strategies for Parallel Processing, D.G. Feitelson and L. Rudolph (eds.), pp. 58-77, Springer Verlpag, 1997. Lecture Notes in Computer Science Vol. 1291.

[42] B. Hamidzadeh, D. J. Lilja, ”Self-Adjusting Scheduling: An On-Line Optimization Technique for Locality Management and Load Balancing”, Int. Conf. on Parallel Processing, vol II, pp. 39-46, 1994.

[43] D. P. Helmbold, Ch. E. McDowell, “Modeling Speedup (n) greater than n”, IEEE Transactions Parallel and Distributed Systems 1(2) pp. 250-256, April 1990.

[44] M. Herlihy, B-H. Lim, and N. Shavit, "Low contention load balancing on large-scale multiprocessors", In 4th Symp. Parallel Algorithms & Architectures, pp. 219-227, Jun 1992.

[45] International Business Machines Corporation. AIX V4.3.3 Workload Manager. Technical White Paper. February 2000.

[46] IRIX 6.5 - Man pages.

[47] H. Jin, M. Frumkin, J. Yan. “The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance”. Technical Report: NAS-99-011, 1999.

[48] D. Klappholz and H-C. Park, "Parallelized process scheduling for a tightly-coupled MIMD machine", In Intl. Conf. Parallel Processing, pp. 315-321, Aug 1984.

[49] S. Krakoviac, "Principles of Operating Systems". MIT Press, 1988.

[50] P. Krueger, T-H. Lai, and V.A. Radiya, "Processor allocation vs. job scheduling on hypercube computers". In 11th Intl. Conf. Distributed Comput. Syst., pp. 394-401, May 1991.

[51] P. krueger and M. Livny, "A comparison of preemptive and non-preemptive load distributing", In 8th Intl. Conf. Distributed Comput. Syst., pp. 123-130, Jun 1988.

[52] J. Labarta, S.Girona, V. Pillet, T. Cortes, L.Gregoris. "DiP : A Parallel Program Development Environment". 2ond Intl. EuroPar Conf. (EuroPar 96), Lyon (France), August 1996.

[53] J. Laudon and D. Lenoski, “The SGI Origin: A ccNUMA Highly Scalable Server”. Proc. 24th Int. Symp. on Computer Architecture, pp. 241-251, 1997.

[54] S.T. Leutenegger and M.K. Vernon, "Multiprogrammed Multiprocessor Scheduling Issues", Research report RC 17642 (#77699), IBM T.J. Watsin Research Center, Nov 1992.

[55] S. T. Leutenegger and M. K. Vernon. “The Performance of Multiprogrammed Multiprocessor Scheduling Policies”, In Proc. of the ACM SIGMETRICS Conference, pp. 226-236, May 1990.

[56] D. Lifka, "The ANL/IBM SP scheduling system". In Job Scheduling Strategies for Parallel Processing", D.G. feitelson and L. Rudolph (eds.), pp. 295-303, Springer-Verlag, 1995. Lectures Nostes in Computer Science Vol. 949.

[57] F.C.H. Lin and R.M. Keller, "The gradient model load balancing method". IEEE TRans. Softw. Eng. SE-13(1), pp. 32-38, Jan 1987.

[58] M. Madhukar, J. D. Padhye, L. W. Dowdy, ”Dynamically Partitioned Multiprocessor Systems”, Computer Science Department, Vanderbilt University, TN 37235, 1995.

[59] S. Majumdar, D. L. Eager, R. B. Bunt, ”Characterisation of programs for scheduling in multiprogrammed parallel systems”, Performance Evaluation 13, pp. 109-130, 1991.

[60] S. Majumdar, D. L. Eager, R. B. Bunt, "Scheduling in multiprogrammed parallel systems". In SIGMETRICS Conf. Measurement & Modeling of Comput. Syst., pp. 104-113, May 1988.

[61] B. D. Marsh, T. J. LeBlanc, M. L. Scott, E. P. Markatos, “First-Class User-Level Threads”. In 13th Symp. Operating Systems Principles, pp. 110-121, Oct. 1991.

[62] X. Martorell, "Dynamic Scheduling of Parallel Applications on Shared-memory Multiprocessors". PhD Thesis, Universitat Politècnica de Catalunya (UPC), July 1999.

[63] X. Martorell, J. Labarta, N. Navarro and E. Ayguade, “Nano-Threads Library Design, Implementation and Evaluation”. Dept. d’Arquitectura de Computadors - UPC, Technical Report: UPC-DAC-1995-33, September 1995.

[64] X. Martorell, J. Labarta, N. Navarro and E. Ayguade, “A Library Implementation of the Nano-Threads Programming Model”. Proc. of the Second Int. Euro-Par Conf., vol. 2, pp. 644-649, Lyon, France, August 1996.

[65] C. McCann, R. Vaswani, J. Zahorjan. “A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors”. ACM Trans. on Computer Systems, 11(2), pp. 146-178, May 1993.

[66] Message Passing Interface Forum. http://www.mpi-forum.org

[67] A. W. Mu’alem, D. G. Feitelson, "Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling". Tech. Report 2000-33, July 2000.

[68] "MIPS R10000 Microprocessor", http://www.sgi.com/processors/r10k/

[69] NANOS Consortium, “Nano-Threads Compiler”, ESPRIT Project No 21907 (NANOS), Deliverable M3D1. Also available at http://www.ac.upc.es/NANOS, July 1999.

[70] T.M. Ni, C-W. Xu, and T.B. Gendreau. " A distributed drafting algorithm for load balancing". IEEE Trans. Softw. Eng. SE-11(10), pp. 1153-1161, Oct 1985.

[71] J. Nieh, M. Lam, “The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications”, Proc. of the 16th ACM Symposium on Operating Systems Principles, St. Malo, France, October, 1997.

[72] J. Nieh, M. Lam, “SMART UNIX SVR4 Support for Multimedia Applications”, Proc. of the IEEE Int. Conf. on Multimedia Computing and Systems, Ottawa, Canada, June 1997.

[73] D. S. Nikolopoulos, T. S. Papatheodorou. "A Quantitative Architectural Evaluation of Synchronization Algorithms and Disciplines on ccNUMA Systems: The Case of the SGI Origin 2000". Proc. of the 13th Intl. Conf. on Supercomputing, Rhodes (Greece), June 1999.

[74] T.D. Nguyen,J. Zahorjan, R. Vaswani, “Maximizing Speedup through Self-Tuning of Processor Allocation”. *IPPS 96*, Technical report UW-CSE-95-09-02. University of Washington

[75] T.D. Nguyen, J. Zahorjan, R.Vaswani, “Parallel Application Characterization for multiprocessor Scheduling Policy Design”. JSSPP, vol.1162 of Lectures Notes in Computer Science. Springer-Verlag, 1996.

[76] T. D. Nguyen, J. Zahorjan, R. Vaswani, “Using Runtime Measured Workload Characteristics in Parallel Processors Scheduling”, in JSSPP volume 1162 of Lectures Notes in Computer Science. Springer-Verlag, 1996.

[77] OpenMP Organization. “OpenMP Fortran Application Interface”, v. 2.0 http://www.openmp.org, June 2000.

[78] J. K. Ousterhout, “Scheduling Techniques for Concurrent Systems”, In Third International Conference on Distributed Computing Systems, pp. 22-30, 1982.

[79] K-H. Park and L.W. Dowdy, "Dynamic partitioning of multiprocessor systems". Intl. J. Parallel Programming 18(2), pp. 91-120, Apr 1989.

[80] E. W. Parsons, K. C. Sevcik. “Benefits of speedup knowledge in memory-constrained multiprocessor scheduling”, Performance Evaluation 27&28, pp.253-272, 1996.

[81] E. W. Parsons, K. C. Sevcik. “Implementing multiprocessor scheduling disciplines". In Job Scheduling Strategies for Parallel Processing, D.G. Feitelson and L.Rudolph (eds.), pp. 166-192, Springer-Verlag 1997. Lecture Notes un Computer Science Vol. 1291.

[82] P. Pierce. The NX/2 operating system. In Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, pages 384--390. ACM Press, 1988.

[83] C.D. Polychronopoulos, "Multiprocessing versus multiprogramming". In Intl. Conf. Parallel Processing, vol. II, pp. 223-230, Aug 1989.

[84] E. D. Polychronopoulos, X. Martorell, D. S. Nikolopoulos, J. Labarta, T. S. 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, July 1998.

[85] E. D. Polychronopoulos, D. S. Nikolopoulos, T. S. Papatheodorou, X. Martorell, J. Labarta, N. Navarro. "An Efficient Kernel-Level Scheduling Methodology for Multiprogrammed Shared Memory Multiprocessors". Proc. of the 12th Intl. Conf. on Parallel and Distributed Computing Systems. Fort Lauderdale, Florida, August 1999.

[86] E. Rosti, E. Smirni, L.W. Dowdy, G. Serazzi, and B.M.Carlson, "Robust partitioning schemes of multiprocessor systems". Performance Evaluation 19 (2-3), pp. 141-165, Mar 1994.

[87] A. Serra, N. Navarro, T. Cortes, “DITools: Application-level Support for Dynamic Extension and Flexible Composition”, in Proceedings of the USENIX Annual Technical Conference, pp. 225-238, June 2000.

[88] S. K. Setia, “Trace-driven Analysis of Migration -based Gang Scheduling Policies for Parallel Computers”, ICPP97, August 1997.

[89] K. C. Sevcik, ”Application Scheduling and Processor Allocation in Multiprogrammed Parallel Processing Systems”. Performance Evaluation 19 (1/3), pp. 107-140, Mar 1994.

[90] K. C. Sevcik. “Characterization of Parallelism in Applications and their Use in Scheduling”. In Proc. of the ACM SIGMETRICS Conference, pp. 171-180, May 1989.

[91] C. Severance, r. Enbody, and P. Petersen. "Managing the overall balance of operating system threads on a multiprocessor using automatic self-allocating threads (ASAT)". J. Parallel & Distributed Comput. 37(1), pp. 106-112, Aug 1996.

[92] SGI Techpubs library, `http://techpubs.sgi.com/`

[93] Silicon Graphics Inc. Origin2000 and Onyx2 Performance Tuning and Optimization Guide. `http://techpubs.sgi.com`, Document Number 007-3430-002, 1998.

[94] F. A. B. Silva, I. D. Scherson, “Improving Throughput and Utilization in Parallel Machines Through Concurrent Gang”, International Parallel and Distributed Processing Symposium 2000.

[95] F. A. B. Silva, I. D. Scherson, “Improving Parallel Job Scheduling Using Runtime Measurements”, 6th Workshop on Job Scheduling Strategies for Parallel Processing, 2000.

[96] J.P. Singh, W.D.Weber, and A. Gupta. “SPLASH: Standford Parallel Applications for Shared Memory”. *Computer Architecture News*, 20(1):5-44, 1992.

[97] J. Skovira, W. Chan, H. Zhou, and D. Lifka, "The EASY- LoadLeveler API project". In Job Scheduling Strategies for Parallel Processing, D.G Feitelson and L. Rudolph (eds.), pp. 41-47, Springer-Verlag, 1996. Lectures Notes in Computer Science Vol. 1162.

[98] M.S. Squillante and E.D. Lazowska, "Using processor-cache affinity information in shared-memory multiprocessor scheduling", IEEE Trans. Parallel & Distributed Syst. 4(2), pp.131-143, Feb 1993.

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

[100] I. Subramanian, C. McCarthy, M. Murphy. “Meeting Performance Goals with the HP-UX Workload Manager”, Proc. of the First Workshop on Industrial Experiences with Systems Software, WIESS 2000, pp. 79-80. October 2000, San Diego, California.

[101] Sun Microsystems. “Solaris Resource Manager [tm] 1.0: Controlling System Resources Effectively”, technical white paper, http://www.sun.com/software/white-papers/wp-srm, 2000.

[102] The Standard Workload Format”, http://www.cs.huji.ac.il/labs/parallel/workload/swf.html

[103] J. Torrellas, A. Gupta, and J. Henessy, "Characterizing the caching and synchronization performance of a multiprocessor operating system". In 5th Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst., pp. 162-174, Oct 1992.

[104] S.K. Tripathi, G. Serazzi, and D. Ghosal. "Processor scheduling in multiprocessor systems". In Parallel Computation, H.P. Zima (ed), pp. 208-255, Springer-Verlag, 1992. Lectures Notes in Computer Science Vol 591.

[105] A. Tucker, A. Gupta, “Process control and scheduling issues for multiprogrammed shared-memory multiprocessors”. In 12th Symposium Operating Systems Principles. pp. 159-166, December 1989.

[106] R. Vaswani and J. Zahorjan, "The implications of cache affinity on processor scheduling for multiprogrammed, shared-memory multiprocessors". In 13th Symp. Operating Systems Principles, pp. 26-40, Oct 1991.

[107] M. J. Voss, R. Eigenmann, ”Reducing Parallel Overheads Through Dynamic Serialization”, Proc. of the 13th Int. Parallel Processing Symposium, pp. 88-92, 1999.

[108] B. Weissman, “Performance Counters and State Sharing Annotations: A Unified Approach to Thread Locality”, Proc. of the 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 127 - 138, 1998.

[109] Workload logs, http://www.ac.upc.es/homes/juli

[110] K. C. Yeager, “The MIPS R10000 Superscalar Microprocessor”. IEEE Micro vol. 16, 2 pp. 28-40, 1996.

[111] I-L. Yen and F.B. Bastani, "Robust parallel resource management in shared memory multiprocessor systems". In 9th Intl. Parallel Processing Symp., pp. 458-465, apr 1995.

[112] K.K. Yue and D.J. Lilja, "Loop-level process control: an effective processor allocation policy for multiprogrammed shared-memory multiprocessors". In Job Scheduling strategies for Parallel Processing, D.G. Feitelson and L. Rudolph (eds.), pp. 182-199, Springer-Verlag, 1995. Lectures Notes in Computer Science Vol. 949.

[113] J. Zahorjan and C. McCann, "Processor scheduling in shared memory multiprocessors". In SIGMETRICS conf. Measurement & Modeling of Comput. Syst., pp. 214-255, May 1990.

[114] Y. Zhu and M. Ahuja, "On job scheduling on a hypercube", IEEE TRans. Parallel & Distributed Syst. 4(1), pp. 62-69, Jan 1993.

[115] B. B. Zhou, D. Walsh, R. P. Brent, “Resource Allocation Schemes for Gang Scheduling”, Job Scheduling Strategies for Parallel Processing, pp. 74-86, Springer-Verlag 2000, Lectures Notes in Computer Science vol. 1911.

[116] Y. Zhang, H. Franke, J. E. Moreira, A. Sivasubramaniam, “ Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques”, IPDPS 2000, Cancun.