Performance-Driven Processor Allocation

Julita Corbalan, Xavier Martorell, and Jesus Labarta

Abstract—In current multiprogrammed multiprocessor systems, to take into account the performance of parallel applications is critical to decide an efficient processor allocation. In this paper, we present the Performance-Driven Processor Allocation policy (PDPA). PDPA is a new scheduling policy that implements a processor allocation policy and a multiprogramming-level policy, in a coordinated way, based on the measured application performance. With regard to the processor allocation, PDPA is a dynamic policy that allocates to applications the maximum number of processors to reach a given target efficiency. With regard to the multiprogramming level, PDPA allows the execution of a new application when free processors are available and the allocation of all the running applications is stable, or if some applications show bad performance. Results demonstrate that PDPA automatically adjusts the processor allocation of parallel applications to reach the specified target efficiency, and that it adjusts the multiprogramming level to the workload characteristics. PDPA is able to adjust the processor allocation and the multiprogramming level without human intervention, which is a desirable property for self-configurable systems, resulting in a better individual application response time.

Index Terms—Operating system algorithms, multiprocessor scheduling, runtime analysis, performance analysis, OpenMP.

1 Introduction

In this paper, we present our proposal for a coordinated scheduler, oriented to efficiently execute workloads of parallel applications in shared-memory multiprocessors. The scheduler is based on the Performance-Driven Processor Allocation policy (PDPA). The goal of PDPA is to minimize the response time, while guaranteeing that the allocated processors are achieving a good efficiency.

PDPA is a processor scheduling policy that decides the processor allocation and the multiprogramming level to be used. PDPA is coordinated with the runtime library and with the queuing system. Coordination means that PDPA informs about its scheduling decisions and it is informed about decisions related to it: On one hand, PDPA informs the runtime library about the number of processors available, preempted threads, etc. And, on the other hand, it informs the queuing system about when to start a new application. Coordination also means that PDPA takes into account the information received to make its decisions.

In this work, we also propose for the processor allocation policy to consider the real performance of parallel applications and impose a target efficiency to avoid the inefficient use of processors. Due to the random nature of such execution environments, there are many factors that can influence the performance of parallel applications. Among them, the size of the input data, the influence of other applications running concurrently, and the fact that users usually are nonexpert and the operating system cannot only rely on the information they provide. Therefore, the performance of parallel applications must be analyzed at runtime and the processors must be redistributed accordingly.

• *The authors are with the Departament d'Arquitectura de Computadors (DAC), Universitat Politècnica de Catalunya (UPC), c/Jordi Girona 1-3, Campus Nord, Mòdul D6, 08034, Barcelona, Spain.*
E-mail: {juli, xavim, jesus}@ac.upc.es.

Manuscript received 20 Mar. 2003; revised 4 Jan. 2004; accepted 10 Sept. 2004; published online 20 May 2005.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0040-0303.

To evaluate our proposal, we have executed four workloads of parallel applications under four scheduling policies on an SGI Origin 2000 with 64 processors: the native IRIX scheduler, Equipartition, Equal_efficiency, and PDPA.

Results show that by imposing a target efficiency to applications, the application execution time is increased in some cases, but the application response time is significantly reduced. The response time is the period of time that starts when the application is submitted and finishes when the application completes.

The rest of this paper is organized as follows: Section 2 presents related work. Section 3 describes the NANOS execution environment in which this work has been developed. Section 4 describes the Performance-Driven Processor Allocation policy. Section 5 evaluates PDPA compared with other dynamic processor allocation policies. And, finally, Section 6 presents the main conclusions and future work.

2 Related Work

Many researchers have studied the use of application characteristics to perform processor scheduling. Majumdar et al. [10], Parsons and Sevcik [17], Sevcik [18], [19], Chiang et al. [3], and Leutenegger and Vernon [9] 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 studies have been carried out using simulations and not through the execution of real applications. They also assume having a priori information available.

Some researchers propose that applications should monitor themselves and tune their parallelism based on their performance. Voss and Eigenmann [23] propose to dynamically detect parallel loops dominated by overheads and to serialize them. Nguyen et al. [15], [16] propose in *SelfTuning* to dynamically measure the efficiency achieved

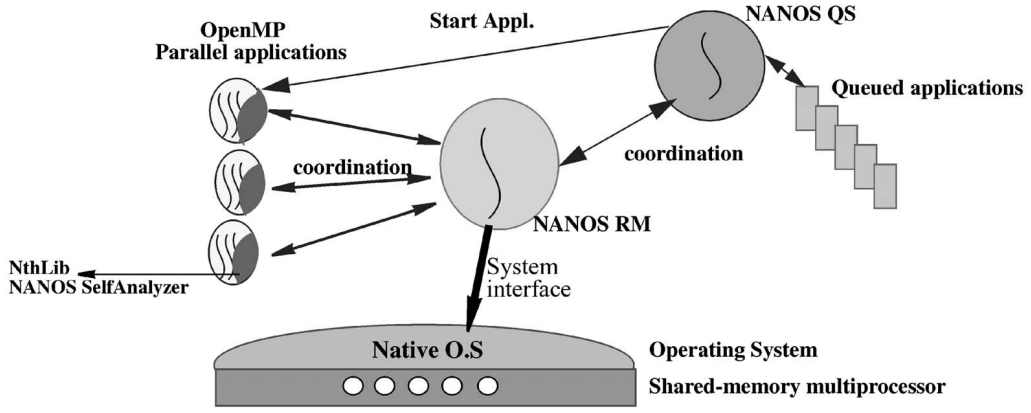

Fig. 1. NANOS execution environment.

in iterative parallel regions and select the best number of processors to execute them by considering the efficiency. SelfTuning is applied at the runtime level.

Other authors propose to communicate the application characteristics to a centralized scheduler which uses these information to allocate processors. Hamidzadeh and Lilja [7] proposes to dynamically optimize the processor allocation by dedicating a processor to search for the optimal allocation. This proposal does not take advantage of the application characteristics and relies only on the system performance (throughput). Nguyen et al. [15], [16] also use the efficiency of the applications, calculated at runtime, to achieve an *Equal_efficiency* in all the processors. The Equal_efficiency policy 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 and Guha [1] use parallel program characteristics in dynamic processor allocation policies also assuming a priori information. McCann et al. [12] 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 idleness, a characteristic provided by each application, to allocate processors, and results in a large number of reallocations.

Recently, Weissmann et al. [24] have proposed Integrated Scheduling, a technique in which the queuing system receives input from the applications running on the system, and adapts their resources accordingly. In this work, applications need to be adapted by the programmer to the scheduler system, providing *sensing* and *actuating* callbacks. The sensing callbacks are used to ask the application about its performance. The actuating callbacks are used to change the resources allocated to applications. As soon as the scheduler has enough information about the running applications, it can decide to move processors from one application to another to improve both applications performance and the overall system utilization. They present the results obtained through simulation, although they are also able to make real executions on clusters of workstations.

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

- With regard to the parameters used by the scheduling policy, our proposal considers two application characteristics: the speedup and the execution time, similar to Eager et al. [4]. However, they propose to work with a priori calculated values. We consider that this is not a desirable precondition because 1) we cannot assume that users will provide this information, and 2) we cannot assume that the information will be correct.
- With regard to the execution environment, we present a practical approach. We have implemented and evaluated our proposal using real applications and a real-commercial architecture, the SGI Origin2000. Most of the previous proposals are based on simulations of the execution environment and the applications. Thus, these previous works are not taking into account important issues related to the architecture, such as the data locality.
- 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.
- We propose to consider the speedup variation compared to the variation in the number of allocated processors. This is the concept of relative speedup presented in Section 4.2.
- 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 improvement of system performance.

3 The NANOS Execution Environment

In this section, we present the particular characteristics of the four elements that constitute our execution environment: the NANOS queuing system (NANOS QS), the NANOS Resource Manager (NANOS RM), the NANOS parallel library (NthLib), and the NANOS SelfAnalyzer. The relation between these elements is outlined in Fig. 1.

3.1 Runtime Libraries

The NANOS parallel library: NthLib. The NthLib parallel runtime library [13], [14] implements the policies and

mechanisms needed for the application-level scheduling. NthLib supports the parallelism specified by users through OpenMP directives, and interacts with the NANOS RM in the following way: It requests for processors and reacts to changes in the number of processors allocated to the application. The processor allocation information is bidirectional, with information flowing both from the NthLib to the NANOS RM and from the NANOS RM to the NthLib.

The NANOS SelfAnalyzer. The NANOS *SelfAnalyzer* [2] is a runtime library that dynamically calculates the speedup achieved by parallel applications and estimates the execution time of parallel applications. The *SelfAnalyzer* exploits the iterative structure of a significant number of scientific applications. The most time-consuming code of these applications is composed by a set of parallel loops inside a sequential loop (named here an *iterative parallel region*). 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. *SelfAnalyzer* is able to work with applications with more than one iterative parallel region and also with nested iterative parallel regions.

The *SelfAnalyzer* controls the execution of several (few) initial iterations of the main outer loop with a small number of processors, called the *baseline* measure. This measure gives *SelfAnalyzer* the reference time to compare with, and to calculate the speedup. Once the *time with baseline* (execution time of one iteration with baseline processors) has been measured, the *SelfAnalyzer* continues with the execution of the application with the number of processors allocated by the NANOS RM. From now on, it measures the execution time of each iteration with the P (allocated) processors, obtaining the *time with P*. The speedup is then calculated as the relationship between the *time with baseline* and the *time with P*. In addition, the *SelfAnalyzer* uses a normalization factor that we call Amdhal's Factor (AF), based on the Amdhal's Law.

The current implementation of *SelfAnalyzer* supports monitoring of applications both when the source code is available, and when only the binary executable is available. In the case when the source code is available, calls to *SelfAnalyzer* can be automatically inserted by the compiler. In the case when the source code is not available, we use a dynamic interposition tool [20] to inject the code with the new code lines. The additional difficulty is to detect those points where the *SelfAnalyzer* must be injected. We have solved this problem by dynamically detecting the iterative structure of the application through a Dynamic Periodicity Detector tool [5]. This tool receives as input the sequence of parallel loops executed (the address of the encapsulated loop), and generates a Boolean indicating if it corresponds with the initial period of a loop or not.

The current version of *SelfAnalyzer* requires applications to be iterative and malleable. We are currently working on extending *SelfAnalyzer* to also work with MPI applications and noniterative jobs. Because we cannot directly measure the speedup in these cases, we plan to provide some hint about application scalability (maybe a combination of the percentage of load imbalance with the percentage of parallelized code).

When codes have an iterative parallel region with a variable working set, this could result in incorrect speedup values when executing the dynamic version because we are not currently able to detect that the application has changed its behavior. However, if calls to *SelfAnalyzer* are automatically inserted by the compiler, this situation could be avoided by resetting data (more information about *SelfAnalyzer* can be found in [2]).

3.2 The NANOS Queuing System

The NANOS Queuing System (NANOS QS) is a user-level submission tool. It implements the job scheduling policy and interacts with the NANOS Resource Manager (NANOS RM) to control the multiprogramming level. The job scheduling policy is in charge of selecting a job from a set of queued jobs. The multiprogramming level is the maximum number of jobs that can be concurrently running at any moment. It is decided by the NANOS RM and enforced by the NANOS QS. The NANOS QS has been implemented to introduce repeatability in the submission of workloads of parallel applications with the aim of evaluating them under different scheduling policies.

3.3 The NANOS Resource Manager

The NANOS Resource Manager (NANOS RM) is the user-level processor scheduler. Once the NANOS QS starts the execution of an application, the application enters under the NANOS RM control. The NANOS RM 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 NANOS RM uses the native operating system interface to manage processes and processors and provides the interface to communicate with the NANOS QS.

The NANOS RM implements PDPA, Equipartition [12], and Equal_efficiency [16]. Equipartition is a dynamic processor allocation policy that decides an equal allocation among running jobs. Reallocations are done at job arrival and job completion. Equal_efficiency allocates more processors to those applications that have the best efficiency using extrapolated values.

4 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. In this section, we describe the PDPA processor allocation policy and the PDPA multiprogramming-level policy.

4.1 Processor Allocation Policy

PDPA is a dynamic space-sharing policy targeted to shared-memory multiprocessors. This kind of policy divides the machine in partitions 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,

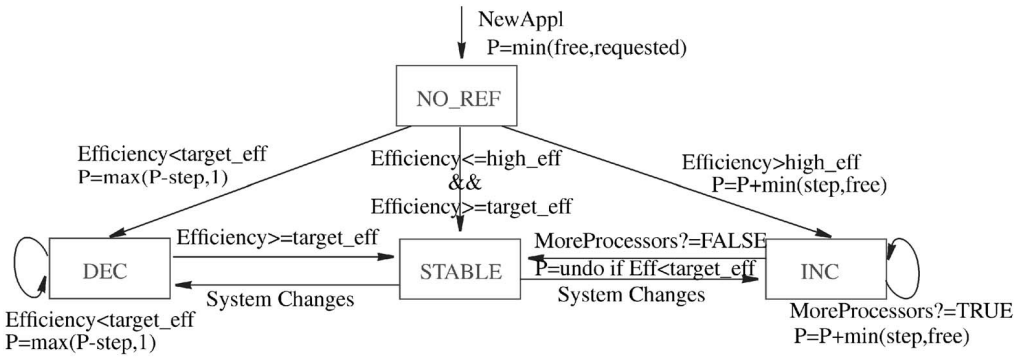

Fig. 2. PDPA: Application state diagram.

while guaranteeing that the allocated processors are being used efficiently. PDPA is activated each time a new application arrives to the system, when an application finishes, or when an 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 allocations different from the current one and the efficiency achieved with them. PDPA uses this information to compare with the actual allocation and performance. The system administrator defines the target efficiency that he/she wants in his/her system. Alternatively, it is dynamically set depending on the load of the system. Currently, it is set the same for all the applications (it is a global target).

4.2 Application State Diagram

PDPA considers each application to be in one of the states shown in Fig. 2. 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) the efficiency considered very good (*high_eff*), 2) the target efficiency (*target_eff*), and 3) the number of processors that will be used to increment/decrement the application processor allocation (*step*). These parameters can be modified at runtime.

PDPA can assign applications to four different states: *NO_REF*, *DEC*, *INC*, and *STABLE* (Fig. 2). Each different state means the knowledge that PDPA has about the performance of each application at the last time PDPA evaluated it. Each time PDPA is activated, it evaluates the performance of each application and decides its next state and the next processor allocation that will receive. Modifications in the processor allocation are associated to state transitions (even if the next state is the same).

4.2.1 No Performance Knowledge: NO_REF State

Applications start in the *NO_REF* state because *PDPA* has no performance knowledge about them (they are in their starting point). PDPA initially allocates the minimum between the number of processors requested and the number of free processors in the system. Once the application informs about its speedup, *PDPA* compares the achieved efficiency[1] with *high_eff* and *target_eff*. If the efficiency is greater than *high_eff*, the next state will be *INC*, (*good performance*). If the efficiency is lower than *target_eff*, the next state will be *DEC* (*bad performance*). If the efficiency is between *high_eff* and *target_eff*, the next state will be *STABLE* (*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.

4.2.2 Good Performance: 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. When a job is in the INC state and informs about its performance, PDPA evaluates: 1) if the achieved efficiency is greater than *high_eff*, 2) that the current speedup obtained is greater than the previous speedup obtained, and 3) if the *RelativeSpeedup* is greater than the percentage of additional processors multiplied by *high_eff*.

The *RelativeSpeedup* measures whether 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 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 superlinear within a range of processors (that means a very high efficiency), but later the speedup progression is not maintained. 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, only if the current efficiency is less than *target_eff*.

4.2.3 Bad Performance: DEC State

The *DEC* state means that the application has not reached the target efficiency the last time PDPA evaluated it. If the current performance is less than *target_eff*, the next state will be *DEC*; otherwise, the next state will be *STABLE*. If the next state is *DEC*, the application will receive the maximum

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

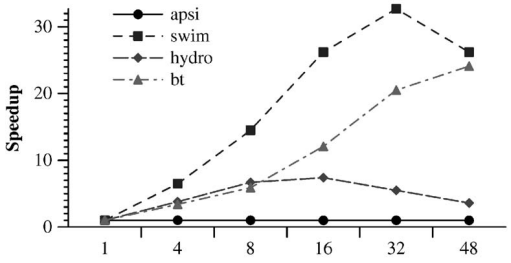

Fig. 3. Speedup curves of the different applications considered.

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

4.2.4 Acceptable Performance: 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 changed, PDPA could change the state of an application from *STABLE* to either *INC* or *DEC*, accordingly. If the application performance changes, the next state and processor allocation could be modified. The number of transitions from STABLE to either DEC or INC may be limited by the system to avoid ping-pong effects.

4.3 Multiprogramming-Level Policy

The multiprogramming level is the number of applications concurrently running in the system. Traditional approaches execute parallel workloads 1) limiting the multiprogramming level, resulting in fragmentation, or 2) without controlling it, resulting in overloading the system. We want to control at any moment the load[2] of the system. Therefore, in this paper, we discard the second option of executing parallel workloads without controlling the multiprogramming level. The first option 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 as the application requests are free. Based on this consideration, we propose to coordinate the two scheduling levels: We leave the decision about when to start a new application to the processor scheduling policy, and we leave the selection of which application to start to the queuing system.

5 Evaluation

We used four workloads to evaluate the proposals of this paper. Each workload consists of combinations of four applications: Swim (SpecFP95 [21]) is superlinear, bt.A (NASPB [6]) has a good scalability, hydro2d (SpecFP95) has a medium scalability, and apsi (SpecFP95) does not scale at all. Fig. 3 shows the speedup curves of the applications used in the paper.

2. The load of the system is the percentage of cpu time used compared to the total amount available.

TABLE 1
Workload Characteristics

	Swim	bt.A	hydro2d	Apsi
w1	50%	50%	-	-
w2	-	50%	50%	-
w3	-	50%	-	50%
w4	25%	25%	25%	25%

The workloads differ in the percentage of each type of application: superlinear, highly scalable, medium scalable, and not scalable. The speedup curves of these applications are presented in Fig. 3 and more information about them can be found in [2].

The scheduling policies that we evaluated in this paper are: the native IRIX scheduling (IRIX), Equipartition (Equip), Equal_efficiency (Equal_eff), and PDPA. We used the NANOS QS as a queuing system, and Equipartition, Equal_efficiency, and PDPA are implemented inside the NANOS RM.

For the native IRIX, we used the native SGI-MP library. This runtime library uses some environment variables that define its behavior. We set the following variables: The OMP_NUM_THREADS variable specifies the number of kernel threads to be used by the application. The OMP_DYNAMIC variable set to TRUE enables the dynamic adjustment of the number of threads available for the execution of parallel regions. We have also tuned the MP_BLOCKTIME environment variable, which controls the amount of time a slave thread waits for work before giving up. This time is specified as the number of times to spin in the wait loop (set to 200.000). Finally, the _DSM_MIGRATION environment variable, which specifies aspects of automatic page migration, was set to ALL_ON to enable migration for all data.

IRIX, Equipartition, and Equal_efficiency were executed with a fixed multiprogramming level, set to four applications. PDPA used also a default multiprogramming level of four applications. The *target_eff* was set to 0.7 and the *high_eff* to 0.9.

We generated workloads where applications are submitted to the system following a Poison interarrival function during 300 seconds. These workloads had an estimated processor demand of 60 percent, 80 percent, and 100 percent of the total capacity of the system. We used workload trace files that specify the arrival sequence of applications to the system. Therefore, the same set of applications was executed in all the scheduling policies evaluated and the submission time of each application was the same. These workload trace files follow the specification proposed by Feitelson in [22] and can be found in [25].

For each workload, we calculated the average response time and the average execution time per scheduling policy and application class. Table 1 presents the composition of the four workloads used in this paper. Each cell indicates the portion of the system load generated by that application in the corresponding workload. In the next sections, we detail the characteristics of the workloads and present the results for each workload.

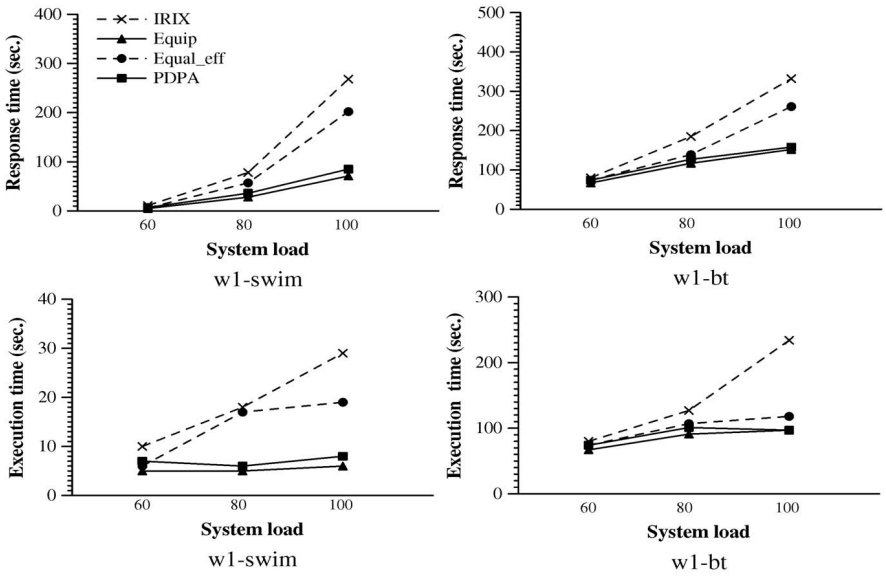

Fig. 4. Results of workload 1. Response and execution time.

In all the workloads, swim, bt, and hydro2d request for 30 processors, and apsi requests for 2 processors due to its poor scalability. We evaluated all these workloads in an SGI Origin2000. This is a CC-NUMA machine with 64 processors. However, we only used 60 processors to evaluate our workloads. We also 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 [8].

5.1 Workload 1

Fig. 4 shows the results obtained from workload 1. The graphs in the top of the figure show the average response time (in seconds) of swim and bt. In the x-axis, we represent the average system load generated by the workload (60 percent, 80 percent, and 100 percent). The graphs in the bottom of the figure show the average execution time (in seconds) of swim and bt. This workload has the characteristic that 1) applications are scalable, 2) they initially 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 causes that applications under the Equipartition execute with 15 processors (we have a total of 60 processors) when the machine is heavily loaded and with 30 processors if the machine is lightly 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 and the execution times 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. When comparing the response time achieved by PDPA and Equipartition, PDPA is 10 percent worse than Equipartition in the case of bt, and approximately 30 percent worse than Equipartition in the case of swim. The execution time of PDPA is 7 percent worse than Equipartition in the case of the bt application, and 30 percent worse in the case of swim.

The Equal_efficiency has the problem that it is too sensitive to small changes in the efficiency measurements. Small variations in the efficiency generate high variances in the processor allocation, resulting in a high number of processor reallocations. The system has to be conscious of the applications being malleable, but know that reallocations are not free, and it is something that must be done "*with care*." Another problem related to 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, when the load of the system is 100 percent, we measured the processor allocation received by swim, and we have found that the Equal_efficiency allocated from a minimum of 2 processors up to a maximum of 28 processors. This is an unfair allocation because two instances of the same application (with the same performance and requesting for the same number of processors) should receive the same amount of processors.

Observe that, in this type of workload, it could be beneficial to reduce the multiprogramming level, improving the execution time. However, in this paper, 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 remains as future work, but this kind of modification will be easily introduced in PDPA.

5.1.1 Related Issues That Affect the System Performance

We have observed that both Equipartition and PDPA significantly improve the results achieved by IRIX and the Equal_efficiency policies. In the case of IRIX, the main reasons are the unresponsiveness of the native runtime

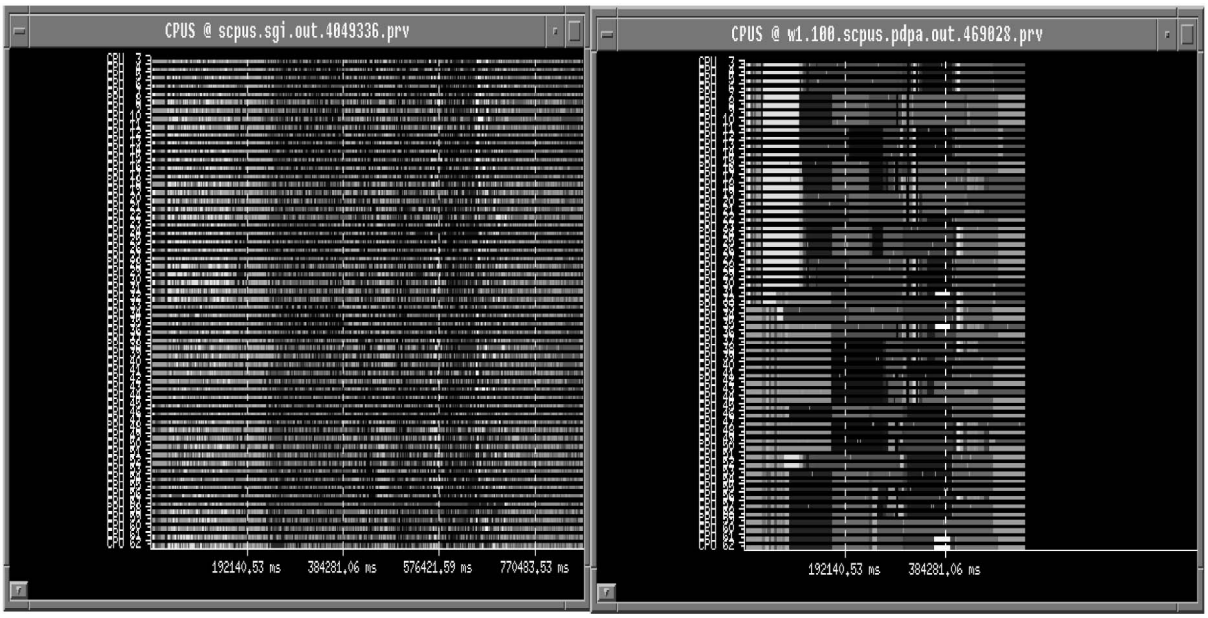

Fig. 5. Execution views for workload 1 under IRIX and PDPA.

system to changes in the system load, and the lack of coordination with the queuing system. The processor placement policy used by IRIX also has a significant influence. This placement policy is based on maintaining the processor affinity as much as possible. However, sometimes it causes that two kernel threads belonging to the same or different applications can be allocated to the same processor, degrading the application performance and generating many process migrations.

Fig. 5 shows the trace file visualization for the workload 1 executed under IRIX, and PDPA (load = 100 percent). These plots were generated with the Paraver Tool [8]. 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 trace files.

We can appreciate that the look of the execution under the native IRIX scheduler is chaotic. The PDPA trace on the right side shows that the execution is quite stable and we can clearly differentiate the execution of the different applications on it. 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.

Using Paraver, we have also measured the total number of process migrations, the duration of the bursts executed by each cpu, and the number of bursts executed per cpu. Table 2 shows the results obtained from the three policies. As we can see, the native IRIX scheduler generates much more kernel thread migrations. The time each cpu is executing the same application under IRIX is approximately 50 times less than under PDPA or Equipartition. The number of cpus allocated under IRIX and under Equipartition is around 15 processors using both policies.

5.2 Workload 2

Fig. 6 shows results from workload 2. The graphs in the top of the figure show the average response time and the graphs at the bottom show execution time of bt and hydro2d applications.

This workload has been designed to evaluate the behavior of the policies when executing a workload where the 50 percent of the load consists of applications with high scalability, and the other 50 percent of the load has medium scalability. 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. With PDPA, the percentage of cpus that are assigned in

TABLE 2
IRIX versus PDPA and Equipartition, Workload 1 (Load = 100 Percent)

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

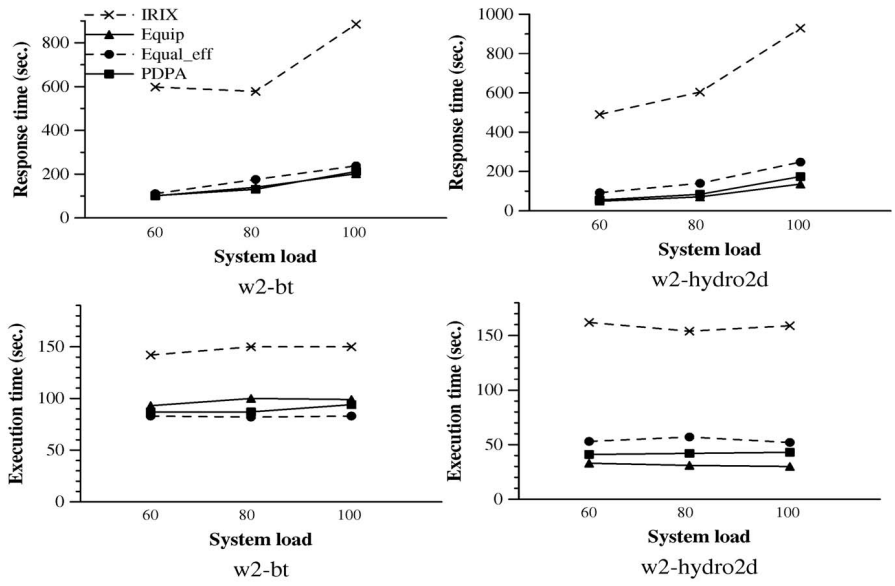

Fig. 6. Results from workload 2. Response and execution time.

average to each type of application is 20 cpus to bt and 9 cpus to hydro2d. The allocation decided by Equipartition is the same to both applications (around 15). This better distribution results in a better execution time of the bt executed under PDPA compared with the bt executed under Equip. In this workload, PDPA outperforms Equipartition by 10 percent in both the response and execution times of bt, but in the case of hydro2ds, Equipartition outperforms PDPA between 23 percent and 30 percent.

In hydro2d, the execution time is slightly worse with PDPA compared with Equipartition, even though the response time is quite the same under both policies. This is 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 the results achieved by PDPA with the Equal_efficiency, we can see that PDPA outperforms the Equal_efficiency in both the response time and 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 percent in the response time of bt and by 58 percent in the response time of hydro2d. When setting the load to 100 percent, the Equal_efficiency has allocated 30 processors in average to bt and 10 processors in average to hydro2d.

In the next section, 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.

5.2.1 Changing the Multiprogramming Level

We executed the second workload setting the multiprogramming level to three and two applications to evaluate the capacity of PDPA to dynamically adjust the multiprogramming level. Fig. 7 shows the execution and response times of bt and hydro2d when executed with ml = 2, 3, and 4, under Equipartition and PDPA, when using different multiprogramming levels and with different system loads. We also show results for the complete workload. We put the execution time (black color) on top of the response time (gray color) to see the time consumed by each component.

From these graphs, 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

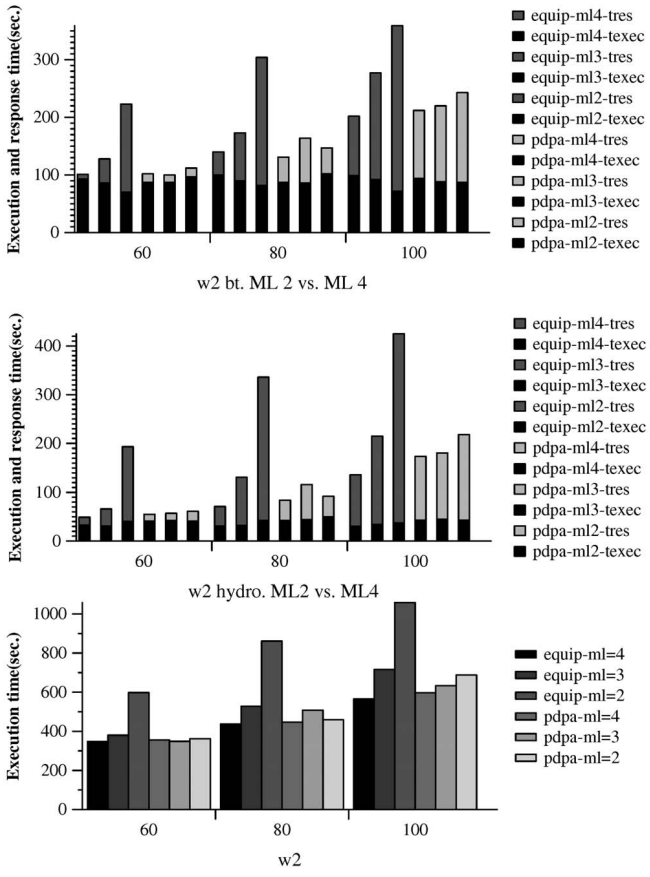

Fig. 7. Results from workload 2. Multiprogramming levels 2, 3, and 4 jobs.

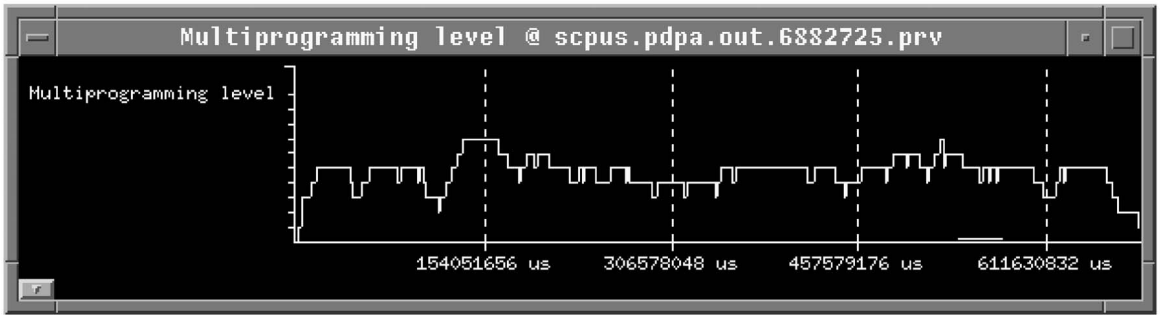

Fig. 8. Multiprogramming level decided by PDPA.

administrator: PDPA dynamically detects the optimal value for any 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 versus using a multiprograming level of two applications, in the cases of load = 80 percent and load = 100 percent.

The second conclusion that we can extract from this experiment is that PDPA is able to adjust the processor allocation of running applications by taking into account their performance. As we can see, when we set the multiprogramming level to two applications, Equipartition allocates the number of processors requested, resulting in better execution times per application.[3] But, this is neither a good result for the system performance nor an acceptable response time for the 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.

The multiprogramming level decided by PDPA reached up to six applications during the workload execution (with load = 100 percent).

Fig. 8 shows the dynamic multiprogramming level decided by PDPA when the load is set to 100 percent. The x-axis presents the time and the y-axis shows the multiprogramming level. The figure shows that PDPA adapts the multiprogramming level to the characteristics of the running applications, in such a way that it changes during the complete execution of the workload.

5.3 Workload 3

Fig. 9 shows the results obtained from the execution of the workload 3. The graphs in the top of the figure show the average response time (in seconds) of bt and apsi applications. The graphs at the bottom show execution times. This workload has been designed to evaluate the behavior of the policies when executing a workload where the 50 percent of the load is composed by scalable applications, and the other half does not scale at all.

In this kind of workload, 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 queuing system. However, since we have performed a previous manual tuning of the processor request, the processor allocation of running applications cannot be significantly improved (apsis could receive one processor instead of two, but this is not a significant change). In any case, the system can be significantly improved if the processor scheduler and the queuing system are coordinated to better use the system in those moments that either there is a single bt executing or there is no bt executing.

These results demonstrate our theory. If we observe the response time graphs, we can observe that PDPA significantly improves the remaining of evaluated policies because both bt and apsi 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 realize that the system is underutilized at times and that a new application can be started. We have analyzed results from PDPA, and we have found that the multiprogramming level was set up to 34 jobs in the experiments with the load set to 100 percent.

Analyzing the results in more detail, we can see that in this workload, PDPA outperforms Equipartition in a 600 percent in both the response time of bt and apsi at the expense of only the 30 percent of slowdown in the execution time in the case of bt and no slowdown in the case of the execution time of apsi. 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 and two processors to apsi. However, we can see that the execution time of bt under PDPA is better than under Equal_efficiency, even though bt received more processors than under PDPA. On average, PDPA improves the execution time of bt 20 percent. Comparing the response time, PDPA improves the response time achieved by bt under the Equal_efficiency 558 percent.

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

We have only executed the first case with the load set to the 60 percent because results were significant enough. Table 3 shows the results achieved in this case. We can see that PDPA significantly improves the Equipartition performance.

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

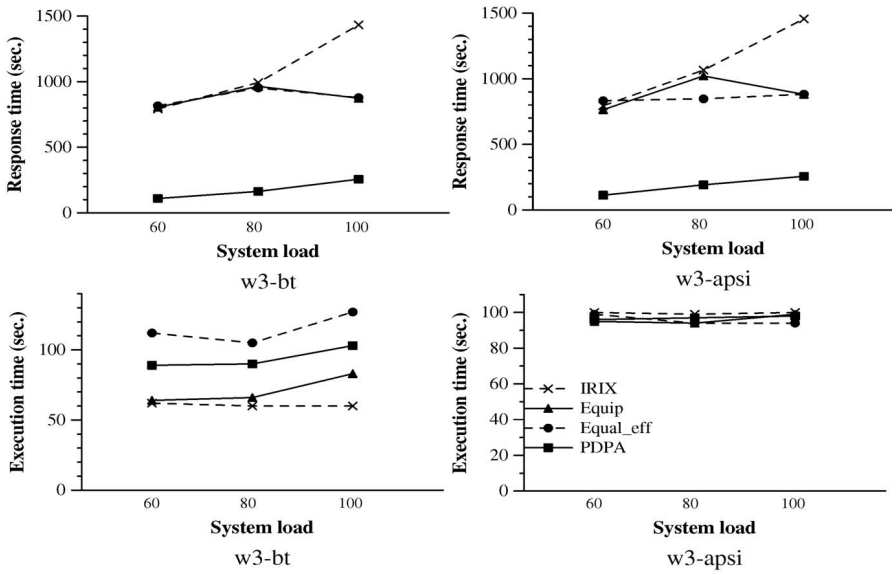

Fig. 9. Results from workload 3. Response and execution times.

5.4 Workload 4

Fig. 10 shows results from workload 4. We show the average response time for swim, bt, hydro2d, and apsi, and the average execution time for the four applications. This workload is a mix of the four types of applications, and each type receives the same amount of CPU percentage. As in the previous workloads, this is not the best case for PDPA because the processor allocation was tuned and the load is quite enough to fill the complete machine. However, we can also observe how the response time achieved by applications when executing under PDPA significantly improves results achieved by other policies, without significantly increasing the execution time.

We have analyzed the processor allocation decided in the case of load = 80 percent, and we have found that swim, bt, hydro2d, and apsi received 17, 20, 10, and 2 processors, respectively. Moreover, we have observed that the maximum multiprogramming level was set up to 14 applications in some periods of the workload execution.

It is surprising that swim receives less processors than bt, having better speedup. This is due to the fact that swim achieves the superlinear speedup in the first range of processors (between 8 and 16 processors). With the rest of processors the achieved speedup is also superlinear, but the relative speedup is not so high. On the other hand, bt has a more progressive scalability, and its relative speedup is better. For this reason, it is more beneficial for the system to allocate more processors to bt than to swim.

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

We compared the response time and the execution time achieved by applications under PDPA and Equal_efficiency. PDPA outperforms Equal_efficiency by 1,095 percent, 502 percent, and 442 percent in the response time of swim, bt, and hydro2d, respectively. And, Equal_efficiency outperforms PDPA by 16 percent, 8 percent, and 1 percent in the execution time of swim, bt, and hydro2d, respectively. However, this is at the expense of 52 percent more processors in the case of swim, 40 percent more processors in the case of bt, and 270 percent in the case of hydro2d. Analyzing these results, we can conclude that PDPA outperforms Equal_efficiency in this workload because we improved the response time of the applications between 500 percent and 1,000 percent, at the only expense of an slowdown in the execution time in the range of 1 percent to 16 percent.

As in the previous workload, we executed a different configuration of this workload, assuming that no previous

TABLE 3
Results from w3, apsi Requesting for 30 Processors (Not Tuned), Load = 60 Percent

	Bt		Apsi		Workload	ML
	Resp. time	Exec. time	Resp. time	Exec. time	Exec. time	
Equip	949 sec.	102 sec.	890 sec.	107 sec.	1993 sec.	4
PDPA	95 sec.	88 sec.	107 sec.	98 sec.	427 sec.	29
Speedup	998 %	15%	831%	9%	466%	

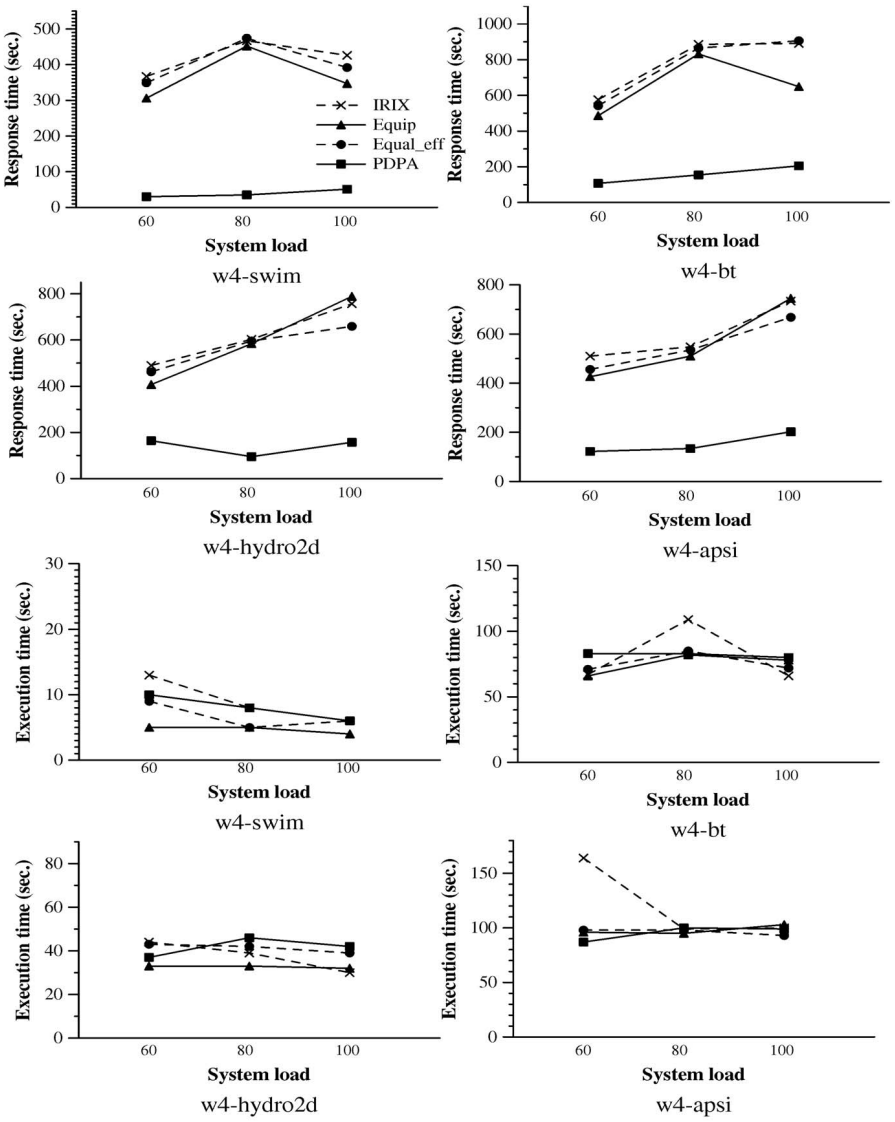

Fig. 10. Results of workload 4. Response and execution time.

tuning of applications has been performed and setting the request of all the applications to 30 processors (the only difference is apsi). Table 4 shows results for load = 60 percent. The last row shows the ratio of PDPA versus Equipartition. In those cases where Equipartition outperformed PDPA, we have reported negative speedups. We can see how PDPA has outperformed the complete execution time of the workload by 282 percent, and the individual response time of applications from a 109 percent up to a 2,830 percent. We can also see that all these benefits have been achieved by only sacrificing a maximum of 30 percent in the execution time of some applications. Analyzing the execution trace file (not presented in the paper), we observed that applications under PDPA have consumed half of the CPU time than under Equipartition to execute the same amount of work, although the CPU

TABLE 4
Results from Workload 4 Not Tuned, Load = 60 Percent

	swim, req=30		bt, req=30		hydro, req=30		apsi, req=30		Total
	T.exec	T.resp.	T.exec.	T.resp.	T.exec.	T.resp.	T.exec.	T.resp.	T.exec.
Equip	6sec.	368sec.	101sec.	568sec.	32sec.	453sec.	104sec.	773sec.	126sec.
PDPA	8sec.	13sec.	81sec.	92sec.	37sec.	45sec.	98sec.	109sec.	496sec.
%	-30%	2830%	-24%	617%	-15%	1006%	6%	109%	282%

utilization under Equipartition has been around the 100 percent and around 70 percent under PDPA.

6 Final Remarks and Future Work

In this paper, we have presented PDPA, a coordinated scheduling policy that bases its decisions on the performance of applications calculated at runtime. With regard to the processor allocation, PDPA tries to allocate to each application the maximum number of processors to reach a specified target efficiency (*target_eff*). PDPA also implements a multiprogramming-level policy that allows the execution of a new application if there are idle processors, and when the allocation of all the running applications is stable or they do not require more processors.

The results obtained 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 introduced a maximum overhead of 10 percent in the total execution time of the workload, and a maximum of 30 percent 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 adapting the multiprogramming level to the workload characteristics. We executed these kind of workloads with and without previous tuning, and we observed that benefits provided by the two points are orthogonal and complementary. In these kind of workloads, PDPA outperformed Equipartition in about 400 percent of the total execution time. For these reasons, we can conclude that by dynamically measuring the performance of applications and imposing a target efficiency we give PDPA a robustness that the rest of policies have evaluated.

The results also show that the first level of coordination between the processor scheduler and the runtime library, and the quality of the processor scheduler, are also very important, as the differences between the Equipartition and the native IRIX scheduler demonstrate. Another important issue is the processor scheduling stability. The processor allocation must be maintained as stable as possible because a high number of reallocations degrades the application and the system performance.

Finally, we also observed that, for the scheduling policies that use extrapolated values to take their decisions, it is very important to verify that these values correspond with the real ones and react accordingly.

Our future work goes in the direction of extending the spectrum of programming models and architectures on which PDPA could be applied. On one hand, we are starting the work with message passing applications based on MPI or MPI+OpenMP. MPI are usually tight to a specific number of processors (i.e., the NAS benchmarks). Introducing a second level of parallelism based on OpenMP makes them more malleable. One first approach for MPI+OpenMP applications is to control the number of processors given to each MPI process to run OpenMP threads. This way, one can achieve better load balancing of the work done for each MPI process. A second approach for MPI applications is to limit the number of processors used by such applications by folding their processes on a number of processors using a binding mechanism and let them run under the policy of the operating system. This approach can be completed by intercepting the calls to the MPI library and suggesting yields of the physical processor at message reception.

On the other hand, we are also extending this work to run on clusters of SMP's, where the resources are physically distributed. We think that adding cooperation between the scheduling policies running on the different machines, we can control enough the scheduling of the physical processors, so that each application is given resources at the same time on all the nodes.

Acknowledgments

The Spanish Ministry of Education has supported this work under grant CYCIT TIC2001-0995-C02-01, and ESPRIT Project POP (IST-2001-33071). The research described in this work has been developed using the resources of the European Center of Parallelism of Barcelona (CEPBA).

References

[1] T.B. Brecht and K. Guha, "Using Parallel Program Characteristics in Dynamic Processor Allocation," *Performance Evaluation,* nos. 27-28, pp. 519-539, 1996.

[2] J. Corbalan, X. Martorell, and J. Labarta, "Dynamic Performance Analysis: SelfAnalyzer," Technical Report UPC-DAC-2002-54, http://www.ac.upc.es/pub/reports/DAC/2002/UPC-DAC-2002-54.ps.Z, 2004.

[3] S.-H. Chiang, R.K. Mansharamani, and M.K. Vernon, "Use of Application Characteristics and Limited Preemption for Run-to-Completion Parallel Processor Scheduling Policies," *Proc. ACM SIGMETRICS Conf.,* pp. 33-44, May 1994.

[4] D.L. Eager, J. Zahorjan, and E.D. Lawoska, "Speedup versus Efficiency in Parallel Systems," *IEEE Trans. Computers,* vol. 38, no. 3, pp. 408-423, Mar. 1989.

[5] F. Freitag, J. Corbalan, and J. Labarta, "A Dynamic Periodicity Detector: Application to Speedup Computation," *Proc. 15th Int'l Parallel and Distributed Processing Symp. (IPDPS 2001),* pp. 2-8, Apr. 2001.

[6] H. Jin, M. Frumkin, and J. Yan, "The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance," Technical Report: NAS-99-011, 1999.

[7] B. Hamidzadeh and D.J. Lilja, "Self-Adjusting Scheduling: An On-Line Optimization Technique for Locality Management and Load Balancing," *Proc. Int'l Conf. Parallel Processing,* vol. 2, pp. 39-46, 1994.

[8] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris, "DiP : A Parallel Program Development Environment," *Proc. Second Int'l EuroPar Conf.,* Aug. 1996.

[9] S.T. Leutenegger and M.K. Vernon, "The Performance of Multiprogrammed Multiprocessor Scheduling Policies," *Proc. ACM SIGMETRICS Conf.,* pp. 226-236, May 1990.

[10] S. Majumdar, D.L. Eager, and R.B. Bunt, "Characterisation of Programs for Scheduling in Multiprogrammed Parallel Systems," *Performance Evaluation,* vol. 13, pp. 109-130, 1991.

[11] X. Martorell, "Dynamic Scheduling of Parallel Applications on Shared-Memory Multiprocessors," PhD thesis, Technical Univ. of Catalonia (UPC), July 1999.

[12] C. McCann, R. Vaswani, and J. Zahorjan, "A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors," *ACM Trans. Computer Systems,* vol. 11, no. 2, pp. 146-178, May 1993.

[13] X. Martorell, J. Labarta, N. Navarro, and E. Ayguade, "Nano-Threads Library Design, Implementation and Evaluation," Technical Report: UPC-DAC-1995-33, Dept. d'Arquitectura de Computadors-UPC, Sept. 1995.

[14] X. Martorell, J. Labarta, N. Navarro, and E. Ayguade, "A Library Implementation of the Nano-Threads Programming Model," *Proc. Second Int'l Euro-Par Conf.,* vol. 2, pp. 644-649, Aug. 1996.
[15] T.D. Nguyen, J. Zahorjan, and R. Vaswani, "Parallel Application Characterization for Multiprocessor Scheduling Policy Design," *Proc. Workshop Job Scheduling Strategies for Parallel Processing,* 1996.
[16] T.D. Nguyen, J. Zahorjan, and R. Vaswani, "Using Runtime Measured Workload Characteristics in Parallel Processors Scheduling," *Proc. Workshop Job Scheduling Strategies for Parallel Processing,* 1996.
[17] E.W. Parsons and K.C. Sevcik, "Benefits of Speedup Knowledge in Memory-Constrained Multiprocessor Scheduling," *Performance Evaluation,* nos. 27-28, pp. 253-272, 1996.
[18] K.C. Sevcik, "Application Scheduling and Processor Allocation in Multiprogrammed Parallel Processing Systems," *Performance Evaluation,* vol. 19, nos. 1/3, pp. 107-140, Mar. 1994.
[19] K.C. Sevcik, "Characterization of Parallelism in Applications and Their Use in Scheduling," *Proc. ACM SIGMETRICS Conf.,* pp. 171-180, May 1989.
[20] A. Serra, N. Navarro, and T. Cortes, "DITools: Application-Level Support for Dynamic Extension and Flexible Composition," *Proc. USENIX Ann. Technical Conf.,* pp. 225-238, June 2000.
[21] Standard Performance Evaluation Corp., SPEC CPU95 Benchmarks, http://www.spec.org/osg/cpu95, 1995.
[22] The Standard Workload Format, http://www.cs.huji.ac.il/labs/parallel/workload/swf.html, 2004.
[23] M.J. Voss and R. Eigenmann, "Reducing Parallel Overheads through Dynamic Serialization," *Proc. 13th Int'l Parallel and Distributed Processing Symp.,* pp. 88-92, 1999.
[24] J.B. Weissman, L.R. Abburi, and D. England, "Integrated Scheduling: The Best of Both Worlds," *J. Parallel and Distributed Computing,* vol. 63, pp. 649-668, 2003.
[25] Workload logs, http://people.ac.upc.es/juli, 2004.

Julita Corbalan received the engineering degree in computer science in 1996 and the PhD degree in computer science in 2002, both from the Technical University of Catalunya (UPC), Spain. Since 2001, she has been lecturing on operating systems in the Computer Architecture Department at the UPC. Her research interests includes processor management of openmp, mpi, and mpi+openmp applications, openmp runtime management, and high performance computing oriented to grid. She has participated in several long-term research projects with other universities and industries, mostly in the framework of the European Union ESPRIT and IST programs. She is currently participating in the HPC-Europa project.

Xavier Martorell received the engineering degree in computer science in 1991 and the PhD degree in computer science in 1999, both from the Technical University of Catalunya (UPC), Spain. Since 1992, he has been lecturing on operating systems and parallel runtime systems. Currently, and since 2001, he is an associate professor in the Computer Architecture Department at UPC. His research interests cover the areas of operating systems, runtime systems, and compilers for high-performance multiprocessor systems. He has participated in several long-term research projects with other universities and industries, mostly in the framework of the European Union ESPRIT and IST programs, and also in collaboration with the IBM T.J. Watson Research Center.

Jesus Labarta has been a full professor of computer architecture at the Technical University of Catalunya (UPC), Spain, since 1990. Since 1981, he has been lecturing on computer architecture, operating systems, computer networks, and performance evaluation. His research interest has been centered on parallel computing, covering areas from multiprocessor architecture, memory hierarchy, parallelizing compilers, operating systems, parallelization of numerical kernels, GRID environments, and performance analysis and prediction tools. He has lead the technical work of UPC in 15 industrial R+D projects. Since 1995, he has been the director of CEPBA where he has pushed the development of own technology with products such as the Paraver and Dimemas performance analysis tools, the NANOS OpenMP environment, and the Grid Supersacalar environment. At CEPBA, he has also been highly motivated by the promotion of parallel computing into industrial practice, and especially within SMEs. In this line, he has been responsible for three technology transfer cluster projects where his team managed 28 subprojects.