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.

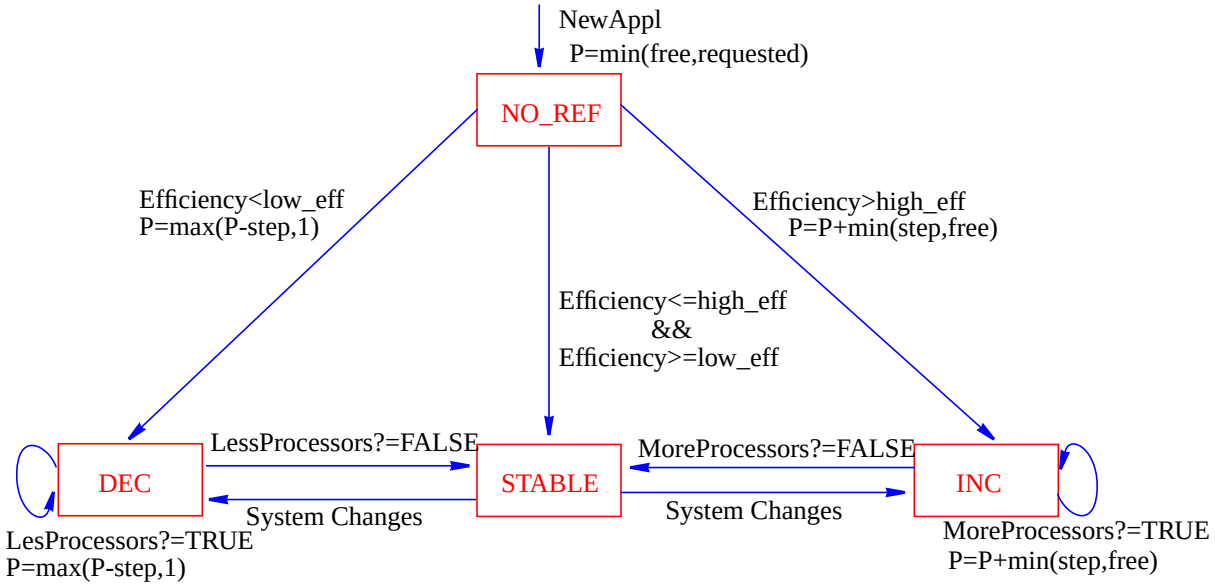

Figure 5.1: *PDPA*: Application state diagram

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.

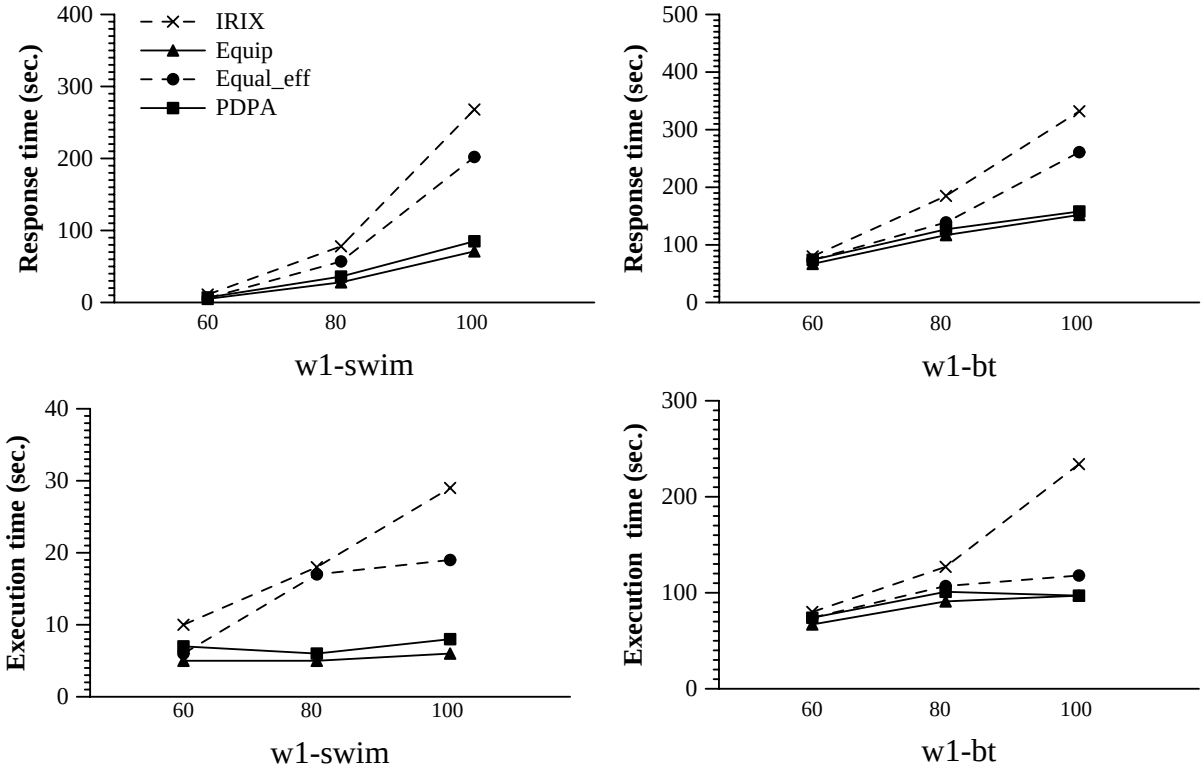

Figure 5.9: Results of workload 1, M.L.=4

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.

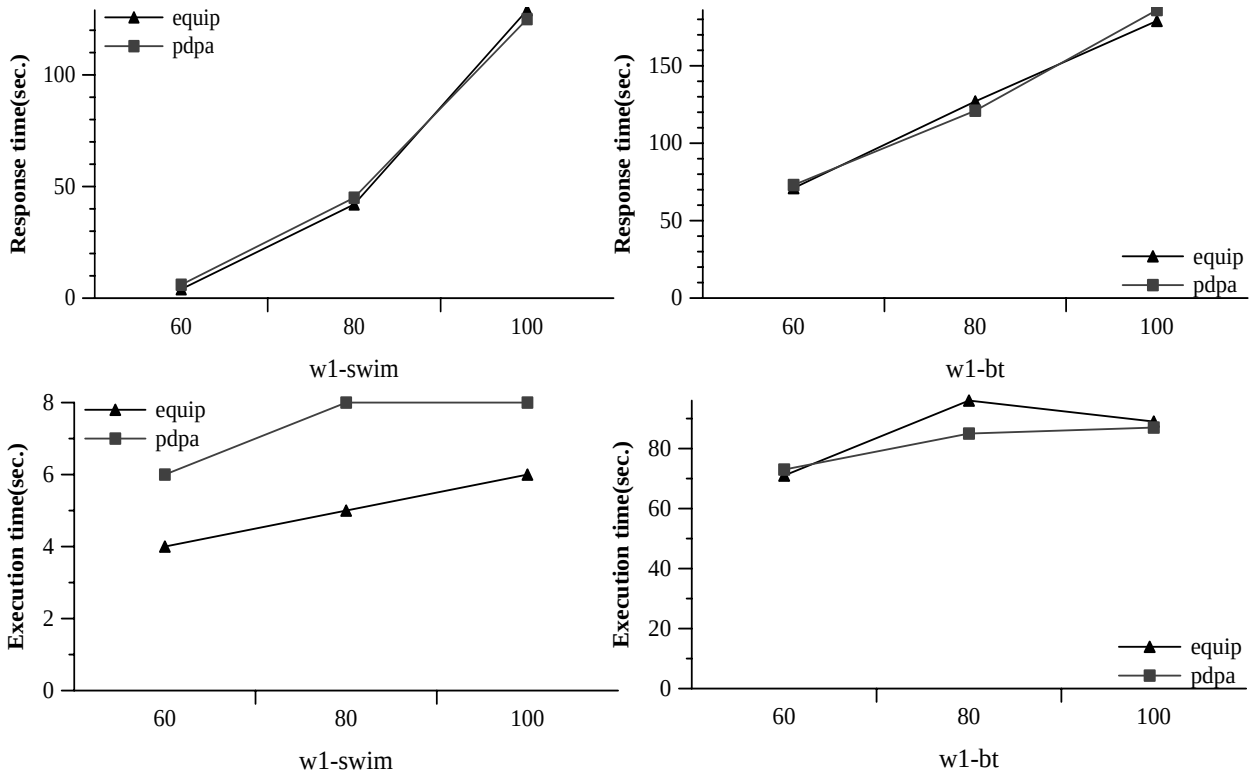

Figure 5.10: Results of workload 1, M.L.=3

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).

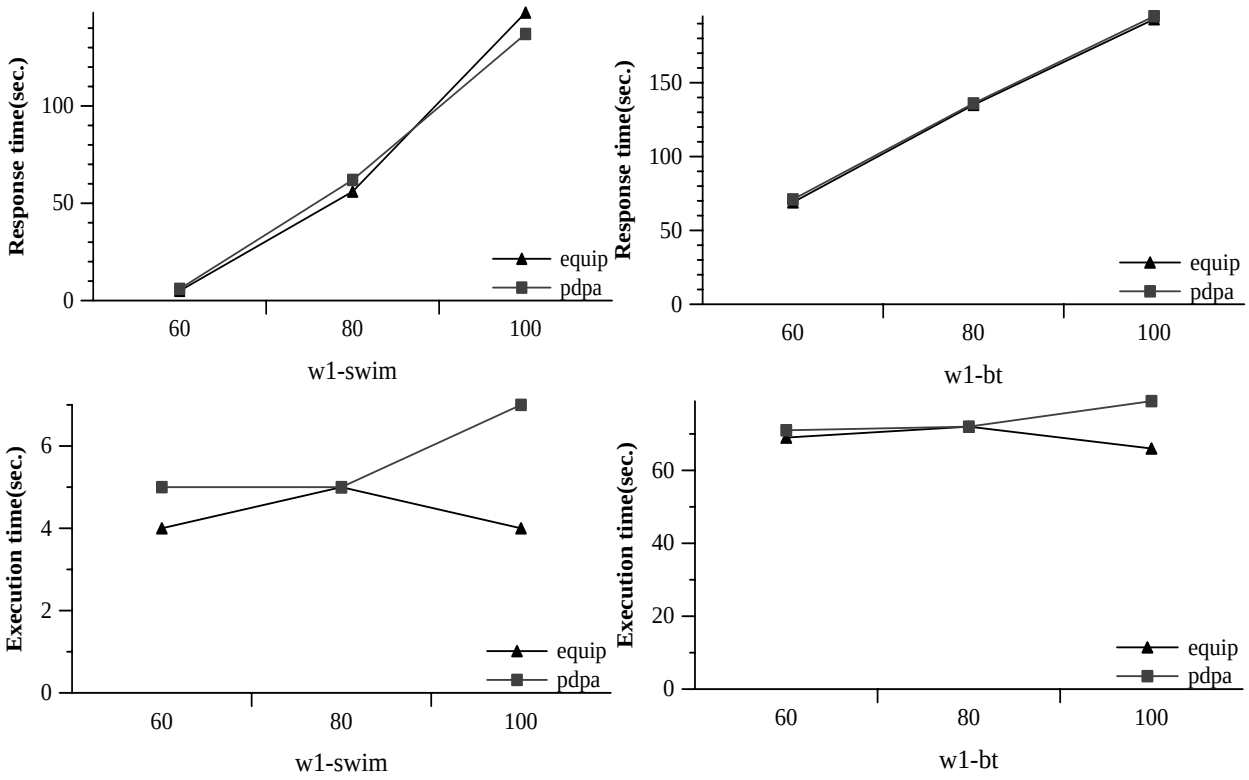

Figure 5.11: Results from workload 1, M.L.=2

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.

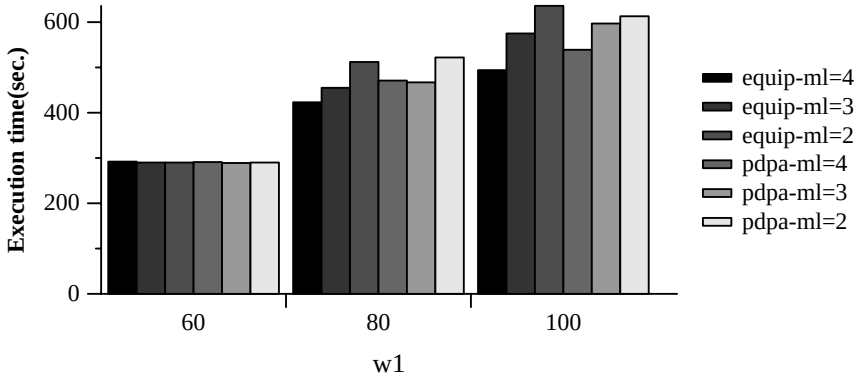

Figure 5.12: Workload execution time: Equipartition vs. PDPA, (workload 1)

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.

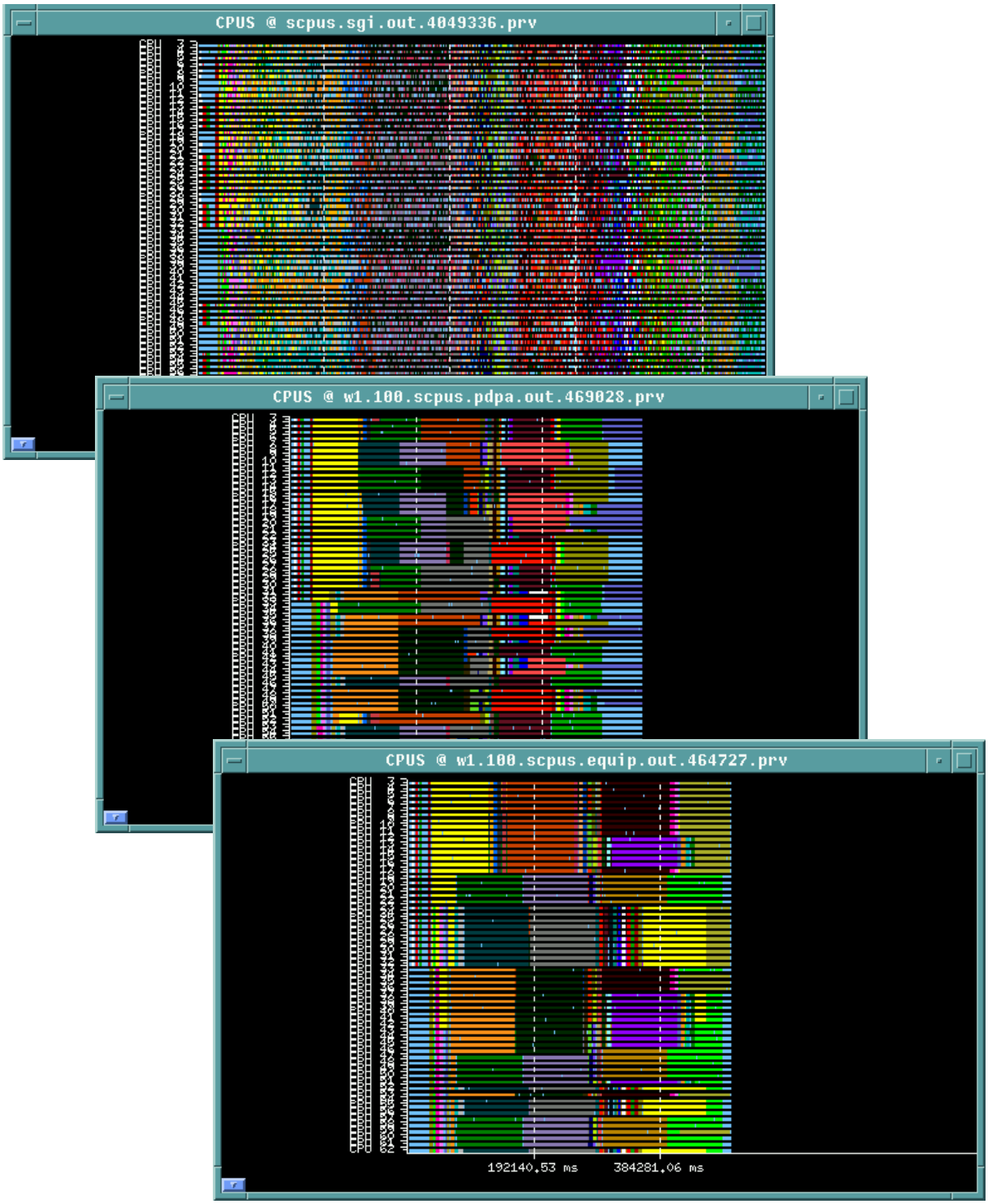

Figure 5.13: Execution views for workload 1 under IRIX, PDPA, and Equipartition (load=100%)

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%.

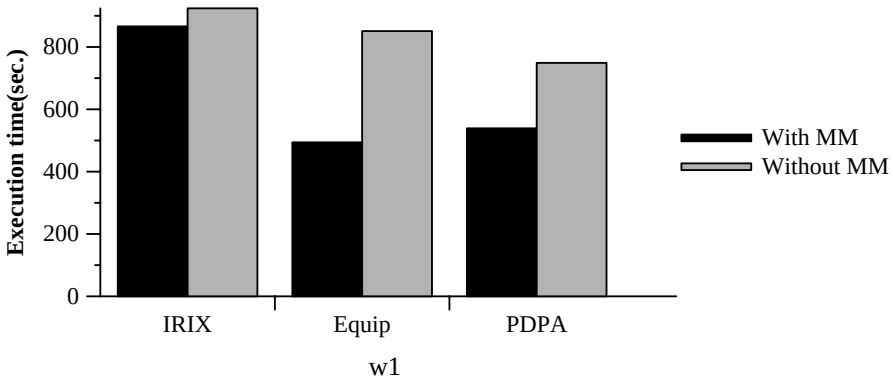

Figure 5.14: Execution time of workload 1 with and without memory migrations

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.

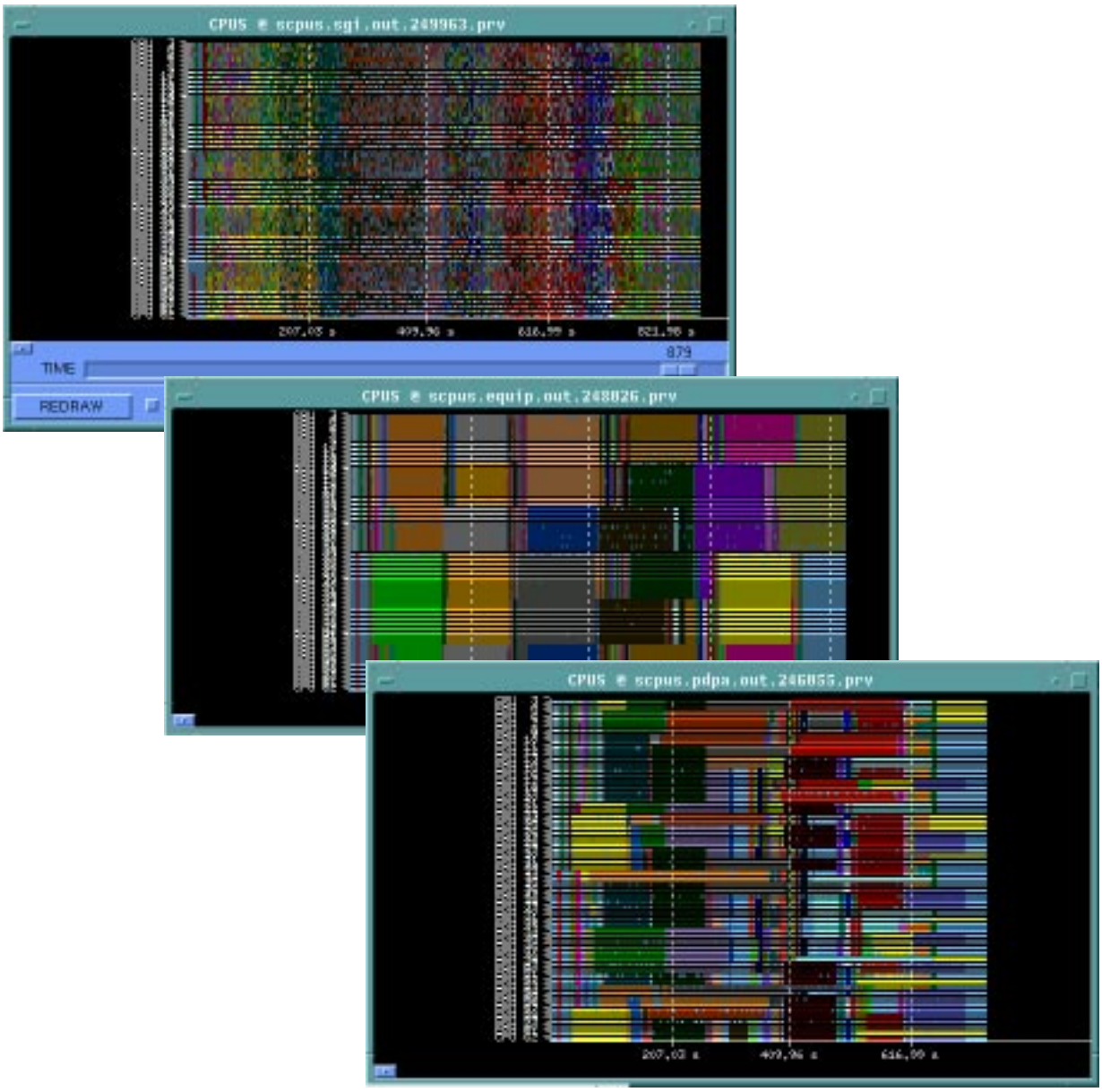

Figure 5.15: Execution of workload 1 without memory migrations under IRIX, Equipartition, and PDPA (load=100%)

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.

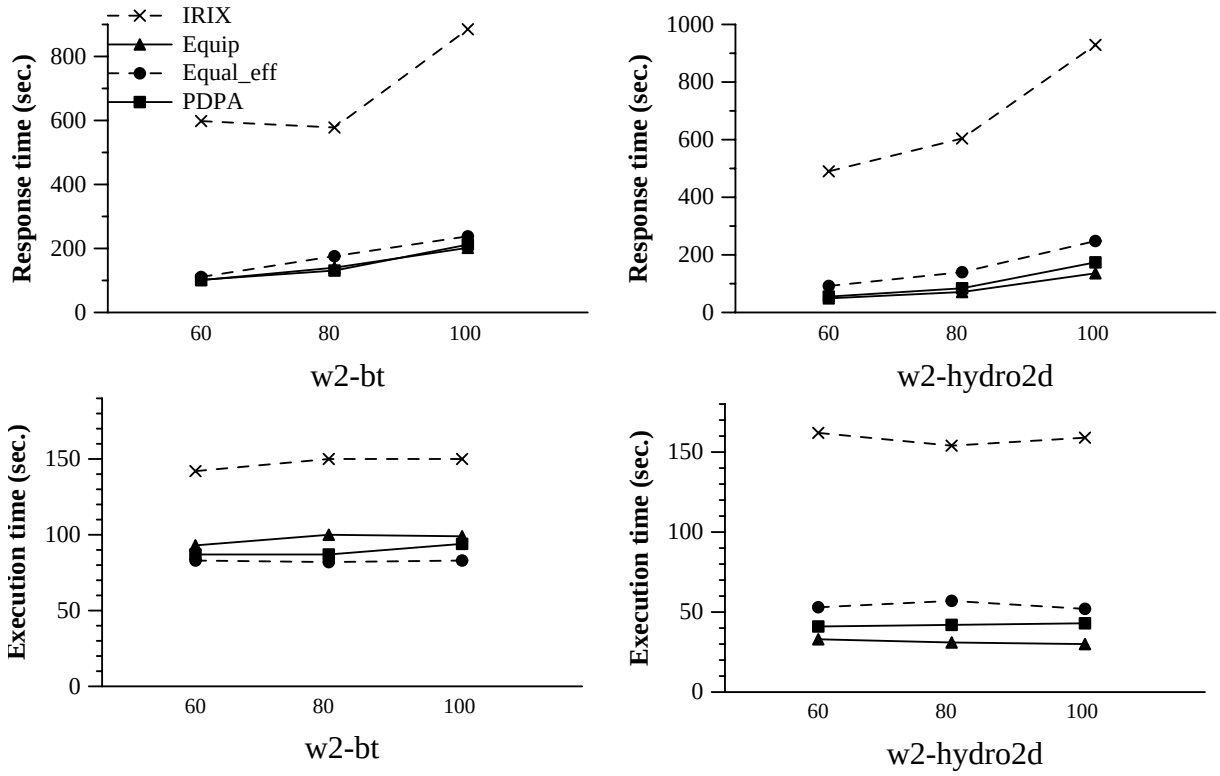

Figure 5.16: Results from workload 2, M.L.=4

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.

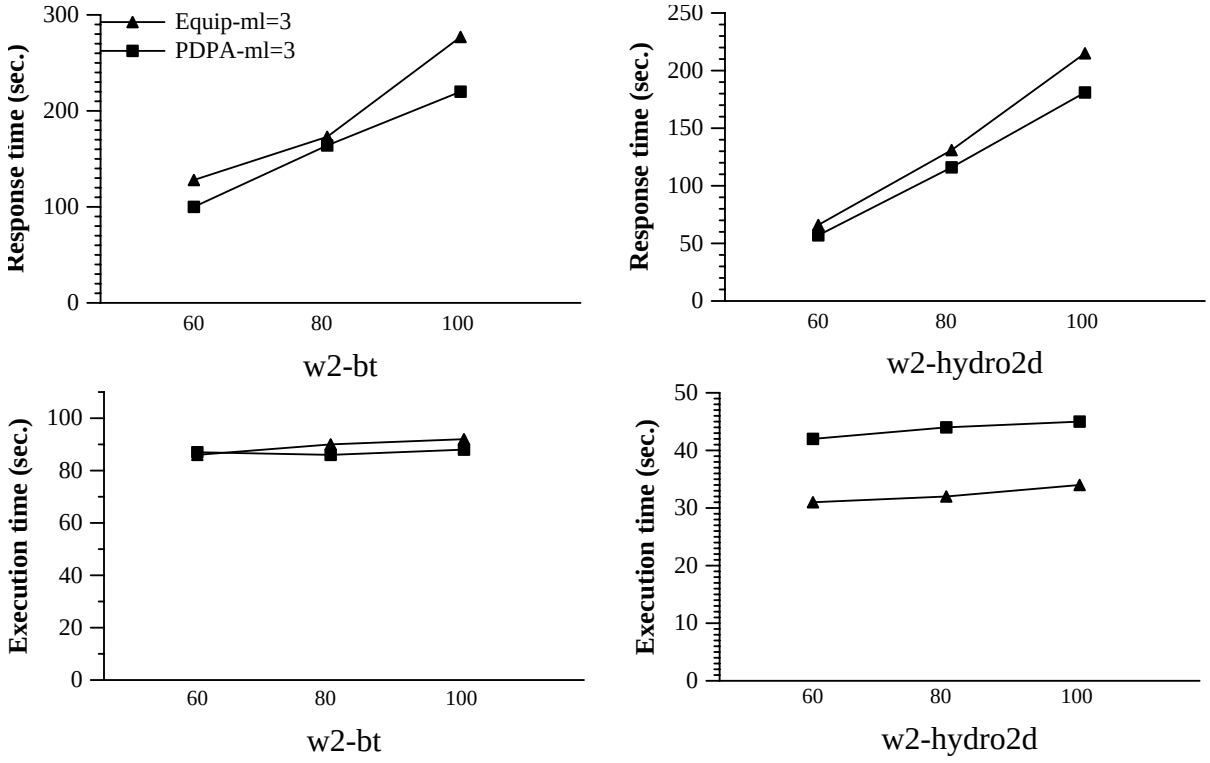

Figure 5.17: Results from workload 2, M.L.=3

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.

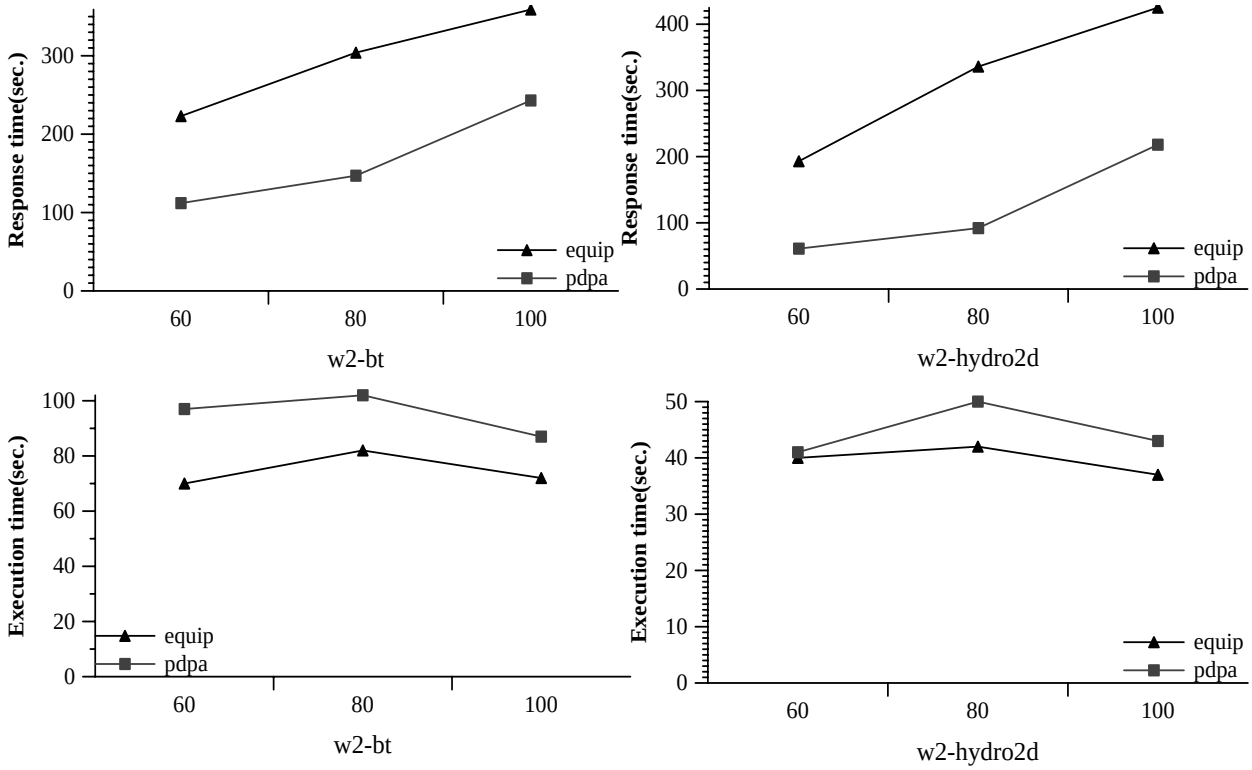

Figure 5.18: Results from workload 2, M.L.=2

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.

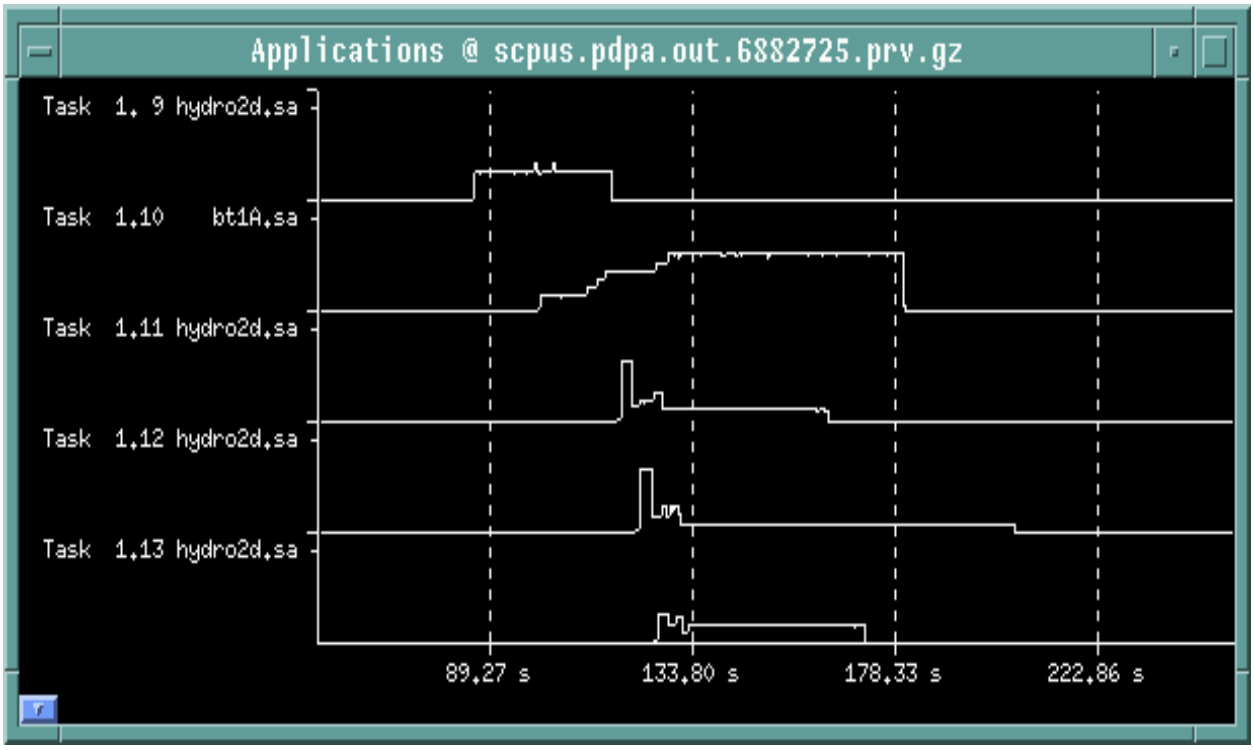

Figure 5.19: Processor allocation decided by PDPA (subset of applications)

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.

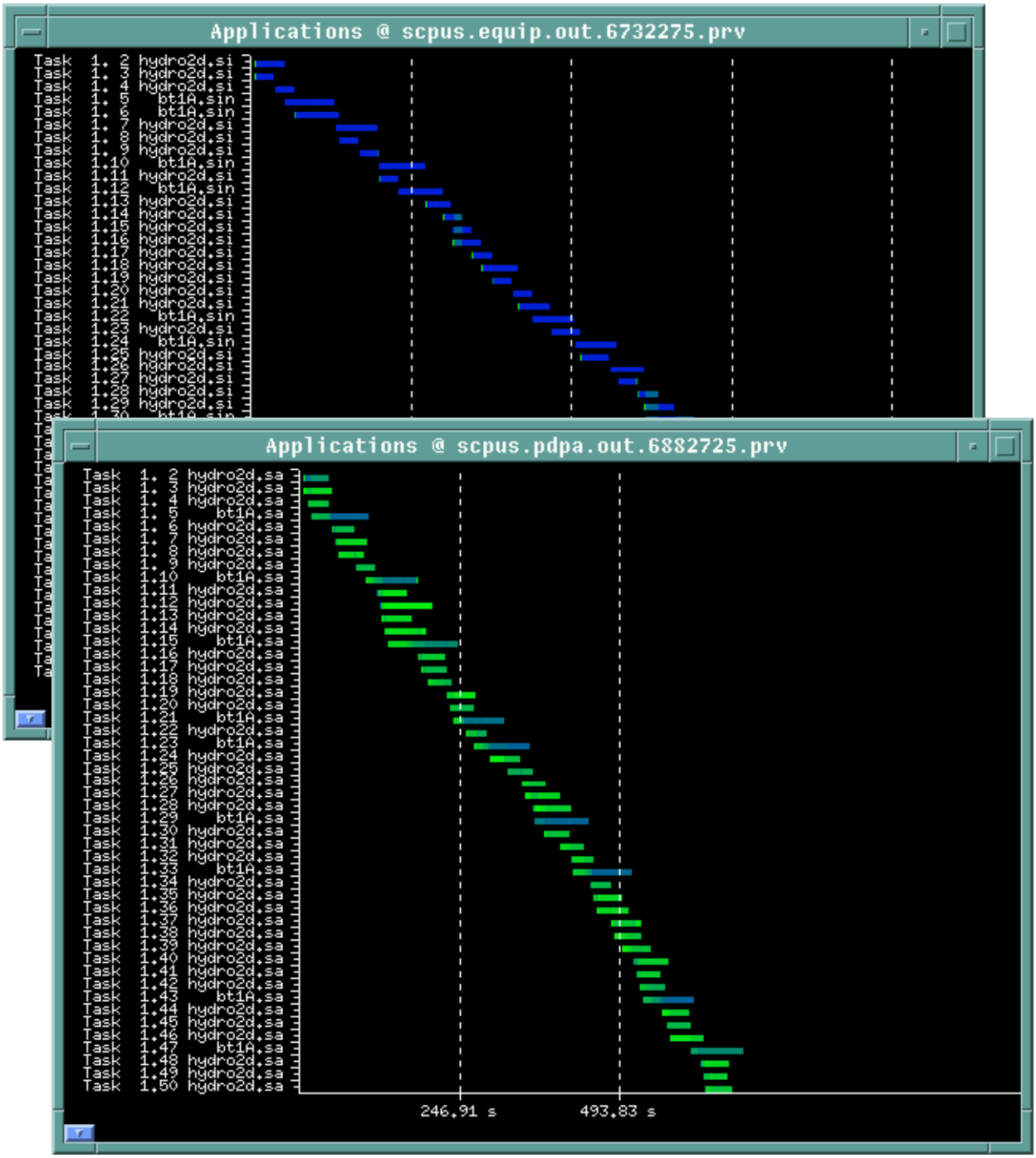

Figure 5.20: Workload 2, processor allocation decided by Equipartiton and PDPA (M.L=2, load=100%)

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.

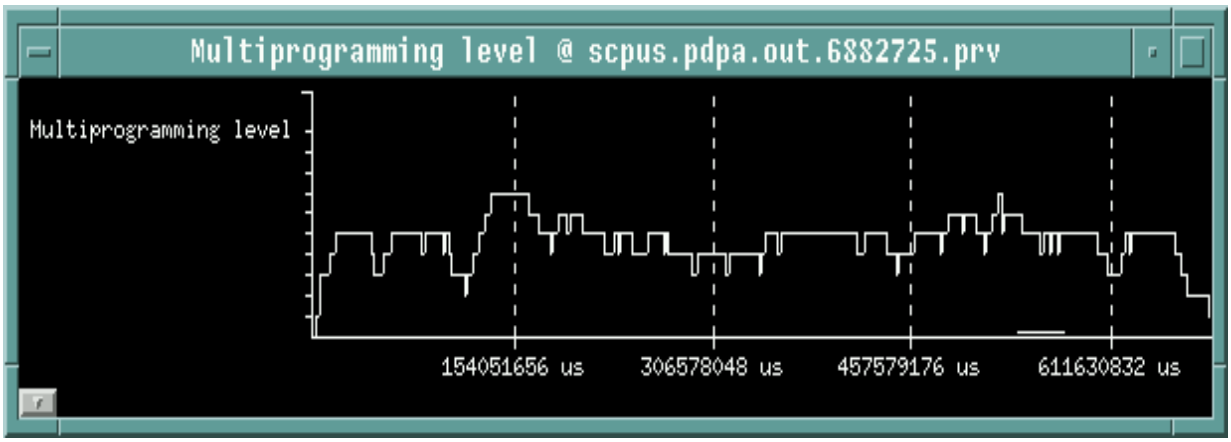

Figure 5.21: Multiprogramming level decided by PDPA

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.

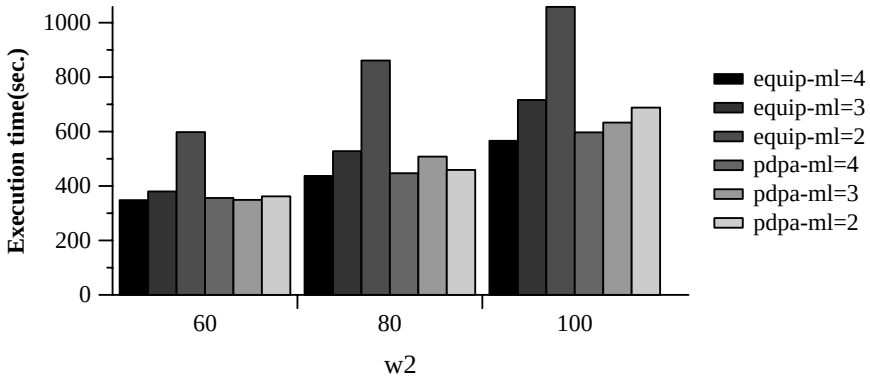

Figure 5.22: Workload 2, Equipartition vs. PDPA

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.

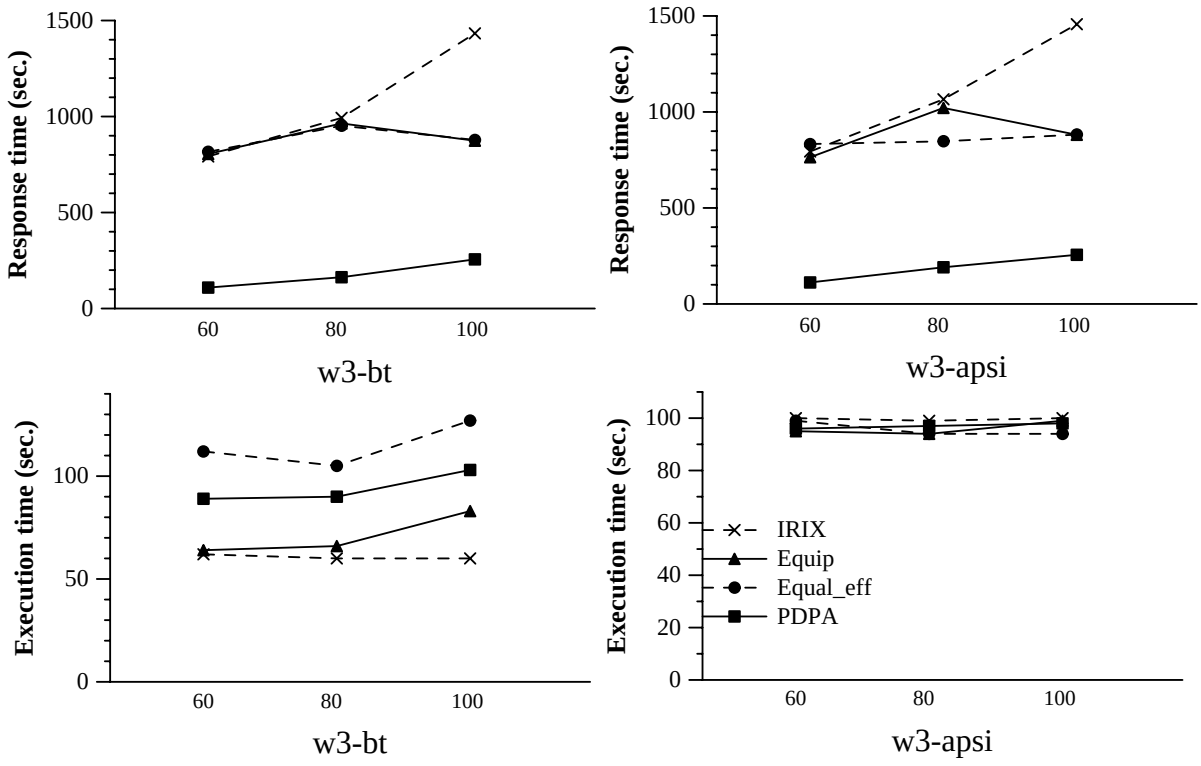

Figure 5.23: Results from workload 3

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.

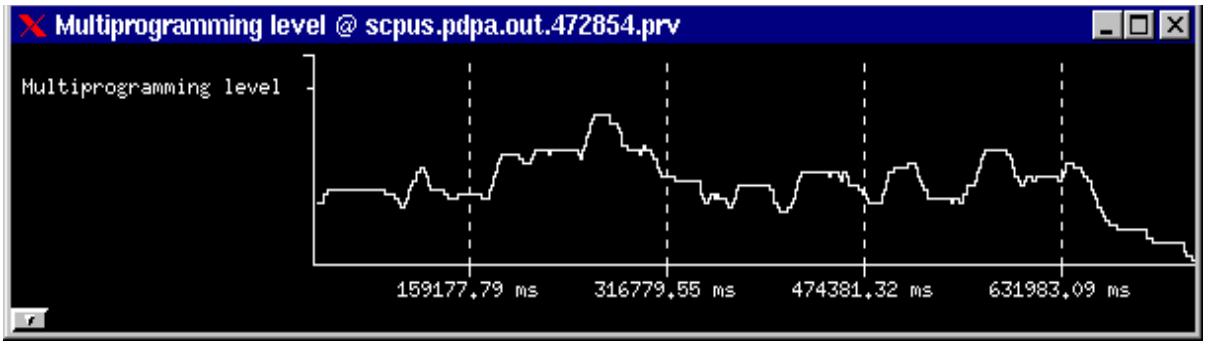

Figure 5.24: Multiprogramming level decided by PDPA, W3 M.L.=4, load=100%

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.

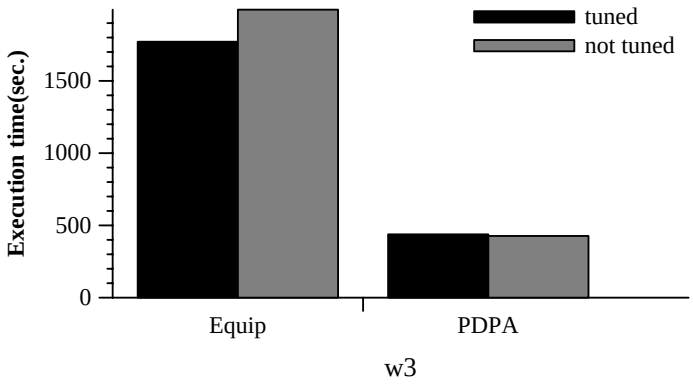

Figure 5.25: Workload 3, tuned vs. not tuned (load=60%)

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%).

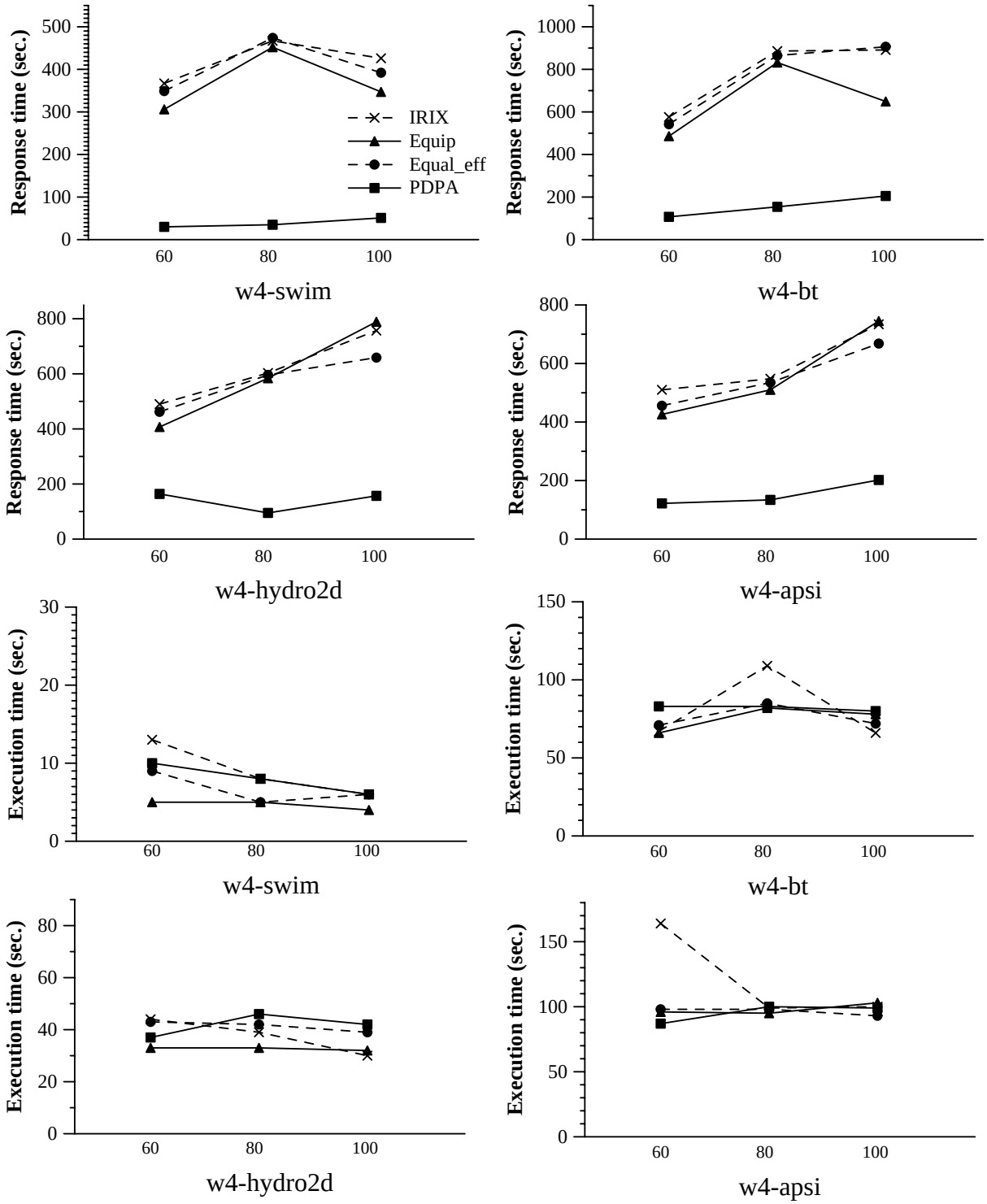

Figure 5.26: Results of workload 4

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.

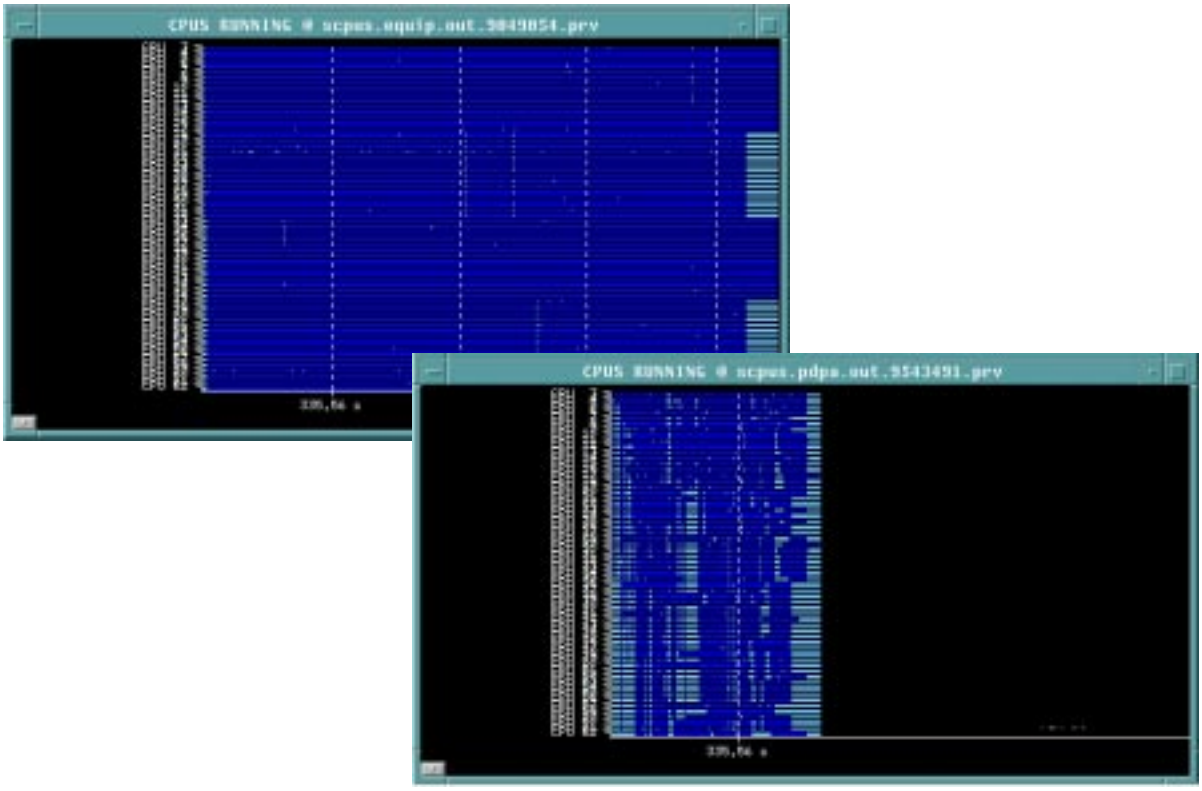

Figure 5.27: Workload 4. CPU utilization, Equipartition vs. PDPA, not tuned. Load=80%

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.

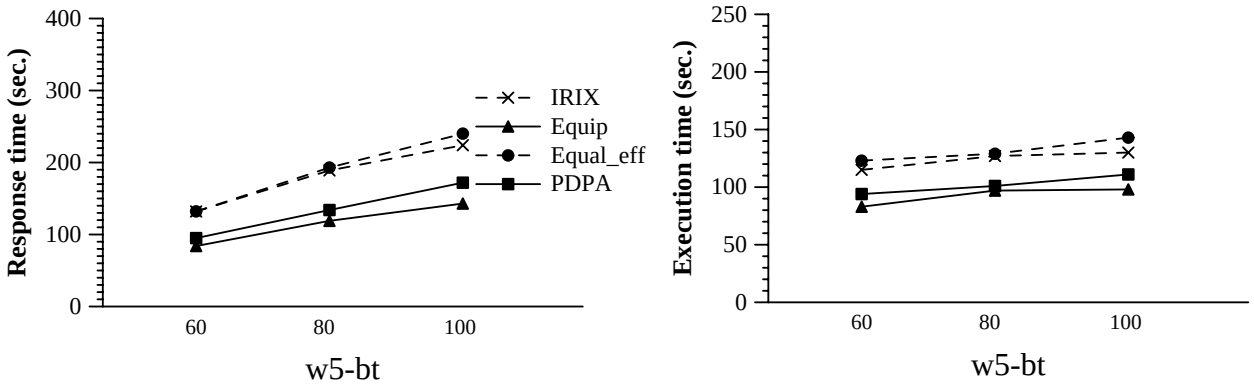

Figure 5.28: Results from workload 5

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.

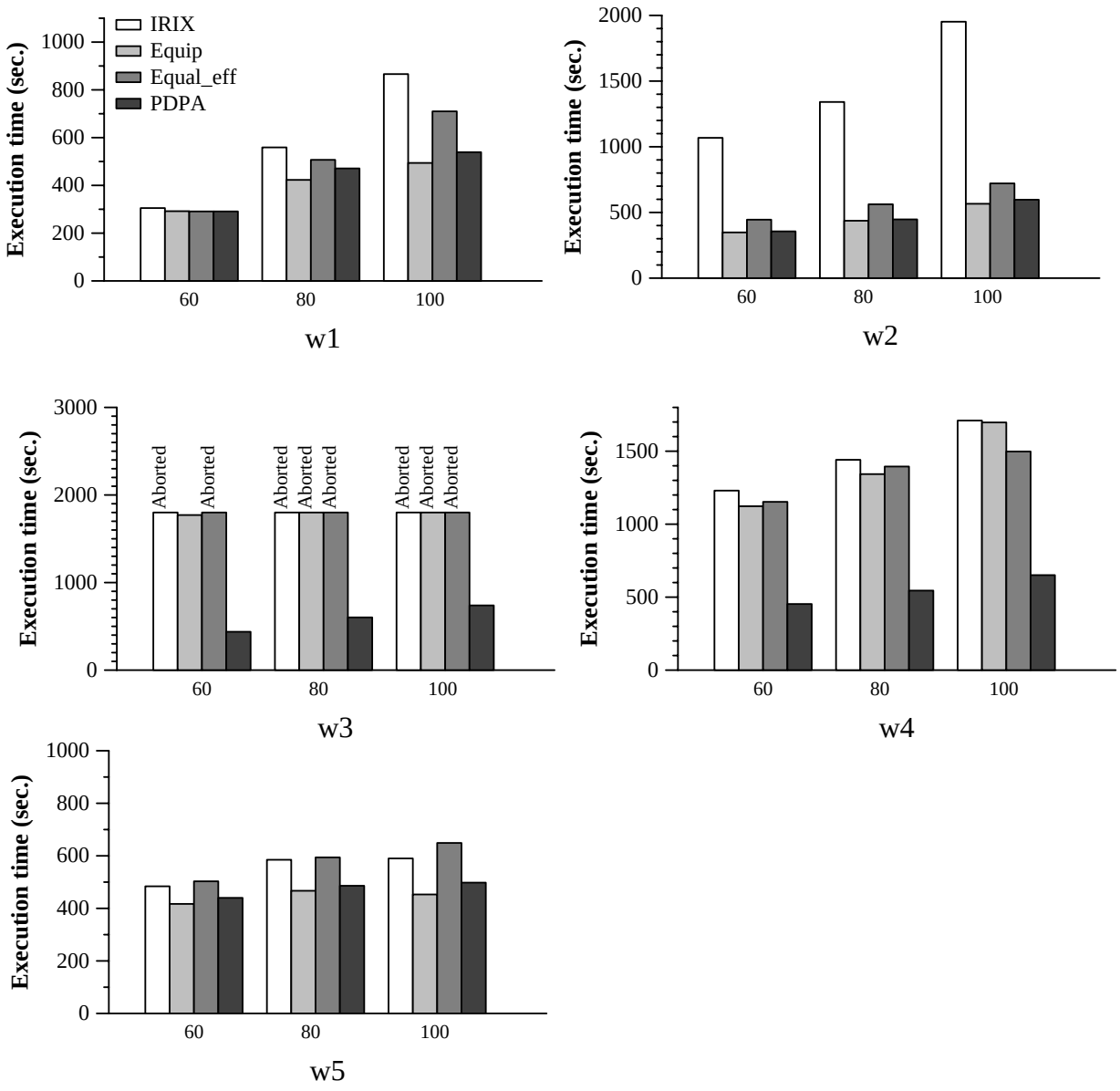

Figure 5.29: Execution time of the five workloads evaluated in this Thesis with M.L.=4

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.