
Performance-Driven Multiprogramming Level

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

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

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

6.1 Introduction

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

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

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

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

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

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

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

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

6.2 Performance-Driven Multiprogramming Level

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

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

to previously proposed processor scheduling policies.

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

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

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

6.2.1 Processor scheduling policy scheme

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

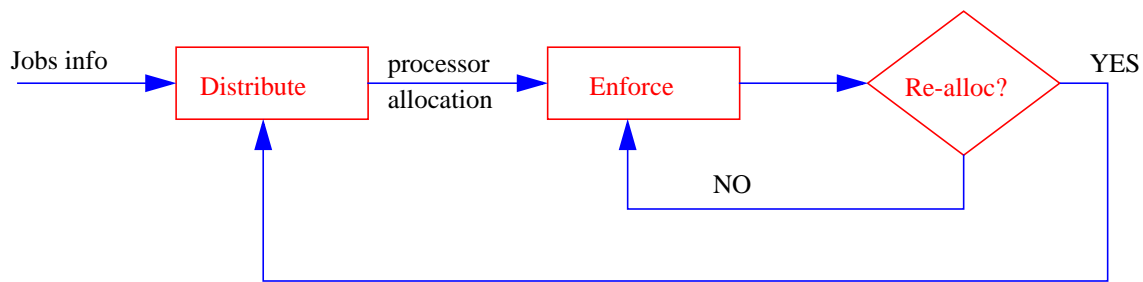


Figure 6.1: Typical processor scheduling policies behavior

Performance-Driven Multiprogramming Level: Processor Allocation

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

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

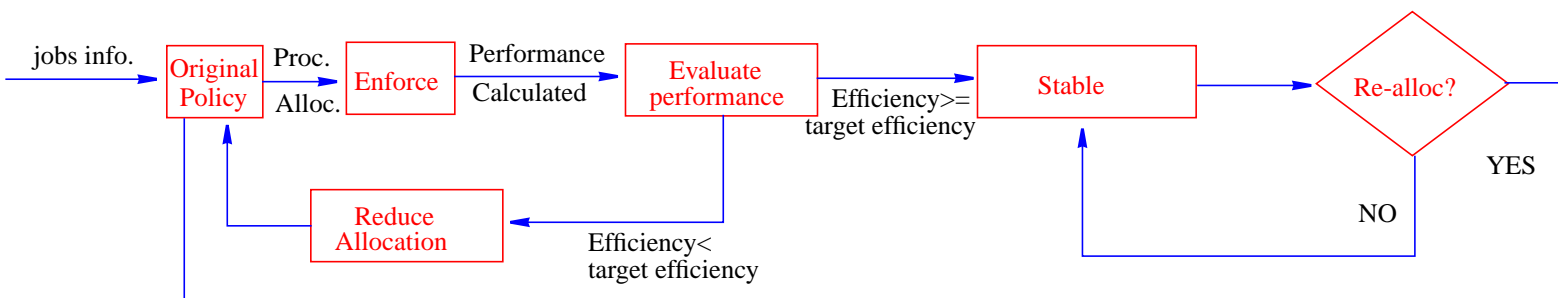


Figure 6.2: Performance-Driven Multiprogramming Level: processor allocation

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

Performance-Driven Multiprogramming Level: Multiprogramming level policy

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

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

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

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

6.3 Scheduling policies

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

6.3.1 Equipartition

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

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

Figure 6.3: Equipartition algorithm

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

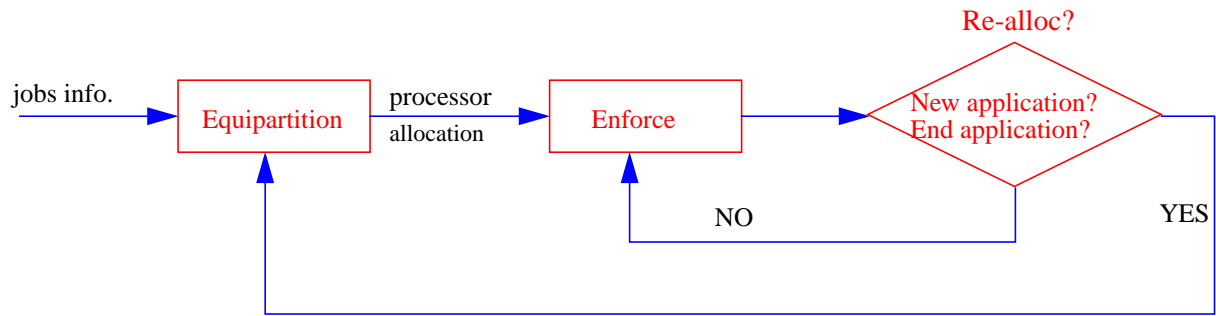


Figure 6.4: Equipartition scheme

6.3.2 Equip++

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

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

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

```

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

```

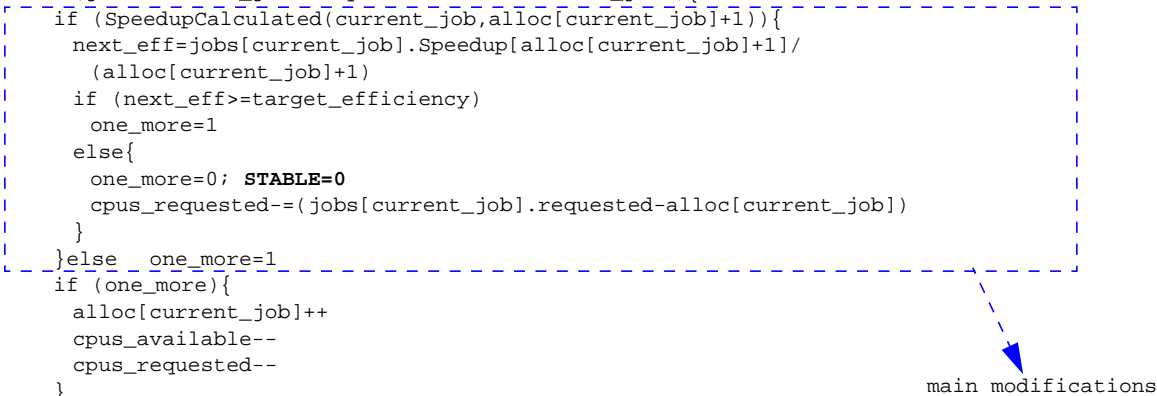


Figure 6.5: Equip++ algorithm

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

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

6.3.3 Equal_efficiency

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

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

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

Figure 6.6: Extrapolated efficiency formulation

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

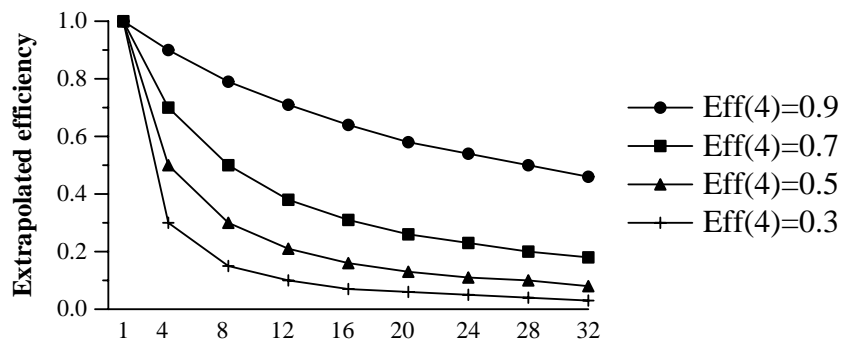


Figure 6.7: Extrapolated efficiency curves

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

Looking at the graph shown in Figure 6.7, it seems that this formulation works quite well. However, we have found several problems when using it. The two main problems related with its use is that it does not accept efficiency values greater or equal to 1.0, and that it does not accept the efficiency of one processor as initial value to extrapolate the curve. For instance, if we introduce as initial value $\text{efficiency}(4)=1.0$, the efficiency values calculated are invalid floating points values (*nan*'s). If we introduce as initial value the efficiency of one processor¹, the extrapolated curve calculates all the values equal to 0.0. Finally, if we introduce the initial value of (for instance) four processors set to 1.1, the extrapolated efficiency with 32 processors is 16.5.

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

```

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

```

Figure 6.8: `Equal_efficiency` algorithm

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

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

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

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

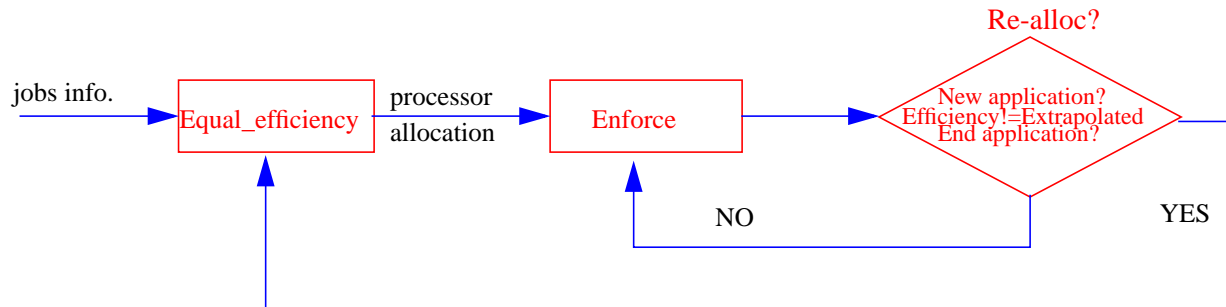


Figure 6.9: Equal_efficiency scheme

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

6.3.4 Equal_eff++

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

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

```

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

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

```

main modifications

Figure 6.10: Equal_eff++ algorithm

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

6.4 Taxonomy

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

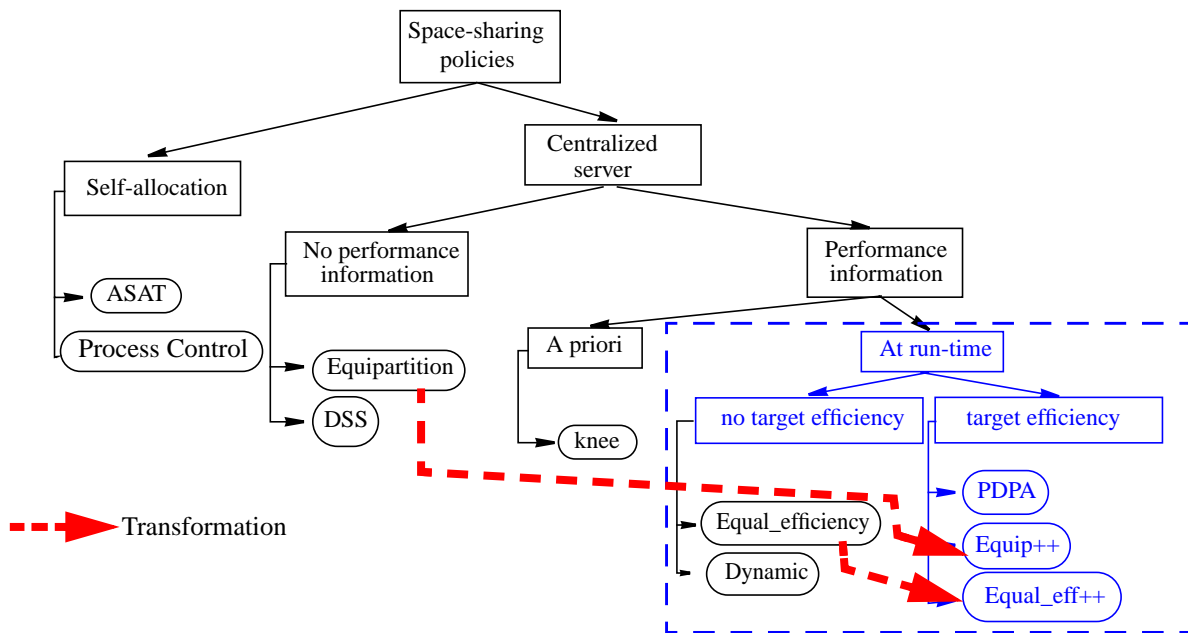


Figure 6.11: Processor scheduling taxonomy

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

6.5 Evaluation

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

Table 6.1: Workload characteristics

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

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

Table 6.2: Configurations

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

6.5.1 Workload 1

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

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

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

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

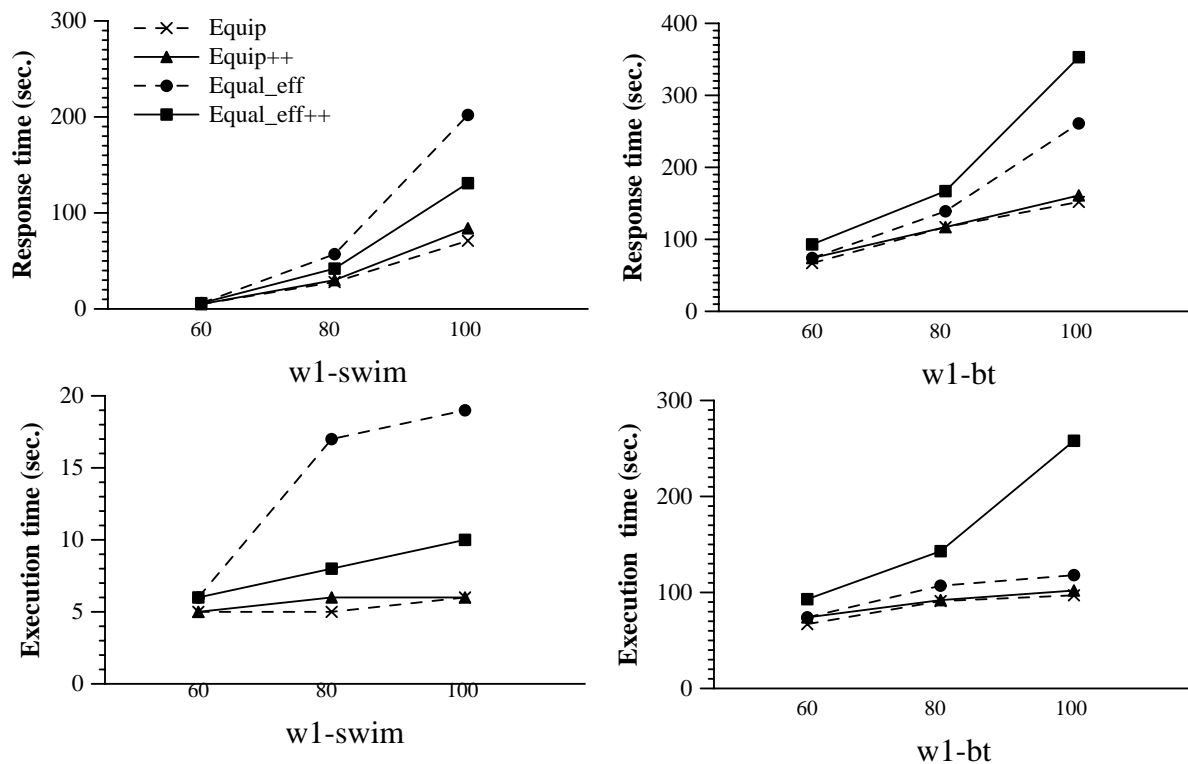


Figure 6.12: Results from workload 1, M.L.=4

6.5.2 Workload 2

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

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

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

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

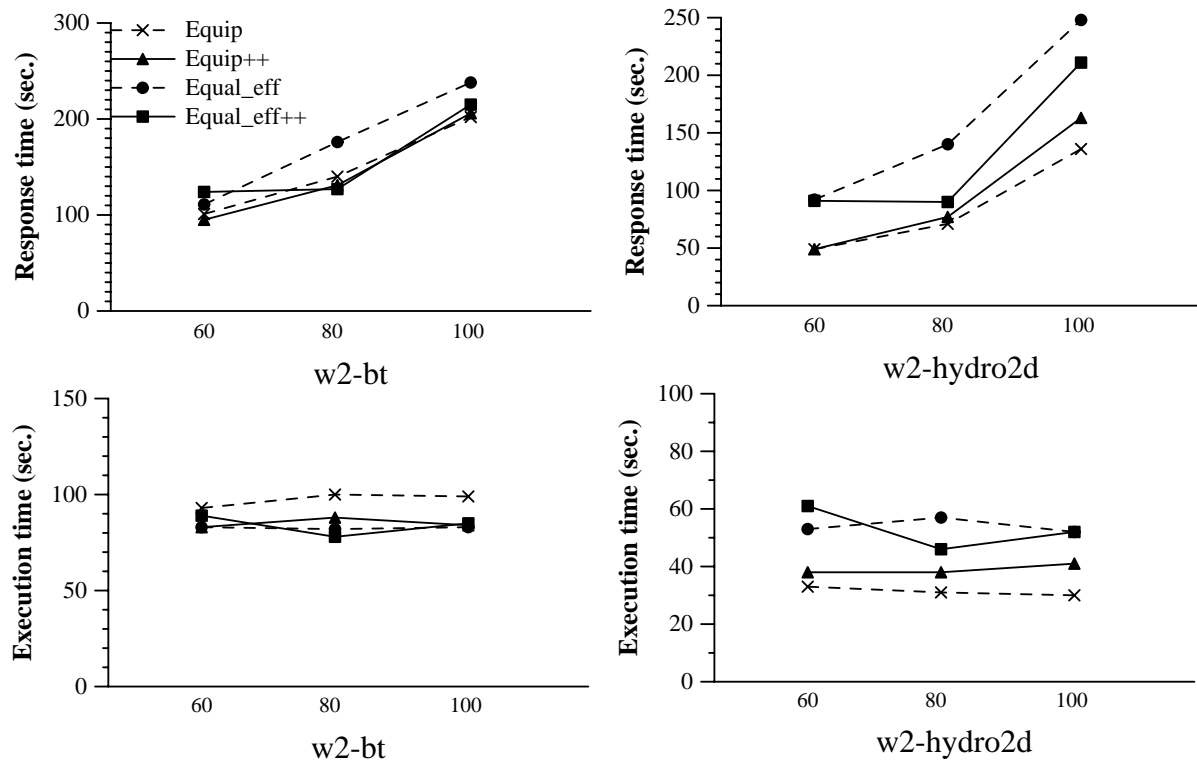


Figure 6.13: Results from workload 2, M.L.=4

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

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

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

Multiprogramming level set to two applications

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

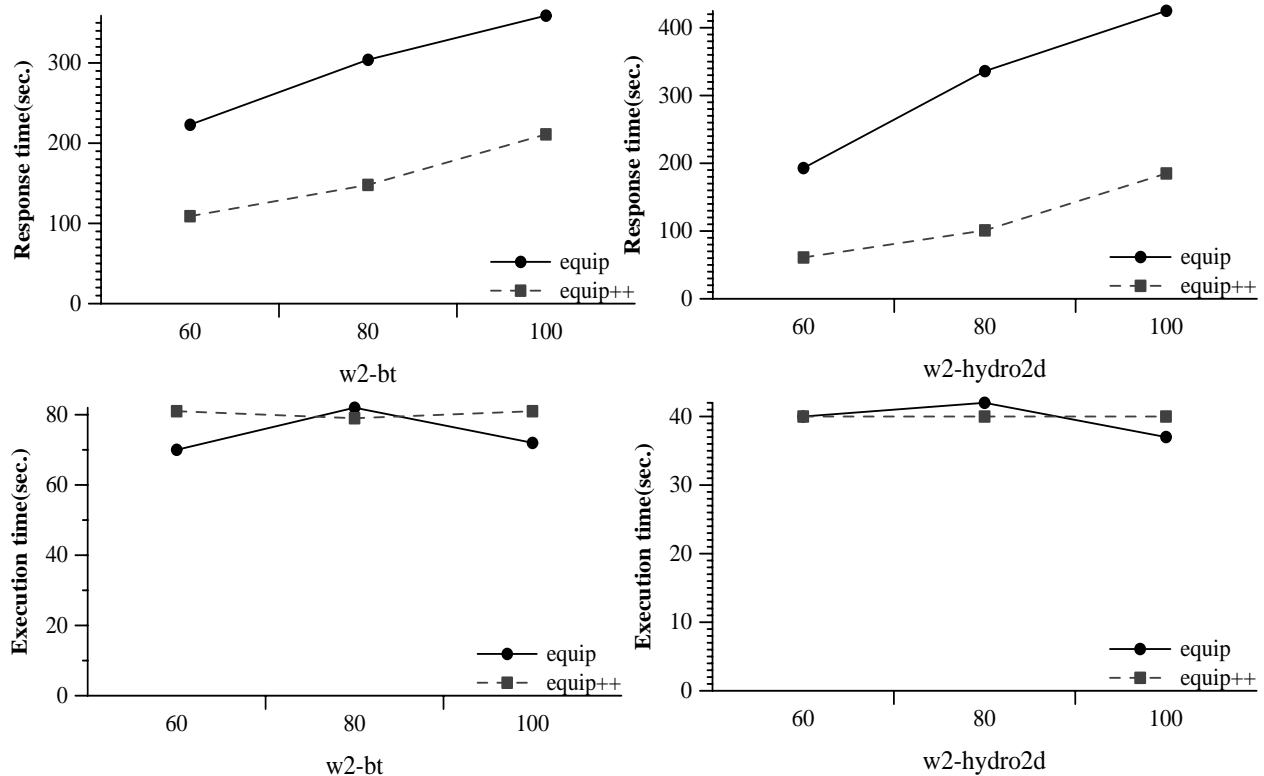


Figure 6.14: Workload 2, M.L.=2 Equipartition vs. Equip++

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

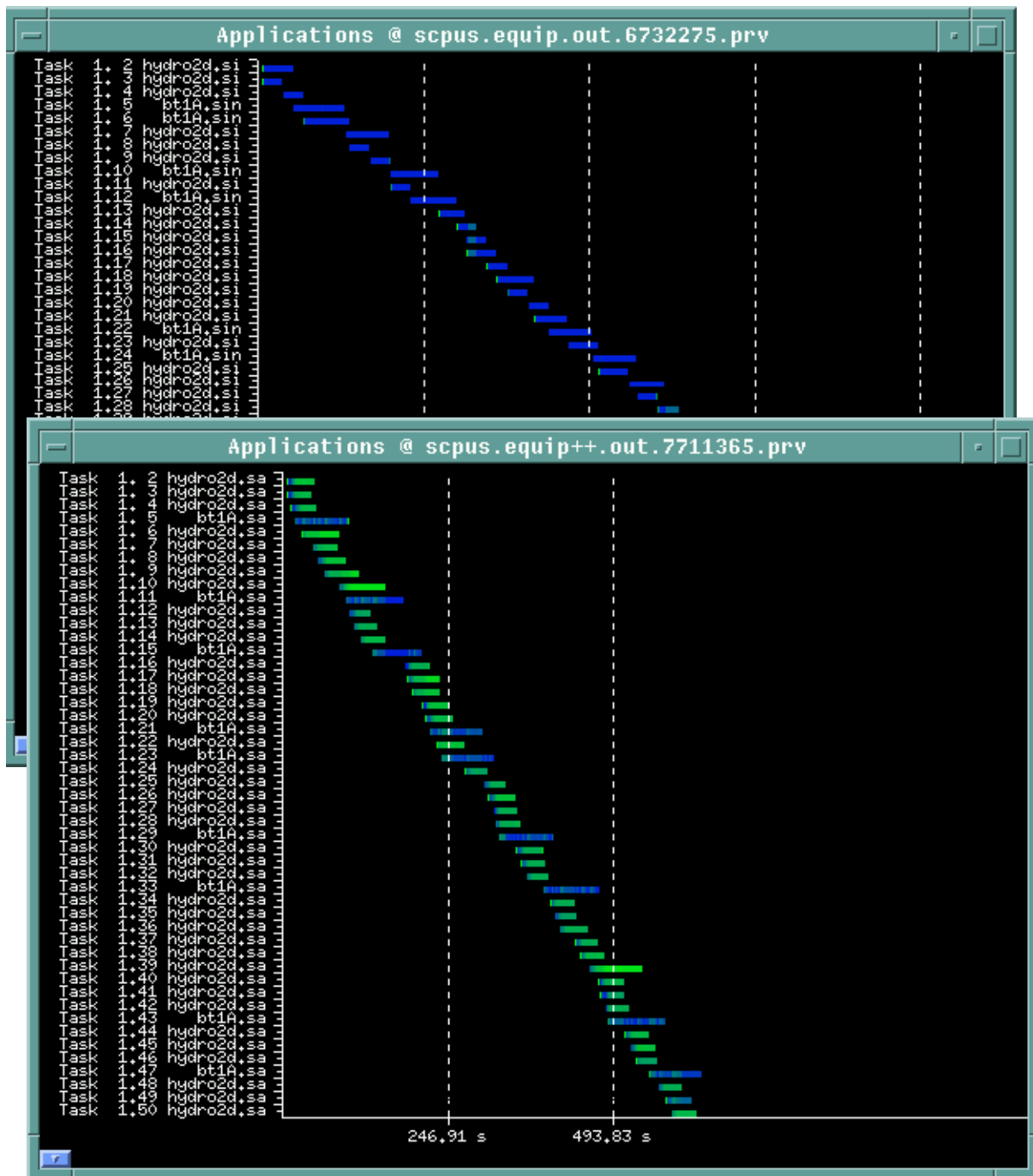


Figure 6.15: Processor allocation decided by Equipartition and Equip++, (M.L=2,load=100%)

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

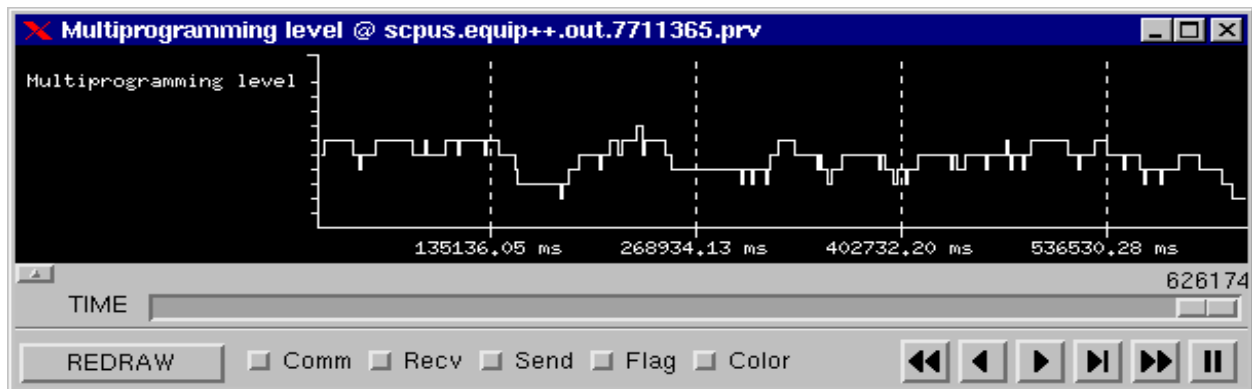


Figure 6.16: Dynamic multiprogramming level decided by Equip++, (M.L.=2, load=100%)

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

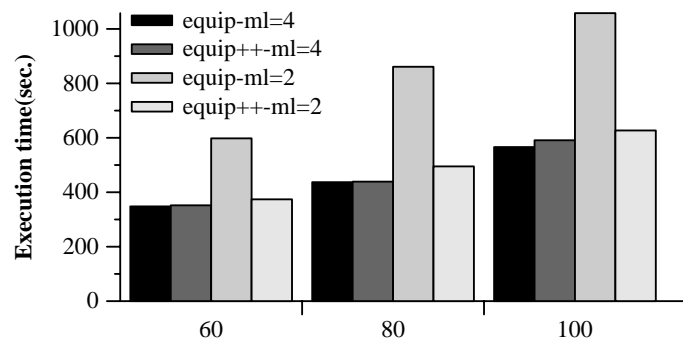


Figure 6.17: Workload execution time, Equipartition vs. Equip++

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

6.5.3 Workload 3

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

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

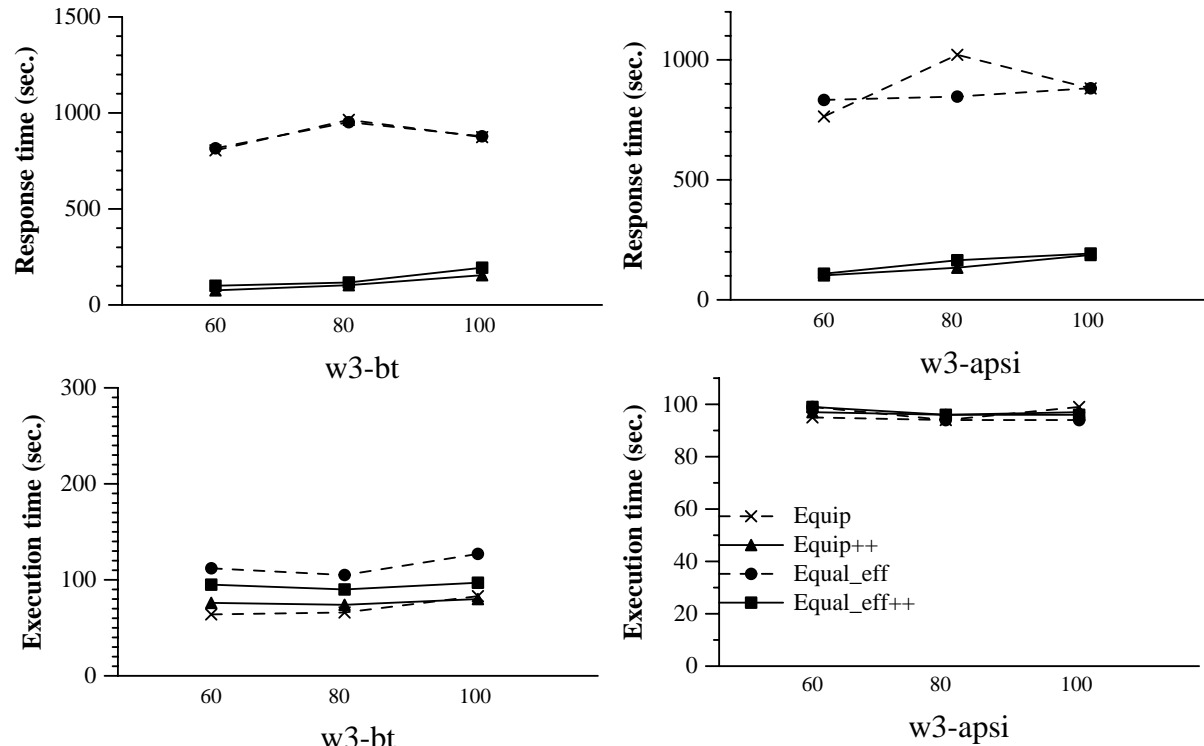


Figure 6.18: Results from workload 3

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

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

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

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

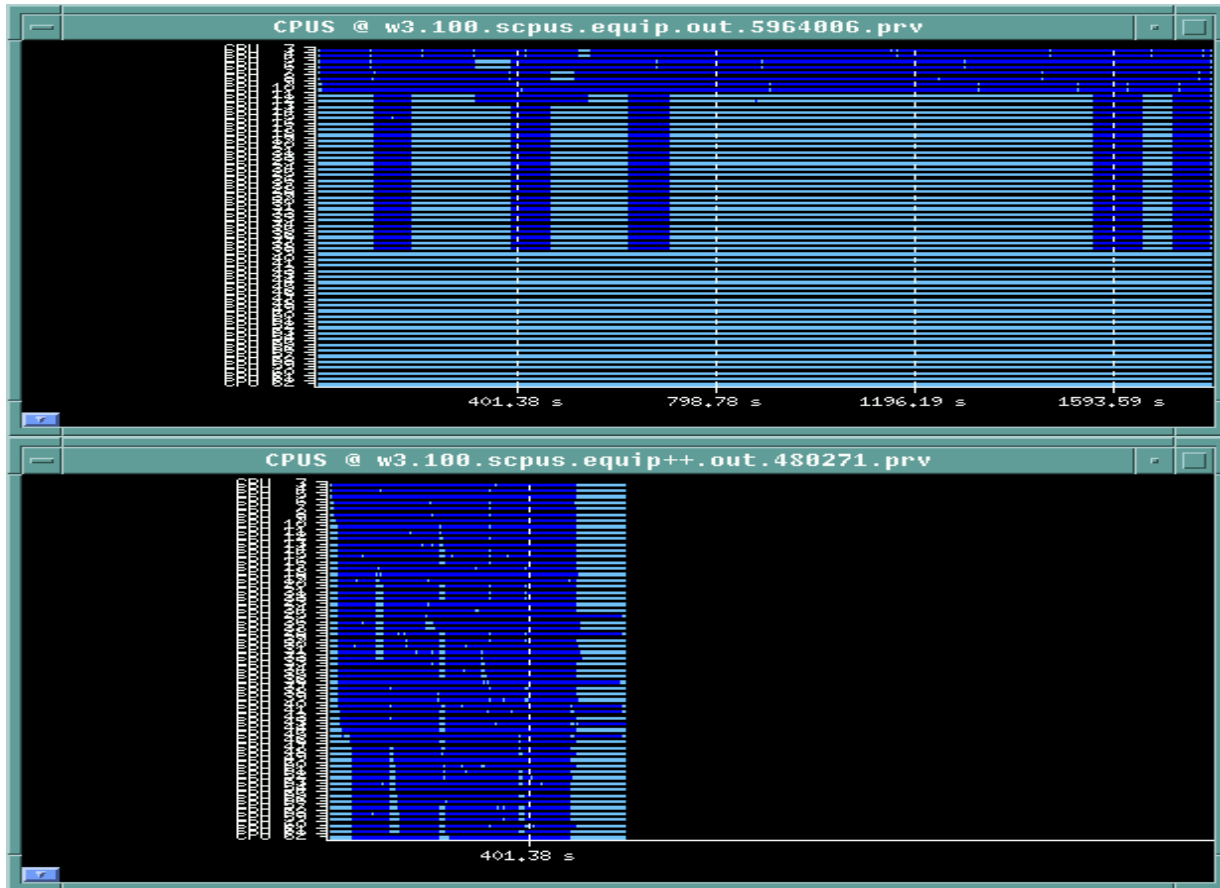


Figure 6.19: Cpu utilization in the case of Equipartition and Equip++, load=100%

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

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

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

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

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

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

6.5.4 Workload 4

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

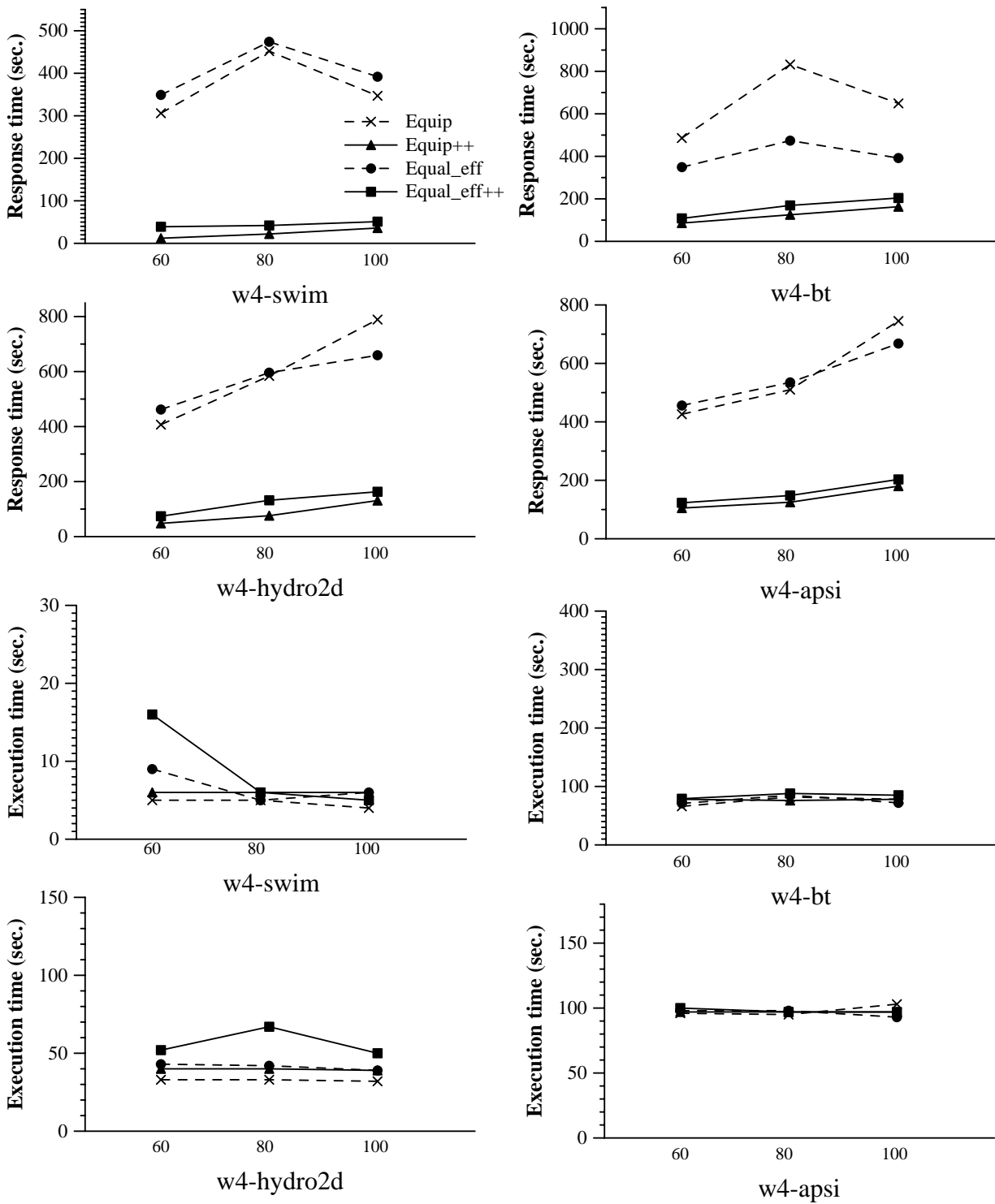


Figure 6.20: Results from workload 4

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

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

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

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

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

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

6.5.5 Workload 5

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

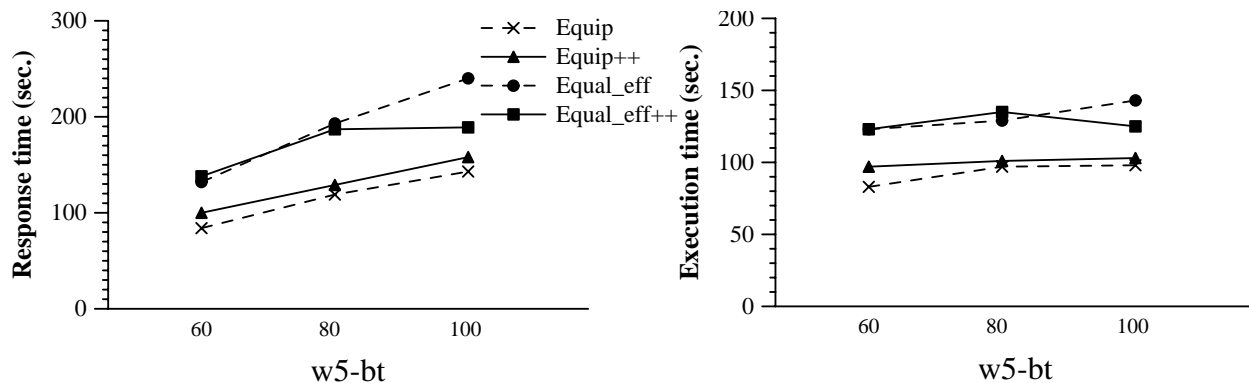


Figure 6.21: Results from workload 5

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

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

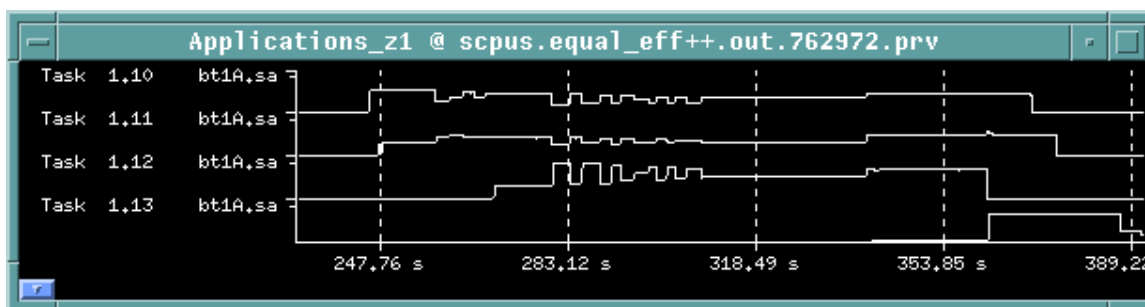


Figure 6.22: Re-allocations decided by Equal_eff++ because of the extrapolation function

6.5.6 Workload execution times

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

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

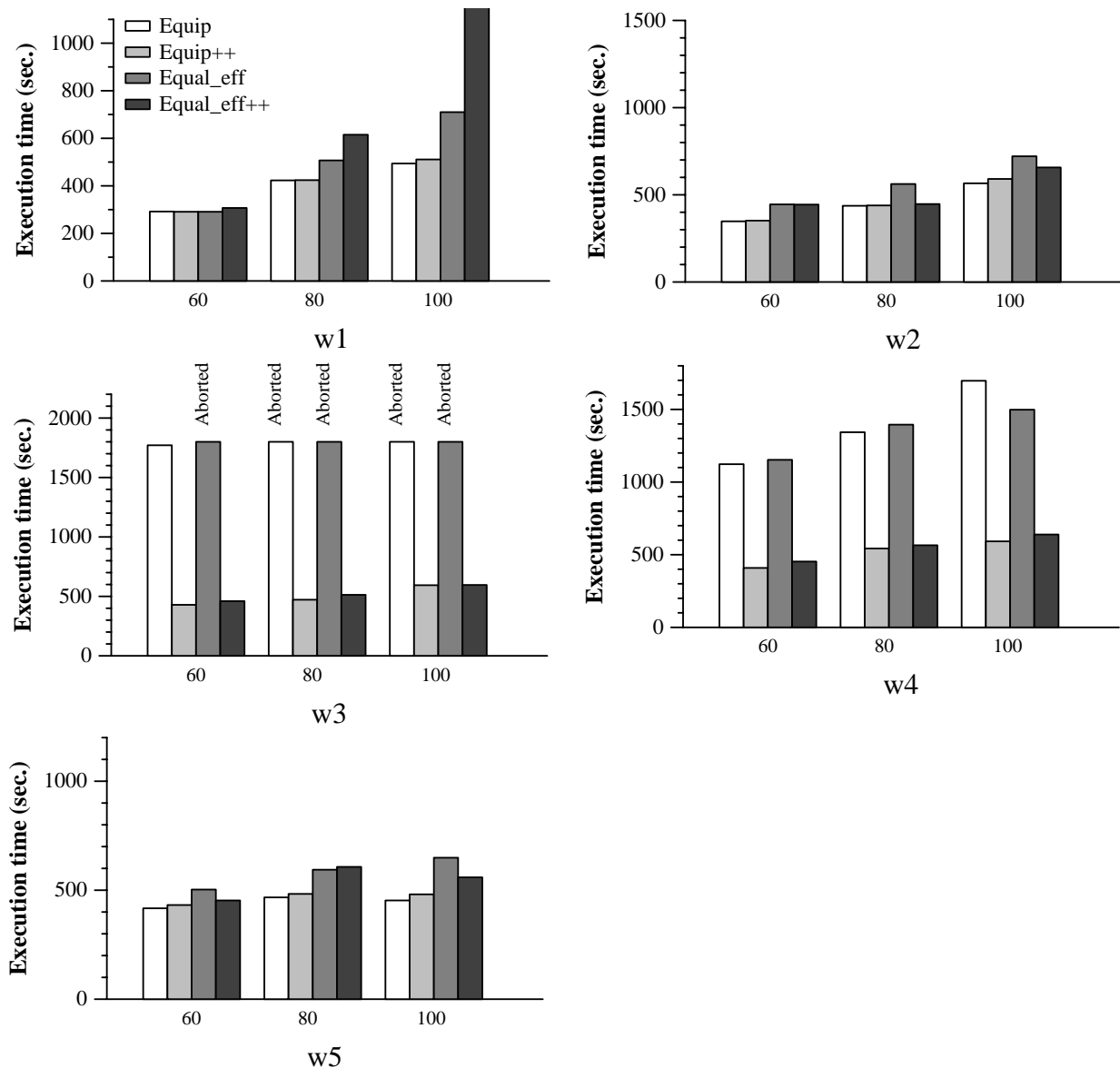


Figure 6.23: Execution time of the complete workloads

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

by the extrapolation function in the case of initial value equal to one processor because we want to remark the negative influence that this kind of formulations can generate if they are not explicitly considered. Authors of the Equal_efficiency do not have the problem to work with super-linear applications because, even they exist, their mechanism does not consider them. Moreover, they do not comment any additional problem and then do not provide any solution.

This effect is more usual under the Equal_eff++ than under the Equal_efficiency because the multiprogramming level is higher under Equal_eff++ than under the Equal_efficiency. Then, the probability to receive less processors is higher with Equal_eff++.

6.6 Summary

In this Chapter, we have presented Performance-Driven Multiprogramming Level (PDML). PDML is a methodology that transforms processor scheduling policies to include feedback based on performance information with the aim of avoiding the inefficient use of processors. PDML modifies processor scheduling policies by periodically evaluating the performance achieved by running applications. If applications reach a given target efficiency, their allocation will be maintained, otherwise it will be adjusted.

PDML also includes a multiprogramming level policy based on the system stability. If all the applications are stable, that is, all of them reach the target efficiency, and there are free processors, the multiprogramming level will be increased. PDML has been applied to the Equipartition and the Equal_efficiency, resulting in the Equip++ and the Equal_eff++.

Results show that the performance of a policy not only depends on the number of processors allocated to each application, but also in the stability of the system, and in the number of reallocations that applications suffer. In the case of Equip++, it reaches the same performance that Equipartition in those workloads that directly perform well in Equipartition, and outperforms Equipartition in those workloads in which Equipartition does not decide an efficient processor allocation.

In the case of Equal_eff++, we can conclude that the main goal of PDML is achieved: to ensure the efficient use of processors and to coordinate the different scheduling levels. However, in some cases the behavior of the original policy generates that some pathological situations are more frequent under Equal_eff++ than under Equal_efficiency, resulting in some incorrect allocations. This situation mainly appears when the performance of parallel applications are initially measured with a small number of processors. In some of the workloads, the benefit is mainly generated by the stability that PDML introduces in the Equal_efficiency. This stability does not have the same impact in the case of the Equipartition because it is much more stable than the Equal_efficiency. In some other workloads the benefit is also due to the multiprogramming level policy. We could conclude that the use of formulations to extrapolate values is possible, but it must be done *with care* and always being conscious that they are not real values.

A difference that we have found between PDPA and these policies is that PDPA has more control about the behavior of the applications and that PDPA is more conscious that the behavior of the application can change. PDPA gives more chances to running applications to receive more processors. This behavior, in some workloads could introduce some overhead, but in other cases could solve incorrect allocations due a punctual bad measurement or medium scalability in a certain range of processors.

