CHAPTER 7

Contributions to Gang Scheduling

In this Chapter, we present two techniques to improve Gang Scheduling policies by adopting the ideas of this Thesis. The first one, Performance-Driven Gang Scheduling, is the result of applying PDML to a traditional gang scheduling policy. The second one is the Compress&Join algorithm, a new re-packing algorithm that exploits the job malleability of OpenMP applications and the use of real performance information.

Performance-Driven Gang Scheduling mainly attacks the problem of the inefficient use of resources by parallel applications, that indirectly results in an excessive number of time slots.

Compress&Join is totally oriented to reduce the number of time slots by reducing the number of processors used by application proportionally to their performance.

These two techniques are orthogonal among them and can be used with any previously proposed scheme.

7.1 Introduction

Gang Scheduling [33][34] is a combination of time-sharing and space-sharing approaches. Characteristics of Gang Scheduling are: (1) threads are grouped into gangs, (2) threads in a gang are executed simultaneously, and (3) time sharing is used among gangs.

Gang scheduling appeared as the solution to the problems of job scheduling policies in those systems where the processor scheduling was a simple dispatch. In this kind of systems, the main problem seems to be the fragmentation, then reasons to use gang scheduling were presented as responsiveness and efficient use of resources. However, gang scheduling still has the problem of the fragmentation [115][33], and the excessive number of time slots [115]. These two problems result in an inefficient use of resources (processors and memory).

In this Chapter, we present two new approaches to improve gang scheduling by exploiting the ideas presented in this Thesis: use of real performance information, imposing a target efficiency, and coordination with the queueing system.

Our first contribution consists of applying PDML to the traditional gang scheduling scheme. This scheme has a scheduling phase that is a simple dispatch. We propose to self-evaluate the allocation decided by this dispatch, to impose a target efficiency, and to coordinate with the queueing system. We have called the resulting policy *Performance-Driven Gang Scheduling*, PDGS.

The main goal of PDGS is to ensure the efficient use of resource. However, one of the main problems of gang scheduling, the excessive number of time slots, is still a problem. Our second approach to improve gang scheduling consist of a new re-packing algorithm, *Compress&Join*, that combines two characteristics of our execution environment (the job malleability and the use of real performance information). The Compress&Join algorithm adjusts the processor allocation of applications based on their performance to fit the same number of applications in less time slots.

Results show that PDGS and the Compress&Join algorithm outperform the gang scheduling approach used as baseline. We will see that the ideas proposed in this Thesis of (1) measuring the performance of applications at run-time and (2) adjust the processor allocation based on this information, are also valid for gang scheduling policies. As in previous Chapters, results show that with our proposals, the execution time of applications is slightly increased because applications usually receive less processors than with other approaches. However, we will see that by adjusting the allocation, the system is efficiently used and that benefits both the system and the individual applications.

The next of this Chapter is organized as follows: Section 7.2 describes modifications introduced in the CPUManager to implement gang scheduling policies and the particular implementation we have done. Section 7.3 presents Performance-Driven Gang

Scheduling. Section 7.4 presents the Compress&Join algorithm. Section 7.5 evaluates PDGS and the Compress&Join algorithm compared with the gang scheduling baseline implemented. Finally, Section 7.6 summarizes this Chapter.

7.2 Gang Scheduling

In this Section, we present characteristics of gang scheduling policies and the particular implementation that we have used as baseline in this Thesis.

Figure 7.1 shows the behavior of generic gang scheduling policies. The different gangs are grouped in time slots following some re-packing algorithm. The total number of processors requested by gangs in a time slot must be less or equal than the total number of processors of the machine. Periodically, at each quantum expiration, the scheduler selects a new time slot to execute all of its gangs. If the workload has changed during the execution of the last quantum, the re-packing algorithm will be re-applied. In any case, the new slot selected is scheduled.

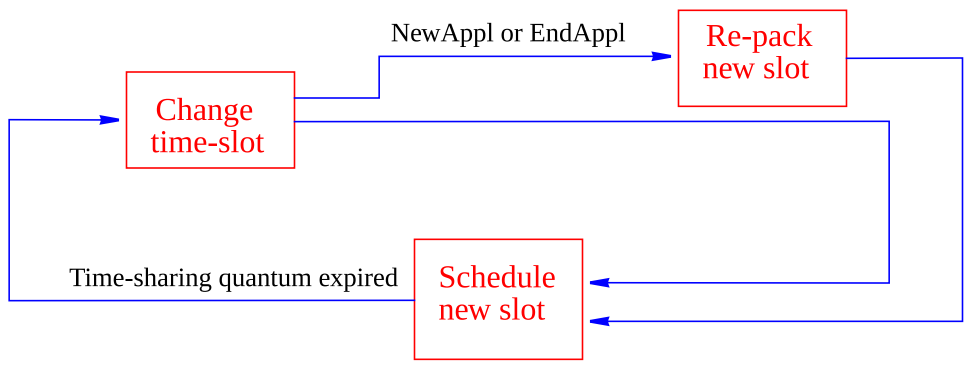


Figure 7.1: Gang scheduling generic scheme

7.2.1 Gang scheduling implementation

There are many versions of gang scheduling. The simplest version of gang scheduling always allocates a gang in the same set of processors. However, most flexible versions have been proposed [81]. One of them is *migratable* [32] preemptions, where jobs that are preempted in a set of processors can be resumed in a different set of processors. A more complex version is *malleable* [32] gang scheduling, where jobs can be resumed in a set of processors of different size, however, malleable preemptions has not been used till now because it is difficult to implement. For this reason, usually the scheduling of the slot is a simple dispatch, where applications receive as many processors as they request. In this Thesis, we will show that malleable gang scheduling is feasible and that to exploit this feature, combined with performance information, improves the system performance. Characteristics of our execution environment (OpenMP applications and space-sharing architecture) make malleable gang scheduling a valid approach.

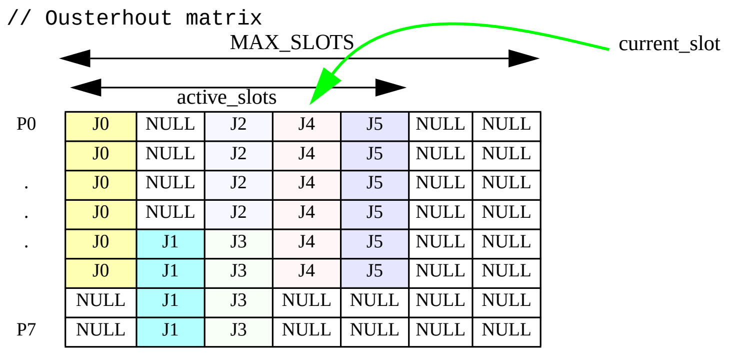
The main difference among gang scheduling policies is the job re-packing algorithm. Feitelson argues in [33] that the best option is to apply a buddy algorithm or to use migration to re-map the jobs (based on a first-fit algorithm). In [88] Setia shows that gang scheduling policies that support migration offer significant performance gains over policies that do not support it.

In the next Section, we present the characteristics of the gang scheduling implementation that we have used as baseline. We have tried to select the best configuration presented in the literature to compare with our proposals. For this reason, we have implemented migratable preemptions [88], that is, threads in a gang can be preempted in a set of processors and resumed in another set based on a first-fit algorithm.

Job organization: Ousterhout matrix

To implement Gang scheduling we have used the mechanism proposed by Ousterhout in [78], the Ousterhout matrix. The Ousterhout matrix defines a two dimensional matrix where one dimension represents processors and the other is time.

Figure 7.2 shows the main data used by the CPUManager to implement the gang scheduling mechanism. The table is an example of a possible configuration of the Ousterhout matrix in a system with eight processors. This table has as many rows as processors and a maximum of columns (MAX_SLOTS). Each column is a time slot, composed by a list of one or several gangs. In the example, slot 0 has one gang (Job_0), slot 1 has one gang (Job_1), slot 2 has two gangs (Job_2,Job_3), and so on. Time-sharing is performed between slots, and the sum of processors allocated to gangs in a slot must be less or equal than the number of processors in the machine. In this particular example there are five *active_slots* (with applications associated to them), and the *current_slot* is the slot 3. Applications in the current slot are the only ones that are running, the rest of applications are stopped.



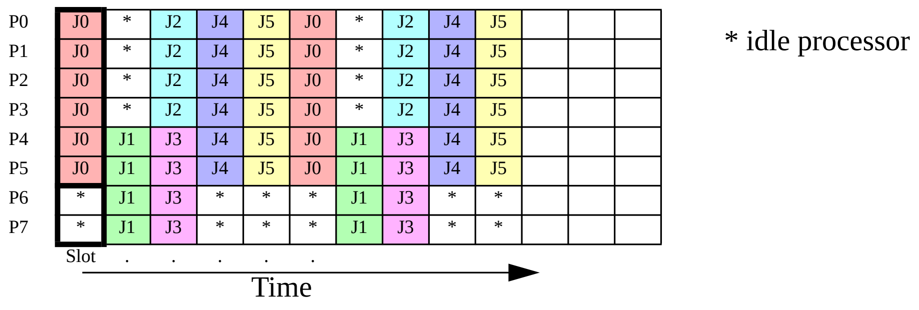
P0	J0	NULL	J2	J4	J5	NULL	NULL
	J0	NULL	J2	J4	J5	NULL	NULL
.	J0	NULL	J2	J4	J5	NULL	NULL
.	J0	NULL	J2	J4	J5	NULL	NULL
.	J0	J1	J3	J4	J5	NULL	NULL
	J0	J1	J3	J4	J5	NULL	NULL
	NULL	J1	J3	NULL	NULL	NULL	NULL
P7	NULL	J1	J3	NULL	NULL	NULL	NULL

Figure 7.2: CPUManager data structures

In the CPUManager implementation we have introduced a small modification with respect to the traditional Ousterhout matrix. In our execution environment, we have the problem that applications start their execution requesting for one processor, and they request for *P* processors when they open their first parallel region.

We have treated this case in a similar way to have two applications, a small sequential application and a new parallel one. The problem is that it is possible that when the application spawns parallelism, it does not fit in the slot. For this reason, we have defined two limits in the Ousterhout matrix. The first one, set to 4 slots, determines the number of slots to which the system will perform the time-sharing. The second one, set to 50 slots, is similar to a buffer of applications that have been started but that do not fit in any of the first four slots when they spawn parallelism. These applications are not scheduled until any of the applications in the first slots finish their execution.

Figure 7.3 corresponds with the execution of the Ousterhout matrix presented in Figure 7.2. In the example, the matrix has five time slots, that means that each application will be executed once every five slots. The system will execute the following sequence of applications: (J0), (J1), (J2/J3), (J4), (J5), (J0), (J1), (J2/J3), (J4), (J5), etc. The example presented in Figure 7.3 does not include the job migration used in this Thesis because we want to present the gang scheduling behavior in a progressive way. The execution including job migration is shown in Figure 7.4, and the job migration algorithm is presented in the next Section.



P0	J0	*	J2	J4	J5	J0	*	J2	J4	J5			
P1	J0	*	J2	J4	J5	J0	*	J2	J4	J5			
P2	J0	*	J2	J4	J5	J0	*	J2	J4	J5			
P3	J0	*	J2	J4	J5	J0	*	J2	J4	J5			
P4	J0	J1	J3	J4	J5	J0	J1	J3	J4	J5			
P5	J0	J1	J3	J4	J5	J0	J1	J3	J4	J5			
P6	*	J1	J3	*	*	*	J1	J3	*	*			
P7	*	J1	J3	*	*	*	J1	J3	*	*			
	Slot	.	.	.	.	.							

Figure 7.3: Gang scheduling behavior

Job re-packing algorithm: first-fit + migrations

The job re-packing policy implemented is a first-fit algorithm [33], (combined with job migrations) implemented in the following way:

- When a new job arrives to the system, it is placed in the first slot with a sufficient number of idle processors.
- A job completion does not imply a job re-packing.

At each time-sharing quantum, the scheduler performs the following steps:

- It stops the running gangs (this implies suspend all the threads)

- It advances the *current_slot* pointer and selects the next slot.
- If there are free processors in *current_slot*, and some application has finished in the last quantum, the job migration algorithm is applied. The job migration mechanism tries to move jobs from low loaded slots to high loaded slots. The load of the slot is computed as used processors divided by total number of processors.

At this point, we have selected a slot that can be dispatched by the scheduler. However, to improve the system utilization, we have introduced a mechanism to fill the remaining holes in the slot. To do that, we have introduced a phase of job replication. We generate a temporal slot, initialized with applications in *current_slot*, and we look for applications that can fit in the slot. The aim is to allow the execution of applications in more than one slot, but maintaining it physically allocated to only one slot in the Oustherout matrix. The Dispatch phase will receive this temporal slot to schedule.

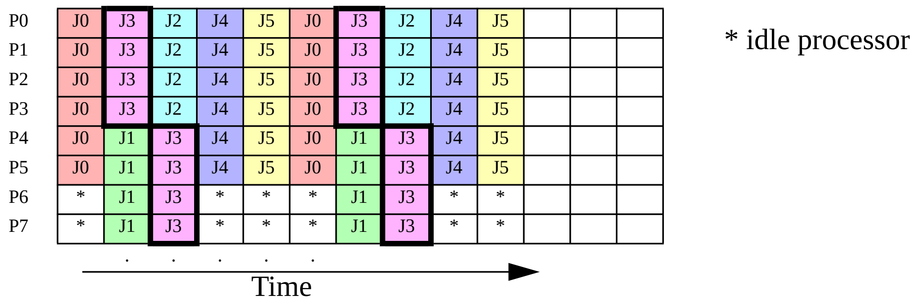

P0 J0 J3 J2 J4 J5 J0 J3 J2 J4 J5
P1 J0 J3 J2 J4 J5 J0 J3 J2 J4 J5
P2 J0 J3 J2 J4 J5 J0 J3 J2 J4 J5
P3 J0 J3 J2 J4 J5 J0 J3 J2 J4 J5
P4 J0 J1 J3 J4 J5 J0 J1 J3 J4 J5
P5 J0 J1 J3 J4 J5 J0 J1 J3 J4 J5
P6 * J1 J3 * * * J1 J3 * *
P7 * J1 J3 * * * J1 J3 * *
* idle processor
Time

Figure 7.4: Migratable gang scheduling

Figure 7.4 shows the behavior of execution of the Oustherout matrix shown in Figure 7.2 with migratable preemptions. We will allow the execution of J3 in two time slots, using the processors more efficiently. Job 3 is only allocated to slot 1 but executed in slot 1 and 2. J3 is not migrated to J1 because the load of slot 1 (50%) is less than load of slot 2 (100%). If the quantum of the time slot is well dimensioned, the system performance can be quite acceptable.

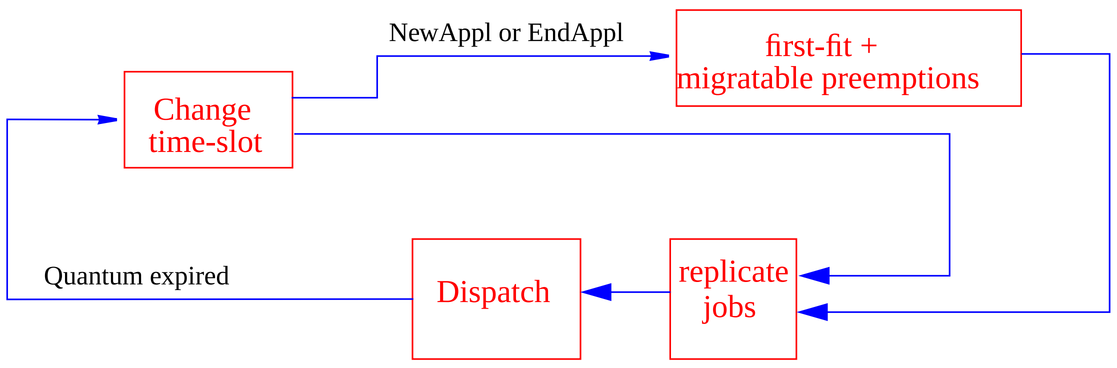


Figure 7.5: Gang scheduling baseline scheme

Figure 7.5 shows the gang scheduling scheme that represents our gang scheduling baseline: first-fit algorithm+migratable preemptions as job re-packing algorithm, and a simple dispatch as scheduling.

7.3 Performance-Driven Gang Scheduling

In this Section, we present *Performance-Driven Gang Scheduling* policy, PDGS. PDGS is the result of applying PDML to a traditional gang scheduling. Figure 7.6 shows the PDGS scheme. In PDGS, scheduling decisions are self-evaluated and corrected based on the performance achieved by running applications compared with the target efficiency. As in dynamic space-sharing policies, PDML introduces a space-sharing quantum to periodically evaluate the application performance. At each space-sharing quantum, the scheduler is activated and it evaluates the achieved performance. The processor allocation of those applications that do not reach the target efficiency is reduced as presented in Chapter 6. In fact, we have applied a processor allocation policy equal to the implemented in PDPA, a bit more complicated than to the one implemented in PDML.

PDGS also modifies the conditions to apply the re-packing algorithm. In this case, the re-packing algorithm must be also applied when the allocation has been adjusted because we consider changes in the processor allocation as a new application arrival.

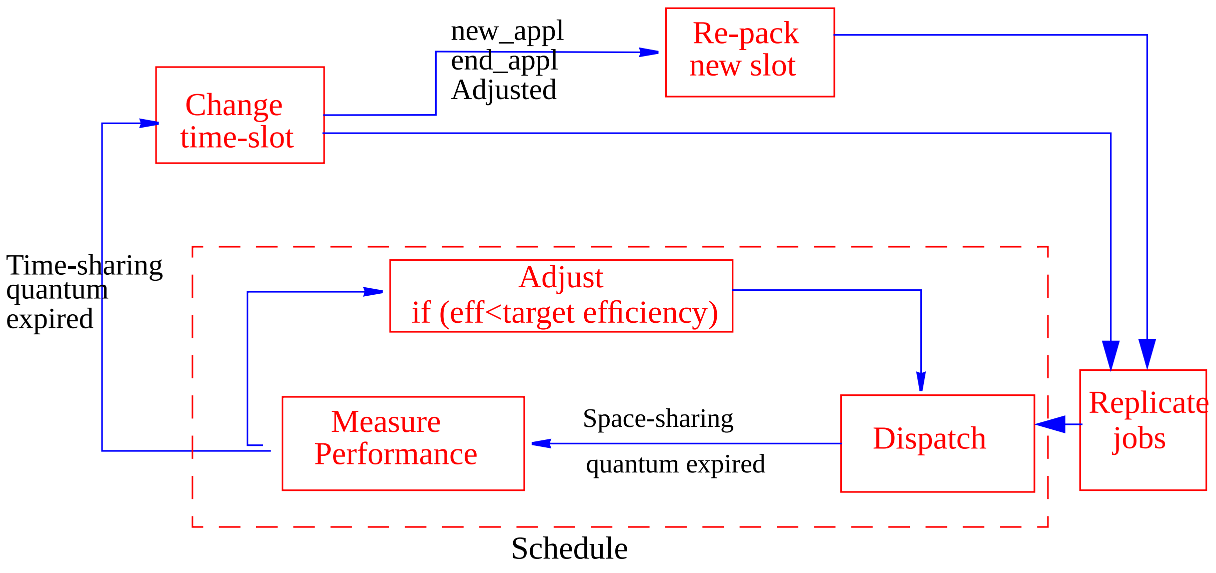
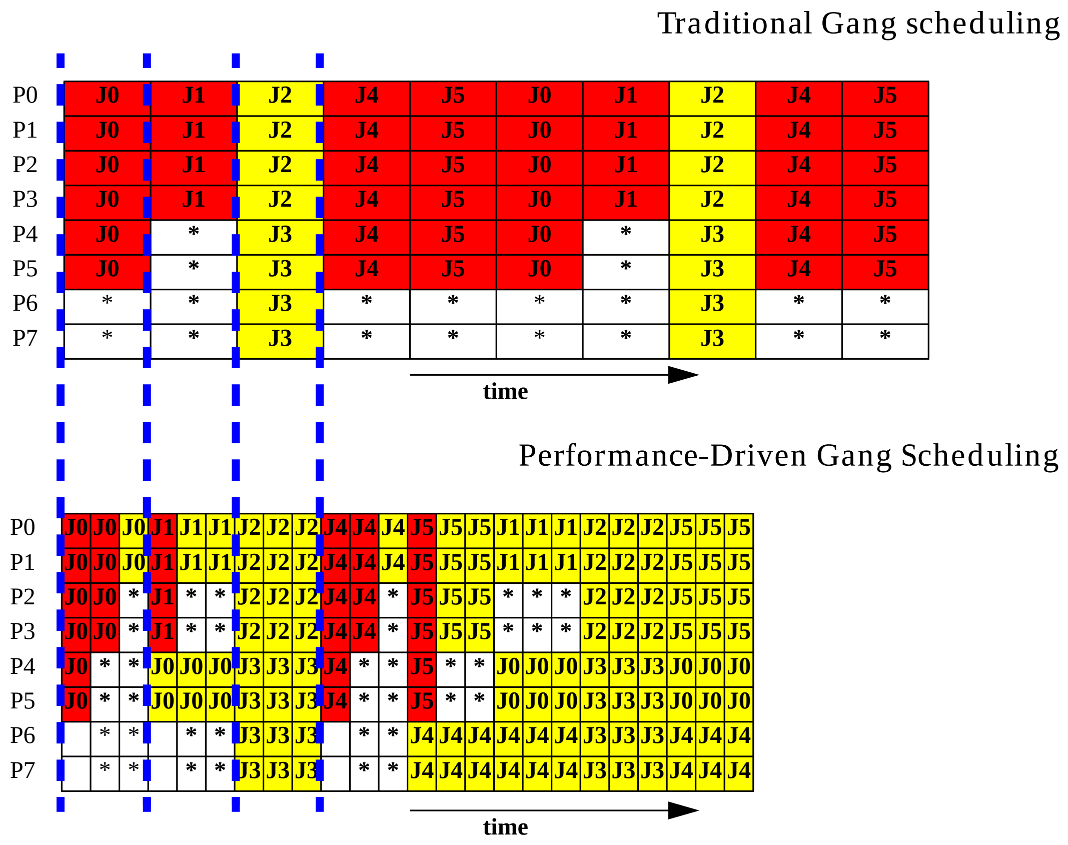


Figure 7.6: Performance-Driven Gang Scheduling scheme

To simplify the implementation, the time-sharing quantum is a multiple of the space-sharing quantum. In the current implementation we have set the space-sharing quantum to 100 ms, as in the previous chapters, and the time-sharing quantum in 6 sec. The space-sharing quantum set to 100 ms. does not imply a re-allocation at each 100 ms. It implies that, at each 100 ms., the processor allocation policy will be applied and, if the performance of running applications have been calculated, the algorithm will evaluate their performance and will adjust their processor allocation to reach the target efficiency (if needed).

Figure 7.7 shows an example about the different behavior of a system with a traditional gang scheduling and a system with PDGS. Initially, there are six jobs executing in a machine with eight processors. The red (dark) color means that the application does not reach the target efficiency and the yellow (light) color means that application reaches the target efficiency.


Traditional Gang scheduling
P0 J0 J1 J2 J4 J5 J0 J1 J2 J4 J5
P1 J0 J1 J2 J4 J5 J0 J1 J2 J4 J5
P2 J0 J1 J2 J4 J5 J0 J1 J2 J4 J5
P3 J0 J1 J2 J4 J5 J0 J1 J2 J4 J5
P4 J0 * J3 J4 J5 J0 * J3 J4 J5
P5 J0 * J3 J4 J5 J0 * J3 J4 J5
P6 * * J3 * * * * J3 * *
P7 * * J3 * * * * J3 * *
time
Performance-Driven Gang Scheduling
P0 J0 J0 J0 J1 J1 J1 J2 J2 J2 J4 J4 J4 J5 J5 J5 J1 J1 J1 J2 J2 J2 J5 J5 J5
P1 J0 J0 J0 J1 J1 J1 J2 J2 J2 J4 J4 J4 J5 J5 J5 J1 J1 J1 J2 J2 J2 J5 J5 J5
P2 J0 J0 * J1 * * J2 J2 J2 J4 J4 * J5 J5 J5 * * * J2 J2 J2 J5 J5 J5
P3 J0 J0 * J1 * * J2 J2 J2 J4 J4 * J5 J5 J5 * * * J2 J2 J2 J5 J5 J5
P4 J0 * * J0 J0 J0 J3 J3 J3 J4 * * J5 * * J0 J0 J0 J3 J3 J3 J0 J0 J0
P5 J0 * * J0 J0 J0 J3 J3 J3 J4 * * J5 * * J0 J0 J0 J3 J3 J3 J0 J0 J0
P6 * * * * J3 J3 J3 * * J4 J4 J4 J4 J4 J4 J3 J3 J3 J4 J4 J4
P7 * * * * J3 J3 J3 * * J4 J4 J4 J4 J4 J4 J3 J3 J3 J4 J4 J4
time

🟥 Does not reach the target efficiency

🟨 Reaches the target efficiency

Figure 7.7: PDGS behavior

In a traditional gang scheduling policy, the scheduler is not conscious of the application performance. It executes each job with the number of processors requested. The number of active slots is five because job 2 and job 3 are executed in the same time slot. Then each job is executed one out of every five time slots. The utilization of the system is the 75% and processors of J0, J1, J4, and J5 are not efficiently used. The percentage of cpus that each job receives is: J0 (15%), J1 (10%), J2 (10%), J3(10%), J4 (15%), and J5(15%).

In this example, each time-sharing quantum has been divided into three space-sharing quanta and the *step* has been set to two processors. PDGS initially allocates to each job as many processors as they request. At each space-sharing quantum, PDGS evaluates the achieved performance. If it does not reach the target efficiency, the processor allocation will be reduced in *step* processors. Once finished the time sharing quantum, the slot is changed. The processor allocation of Job_2 and Job_3 is not changed because they reach the target efficiency. The reduction in the processor allocation indirectly favours the migration of jobs and the reduction in the number of time slots. In the example, when Job_1 is started, the re-packing algorithm notes that Job_0 now only uses 2 processors, and that it can be migrated to this slot. This process (evaluation / adjust / migration / re-packing) is continuously repeated. In this example, the stable configuration has only three time slots, the system utilization is 91%, and the utilization per job is: Job_0 (16%), Job_1(8%), Job_2 (16%), Job_3(16%), Job_4 (16%), and Job_5(16%). Even receiving less processors than in the traditional gang scheduling, most of the jobs receives a higher percentage of cpu. The system utilization is better, and cpus are more efficiently used. This is just an example to show the expected benefits due the PDGS. In the evaluation Section we will really demonstrate that PDGS improves gang scheduling approaches that do not consider the application performance.

7.3.1 Multiprogramming level policy

PDGS also includes a multiprogramming level policy. In this case, the policy adopted is the same than in PDPA. Inside each space-sharing quantum, the scheduler evaluates if all the applications assigned to this slot are stable and there are free processors in the current slot. PDGS also evaluates if the number of active slots does not exceed a maximum. If these two conditions are TRUE, a new application is allowed to join the slot. We have set the maximum number of time slots to 4.

We have experimentally observed that it is important to limit to a small number of slots the Ousterhout matrix because of two main reasons. The first one is that the amount of resources used by running applications can saturate the system. We have executed some experiments that show very bad results because the system does not have more processes to execute applications, even when there are empty slots in the Ousterhout matrix. This situation is not so strange because parallel libraries usually generate more processes than they really use. This was our main motivation to modify the NthLib to include the dynamic thread creation (to adjust the number of processes created to the number of allocated processors). The second problem is related to the amount of memory used . We have observed that an application executes faster when executed in standalone mode than when executed inside a multiprogrammed workload because of the interferences that the rest of applications generate, mainly in memory. In gang scheduling policies, memory interferences are generated by running and active applications, which are proportional to the number of time slots.

7.4 Compress&Join: Malleability based on performance information

With PDGS, the number of time slots can be still a problem because the reduction in the number of slots that PDGS generates depends on the performance of running applications. It is possible to get a resulting Ousterhout matrix equal to the original matrix if the performance of running application reaches the target efficiency with the number of requested processors. Moreover, PDGS does not modify the allocation to fit applications in the resulting holes.

In an execution environment with malleable applications, the system fragmentation has no sense because jobs can adapt their parallelism and fit holes in the Ousterhout matrix. Based on this consideration, we propose a new re-packing algorithm that "*compresses*" applications to fit them in a reduced number of slots. The "*application compression*" is made based on the achieved performance and to "*compress*" an application means to reduce its processor allocation.

Compress&Join re-generates the complete Ousterhout matrix. The goal of this algorithm is to minimize the delay introduced by the alternation between slots by reducing their number. We are assuming the benefit provided by reducing the number of slots is greater than the slowdown produced by the reduction in the number of processors allocated to each application.

For instance, consider a simple case when we have one time slot that runs a parallel application with 64 processors. In that case this application does not suffer slowdown because with one time slot there are not context switches. If a new application arrives requesting 64 processors, a normal packing algorithm opens a new time slot and performs time-sharing between the two applications. In that case each one receives 50% of the cpu time and suffers and slowdown of 2. The Compress&Join algorithm will adjust the processor allocation of each application to 32 processors, reducing the number of slots from two to one. After applying the Compress&Join each application will receive half of the number of processors that in a normal algorithm, but (1) they will suffer no context switches, and (2) their efficiency will be also greater with 32 processors than with 64 processors. We assume that the benefit generated by reducing the number of time slots is greater than the penalty by reducing the processor allocation. One important thing is that all the applications collaborate to reduce the number of slots, not only those applications that we try to fit in holes, the rest of applications in the slot are also compressed.

In a general case, this algorithm will significantly reduce the number of context switches, and will eliminate most of the holes in the Ousterhout matrix because the algorithm will try to compress applications in these holes.

Since theoretically a malleable application could be reduced until it only uses one processor, we define a limit in the compression that an application can suffer. This limit is based on the speedup that the application will reach compressed (this may be an extrapolated value), compared with the speedup achieved with the current allocation (which is a calculated value).

Figure 7.8 shows the scheme that results of including the Compress&Join algorithm in a gang scheduling policy. Note that Compress&Join substitutes the re-packing algorithm but that it is totally compatible with the fact of applying a gang scheduling policy such as PDGS.

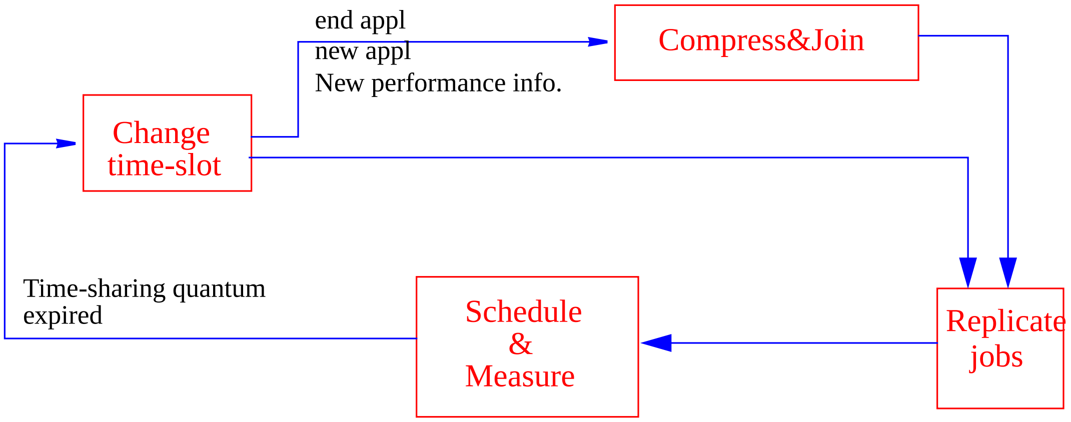


Figure 7.8: Compress&Join scheme

Figure 7.9 shows the main loop that re-generates the Ousterhout matrix in the Compress&Join algorithm. It starts by removing the complete matrix, then it tries to fit each application in some of the active slots. The number of active slots is initially set to zero since the matrix has been cleared. If the job can be compressed in the slot currently processed, the algorithm will go on with the next job. Otherwise, the algorithm tries to compress the job in the next active slot. Finally, if the job has not been compressed in any active slot, the algorithm will activate a new slot and will fit the job in this new slot. Each time the algorithm processes a new job it starts from the first slot. The idea is to first fit holes, rather than to compress jobs.

```
void Compress_and_join()
{
  last_slot=0
  RemoveTimeSlotTable()
  for(job=0;job<total_jobs;job++){
    for (slot=0;slot<last_slot;slot++){
      if (Compressed(job,slot)break;
  }
  if (slot==last_slot){
    last_slot++
    new_slot(appl,slot)
  }
}
```

Figure 7.9: Main Compress&Join loop

Figure 7.10 shows the algorithm that implements the compression of a particular job in a time slot, the *Compressed*(...) function in Figure 7.9. This function receives as parameter a slot identifier and a job identifier. A slot has a set of applications associated to it. This set always will have at least one job because new slots are initialized through the *new_slot*(...) function.

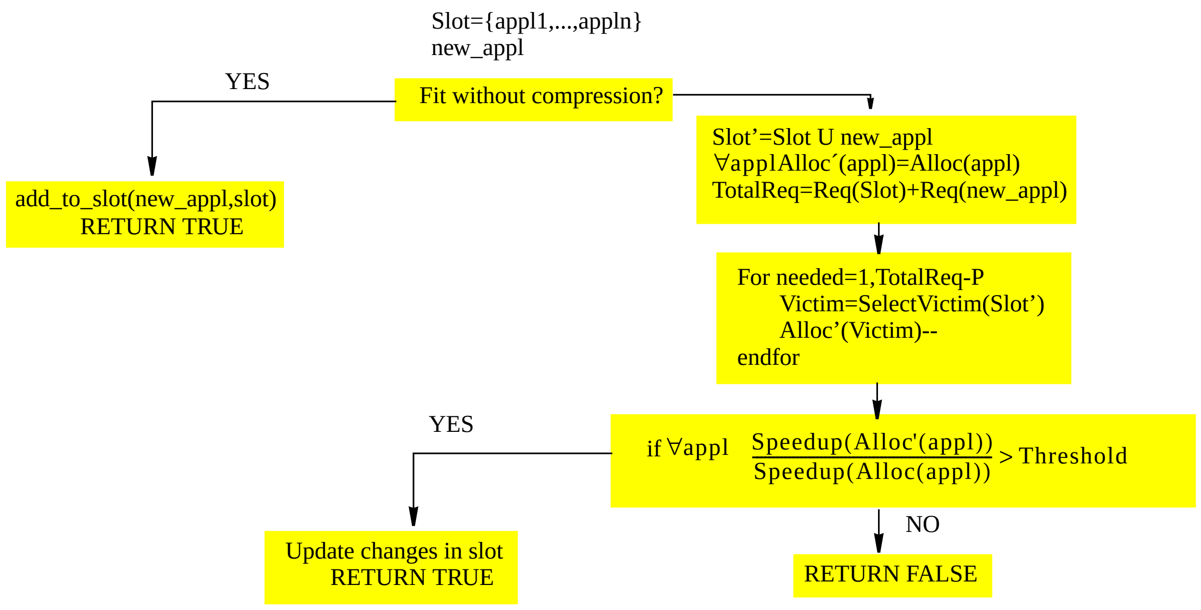


Figure 7.10: Algorithm to compress one application in one time slot

The algorithm initially tries to directly fit the application in the slot. If it fits, the slot will be updated with the new information and the *Compressed* function will return TRUE indicating that the job has been "compressed" in the slot. If the application requests for more processors than the currently available in the slot, the algorithm will try to fit it in the slot by compressing all the applications in the slot. This compression is proportional to the performance of applications in the slot. We compute a temporal slot, *Slot'*, composed by the new application and all the applications in *Slot*, and a temporal allocation, *Alloc'*, initialized to the current allocation of each application, *Alloc*. We also compute the total number of processors requested by *Slot´*, which is the sum of the processors requested by applications in *Slot* and the processors requested by *new_appl*. This value gives us an idea about how many processors we need, which is the difference between *TotalReq* and *P*. For instance, if we have eight processors in our system, six processors allocated to the processed slot, and the new application requests for four processors, we have a total request of ten processors, that means that we need to reclaim two processors to jobs in *Slot'* to have eight processors as maximum allocation. The question now is to check whether we can reduce the allocation of some applications in the slot, including the new one, to fit the new application in this slot.

The criteria used is to select, one by one, the application in *Slot*' with the small efficiency, and to reduce its processor allocation in one processors. The idea is that we will always increase the efficiency of an application if we reduce its allocation[1]. We repeat this process as many times as processors needed. Since the complete efficiency could be not known at this point, the algorithm extrapolates those values that have not been calculated by SelfAnalyzer.

Once computed the new (even temporal) allocation, we evaluate if the reduction in the speedups that this compression will generate is considered acceptable by the algorithm. To decide if a compression is acceptable the Compress&Join defines a threshold. If the ratio between the speedup achieved with the new allocation and the original allocation achieved by all the applications in *Slot*' is greater than this threshold, the compression will be considered acceptable. Otherwise, it is discarded. If it has been acceptable, the *Slot* is updated with the changes, the new application and the new allocation associated to applications in the *Slot,* and the algorithm returns TRUE. If changes has been discarded, the algorithm will return FALSE.

The threshold is a parameter of the algorithm. In our current implementation we have set it to a reduction in 50% in speedup.

The last question is how do we know the speedup achieved with the new allocation. A possible solution is to maintain the compressed allocation until the application informs about its new speedup. However, this solution has the problem that we have to apply the changes in any case, then measure, and then undo the compression if we found that it was an incorrect decision. To do something more "*intelligent*", we have used the formulation proposed by Dowdy in [25] and used in the Equal_efficiency policy. We are conscious that, in some cases, this function can not be representative of the real behavior of the application, but we use it just as a hint to take an initial decision. In any case, once the real performance information is available, we check our decisions. In addition, if the real performance information is available, we use this information, not the extrapolated value.

1. This is not always true but it is a good approximation

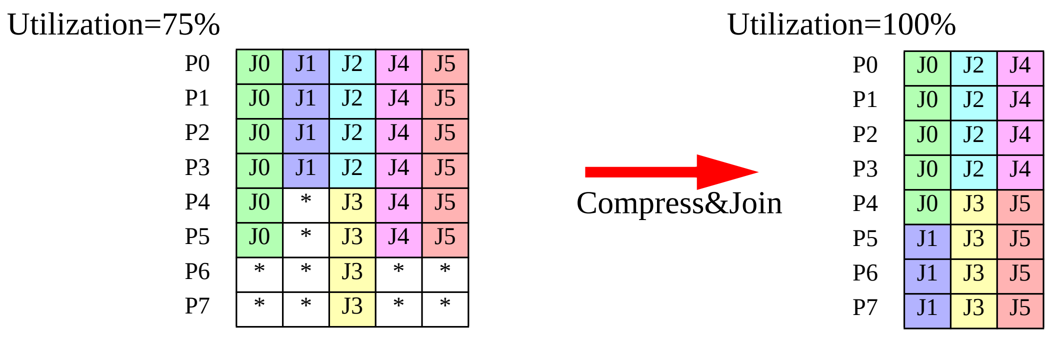


Figure 7.11: Ousterhout Matrix generated by the *Compress&Join*

Figure 7.11 shows the resulting matrix after applying the *Compress&Join* algorithm to the matrix presented in Figure 7.2. We can see how after applying the *Compress&Join* algorithm jobs have been proportionally reduced. In this example, system utilization goes from 75% to 100%, and the cpu percentage that each application receives is shown in Table 7.1. In this case, the different colors do not represent any efficiency values, they are only to better differentiate the different jobs.

Table 7.1: Cpu percentage per job

	J0	J1	J2	J3	J4	J5
Gang	15%	10%	10%	10%	15%	15%
Compress&Join	20.8%	12.5%	16.6%	16.6%	16.6%	16.6%

Observe that, even receiving less processors, each job has increased its percentage of cpu utilization. This is because the number of cpus that each application receives per unit of time is greater than in the Gang version with a simple first-fit algorithm.

Rather than the PDGS that is applied to any running application, the Compress&Join algorithm should be executed when the system conditions require it. These conditions are:

- A new application arrives to the system
- An application finishes its execution
- New real performance information is available and there are significant differences with some extrapolated values
- The allocation of some of the running applications has changed

The two firsts conditions are traditional conditions to re-apply any job re-packing algorithm. The two last are specific of the Compress&Join algorithm.

7.5 Evaluation

In this Section, we compare results achieved by the gang scheduling policy selected for baseline, with results achieved by PDGS and gang scheduling plus the Compress&Join algorithm. As in the previous Chapter, we have used the five workloads presented in Chapter 3. Table 7.2 resumes their main characteristics.

Table 7.2: Workload characteristics

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

Table 7.3 resumes characteristics of the different configurations evaluated in this Chapter. The multiprogramming level is not fixed because gang scheduling always includes a variable multiprogramming level. The Compress&Join configuration includes the gang scheduling policy presented in this Chapter and used as baseline plus the Compress&Join algorithm.

Table 7.3: Configurations

Policy	Queueing system	Processor scheduler	Run-time library
Gang	Launcher	CPUManager	NthLib
PDGS	Launcher	CPUManager	NthLib
Compress&Join	Launcher	CPUManager	NthLib

7.5.1 Workload 1

Figure 7.12 shows results for workload 1. Workload 1 is a mix of a 50% of super-linear applications (swim's), and a 50% of highly scalable applications (bt's). The number of processors requested by these applications (30 each one of them), implies that the workload does not generate fragmentation with Gang scheduling. The two graphs in the top of the figure show the average response time of swim's and bt's. The two graphs in the bottom of the figure shows the average execution time of swim's and bt's.

PDGS ensures the efficient use of resources by imposing a target efficiency. The reduction in the resulting processor allocation, sometimes generates the indirect benefit of a reduction in the number of slots in the Ousterhout matrix. However, this is not the goal of PDGS. Taking into account that this workload does not generates fragmentation, and applications have been previously tuned, we do not expect significant benefits such as in the previous chapters. Results show that PDGS reaches the same performance than gang in the response time of applications, and slightly worse than gang comparing the execution time of applications. This is quite normal because if (1) the reduction in the allocation does not imply a reduction in the number of slots, and (2) does not imply an improvement in the execution time, the benefit will not be significant.

The Compress&Join algorithm introduces benefits in the system performance. Nevertheless, in this case, we have introduced some undesired fragmentation. After compressing applications and joining slots, some applications have not been reduced and have been executed using one slot for a single application, whereas in the baseline gang scheduling there is no fragmentation in this workload.

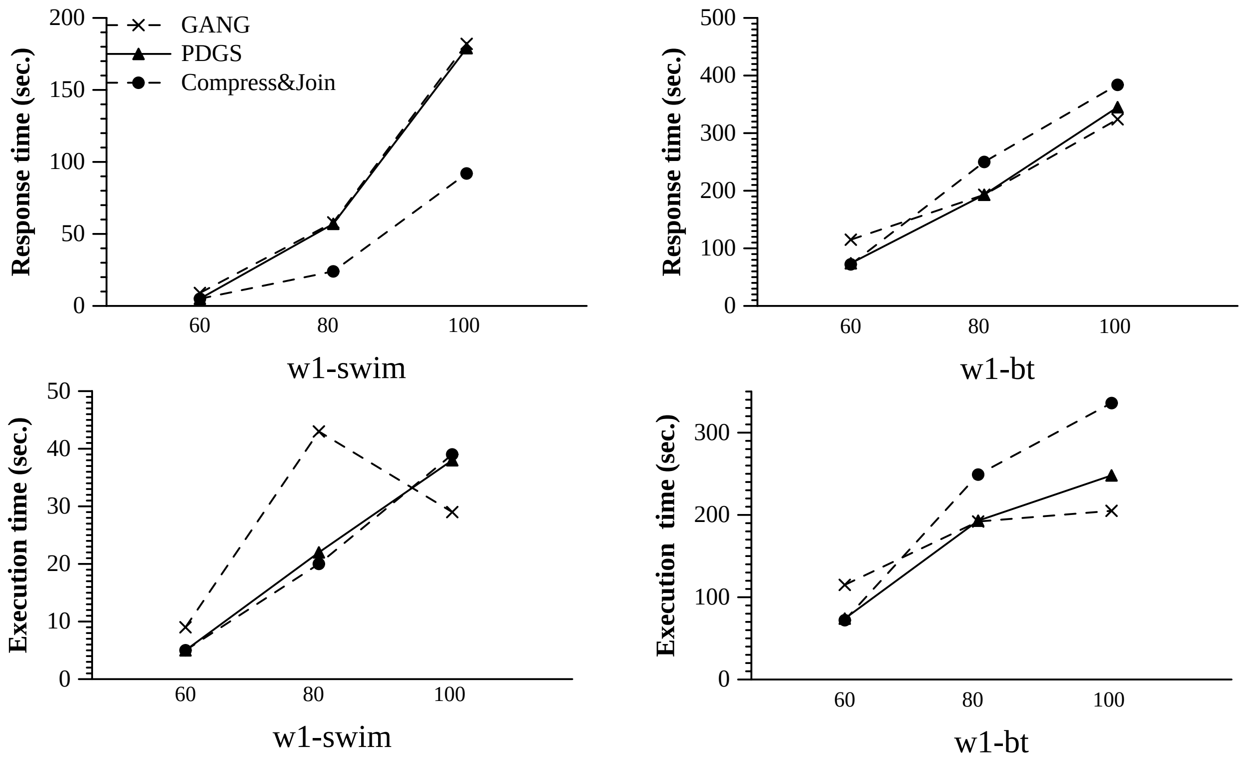


Figure 7.12: Results from workload 1

7.5.2 Workload 2

Figure 7.13 shows results for workload 2. Workload 2 is a mix of a 50% of highly scalable applications (bt's), and a 50% of applications with medium scalability (hydro2d's). In this workload, applications also request for 30 processors each one. As in the previous workload, this workload does not generate fragmentation during its execution. The reason is that all the applications request for 30 processors and the system has 60 processors, then any combination of applications fits in the system.

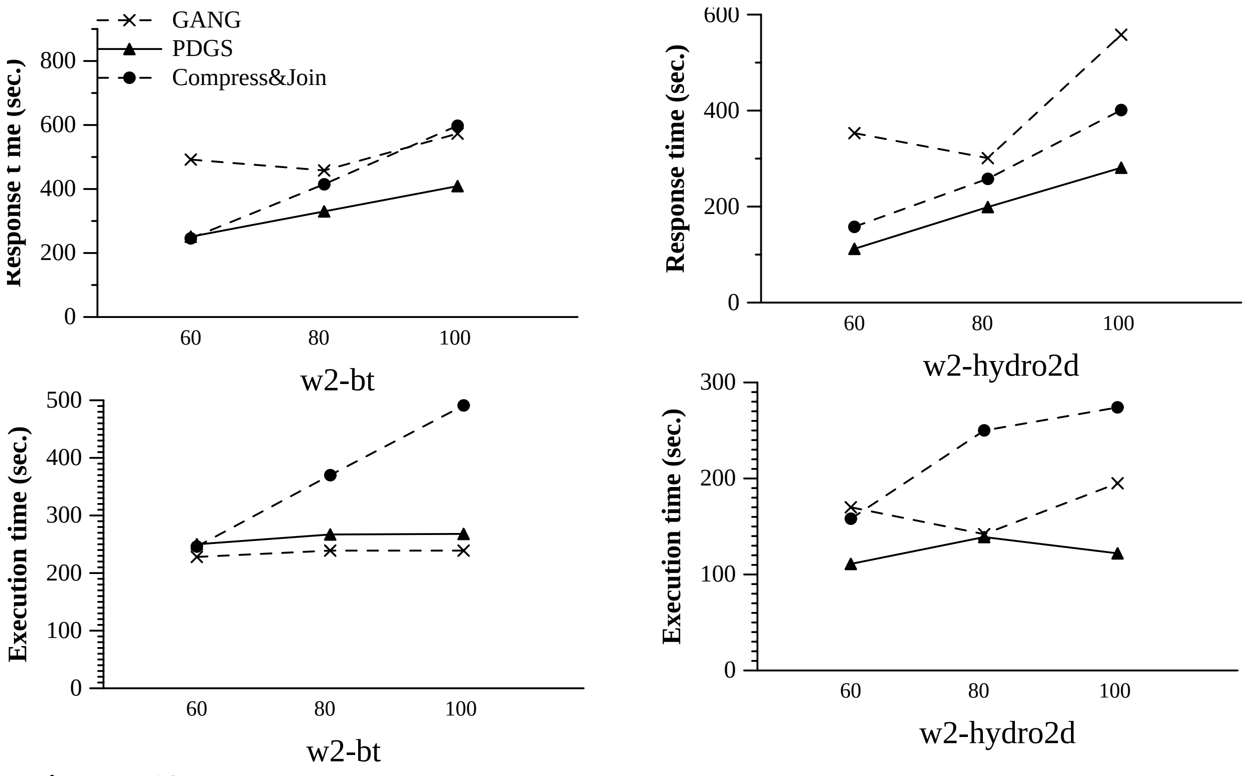


Figure 7.13: Results from workload 2

Comparing the results achieved by PDGS with gang scheduling, we can see that PDGS outperforms gang. In the response time of applications, PDGS outperforms gang by 58% (on average). Gang scheduling improves the execution time of bt's by 10% (on average) compared with PDGS. And PDGS improves the execution time of hydro2d's by 38% (on average) compared with gang scheduling.

In the case of the Compress&Join algorithm, we can observe that it has reduced the response time of applications but at the expense of increasing the execution time by 50% (on average). The response time of applications has been reduced also by 50% (on average).

Table 7.4: PDGS vs. Compress&Join (load=100%)

	PDGS		Compress&Join	
	cpus	time running	cpus	time running
bt's	24	91	17	160
hydro2d's	10	59	10	81

Table 7.4 compares PDGS with Compress&Join. The *cpus* column shows the number of cpus allocated on average by each policy to each application. The *time running* column shows the time the application has been executing, that is, it is not considered the time that the application has been active but not running. As we can see, PDGS allocates more

processors to bt's than gang+Compress&Join. This is because the goal of PDGS is to ensure that applications reach the target efficiency, and the goal of the Compress&Join is to reduce the number of slots, at the expense of reducing the allocation of running applications.

An interesting effect is that PDGS and Compress&Join has allocated the same number of cpus to hydro2d's, but hydro2d's under Compress&join has consumed a 37% more cpu time that under PDGS. This is because the influence of the number of simultaneously running applications. Figure 7.14 shows the multiprogramming level during the execution of the workload under gang+Compress&Join (top of the figure), and under PDGS (bottom of the figure). The x axis is time and the y axis is the number of applications concurrently running at each moment. We have fit the x scale to the duration of the workload but the y scale is the same in both graphs. We can appreciate how under PDGS there are less applications than under gang+Compress&Join.

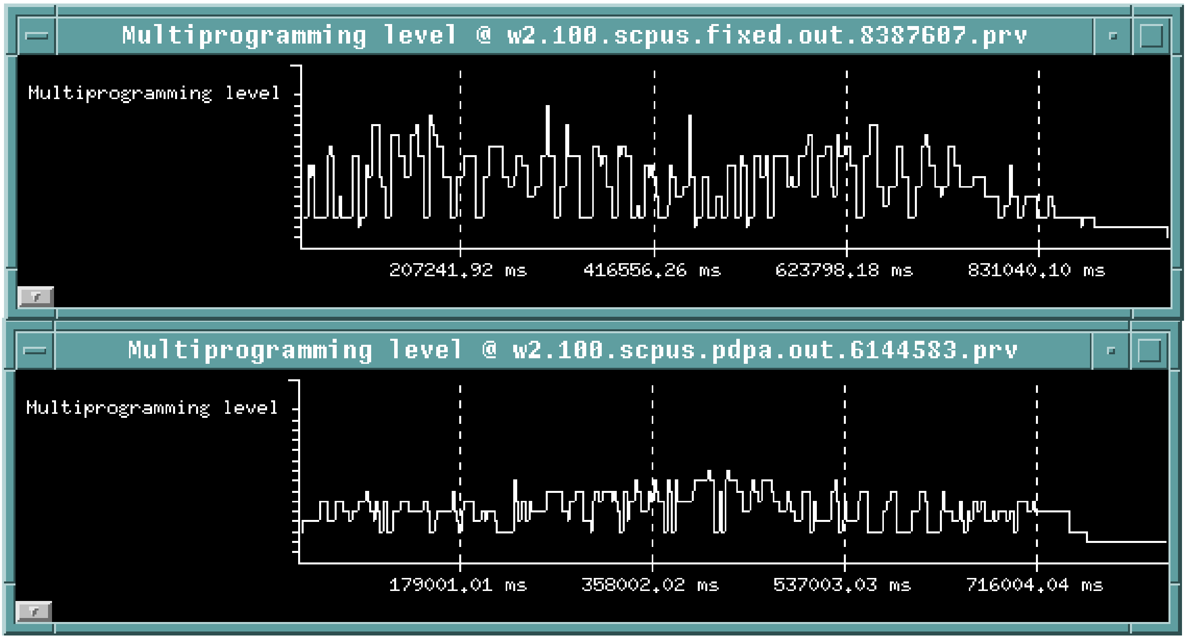


Figure 7.14: Multiprogramming level with Compress&Join and PDGS (load=100%)

Effect of the fragmentation in the workload execution

Workloads used in this Thesis do not generate fragmentation because in most of them applications request for 30 processors and the system uses 60 processors. To give an insight about the potential of exploiting the job malleability, we have executed the workload 2 in such a way that we have generated a bad case for a traditional gang scheduling policy: we have set the request of bt's and hydro2d's to 32 processors. With this modification, a traditional gang scheduling policy is not able to run more than one job per time slot. This situation is possible because in a real system, applications are submitted by different users, and they are not going to tune the request of their applications to fit with the rest of jobs. However, it is obvious that just by reducing in two processors the allocation of each application we could run, at least, two jobs per time slot. We have performed this experiment with gang scheduling and with gang+Compress&Join.

Figure 7.15 shows the trace file visualization of the execution of the workloads with the two configurations. Each line shows the cpu activity, each color represents a different application. We have set the same x scale to compare them. The first trace file corresponds with the execution with gang scheduling. Blue light color means that the corresponding cpu is idle and each other color represents a different job. The system utilization under gang is the 52% and under Gang+Compress&Join is the 91%.

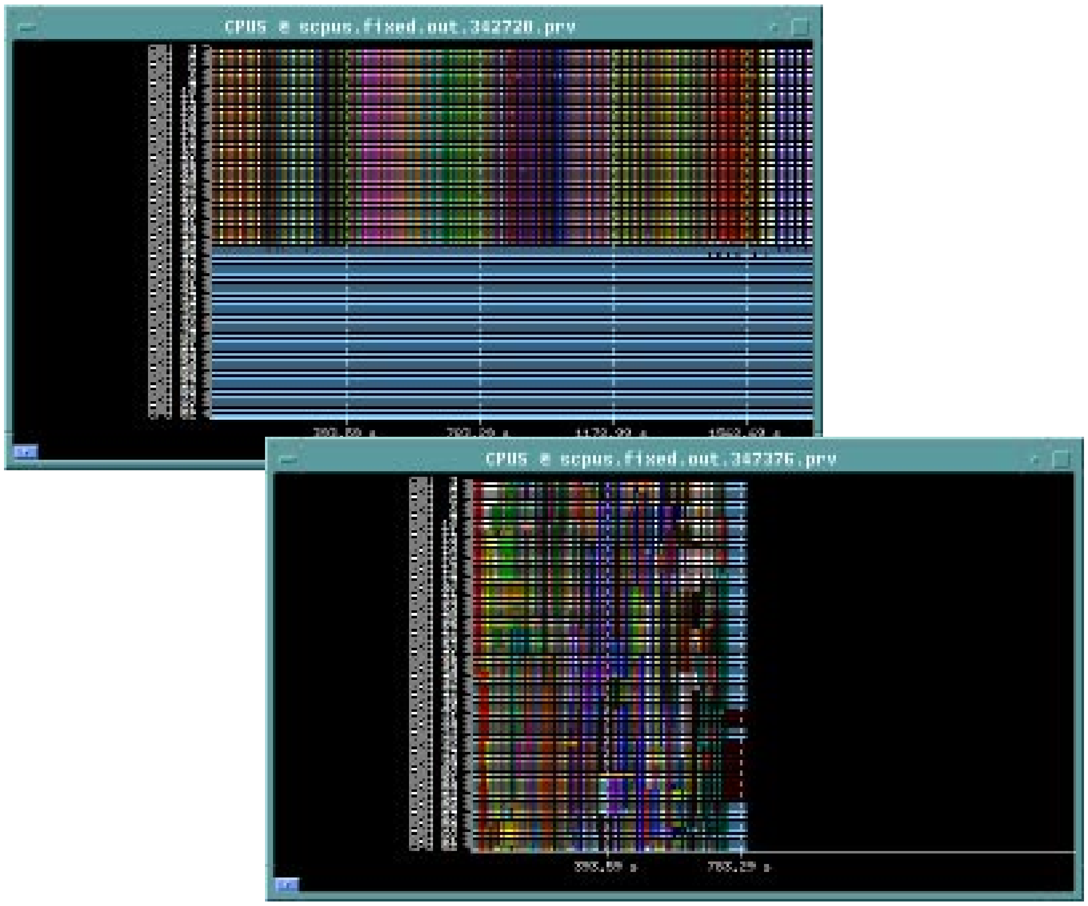

CPUS @ scpus.fixed.out.342720.prv
CPUS @ scpus.fixed.out.347376.prv

Figure 7.15: Workload 2 (req.=32 proc., load=100%) Gang vs. Gang+Compress&Join

We have also measured the number of cpus allocated by the gang+Compress&Join version. Bt's have received 19 processors on average and hydro2d's 11 processors. These values are very similar to those achieved by the workload execution where applications request for 30 processors. In the previous execution bt's receive 17 processors and hydro2d's receive 10 processors, see Table 7.4. We believe that the small difference is due to the different application speedup when executing in alone or with other applications at the same time. As we have commented previously, the speedup of a parallel application not only depend on the number of processors received to run.

Figure 7.16 shows the execution time of workload 2 when setting the request of applications to 30 processors compared with the execution time when setting the request of applications to 30 processors. In gang scheduling the execution time has been increased by 33%. In the case of Compress&Join the execution time has been even reduced by 20%. Comparing gang with gang+Compress&join, gang+Compress&Join has speedup the execution of gang by 219%.

What is very important is that gang+Compress&Join is not significantly affected by the user request. This is a common characteristic of all our proposals: they are very robust to changes in the application request and to changes in the system parameters such as multiprogramming level.

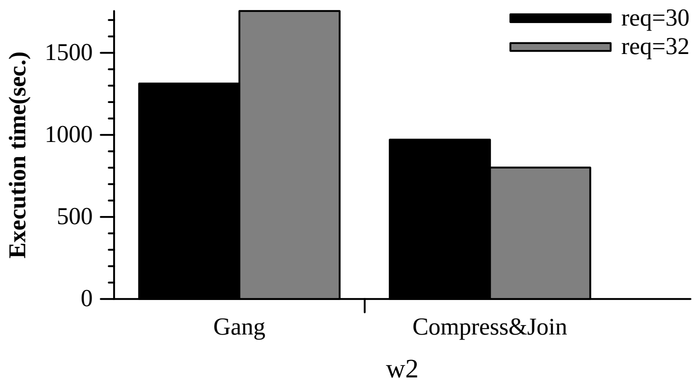


Figure 7.16: Workload 2, (req.=30, load=100%). Gang vs. Gang+Compress&Join

With Compress&Join we can not only improve the system by reducing the number of time slots, but also by reducing the fragmentation that can appear if applications are rigid or the processor allocation policy is not dynamic.

7.5.3 Workload 3

Figure 7.17 shows results for workload 3. Workload 3 is composed by a mix of a 50% of scalable applications (bt's), and a 50% of not scalable applications (apsi's).

Comparing PDGS with gang, we can see that PDGS reaches a similar performance to gang in the response time of bt's, but PDGS outperforms the response time of apsi's. This is because PDGS allocates less processors to bt's, improving the response time of the rest of applications because they can be started before. PDGS improves the response time of bt's by 10%, respect to gang, and the response time of apsi's by 27%.

Compress&Join has introduced an average slowdown of 1% in the response time of bt's, and has improved the response time of apsi's by 22%. In this workload, the fact of having applications requesting two processors generates that the multiprogramming level is very high, in some moments of the workload execution up to 32 applications.

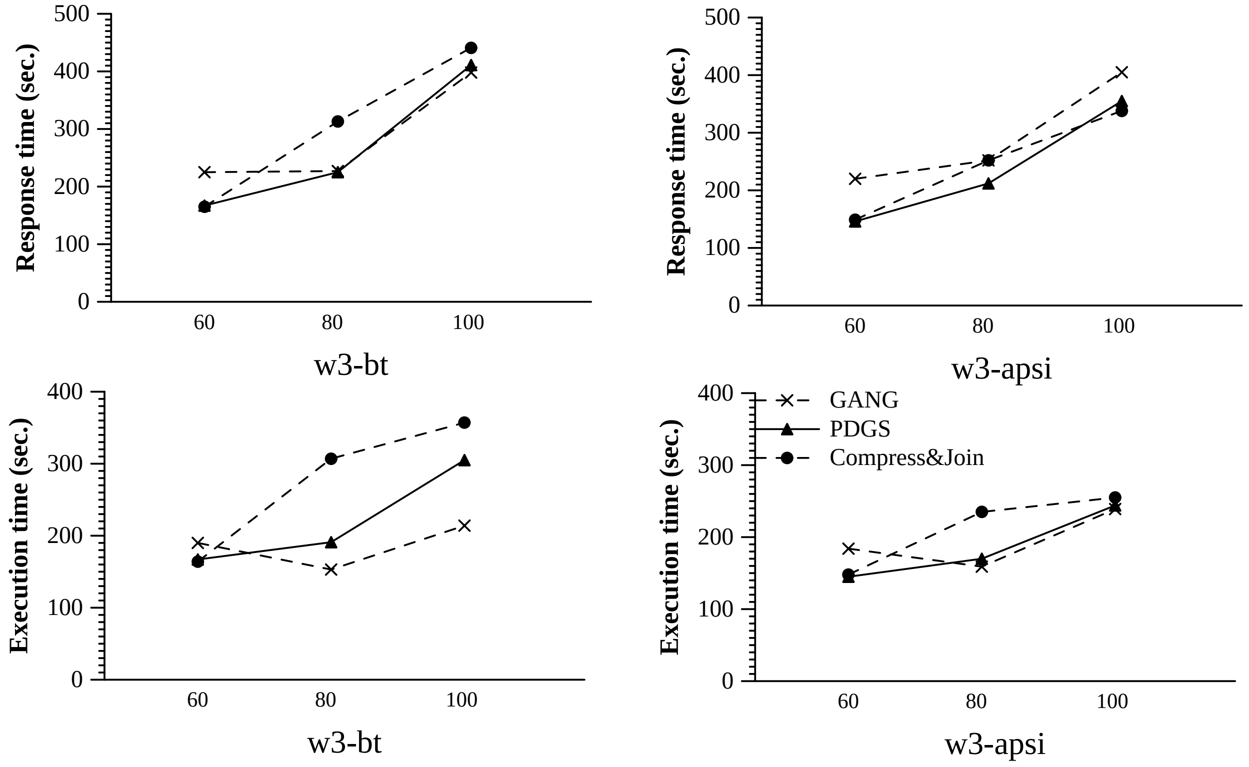


Figure 7.17: Results from workload 3

7.5.4 Workload 4

Figure 7.18 shows results for workload 4. Workload 4 is a mix of the four types of applications: 25% of super-linear applications, 25% of scalable applications, 25% of medium-scalable applications, and 25% of not scalable applications.

Results show that both approaches, PDGS and Compress&Join outperform the baseline gang scheduling. When the load is set to 80%, the particular concurrency of applications generates that results are slightly different than those achieved with the load set to the 60% and 100%. However, on average, PDGS outperforms gang by 248% and the Compress&Join algorithm outperforms gang by 188%. Comparing PDGS and Compress&Join, PDGS shows better results than gang+Compress&Join.

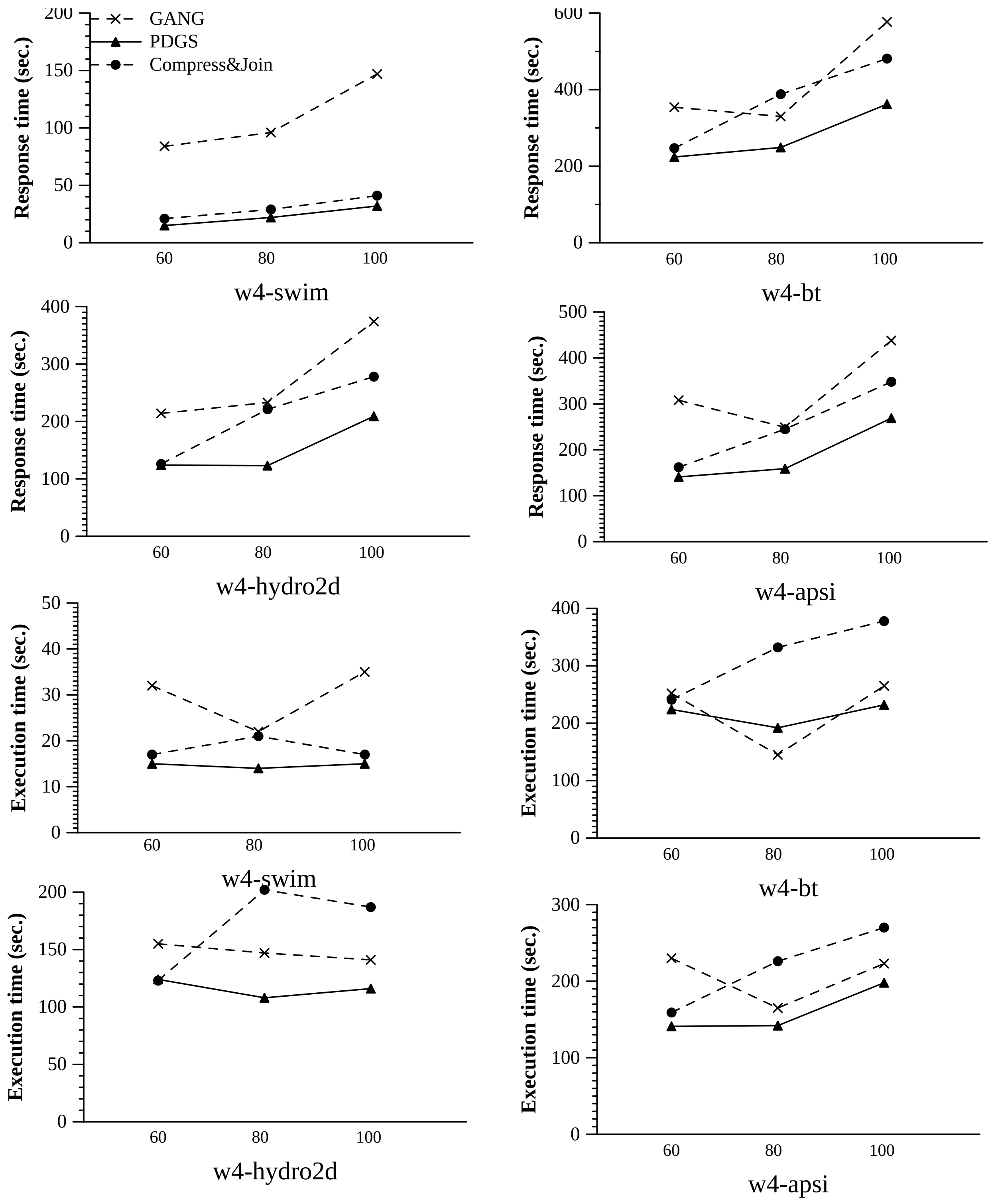


Figure 7.18: Results from workload 4

Table 7.5 shows the number of cpus allocated (on average) to each application per policy, the running time (on average), and the cpu time consumed by these applications when the load is set to the 100%. The cpu usage has been calculated as the number of cpus multiplied by the running time. We have marked in bold font the best result per application. We want to show that using a reduced number of processors, applications can be executed using less total cpu time with a similar execution time. To show that we have analyzed in detail results for workload 4 when the load is set to the 100%.

In the case of swim's, the three policies allocate a high number of processors, and the running times are similar. The policy that achieves the best cpu usage is the baseline, gang scheduling. Swim's under gang consume 11% less cpu time than under PDGS and 6% less than Compress&Join. However, if we analyze the response time achieved by swim's in this workload, Figure 7.18, we can see that both PDGS and Compress&Join significantly outperform gang. This is because PDGS and Compress&Join significantly improve the cpu usage of the rest of applications, resulting in a benefit in the response time of swims. We can see that in the other two policies PDGS and Compress&Join consume much less cpu time than gang, 266% in the case of PDGS and 92% in the case of Compress&Join, and that the running time is also better. We have not presented results for the apsi application because it requests for two processors and then there are no chances to adjust its allocation.

Table 7.5: Cpu usage and running time in workload 4, load=100%

	SWIM			HYDRO2D			BT		
	cpus	running time	cpu usage	cpus	running time	cpu usage	cpus	running time	cpu usage
CJOIN	28	5.65 sec.	158.2 sec.	12	61.3 sec.	735.6 sec.	18	114 sec.	**2052 sec**.
PDGS	27	5.6 sec.	151.2 sec.	11	48.22 sec.	**530.4 sec**.	23	91sec.	2093 sec.
GANG	27	5.29 sec.	**142.8 sec**.	27	52.36 sec.	1413.7 sec.	26	104 sec.	2704 sec.

Figure 7.19 shows the multiprogramming level generated by each policy when the load is set to the 100%. The x axis is the time and the y axis is the multiprogramming level. The graph in the top shows the multiprogramming level generated by gang. Its maximum value has been 20 applications. The second one shows the PDGS multiprogramming level. Its maximum value has been 26 applications. And the last one shows the Compress&Join multiprogramming level. Its maximum value has been 27 applications.

If we observe Table 7.5, the execution time achieved by applications under the different approaches, we will see that PDGS achieves the best results, and that gang reaches better execution times than gang+Compress&Join. As we have commented previously, this is because applications receive less processors when using the Compress&Join algorithm.

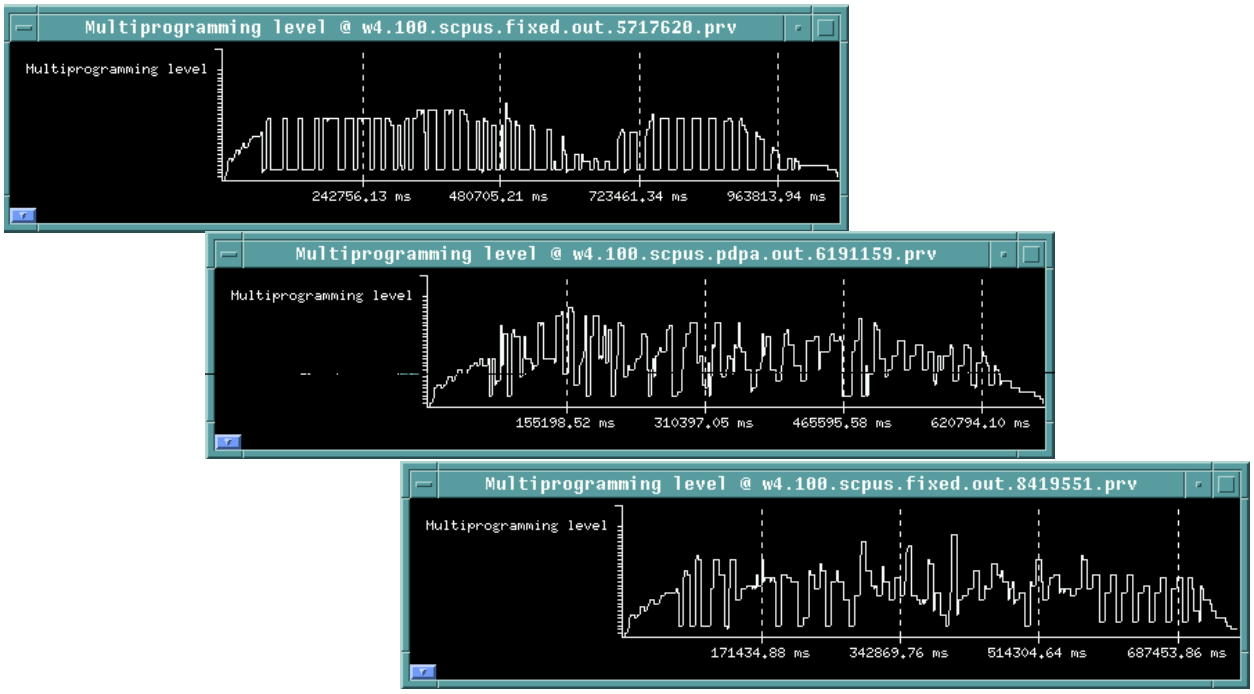


Figure 7.19: Multiprogramming level decided by Gang, PDGS, and Compress&Join, load=100%

7.5.5 Workload 5

Figure 7.20 shows the results for workload 5. Workload 5 is composed by only bt's. As in the previous workloads, the best results are achieved by PDGS. On average, gang+ Compress&Join outperforms gang by 32%, and PDGS outperforms gang by 19%.

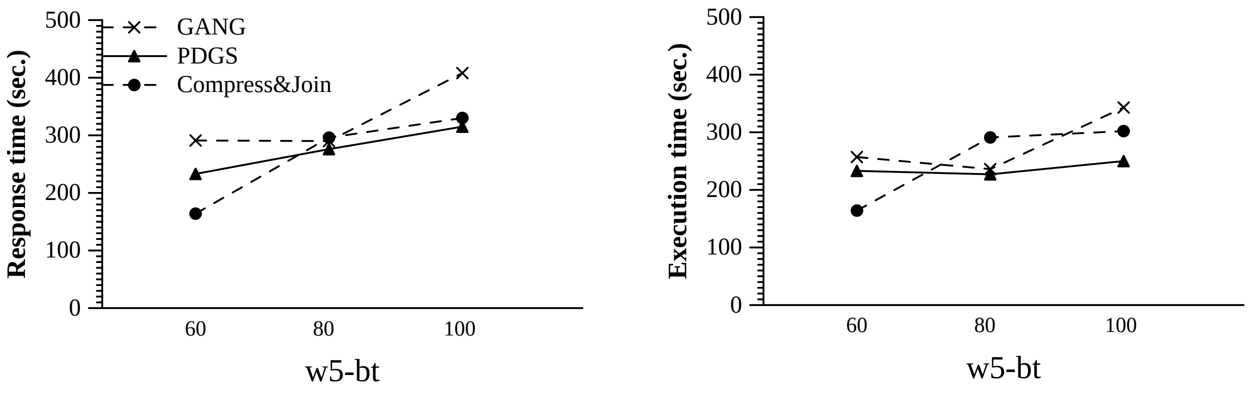


Figure 7.20: Results from workload 5

Table 7.6 shows the number of cpus allocated (on average) by each policy to bt's in the workload 5. As in the previous workloads, PDGS allocates less processors than Gang to applications but consumes less cpu time than Gang, and also consumes less cpu time than

Gang+Compress&Join. Under gang+Compress&Join, applications receive less processors than with the other approaches. If we compare the cpu usage we can see that the best result is achieved by Compress&Join. However, PDGS consumes a 1% more cpu time than Compress&Join, but PDGS outperforms Compress&Join by 11%.

Table 7.6: Cpu allocation and running time in workload 5, load=100%

	BT		
	cpus (avg.)	running time (sec.)	cpu usage
GANG	29	100.51 sec.	2900 sec.
PDGS	24	**94.8 sec.**	2256 sec.
Compress&Join	21	106 sec.	**2226 sec**.

7.5.6 Workload execution times

Figure 7.21 shows the execution time of the five workloads evaluated in this Thesis. We can see that both approaches, PDGS and the Compress&Join algorithm introduce benefits in the execution time of the workload, which is the main goal of this Thesis. We have shown that PDGS and Compress&Join improve the response time of applications, and this has a direct effect in the total workload execution time.

Table 7.7 compares the execution time under gang compared with the execution time under PDGS and Gang+Compress&Join. We show the ratio of the execution time with Gang scheduling and the execution time under each policy. We show the percentage of improvement of our approaches with respect to Gang scheduling. For instance, in the first row, PDGS executes the workload 1 a 14% faster than Gang with load=60%, and a 5% slower with load=80%.

Table 7.7: Percentage of improvement, PDGS and Compress&Join vs. Gang

	60%	80%	100%	AVG.
w1-Gang/PDGS	14%	-5%	-7%	0%
w1-Gang/Gang+CJoin	13%	-3%	-3%	**2%**
w2-Gang/PDGS	75%	36%	56%	**55%**
w2-Gang/Gang+CJoin	76%	25%	35%	45%
w3-Gang/PDGS	42%	17%	21%	**26%**
w3-Gang/Gang+CJoin	38%	7%	13%	19%
w4-Gang/PDGS	54%	38%	54%	**48%**
w4-Gang/Gang+CJoin	45%	21%	40%	35%
w5-Gang/PDGS	15%	0%	25%	13%
w5-Gang/Gang+CJoin	32%	-1%	27%	**19%**

As we can see, both approaches outperform the baseline Gang scheduling, showing the benefit that results from measuring the performance of running applications and adjusting the allocation based on this information in Gang scheduling policies. We have marked in bold type the approach the reaches the best performance (on average) per workload. In three of the five workloads the best performance is reached by PDGS and in two of them by gang+Compress&Join.

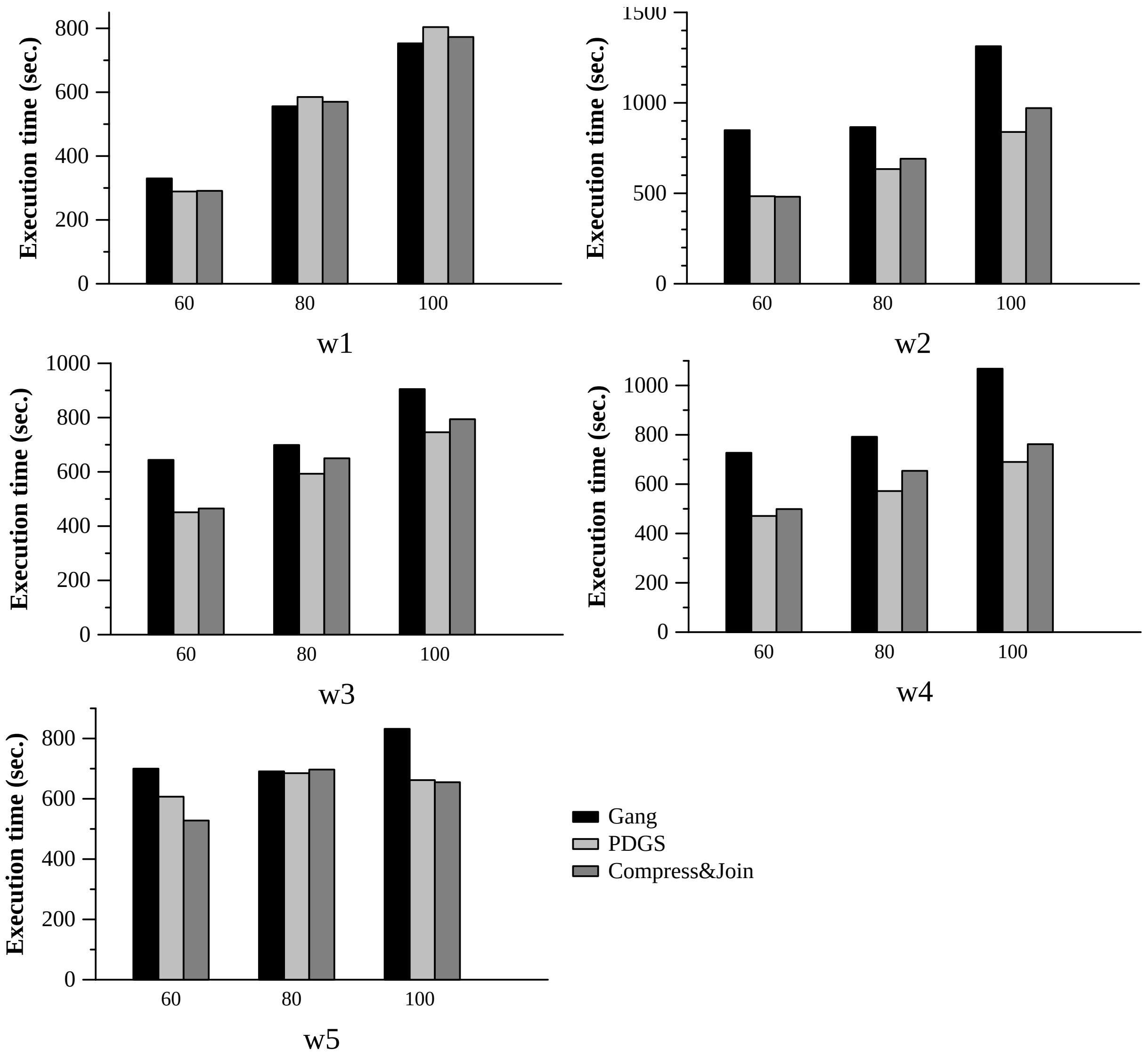


Figure 7.21: Execution time of the complete workloads

7.6 Summary

In this Chapter, we have presented two approaches to improve gang scheduling. Our first proposal consist of adjusting the allocation of running applications based on the performance achieved by them. Applications initially receive the number of processors requested and the system adjust their allocation until they achieve the target efficiency. Once the target efficiency is achieved, the application becomes stable and the system re-applies the re-packing algorithm with the new allocation. This approach is called Performance-Driven Gang Scheduling, PDGS.

The second approach is a job re-packing algorithm, Compress&Join, totally oriented to reduce the number of time-slots in the system. The Compress&Join algorithm reduces the application processor allocation based on the performance achieved. The algorithm imposes a maximum slowdown with respect to the speedup achieved with the requested number of processors. With this reduction, the same set of application can be placed in a reduced number of slots. The aim of this algorithm is both to reduce the fragmentation and to reduce the number of time slots.

We have compared both approaches with a gang scheduling used as baseline that includes a first-fit algorithm, job migration, and the execution of jobs in multiple slots. Results show that both approaches introduce benefits in the execution of the workloads. In some of the cases, the execution time of applications has been increased, but the response time of applications has been improved, resulting in a better workload execution time. We have also shown that the cpu usage is better under PDGS and Compress&Join than under the baseline gang scheduling. This is because processors are more efficiently used.

Finally, an observation based on our experience is that this kind of policies can solve some of the problems presented by space-sharing policies that have a fixed multiprogramming level, such as Equipartition, or Equal_efficiency. On the other hand, results achieved in previous Chapters show that, comparing gang scheduling with dynamic space-sharing policies that have a dynamic multiprogramming level such as PDPA, or Equip++, dynamic space-sharing policies reach better results and are easier to implement.

Our experience also demonstrates us that it is important to dimension the time-sharing quantum to a relatively high value because, otherwise applications can significantly degrade their performance. In the evaluation presented in this Chapter we have used a quantum of six seconds, showing good results. However, in previous evaluations we used a quantum of three seconds and most of them showed bad execution due to a system performance degradation. Reasons were that the overhead generated by the context switch of applications, mainly because of the loss of memory locality, was more significant than the benefit introduced by having a single time slot to execute.