CHAPTER 4

Dynamic Performance Analysis: SelfAnalyzer

Abstract

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

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

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

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

4.1 Introduction

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

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

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

Figure 4.1: Speedup equation

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

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

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

4.2 Related Work

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

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

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

Figure 4.2: Speedup and efficiency definitions

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

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

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

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

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

Figure 4.3: Nguyen proposal to calculate the efficiency

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

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

4.3 Dynamic Performance Analysis: The SelfAnalyzer

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

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

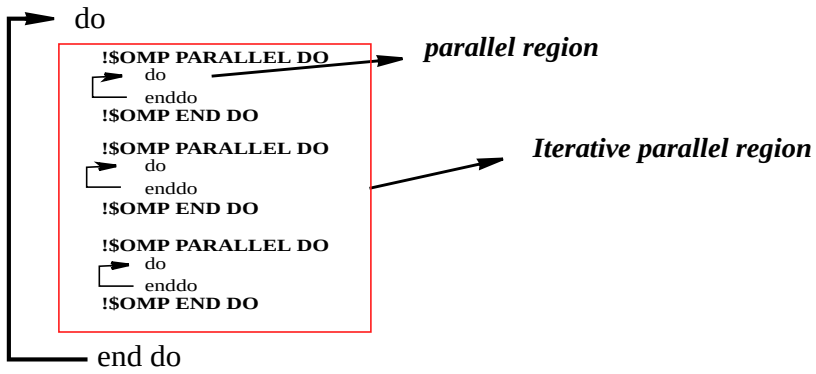


Figure 4.4: Structure of a *Iterative parallel region*

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

4.3.1 Dynamic speedup computation

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

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

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

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

Figure 4.5: Speedup calculation

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

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

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

Amdahl's Factor

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

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

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

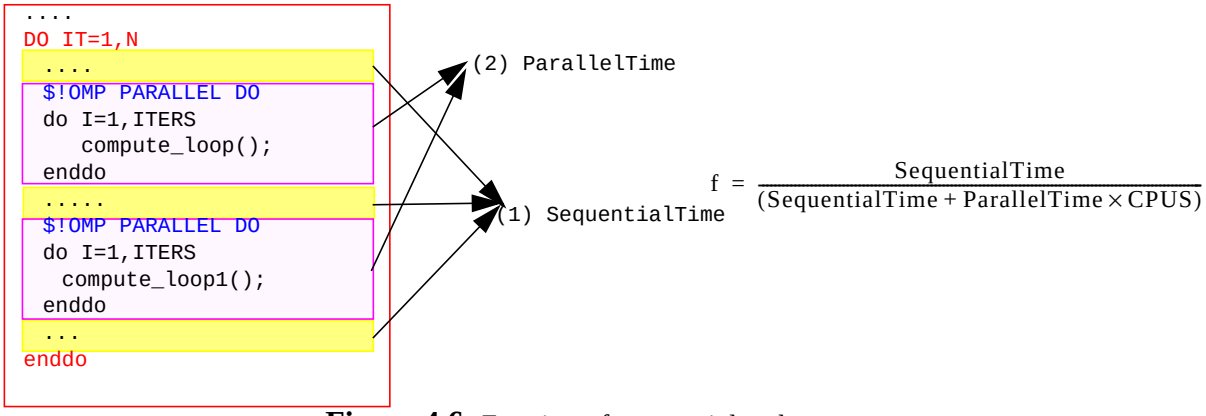


Figure 4.6: Fraction of sequential code

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

Implementation issues

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

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

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

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

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

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

4.3.2 Execution time estimation

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

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

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

Figure 4.8: Execution time estimation

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

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

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

4.4 Application instrumentation

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

4.4.1 SelfAnalyzer interface

Table 4.1 shows the SelfAnalyzer interface.

Table 4.1: SelfAnalyzer API

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

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

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

4.4.2 Static instrumentation

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

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

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

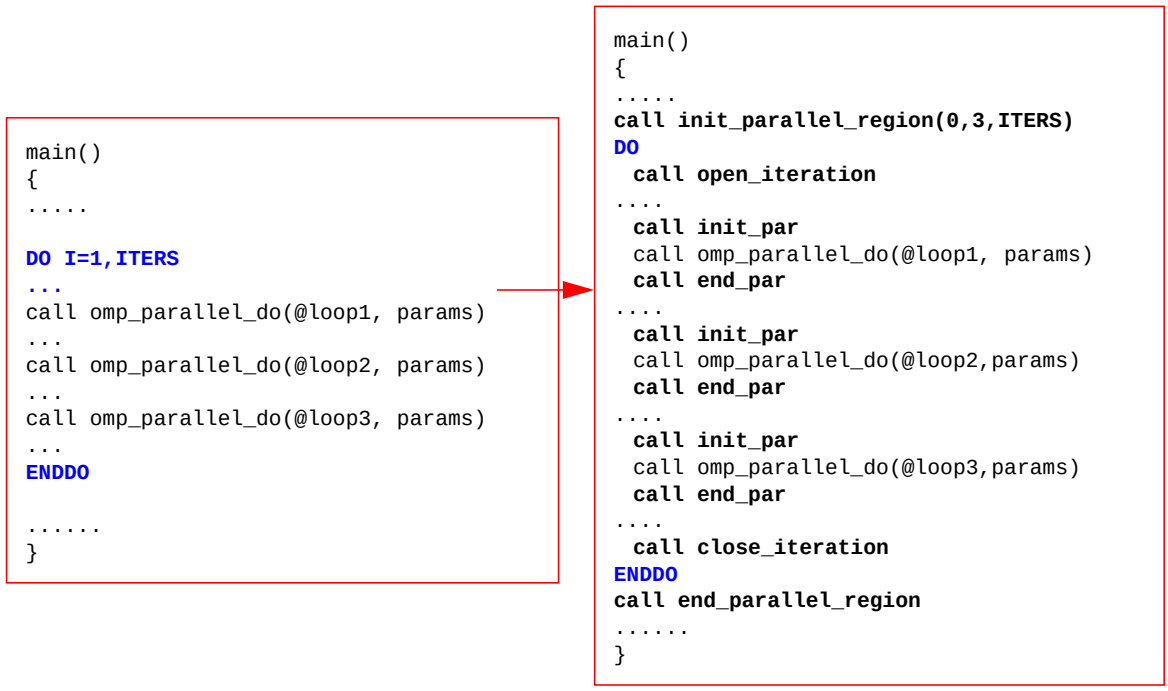


Figure 4.9: Static instrumentation

4.4.3 Dynamic instrumentation

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

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

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

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

Table 4.2: DPD interface.

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

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

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

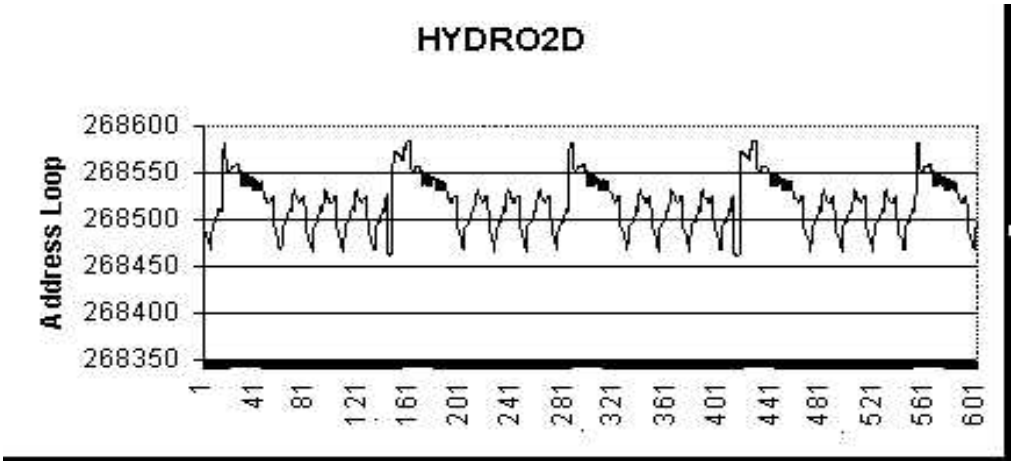


Figure 4.10: Iterative behavior of hydro2d (SpecFP 95)

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

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

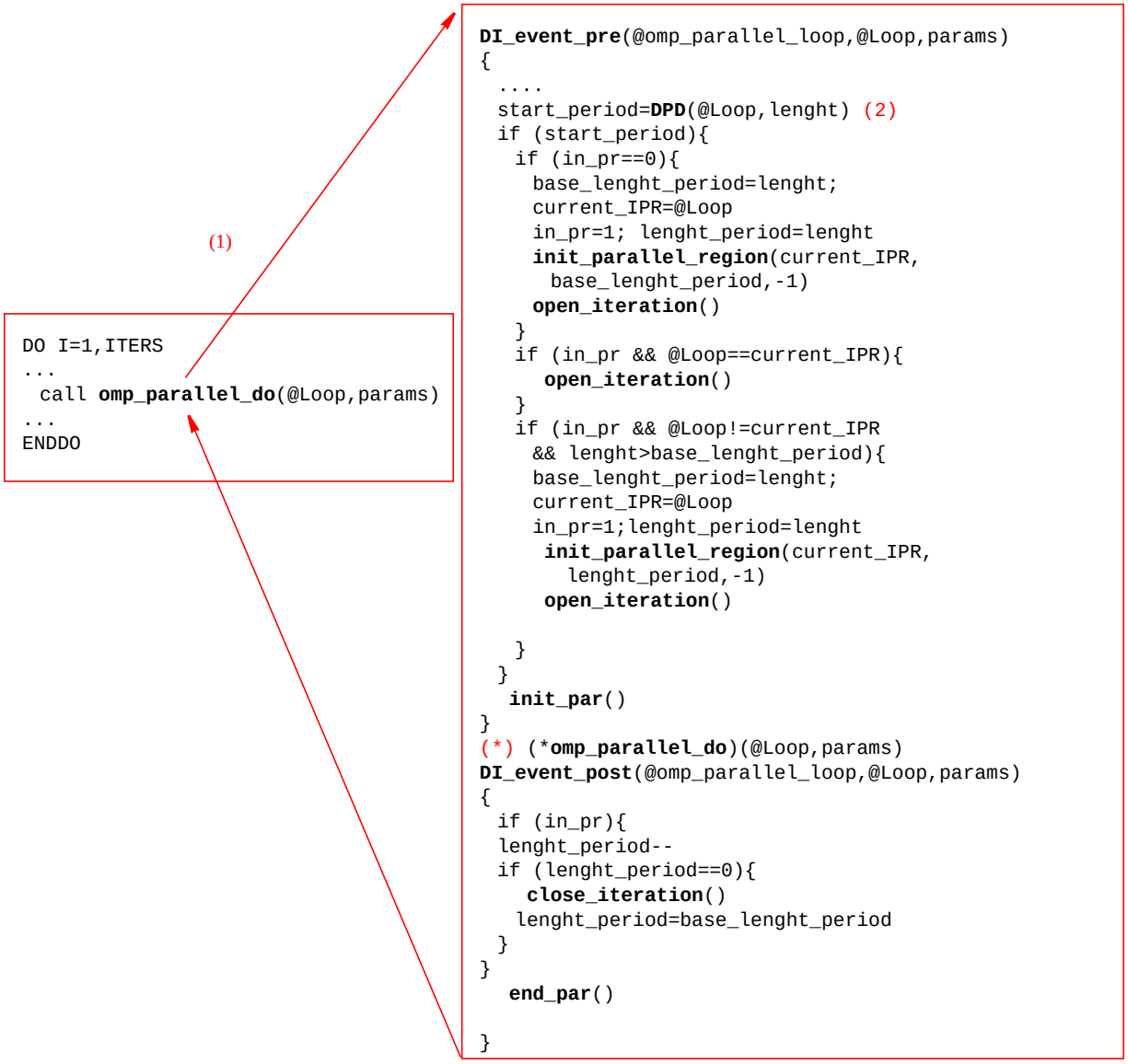
1.It is the first time we detect this IPR. We initialize the IPR by calling *init_parallel_region* and *open_iteration* functions.

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

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

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

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



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

Figure 4.11: Dynamic Instrumentation (DITools + DPD)

4.5 Integration in the execution environment

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

Table 4.3: New functionality

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

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

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

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

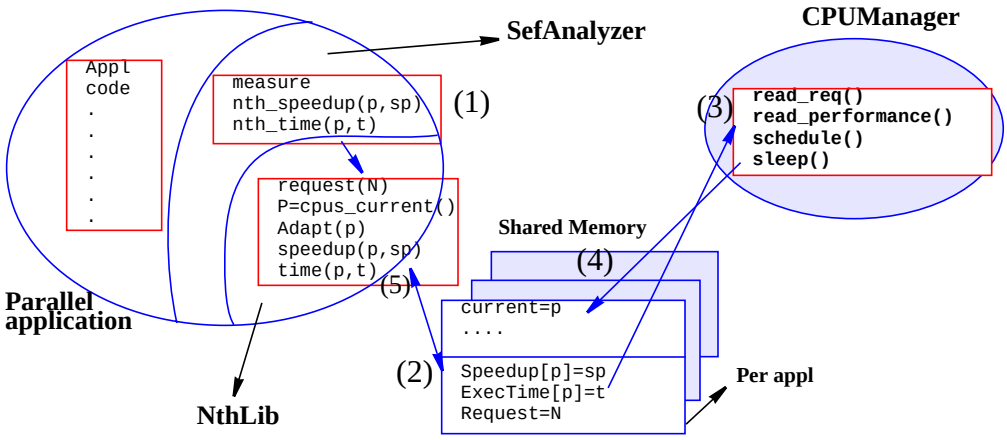


Figure 4.12: SelfAnalyzer integration in the execution environment

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

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

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

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

4.6 Evaluation

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

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

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

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

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

We have executed seven different configurations:

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

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

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

4.6.1 Tomcatv

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

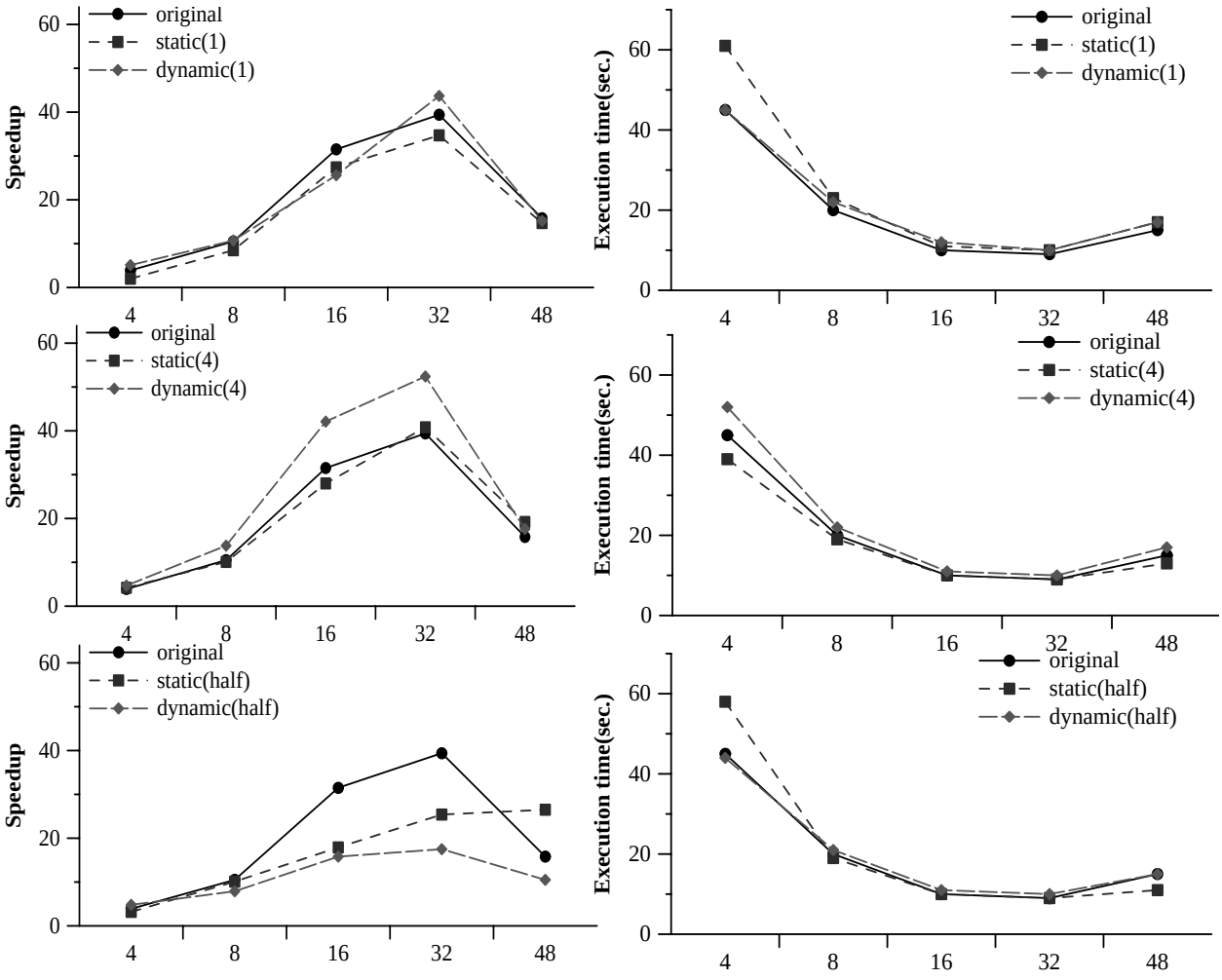


Figure 4.13: Tomcatv evaluation

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

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

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

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

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

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

4.6.2 Hydro2d

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

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

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

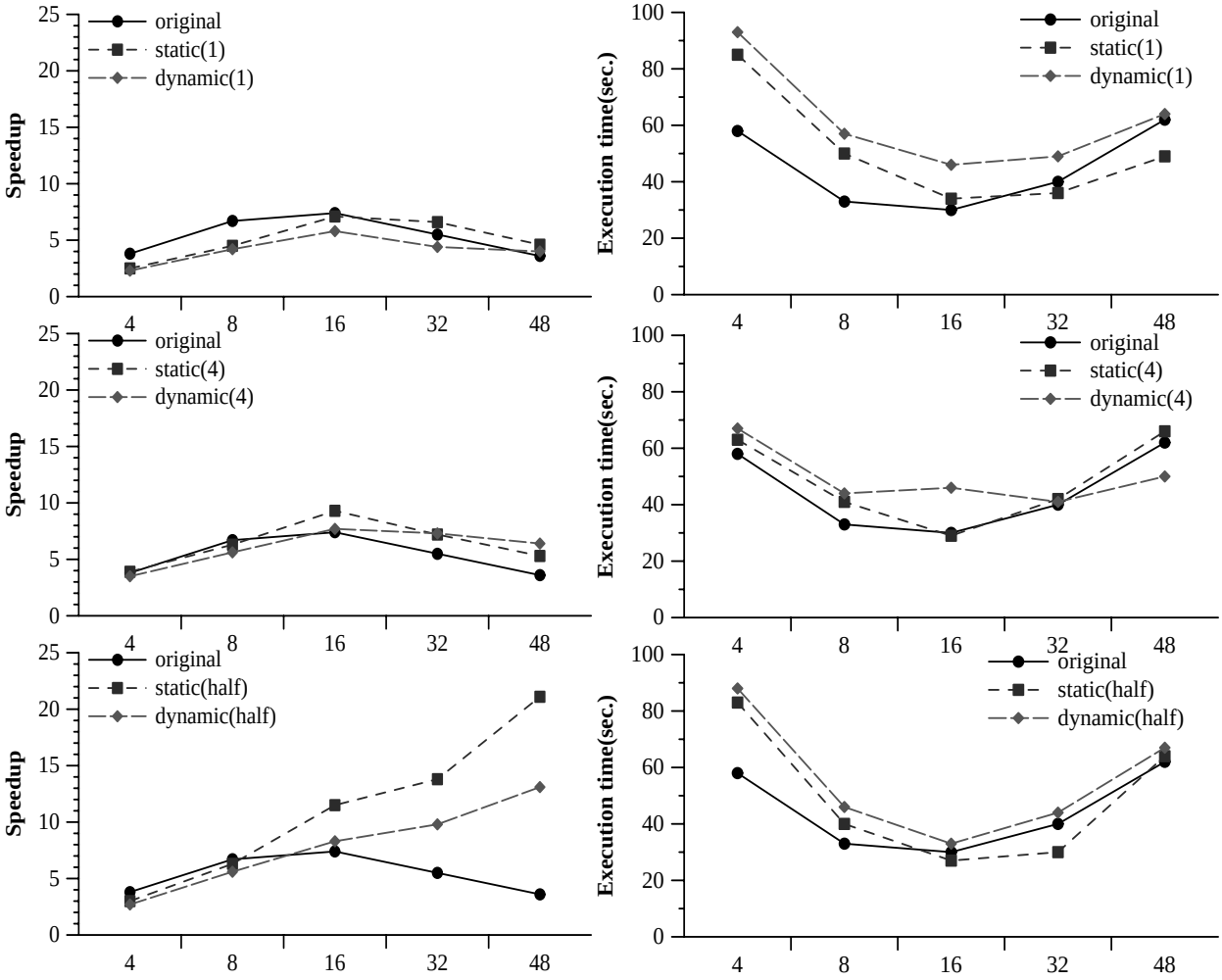


Figure 4.14: Hydro2d evaluation

4.6.3 Bt

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

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

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

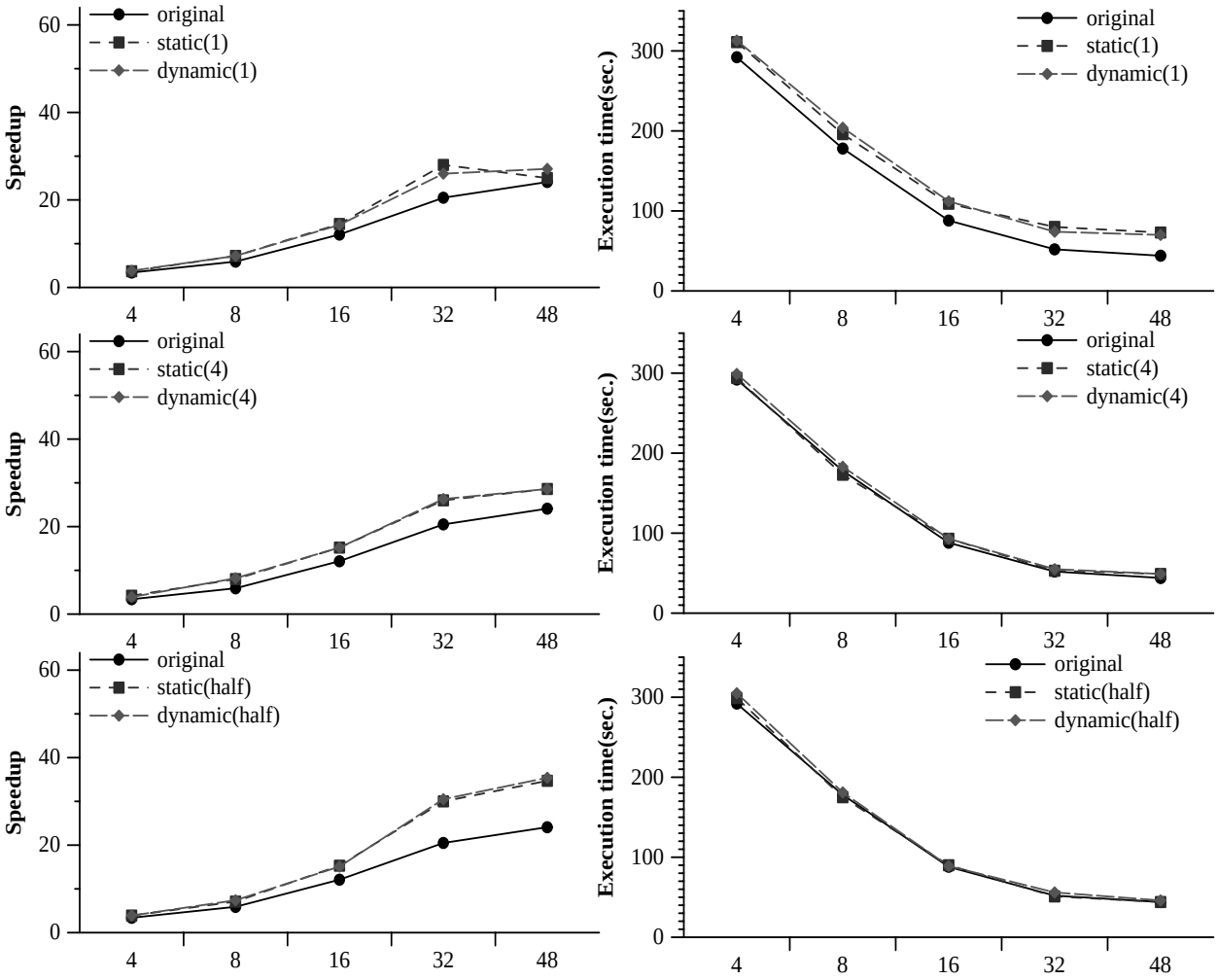


Figure 4.15: Bt evaluation

4.6.4 Swim

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

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

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

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

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

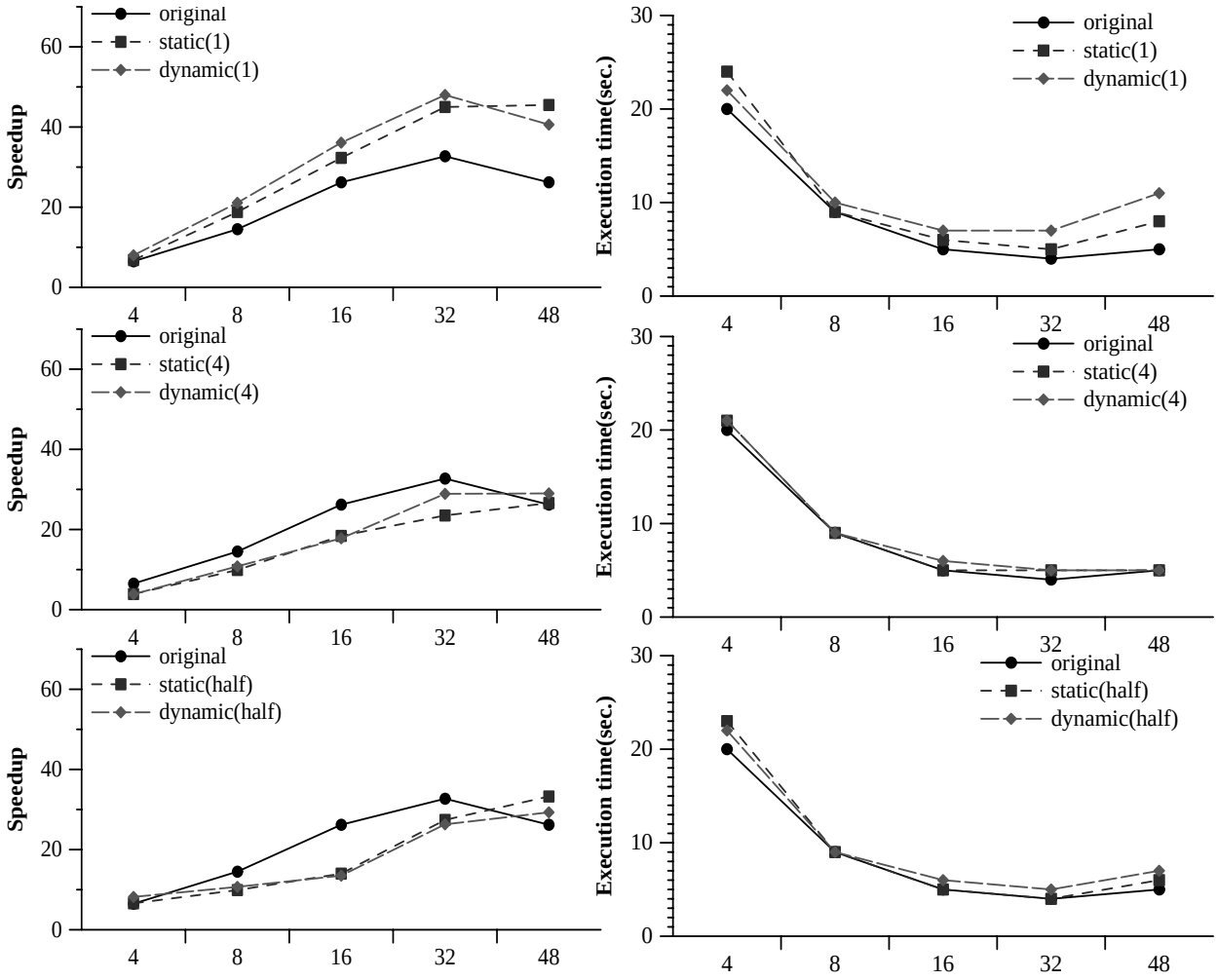


Figure 4.16: Swim evaluation

4.6.5 Apsi

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

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

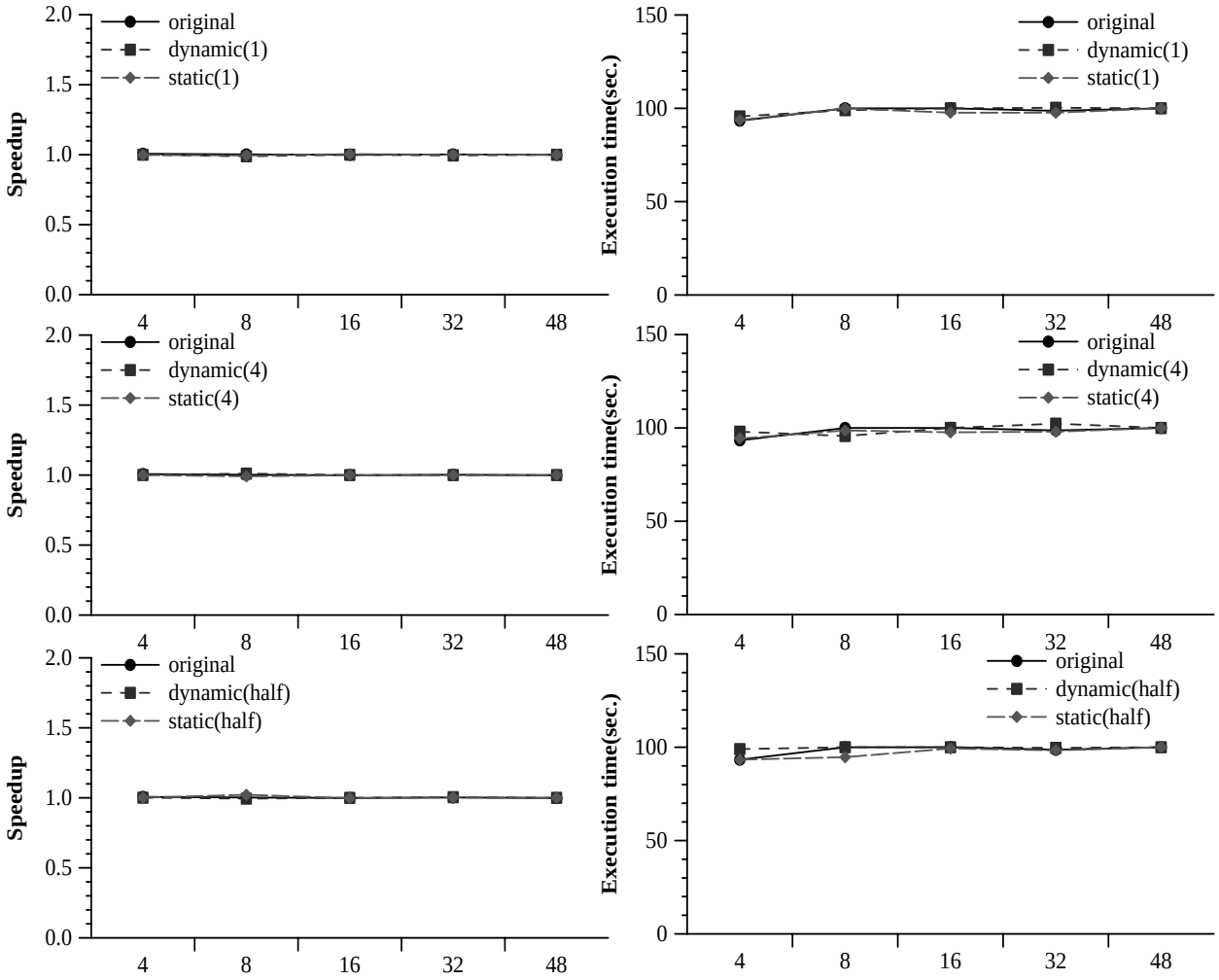


Figure 4.17: Apsi evaluation

4.7 Summary

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

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

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

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

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