Spectral Analysis of Executions of Computer Programs and its Applications on Performance Analysis

Marc Casas Guix

Barcelona, 2009

Advisors
Rosa M. Badia Sala
Jesús Labarta Mancho

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor per la UPC
Barcelona Supercomputing Center / Department of Computer Architecture
Technical University of Catalonia

Abstract

This work is motivated by the growing intricacy of high performance computing infrastructures. For example, supercomputer MareNostrum (installed in 2005 at BSC) has 10240 processors and currently there are machines with more than 100.000 processors. The complexity of this systems increases the complexity of the manual performance analysis of parallel applications. For this reason, it is mandatory to use automatic tools and methodologies. The performance analysis group of BSC and UPC has a large experience in analyzing parallel applications. The approach of this group consists mainly in the analysis of tracefiles (obtained from parallel applications executions) using performance analysis and visualization tools, such as Paraver. Taking into account the general characteristics of the current systems, this method can sometimes be very expensive in terms of time and inefficient.

To overcome these problems, this thesis makes several contributions. The first one is an automatic system able to detect the internal structure of executions of high performance computing applications. This automatic system is able to rule out non-significant regions of executions, to detect redundancies and, finally, to select small but significant execution regions. This automatic detection process is based on spectral analysis (wavelet transform, fourier transform, etc..) and works detecting the most important frequencies of the application's execution. These main frequencies are strongly related to the internal loops of the application' source code. Finally, it is important to state that an automatic detection of small but significant execution regions reduces remarkably the complexity of the performance analysis process.

The second contribution is an automatic methodology able to show general but non-trivial performance trends. They can be very useful for the analyst in order to carry out a performance analysis of the application. The automatic methodology is based on an analytical model. This model consists in several performance factors. Such factors modify the value of the linear speedup in order to fit the real speedup. That is, if this real speedup is far from the linear one, we will detect immediately which one of the performance factors is undermining the scalability of the application. The second main characteristic of the analytical model is that it can be used to predict the performance of high performance computing applications. From several execution on a few of processors, we extract model's performance factors and we extrapolate these values to executions on higher number of processors. Finally, we obtain a speedup prediction using the analytical model.

The third contribution is the automatic detection of the optimal sampling frequency of applications. We show that it is possible to extract this frequency using

spectral analysis. In case of sequential applications, we show that to use this frequency improves existing results of recognized techniques focused on the reduction of serial application's instruction execution stream (SimPoint, Smarts, etc..). In case of parallel benchmarks, we show that the optimal frequency is very useful to extract significant performance information very efficiently and accurately.

In summary, this thesis proposes a set of techniques based on signal processing. The main focus of these techniques is to perform an automatic analysis of the applications, reporting and initial diagnostic of their performance and showing their internal iterative structure. Finally, these methods also provide a reduced tracefile from which it is easy to start manual fine-grain performance analysis. The contributions of the thesis are not reduced to proposals and publications. The research carried out these last years has provided a tool for analyzing applications' structure. Even more, the methodology is general and it can be adapted to many performance analysis methods, improving remarkably their efficiency, flexibility and generality.

Contents

1 Introduction **9**

1.1 General Considerations . . . 9

1.2 Why is Spectral Analysis Useful to Study Performance of Applications? 10

1.3 Contributions . . . 11

1.3.1 Automatic Phase Detection and Structure Extraction of MPI Applications . . . 12

1.3.2 Speedup Analysis of MPI Applications . . . 13

1.3.3 Extraction of Optimal Sampling Frequency of Applications . . . 14

1.4 Thesis Organization . . . 15

2 Background and Context **17**

2.1 Parallel Programming . . . 17

2.1.1 Shared-Memory Programming . . . 17

2.1.2 Message Passing . . . 19

2.1.3 Data Parallel Languages . . . 22

2.1.4 Remote Procedure Call and Active Messages . . . 23

2.2 Performance Analysis Tools . . . 24

2.2.1 Overview of Performance Analysis Tools . . . 24

2.2.2 Collecting and Studying Performance Data . . . 25

2.2.2.1 Tracefile Generation . . . 26

2.2.2.2 Tracefile Format . . . 27

2.2.2.3 Visualization and Analysis . . . 29

2.3 Signal Processing . . . 30

2.3.1 Wavelet Transform . . . 31

2.3.1.1 Definition . . . 31

2.3.1.2 Fast Wavelet Transform . . . 32

2.3.1.3 General Considerations . . . 35

2.3.2 Cross-correlation and Autocorrelation . . . 35

2.3.2.1 Definition 36
2.3.2.2 Autocorrelation 36
2.3.2.3 Computation of Cross-correlation 38
2.3.2.4 General Considerations 38
2.3.3 Mathematical Morphology 39
2.3.3.1 Definitions 39
2.3.3.2 Computation 40
2.3.3.3 General Considerations 40

3 Automatic Phase Detection and Structure Extraction of MPI Applications 43
3.1 Introduction 43
3.1.1 Definition of Structure Extraction 43
3.1.2 Overview of the Structure Extraction Process 44
3.1.3 Applications of Structure Extraction in Performance Analysis . . 45
3.2 Related Work 46
3.3 Metrics Generation 48
3.3.1 Number of Processes Computing 49
3.3.2 Sum of Durations of Computing Bursts 49
3.3.3 Number of Point to Point MPI Calls 50
3.3.4 Number of Collective MPI Calls 50
3.3.5 Instructions per Cycle 50
3.3.6 Communications 50
3.3.7 Examples 51
3.4 Program Phases 51
3.4.1 Description of the Program Phases 53
3.4.2 Detection of the Program Phases 53
3.5 Corrupting Factors 56
3.5.1 Flushing 56
3.5.2 Preemptions 57
3.5.3 Clogged System 59
3.6 Structure 60
3.6.1 Iterative Pattern Detection 61
3.6.2 Representative Iterations 63
3.6.3 Hierarchy detection 63
3.7 Evaluation 64
3.7.1 Evaluation of Complete Internal Structure and Hierarchical Searching 64

3.7.2 Evaluation of Metrics 68
3.7.3 Evaluation of Tracefile Size Reductions 73
3.7.4 Scalability of the Analysis 76

4 Speedup Analysis of MPI Applications 77
4.1 Introduction 77
4.2 Related Work 78
4.3 Parallelization Efficiency 81
4.3.1 Speedup Extraction 81
4.3.2 Speedup Model 81
4.3.2.1 Initial Parameters 82
4.3.2.2 Derivation of a Partial Model 83
4.3.2.3 Derivation of the Final Model 85
4.4 Evaluation 86
4.4.1 Size Reduction 86
4.4.2 Results Obtained with the Performance Model 87
4.5 Prediction 93
4.5.1 Prediction of the Execution Time 93
4.5.2 Prediction of the Performance Undermining Factors 95

5 Optimal Frequency Extraction 99
5.1 Introduction 99
5.2 Related Work 101
5.3 Minimal Representative Subset Detection 102
5.3.1 Results 103
5.3.1.1 Sequential Benchmarks 103
5.3.1.2 Taking the Length Arbitrarily 106
5.3.1.3 Parallel Benchmarks 109
5.4 Clustering 111
5.4.1 Description of the Experiments 112
5.4.2 Evaluation 115
5.5 Optimal Information Extraction 119

6 Conclusions and Future Work 123
6.1 Conclusions 123
6.1.1 Automatic Phase Detection and Structure Extraction of MPI Applications 123
6.1.2 Speedup Analysis of MPI Applications 124
6.1.3 Extraction of Optimal Sampling Frequency of Applications 125

6.2 Future Work 126

Chapter 1
Introduction

1.1 General Considerations

In the last years, high performance computing has experienced major successes. On the one hand, parallel computing infrastructures have achieved performance improvements which were unthinkable several years ago [7]. On the other hand, high performance computing has become an essential tool for a wide set of topics such as meteorology, genomics, computational mechanics, cosmology, nuclear energy, aeronautic engineering, etc...

These achievements have brought several challenges which have to be addressed in order to maintain high performance computing in the way of success. First, the increase of computational power brings more hardness to obtain optimized codes because there are more issues related to parallel machines which have to be addressed. Second, the spread of high performance computing solutions brings problems to researchers of other scientific disciplines which are not familiarized to work with supercomputers. They design the codes which have to be executed on parallel infrastructures but, due to their lack of experience, they spend a lot of time implementing and optimizing it. In summary, we have that it is more difficult to implement parallel computing infrastructures and that the people who have to do that it is not familiarized with high performance computing world.

The solution of the problems described above is to provide tools and methodologies which can make easier the process of developing and tuning parallel codes. These tools have to be able to provide a friendly and easy to understand environment to scientists in order to warrant a fast and easy developing process. The objective is to eliminate the tedious and repetitive tasks and to automatize as much as possible the performance analysis of parallel codes. Human intervention can not be, for the moment, totally eliminated from the process of code implementation and performance evaluation of it. However, things like the size of data which have to be analyzed or the tedious work focused on the understanding of very general performance properties can be remarkably reduced, as we demonstrate in this work.

1.2 Why is Spectral Analysis Useful to Study Performance of Applications?

There are a lot of tools that come from the world of Data Mining [37] or Multivariate Statistics [92] which can be applied to data extracted from executions of high performance computing applications. However, it is important to take into account the general properties of these applications in order to determine the kind of techniques which can be useful for our proposals.

Remember that a remarkably relevant part of high performance computing applications are focused on the resolution of numerical issues such as systems of partial differential equations, systems of linear equations, non-linear equations, ... Typically these numerical issues are related with problems arisen from meteorology, mechanical engineering, cosmology, quantum dynamics... These algorithms, designed from numerical analysis theory, have a strong iterative character because they try to find a good approximation of the solution improving, iteration after iteration, an initial value. The executions of these codes spend most of the time on the execution, one time after another, of code inserted in a loop.

For example, many of the applications included in the well known NAS Parallel Benchmarks [3] carry out this kind of calculations. Concretely, the codes Multigrid (MG), Conjugate Gradient (CG), 3-D FFT PDE (FT), LU solver (LU), Pentadiagonal Solver (SP) and Block Tridiagonal Solver (BT) have strong iterative behavior because they are based on typical numerical algorithms, like many high performance computing applications. In chapter 3 we explain how it is possible to detect these iterations and we provide many concrete examples.

Due to this typical periodic behavior, the study of executions of applications from the frequency-domain point of view becomes interesting. Assume that we extract a time-varying metric from the application's execution. We call signal to this time varying metric. Typically, a time-domain representation shows how this signal changes over time. On the other hand, a frequency-domain representation shows how much of the signal remains within each given frequency band over a range of frequencies. A frequency-domain representation can also provide information on the phase shift that must be applied to each sinusoid in order to be able to recombine the frequency components to recover the original time signal. Therefore, frequency domain representation shows a complete depiction of iterative behavior of the application, provides information about the loops, about how they behave and enables to know what is the application doing during its execution. If it is possible to summarize the main behavior of an application's execution using a minimal and representative segment of the execution which is repeated many times, spectral analysis and frequency-domain decomposition will show the path to do this.

The potential and usefulness of spectral analysis applied to study the executions of computer applications is clear. First, it makes possible to detect the iterative behavior, very common in high performance computing applications, enabling the possibility to reduce the amount of data which has to be analyzed to understand the main interesting properties of the execution. Second, it also provides the value of the main frequency detected on the execution. Since this main frequency provides a length that summarizes the behavior of the whole execution, it is useful to determine a minimal representative execution segment and to perform an optimal sampling of the execution. All these ideas are amplified, discussed, implemented and evaluated in this thesis, providing evidence of the advantages of spectral analysis techniques in this context.

1.3 Contributions

In this section, we describe the main contributions of this thesis. Mainly, these contributions can be classified in three categories. The first one makes reference to the application of spectral analysis techniques to message passing programs in order to automatically detect their iterative structure. This detection provides a remarkable reduction of several orders of magnitude of the size of data which has to be analyzed to study the performance of these applications. The second category makes reference to the formulation and evaluation of an analytical model of the speedup of message passing applications. This model is extracted from the reduced information obtained applying the contributions of the first category. It provides an automatic understanding of general but non-trivial performance problems of message passing applications. Finally, the third category is focused on the application of spectral analysis to detect the optimal sampling frequency of applications. This information enables us to select a minimal representative subset of applications and remarkably improves existing techniques of analysis and selection of representative segments of applications.

1.3.1 Automatic Phase Detection and Structure Extraction of MPI Applications

The process of obtaining useful tracefiles from message passing applications for performance analysis in supercomputers is large and tedious. If we execute applications on hundreds or thousands of processors, the tracefile size will grow up to 10 or 20 GB. These amounts of data cannot be directly handled by the typical visualization tools [62, 52]. The analyst can reinstrument the source code in order to obtain a reduced amount of data. This option implies execute again the application and requires a deep code understanding. The analyst can try to reduce directly the size of tracefile cutting or filtering it. However, important information can be lost in this process. In summary, there is a need of reduction the size of tracefiles avoiding the losing of data and the wasting of time.

The first contribution of this thesis is a methodology, based on spectral analysis techniques, which is able to obtain internal structure of tracefiles and to select meaningful parts of it. The underlying philosophy is to rule out the useless and redundant information. The techniques used to perform this reduction are based on several mathematical concepts, such us wavelet transform, mathematical morphology and cross-correlation of signals.

This search for structure has two important advantages. First, its hierarchical character allows to exploit the existence of nested loop in order to assure the maximum possible reduction of the tracefile size. Second, this methodology can use any kind of metric as input, assuring its generality and flexibility.

In summary, the automatic methodology derived from this contribution gives the internal structure of the tracefile, provides the most representative regions of it and, therefore, reduces the problem of analyzing a huge tracefile (10 or 20 GB) to the study of several hundreds of MB. The work performed in this area has resulted in the following publications:

Marc Casas, Rosa M. Badia, Jesús Labarta, **Automatic Extraction of Structure of MPI Applications Tracefiles**, In Proceedings of 13th International Euro-Par Conference (Euro-Par), 2007. *Best Paper Award*

Marc Casas, Rosa M. Badia, Jesús Labarta, **Automatic Phase Detection of MPI Applications**, In Proceedings of Parallel Computing: Architectures, Algorithms and Applications (ParCo), 2007

1.3.2 Speedup Analysis of MPI Applications

The elaborateness of high performance computing applications has been growing very fast in the last years. Only experimented analysts are able to determine the factors that are undermining the performance of up-to-date applications. Analyst time is a very expensive resource and, for that reason, a strong effort to develop automatic performance analysis methodologies has been made by the scientific community.

In this contribution, we propose a methodology that is able to automatically detect the main performance problems of applications. These problems are very general but they are not trivial. The main goal is to automatize as much as possible the analyst time, eliminating tedious and repetitive work. The methodology is based on, first, a size reduction of the performance data obtained from the executions and, second, an analytical model obtained from this performance data which fits the speedup of the applications in terms of several parameters. Such parameters are related to several performance issues. This analytical model gives general performance trends, guiding the analyst and reducing the initial search for performance problems. We have obtained results from real up-to-date applications and we have validated the conclusions automatically derived from the methodology.

Another significant application of the analytical model is performance prediction. It is possible to predict values of performance factors of executions on many processors. These predictions are carried out from real values obtained from execution on a few of processors. Finally, the model is applied to extract a speedup prediction.

In summary, this contribution provides a methodology which is able to automatically detect non-trivial performance characteristics of applications and to predict their behavior if we increase the number of processors. Since the methodology is based on, first, a size reduction performed using fast signal processing algorithms and, second, on a direct extraction of parameters, the automatic analysis is obtained very quickly. On the other hand, our analysis scheme is flexible in the sense that any performance property could be taken into account and its impact in the speedup could be evaluated numerically. Finally, this methodology can be applied either after the execution of the application or during the execution. The work performed in this area has resulted in the following publications:

Marc Casas, Rosa M. Badia, Jesús Labarta, **Automatic analysis of speedup of MPI applications**, In Proceedings of the 22nd ACM International Conference on Supercomputing (ICS), 2008.

Marc Casas, Rosa M. Badia, Jesús Labarta, **Prediction of Behavior of MPI Applications**, In Proceedings of the 2008 IEEE International Conference on Cluster Computing (Cluster), 2008.

1.3.3 Extraction of Optimal Sampling Frequency of Applications

Several applications have been proposed as benchmarks to evaluate the adequateness of architectures. The performance of these benchmarks is used to determine the strength and the weaknesses of the architectures. Therefore, the performance evaluation of benchmarks is a key factor in the process of designing new architectures. The problems are the high amount of data generated by real executions and the impossibility to perform a detailed simulation of the whole execution in a theoretical architectures. To address these problems, a lot of approaches have been proposed to reduce the application's instruction execution stream [83], [103], [50]. However, none of the existing proposals address the problem of obtaining the optimal sampling frequency. They consider the sampling frequency as an input parameter that has to be provided by the user or they simply modify it to obtain lower errors.

In this contribution, we demonstrate that obtaining the optimal sampling frequency improves remarkably the accuracy of the existing techniques and also enables the possibility to obtain consecutive representative subsets of the application's instruction execution stream. We show how the methodology developed to reduce the size of tracefiles, based on spectral analysis, is also useful to obtain the optimal frequency and we evaluate it on SPEC2000 benchmarks [93]. We also evaluate the interactions between our proposal and the existing techniques.

We also show how it is possible to extract relevant performance data from executions of applications in a very efficient way. It reduces the computational power required to extract some metrics. We apply these ideas to NAS Parallel Benchmarks.

In summary, spectral analysis is a powerful tool that has to be used in the context of analysis of applications because it improves the current methodologies if it is used as input of the currently existing tools, opens new perspectives in this topic if it is used alone and, finally, does not increase significantly the overhead of the analysis. The contributions in this area have resulted in the following work:

Marc Casas, Harald Servat, Rosa M. Badia, Jesús Labarta, **Analyzing the Temporal Behavior of Applications Using Spectral Analysis** Technical Report: UPC-DAC-RR-CAP-2009-9

1.4 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 introduces and briefly explains several generalities which are widely used in the rest of the thesis. Chapter 3 explains the automatic search of MPI applications' tracefiles structure and an evaluation of it. Chapter 4 explains and demonstrates the analytical speedup model and shows several cases of study in order to illustrate its usefulness. Chapter 5 addresses the problem of detect the optimal sampling frequencies and its interactions with other approaches. This chapter also shows an evaluation of the techniques proposed. Finally, 6, explains the conclusions that can be extracted from this thesis.

Chapter 2 Background and Context

In this chapter, we describe several generalities and concepts which have been widely used in this thesis. First, we talk about several basic concepts of parallel programming, explaining the main programming paradigms which have been used to program parallel machines. After, we talk about several concepts related to performance analysis of high performance computing applications and we give an overview about several well known performance tools. Finally, we explain several generalities of signal processing and spectral analysis and how they can be useful for the main goals of this thesis.

2.1 Parallel Programming

Parallel computing has become a widespread computing paradigm in the current world. A lot of parallel systems have been installed in the main research and industrial centers around the world. To program this heterogeneous set of machines, several programming models have been developed. In this section, we give a general overview about the state of the art in methods for parallel programming. For the moment, we have to introduce an important concept in parallel programming, the concept of process. It is the fundamental unity in most of parallel programming methods. A process is an instance of a program that is executed in serial form. We say that a program is parallel if it executes more than one active process at any time of its execution. Therefore, to be useful, parallel programming paradigms have a well defined set of tools able to create, destroy and coordinate the different processes during the execution of a parallel program.

2.1.1 Shared-Memory Programming

In software, shared-memory is a method of communication between processes running at the same time. One process will create an area in random access memory

(RAM) which can be accessed by all the processes. From the hardware point of view, shared memory machine is one in which all the processors have access to all memory locations. However it is also possible to emulate shared-memory machines with distributed memory systems if we can create a global address space.

Shared-memory programming paradigms have several basic operations. First of all, they have two kind of operations to create processes. These two ways of process creation are called static and dynamic. The first one consists in the creation of a process at the beginning of the execution by a directive of the operating system. The second one consists in the creation of processes during the execution of the application.

Second, there are three basic primitives to coordinate processes. The first specifies the variables that can be accessed by all the processes, the second avoids problematic accesses to shared resources and the third synchronizes the processes.

In the last years, OpenMP [68] has become the most extended standard for parallel programming in shared-memory machines. To create the different processes, OpenMP uses dynamic creation. The execution starts as a single process, after several processes are created (fork point), starting the parallel region. This region ends when they are killed (join point) and the serial execution continues. An execution can have several parallel regions and each region can have different numbers of parallel processes. Obviously, the parallel regions start in a fork point and end in a join point. For this reason, the parallel model of OpenMP is called fork-join model.

Suppose that we want to parallelize the typical algorithm of matrix multiplication. In a very simple and non-optimized form, this algorithm can be written in C language as follows:

```
for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
        sum=0;
        for (k = 0; k < rows; k++) {
            sum = sum + array[i][k] * array[k][j];
        }
        array2[i][j]=sum;
    }
}
```

It is important to state that each level in the nested loop can be executed independently of each other. For this reason, the parallelization using OpenMP is very simple: We indicate the fork point inserting the directive ”pragma omp parallel for” just before the first level of the loop. It is also possible to insert this directive inside the internal levels of the loop. However, the performance gain will be better if we insert the directive at the outermost level. In the parallelized loop, variables array, array2, cols and rows are shared, that is, their values are shared by all the threads. Variables i, j, k and sum are private, that is, each thread assigns values to them independently of each other. The joint point is reached when the program finishes the execution of the nested loop.

```
pragma omp parallel for shared(array, array2, cols, rows) private(i, j, k, sum)
for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
        sum=0;
        for (k = 0; k < rows; k++) {
            sum = sum + array[i][k] * array[k][j];
        }
        array2[i][j]=sum;
    }
}
```

Typically, OpenMP programs have a global structure which is composed by sequential and parallel regions. As we have said above, parallel regions start in a fork point and end in a joint point. These regions are typically composed by fine-grain parallel chunks of computations. Once the joint point is reached, there is an implicit barrier that synchronizes all the threads. After, parallel region ends and a new sequential execution segment starts.

2.1.2 Message Passing

The most extended paradigm to program distributed-memory systems is message passing or a variant of it. This paradigm consists in direct communications between the processes, that is, they explicitly send and receive messages. When the execution starts, many different processes are created. All of them execute the same program with different data. They are allocated at the beginning of the execution and they are alive until the execution ends. Therefore, the message passing application has always the same number of processes created and cannot reallocate resources. This is a difference between message passing and the OpenMP model.

The Message Passing Interface (MPI) [58] is the most extended library interface which provides an implementation of the message passing paradigm to parallelize applications. The main goal of MPI is to become a widely used standard for writing message passing applications. Other important goals of MPI are, first, provide efficient communication, second, provide implementations that can be used in a wide range of machines, third, provide C and Fortran bindings, fourth, provide a reliable communication interface and fifth, provide a friendly interface.

In MPI each process has an identifier associated. For example, if we execute a MPI application with 8 processes, each process has a rank between 0 and 7. It is possible to define communication groups. The process assigned within a communication group can only communicate between them. This process group is ordered and process are identified by their rank within this group. A predefined communication group which contains all the processes is provided by MPI. Its name is MPI_COMM_WORLD. MPI has two basic functions. The first is used to send a message and the second enables to receive it:

```
int MPI_Send( void*         start_buffer    /* input */
              int           count           /* input */
              MPI_Datatype  datatype        /* input */
              int           destination     /* input */
              int           tag             /* input */
              MPI_Comm      communicator)   /* input */
```

Parameter start_buffer is the initial address of the sent buffer, count is the number of elements contained in buffer, datatype is the datatype of each send buffer element, destination is the rank of the destination, tag is the tag of the message and communicator is the communication group.

```
int MPI_Recv( void*         start_buffer    /* output */
              int           count           /* input */
              MPI_Datatype  datatype        /* input */
              int           destination     /* input */
              int           tag             /* input */
              MPI_Comm      communicator    /* input */
              MPI_Status    status)         /* output */
```

The receive buffer is composed by the storage of count consecutive elements. The type of these elements is specified by datatype. The receive buffer starts at address start_buffer. If the length of the received message is greater than the length of the receive buffer, an overflow error will occur. Status parameter returns information about the received message.

For example, suppose that process 0 sends and int i to process 2. The call of MPI_Send is as follows:

```
MPI_Send(&i, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
```

In order to receive the messages, process 2 has to call MPI_Recv. To assure a right communication between process 0 and 2, the call of MPI_Recv has to match tag and communicator parameters and the memory available for the message has to be greater or equal than the message sent. Remember that this memory is specified by the buffer, count and datatype parameters. The call of MPI_Receive can be as follows:

```
        MPI_Recv(&i, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
```

Now several problems have to be addressed. The first one happens if process 0 calls MPI_Send but process 2 does not call MPI_Recv. The solution of this problem depends on the system that is executing the application. Basically, there are two solutions: The system software can buffer the message, that is, it can save the message on the memory and process 0 can continue the execution. When process 2 calls MPI_Recv, the message is transfered to it. This kind of communication is called buffered communication. Other solution is to synchronize the execution, that is, process 0 can send a request to 1 and wait until it receives an authorization to begin the communication from 1. This kind of communication is called synchronous communication.

The second problem happens if process 2 executes the receive call before that process 0 starts the transmission of the message. The function we called, MPI_Recv, is blocking. It means that process 2 will be waiting, remaining idle, until process 0 starts the transmission of the message. It is important to state that this is not synchronous communication because it is not mandatory for process 0 to receive permission to start the send. This type of communication is called blocking communications. On the other hand, it is also possible to perform a nonblocking receive communication. It consists in another type of receive call that sends a request to the system to inform that the process has started a receive call. This process can perform calculation which do not need the information contained in the message and check later if the message have arrived. This receive function is called MPI_Irecv:

```
int MPI_Irecv(void*         start_buffer    /* output */
              int           count           /* input */
              MPI_Datatype  datatype        /* input */
              int           destination     /* input */
              int           tag             /* input */
              MPI_Comm      communicator    /* input */
              MPI_Request   *request)       /* output */
```

Non-blocking communications can reduce substantially the overhead due to communication. If a node of a parallel machine is able to compute and communicate simultaneously, the performance of message passing programs will improve a lot. While computations are being performed, it is also possible to do the work needed by the non blocking operation. It is well known that communications are more expensive than computations. Therefore, overlapping communication and computation can provide important performance improvements.

For the moment, we have talked about communications that involve two processes. For this reason, this communications are called point-to-point communications. However, MPI provides another kind of communications which involve more than two processes. These communications are called collective communications. A communication is called collective when involves a group of processes. The existence of collective communications are motivated by several performance problems that can appear usually. For example, sometimes one process has to read all the data from one file and after send data to the other processes to allow the beginning of computations. Suppose that there are n processes. If the process 0 has to send all the information to the others, it has to perform $n - 1$ sends and they cannot be overlapped. It's better to perform the sends using a tree approach, that is, process 0 sends to process 1, after process 0 sends to process 2 and process 1 sends to process 3 and so on. This scheme allows us to perform the message exchanges in logarithmic time because several communications are overlapped in time. However, the implementation of this tree approach is quite complicated if we only have the point-to-point functions we have seen. To overcome this problem, there is a function called MPI_Bcast which performs an optimum broadcast of a message. This is the syntax of MPI_Bcast:

```
int MPI_Bcast(void*         message        /* input/output */
              int           count          /* input */
              MPI_Datatype  datatype       /* input */
              int           root           /* input */
              MPI_Comm      comm)          /* input */
```

This call simply sends a message from the root process to the set of processes contained in the communication group. To receive the message, the call should be performed by all the processes involved in the communication: The root and the receiving processes. The parameter message will be an input parameter for the root process and an output parameter for the other processes.

This is only an example of collective call. MPI provides more functions which are able to perform other kinds of data exchange between processes. The common denominator of these collective calls is that they involve a communication group, that is, a set of processes.

Typically, message passing executions have computation regions interspersed with communication ones. Point to point calls are frequently used to implement communications between tasks. Collective calls are focused on synchronize the tasks or on perform efficient and complex communicating operations.

2.1.3 Data Parallel Languages

Data parallelism [38] , also known as loop-level parallelism, is a simple approach to program parallel systems. It consists in distribute data among the processes. Each process executes the same program on the part of data that has been assigned to it. This approach is very useful if we have to compute on very regular data structures. In this case, the programmer only has to indicate, adding some directives, that the structure has to be distributed among the processes. Compiler automatically replaces these directives with code that distributes the data. Obviously, as more regular is data, more efficient is its distribution among the processes.

It is important to state that, in general, the problem of mapping data structures to processors is very hard. The only particular case that can be easily solved is when data structure is very regular, for example, a dense matrix. This is a very important issue for any kind of parallel program. However, in data-parallel programs this problem becomes crucial because the mapping of data structures is specified during the compiling time.

An example of data parallel languages are Partitioned Global Address Space programming models (PGAS). These programming models provide high performance computing programmers with a shared address space model which simplifies programming while exposing data/thread locality to enhance performance. This facilitates the development of programming languages that can reduce both development time and execution time.

Data Parallel Language approach is specially useful to program architectures such as Compute Unified Device Architecture (CUDA) [36], a parallel computing architecture based on GPU cores. CUDA uses general purpose computing on graphics processing units (GPGPU). This technique consists in using a GPU, which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the CPU. Finally, we want to mention that data parallel language approach it is also useful to program multicore architectures [28] or reconfigurable devices [70] (FPGA's, ...).

2.1.4 Remote Procedure Call and Active Messages

Message passing and data parallelism are the two most used approaches to program distributed memory systems. However, we want to mention two approaches more that have been successful: Remote Procedure Call (RPC) and Active Messages. In these two models, the communication between processes consists not only in transmission of data but also in execution of subprograms in other address spaces (another computer) without the need of an explicit codification by the programmer of this remote interaction.

The differences of these two approaches remain on the fact that, on the one hand, RPC follows a synchronous communication model, that is, the main process is idle while it is waiting for the output of the subprogram executed by the remote processor. This synchronous communication model comes from the fact that RPC was initially designed for use in distributed computing. While the remote process is performing the execution of the subprogram, the host system of the main process can carry out other jobs. If the host system is composed by processors exclusively dedicated, this communication model will not be optimal. To solve this problem, we have active messages: When the main process has sent the message through the interconnection network, it continues its computations. When the remote receives the message, it stops the computations which were being performed on it, reads the information contained in the message and executes the subprogram.

2.2 Performance Analysis Tools

This section is focused on Performance Analysis Tools. First, we make a general overview of the existing tools, explaining how we classifying them according to their general characteristics. After, we explain some generalities about the collection of performance data in this thesis. We explain how tracefiles are generated, we describe their format and, finally, we talk about the possibility to visualize these files.

2.2.1 Overview of Performance Analysis Tools

Understand the behavior of an application is mandatory to evaluate its suitability and applicability on parallel machines. To acquire this understanding, it is mandatory to capture from the program execution the information from which the performance metrics will be obtained. Performance analysis tools need some mechanisms to capture information. Typically, these mechanisms are inserted in segments of code, enabling the possibility of extract and summarize information through a mechanism implemented at a lower level. For example, hardware counters [66] are based on mechanisms which are able to summarize architecture accumulate occurrences of specific events in registers that can be consulted by higher level mechanisms.

The information extracted from real executions of applications can be described like a set of three dimensional points. Each point corresponds to an event occurred during the execution of the application. The three dimensions are, first, the temporal dimension (when an event happens), second, the spatial dimension (the process where it happens) and, third, the type of event. To capture these points, two basic techniques can be used: Instrumentation and sampling. The first one consists in capture each relevant event to the performance analysis. For example, if we are interested in the performance of a supercomputer's communication network , we will capture all the communication events. The second technique, sampling, consists in extract the events according when an external mechanism determines. For example, we can extract data each second.

The obtained points have to be presented to the analyst in such a way that simplifies the understanding. Mainly, there are two forms to summarize data: the first one consists in collapse the time dimension, the space dimension or both. This summarization can be done online, that is, during the executions of the application or off-line. Data presentation can be done in text format [44] or using a GUI [104, 90]. This last presentation method offers a clearer presentation to the user. The second method to summarize data consists in time-dependant visualizations [52, 62, 107]. The underlying idea is that to observe the time evolution of the events can strongly improve the understanding of the behavior of a parallel application.

An important consideration that we want to make is the importance of the analyst time dedicated to performance analysis. This analysis can be expensive if the analyst does not have previous knowledge about the studied application. Minimizing this time is one of the most important goals of the analysis tools and methodologies. The automatic capture of the internal structure of an application is a way to reduce this analyst time, as we will see in this thesis.

One of the main advantages of the automatic structure extraction of applications' execution is that it enables the possibility to acquire a first general overview of the executions without information about the source code of the application. This information is typically very expensive to obtain because it requires a careful observation of the main functions of the application source code. Up-to-date high performance computing application have large source codes. Therefore, to study these source codes is not an easy task.

Other important advantage of our approach is that it simplifies the instrumentation process because the generation of large data sets is not problem. The automatic reduction process of data sets makes possible to handle with this kind of sets. In the next chapters, these ideas are explained and developed in detail.

2.2.2 Collecting and Studying Performance Data

To detect the structure of an application's execution, we need first to generate data from the execution which contains information related to its evolution. In order to perform a time analysis of the execution, we need to generate a time stamped sequence of events from the execution of the application. This complete time stamped sequence of events is called tracefile. Each tracefile contains information of an execution of an application. Our approach consists in generate and analyze time-varying signals from these tracefiles. However, it is important to mention that our approach can be applied to other kinds of input data. Input information has to be time stamped. This is the only condition that has to be satisfied by the input data.

2.2.2.1 Tracefile Generation

To generate a tracefile of an application, we need to instrument its source code and perform a real execution of it. The instrumentation of the source code is needed in order to use the functionality of a tracing package. Basically, tracing packages are libraries which provide functions able to collect fine grained information of a sequential or a parallel application. There are a lot of examples of tracing packages focused on the analysis of parallel programs: VampirTrace [61], TAU [82] or MPItrace [59]. In this work, we use the last one.

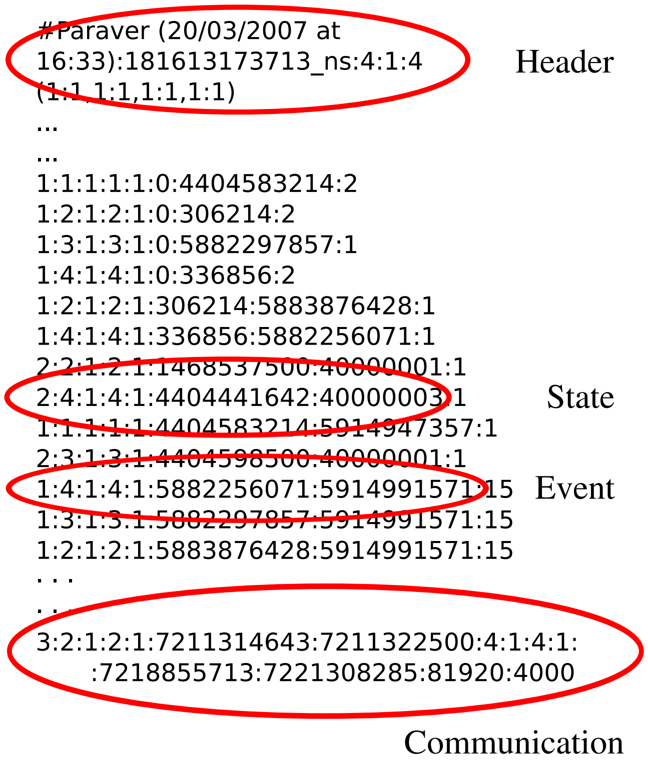
The MPItrace tool has been designed for parallel codes that use the message passing (MPI) programming model. This tool generates a Paraver trace file. In this file, the tracing package records the basic activity of the MPI program. The MPItrace tool assumes each MPI process has only one thread. A tracefile represents a single MPI application execution, therefore it includes only one program with several tasks and one thread per task. This tracing package records many types of information. These are the three major types of information we can get using MPItrace:

States: They provide information about the activity of each thread. They can be computing, waiting for a message or performing an input/output operation.

Communication: They provide information about point to point communications. They have identifiers for the starting and for the end of the sending and receives. They also provide the values of parameters tag and size according to their values in the MPI calls (see section 2.1.2). Since it is not possible through the MPI instrumentation to find out when the actual data transfer takes place, physical communication is assumed to be identical to logical one.

Events: They provide information about the beginning and the end of several operations that the applications performs during its execution. One example to tag the beginning and the end of MPI operations, such as Barriers, Broadcast, AlltoAll, and all kind of Send - Receive calls. Another example is to tag the beginning and the end of flushing data to disk (see section 3.5).

In this section we have talked about tracing packages focused on the analysis of MPI programs. However there are many packages focused on the study of other kind of MPI programs. For example, OMPtrace [16] instruments parallel code of OpenMP applications. Basically, this package provides information in terms of states and events. In this context, state information means information about if a given thread



```
#Paraver (20/03/2007 at
16:33):181613173713_ns:4:1:4
(1:1,1:1,1:1,1:1)
...
...
1:1:1:1:1:0:4404583214:2
1:2:1:2:1:0:306214:2
1:3:1:3:1:0:5882297857:1
1:4:1:4:1:0:336856:2
1:2:1:2:1:306214:5883876428:1
1:4:1:4:1:336856:5882256071:1
2:2:1:2:1:1468537500:40000001:1
2:4:1:4:1:4404441642:40000003:1
1:1:1:1:1:4404583214:5914947357:1
2:3:1:3:1:4404598500:40000001:1
1:4:1:4:1:5882256071:5914991571:15
1:3:1:3:1:5882297857:5914991571:15
1:2:1:2:1:5883876428:5914991571:15
. . .
. . .
3:2:1:2:1:7211314643:7211322500:4:1:4:1:
    :7218855713:7221308285:81920:4000
```

Figure 2.1: **Paraver Tracefile Example**

is computing, communicating, performing an input/output operation or scheduling (generating work/notifying termination). Information in terms of events means essentially information about if a given thread entries / exits from a parallel region or from a routine that includes OpenMP directives.

Finally, we have to mention OMPItrace [67], a tracing package able to trace application which contain MPI and OpenMP directives, Java Automatic Code Interposition Tool (JACIT) [98] a tool which enables the user to insert probes on Java codes, the infoPerfex [96] tool, which enables the user to obtain information from hardware counters sampled periodically, and SCPUs [77] a tool that instruments the operating system scheduling.

2.2.2.2 Tracefile Format

Tracefiles contain timestamped sequences of events of a given execution of an application. Obviously, the format of this sequence is not important from the conceptual point of view in order to generate signals. However, in order to provide a general idea of the kind of information contained in tracefiles and, therefore, of the kind of information needed by our approach in order to detect execution structure, we show an overview of the tracefile format in figure 2.1. It is the typical Paraver [52] tracefile format.

First, we highlight the header of the tracefile. It contains information about the total application time in nanoseconds, about the number of processors in the tracefile, about the number of applications in it (usually we have only one application for each tracefile), the number of tasks for each application and the number of threads for each task. In this case we have that the total application time is equal to 181613173713 nanoseconds, the number of processors in the tracefile is equal to 4, the number of applications in the tracefile is equal to 1 and the number of tasks of the application is equal to 4.

After, we show an example of a state. The first number, 1, means that this record is a state. The second, third, fourth and fifth numbers are equal to 2, 1, 2, 1 respectively. These four numbers have to be read from right to left. The last pair 2, 1 means that this event provides information about the first thread of the second task. The first pair 2, 1 means this task belongs to application 1 and that it is mapped to processor 2. Remember that in this trace we only have 1 application and that each thread has only 1 task. For this reason, the third and the fifth numbers are always equal to 1. The sixth number, 306214, shows the beginning of the state in milliseconds and the seventh, 5883876428, provides its end. Finally, the eighth number gives the type of the state. In MPI tracefiles, state 1 is usually interpreted as a computing state.

After we show an example of an event. The first number, 2, means that this record is an event. The second, third, fourth and fifth numbers mean the same as above. The sixth parameter is the instant of time of the event occurrence, in nanoseconds. After, the seventh parameter, 400000003 is an identifier of the event type. This identifier means the task is flushing its records to disk (see section 3.5). Finally, we have the value the event. In this case, 1 means beginning of flushings.

Finally, we show an example of a point-to-point communication record. The first number, 3, means that this line provides information about a communication. We have two sets of four numbers, 2:1:2:1 and 4:1:4:1. The first set identifies the sending task and the second set identifies the receiving task. On the one hand, the two numbers following the set 2:1:2:1, 7211314643 and 7211322500, provide the beginning and the end of the send. On the other hand, the two number following the set 4:1:4:1 provide the beginning and the end of the receive. The last two numbers, 819200 and 4000, provide the size, in bytes, and the tag of the message.

Note that it is easy to generate time dependent signals from this kind of information. For example, we can generate a signal which contains how many tasks are performing a point to point send call in each instant of time or how many tasks are in a computing state. From these signals, we extract the structure of the execution of the application.

2.2.2.3 Visualization and Analysis

We want to mention another important fact related to tracefile and related to performance analysis: the possibility of generate visualizations of the tracefile in order to visually study characteristics of the execution of the application such as the performance properties or the general structure of the application. The process of performance analysis of application can be remarkably accelerated with the help of visualization tools such as Paraver [52].

Paraver is a malleable performance visualization and analysis tool. It can be used to analyze performance properties related to MPI, OpenMP, MPI+OpenMP, Java, Hardware counters profile, operating system activity and many others. It provides a qualitative global perception of the whole execution of the application. It also provides the possibility of focussing the analysis on a small region of the execution and its quantitative properties. Finally, Paraver enables the analyst to determine where to invert the programming effort to improve the performance of the application.

The graphical views provided by Paraver represent the behavior of the application along time. These views characterize the temporal evolution of the application and its behavior. It is also possible to identify patterns and detect causality relationships by the analyst who is looking at the Paraver graphical representation.

For example, in figure 2.2 we have an example of several Paraver representations. The application is NAS-BT class A [3] executed on 4 processors. We do not show the whole execution, only a small fraction of that. At the top, we have a typical view. In this representation we have, in the x-axis, the time and in the y-axis the set of tasks. Computing bursts are drawn in blue, communications in yellow and waiting time in red. As you can see, we can follow the temporal evolution of each thread detecting by visual inspection when it is communicating or computing. Please, note that this is the graphical expression of information contained in tracefiles, that is, each communication, event or state is codified in the terms we explained in section 2.2.2.2. Secondly, we have a graphical representation of a signal. This signal summarizes information about the number of communications, that is, it provides for each instant of time the number of flying messages which the application is transmitting. In the third place, we have a signal generated from computing states: The sum of durations of computing states. Note that, for this case, this signal is complementary of the above signal, that is, when we have a lot of communications, there are a few computing bursts and, on the other hand, when we have long computing bursts there are a few communica-

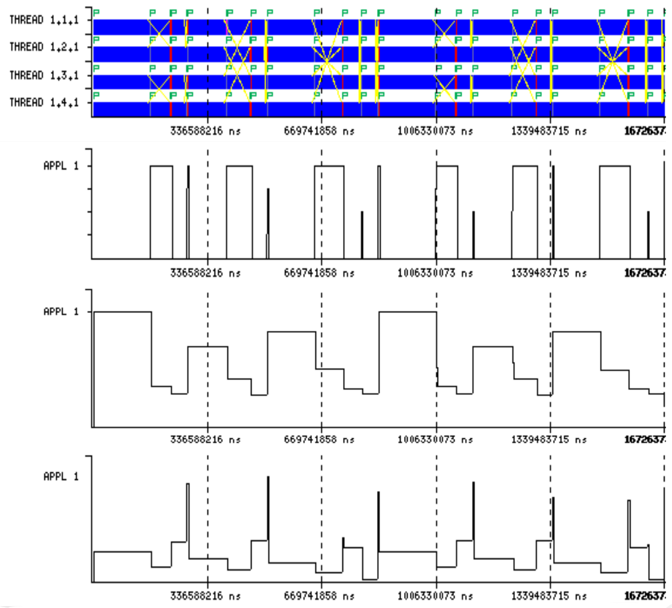


Figure 2.2: **Paraver views of two iterations of NAS-BT class A executed on 4 processors.**

tions. Finally, in the fourth place, we have a signal which indicates, for each instant of time, the mean IPC achieved by the set of threads. All these signals can be generated with Paraver from the original tracefile by filtering and interpreting the trace information. With Paraver, the analysis requires an interactive user intervention. In this thesis, we pursue to perform this automatically for large tracefile.

With the above example, we want to emphasize the flexibility and malleability of Paraver. It is possible to visualize many kinds of information graphically or numerically. Our approach has the same philosophy from the point of view of flexibility of metrics and signals. As we explain in the next chapters, our approach is general in the sense that is possible to analyze signals independently of the information they contain or the methodology we have followed to obtain them.

2.3 Signal Processing

In this work, signal processing techniques play a key role because, as we explain in the next chapters, we characterize executions of applications using signals. For this reason, we dedicate this section to explain several well known signal processing techniques which are used in this thesis widely. Furthermore, we explain why they have been used and the advantages and disadvantages of them from out point of view. These techniques are closely related with wavelet transform, cross-correlation function and mathematical morphology.

2.3.1 Wavelet Transform

Wavelet transform is a well known signal processing tool used in a wide and heterogeneous range of topics. This transform and the techniques based on it have become useful for relevant issues such as image compression [10], sensor networks [99], signal processing, seismic geophysics, molecular dynamics, speech recognition, climatology, turbulence analysis, quantum mechanics, optics, ... The most important feature of Wavelet Transform is that it captures information not only about the values of the frequencies of the input signal but also about the physical location of these frequencies, that is, using wavelet transform it is possible to know estimations of values of the main frequencies of the signal and the places within the input signal's domain where these frequencies occur. Remember that the traditional Fourier Transform does not give information about the physical location of the frequencies, only gives the spectrum of frequencies as output. Another important difference is that Fourier Transforms represents signals in terms of sums of sinusoids and Wavelet Transform can represent signals using a more wide range of functions, providing a more general analysis environment.

The history of wavelets starts with Haar's contributions at the beginning of the 20th century. He developed the well known Haar's transform [100], a particular case of wavelet transform. The following step was done by Zweig, discovering the continuous wavelet transform in 1975 while he was studying the interactions between the ear and the sound. We have to mention Stromberg [94], for his contributions to discrete wavelets in 1983 and Daubechies' discovering of continuous wavelets with compact support in 1988 [22]. Many others [35] have contributed decisively in the development of wavelets theory.

2.3.1.1 Definition

Wavelet transform expands a function in the spatial domain to an adding of orthogonal functions. It is a particularly interesting operator because wavelet expansions require very few terms to approximate accurately the functions. This is the reason why wavelet transform is widely used for compressing images or signals. In terms of formulas, the set of orthogonal functions is $\{\psi_{k,n} | k \in \mathbb{Z}, n \in \mathbb{Z}\}$ and the discrete wavelet transform (DWT) is a representation of a signal $x(t)$:

$$x(t) = \sum_{k=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} d_{k,n} \psi_{k,n}(t) \tag{2.1}$$

$$d_{k,n} = \langle \psi_{k,n}(t), x(t) \rangle = \int_{-\infty}^{\infty} \psi_{k,n}(t) x(t) dt \tag{2.2}$$

where $d_{k,n}$ are the wavelet coefficients. The set of orthogonal functions is defined shifting and stretching a mother function, ψ, called mother wavelet. All the functions belonging to the set $L_2(\mathbb{R})$ (functions whose square is integrable) are eligible to become mother wavelets. For each one of them, we can define a family of wavelets

$$\psi_{k,n} = 2^{-\frac{k}{2}} \psi(2^{-k} t - n) \tag{2.3}$$

increasing n, the function is switched to the right and, otherwise, decreasing it, the functions is switched to left. On the other hand, increasing k the function is stretched and its value is decreased.

Note that the set $\{\psi_{k,n} | n \in \mathbb{Z}\}$, that is, the subset of wavelets with a fixed value of k, describes a given level of detail of the input signal. As the value of k increases, the wavelets become more stretched and describe a higher level of detail of the signal. This fact allows DWT to provide a multi-resolution analysis of the input signal. In summary, DWT provides a multi-resolution analysis of a continuous signal in $L_2(\mathbb{R})$.

2.3.1.2 Fast Wavelet Transform

As we have seen in the past section, the computation of each wavelet coefficient requires the numerical evaluation of an integral. It is not possible to compute these integrals from the computational point of view. To overcome this problem, an algorithm called Fast Wavelet Transform was developed. Its theoretical basis is functional analysis [23], [100]. It is recursive and based on two filters: a low pass filter and a high pass one. Each filter is characterized by a set of coefficients: the set $\{a_d\}$ characterizes the low pass filter and $\{b_d\}$ the high pass one. The values of these coefficients are derived from a function $\phi \in L_2(\mathbb{R})$, called scaling function. That is, each scaling function defines a set of two filters that make possible a quick calculation of the Discrete Wavelet Transform. The theoretical definitions we made in the past section are implicit in this scheme, that is, from the scaling function ϕ it is possible to define a mother wavelet and the subsequent set of functions. However, not all the functions in $L_2(\mathbb{R})$ are able to provide a multi-resolution analysis based on DWT. There are several non-trivial mathematical conditions, described in [45], that the scaling function has to fulfill.

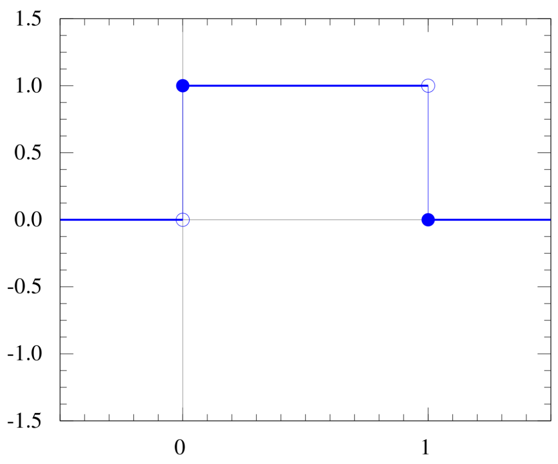


Figure 2.3: **Haar Scaling Function**

The simplest scaling function $\phi(t)$, called Haar' scaling function, can be defined as:

$$\phi(t) = \begin{cases} 1 & 0 \leq t < 1, \\ 0 & \text{otherwise.} \end{cases} \tag{2.4}$$

From this scaling function, it is possible to obtain the Haar's mother wavelet function, $\psi(t)$. The mathematical methodology that enables to generate the wavelet mother function from a well-defined scaling one can be found in [45]. Haar's wavelet mother function can be described as

$$\psi(t) = \begin{cases} 1 & 0 \leq t < 1/2, \\ -1 & 1/2 \leq t < 1, \\ 0 & \text{otherwise.} \end{cases} \tag{2.5}$$

In figure 2.3 we show the simplest scaling function, Haar' scaling function and in 2.4 we have the wavelet derived from it.

As we have said above, two filters can be obtained from the scaling function, $\phi(t)$. These two filters are the basis of the FWT algorithm. This algorithm consists in apply recursively an operation that transforms a set of $N = 2^J$ coefficients, $s_0, s_1, ..., s_{N-1}$ into two sets of $\frac{N}{2}$ samples. Each recursive application of the operation is called level. The coefficients used as input to the transform are expressed $s_0^0, s_1^0, ..., s_{N-1}^0$. In general, level l coefficients, those are, $s_0^l, s_1^l, ..., s_{n-1}^l$ and $d_0^l, d_1^l, ..., d_{n-1}^l$ are obtained

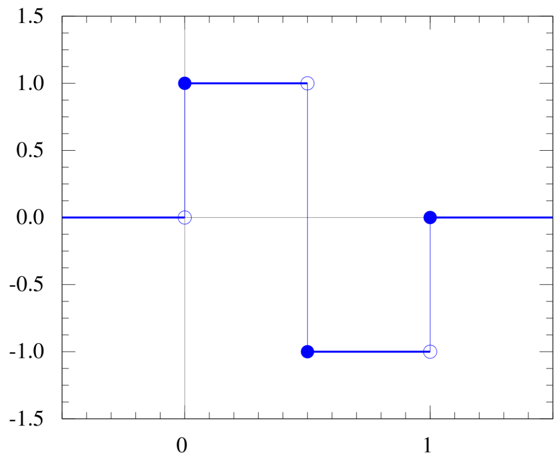


Figure 2.4: **Haar Wavelet**

from level $l-1$ coefficients $s_0^{l-1}, s_1^{l-1}, ..., s_{2n-1}^{l-1}$ according to two recurrence relations:

$$s_i^l = \sum_{d=0}^{D-1} a_d s_{\langle 2i+d \rangle_{2n}}^{l-1} \tag{2.6}$$

$$d_i^l = \sum_{d=0}^{D-1} b_d s_{\langle 2i+d \rangle_{2n}}^{l-1} \tag{2.7}$$

where $n = \frac{N}{2^l}$, $i = 0...n-1$, a_d and b_d are coefficients for low and high pass wavelet filters, D is the number of values in the wavelet filters. $\langle x \rangle_m$ is the modulus function defined so that $\langle x \rangle_m \in [0, m-1]$ for all $x \in \mathbb{Z}$. To execute correctly this algorithm, N has to be power of 2, that is, $N = 2^J$. Note that it implies that the recursive call will be executed J times. It is important to state that s_i^l contain information about low frequencies and d_i^l about high frequencies.

When the execution of the fast wavelet transform is finished, we will obtain a set of coefficients. First, $\frac{N}{2}$ values of the level 1, $s_0^1, s_1^1, ..., s_{N/2-1}^1$, after, $\frac{N}{4}$ coefficients of the level 2, $s_0^2, s_1^2, ..., s_{N/4-1}^2$, ..., until 2 coefficients of level J, s_0^J, d_0^J. According to the theory, the coefficients of level i contain information about frequencies between $[\frac{\alpha}{2^{i+1}}, \frac{\alpha}{2^i}]$ where $1 \leq i \leq J$ and α is the sampling frequency of the input signal. Note that the higher frequency considered by this multi-resolution scheme is $\frac{\alpha}{2}$. This fact is coherent with the Nyquist-Shannon sampling theorem [80, 65] which states that is not possible to obtain information from the spectral analysis of a signal of frequencies higher than $\frac{\alpha}{2}$ if the sampling frequency is α.

2.3.1.3 General Considerations

As we have seen in the past section, the analysis using Discrete Wavelet Transform (DWT) allows us to obtain a multi-resolution description of the signal. We obtain $\frac{N}{2}$ coefficients that give information about the physical localization of frequencies from $\frac{\alpha}{2}$ to $\frac{\alpha}{4}$, $\frac{N}{4}$ coefficients that give information about the physical localization of frequencies from $\frac{\alpha}{2}$ to $\frac{\alpha}{4}$, and so on ... Note that for high frequency values we have a lot of information about the physical location of them and for lesser frequency values we have less information about their physical location. On the other hand, the resolution of the frequencies' values is higher for low values of them because interval $[\frac{\alpha}{2^{i+1}}, \frac{\alpha}{2^{i}}]$ is larger than interval $[\frac{\alpha}{2^{i+2}}, \frac{\alpha}{2^{i+1}}]$.

This is the main problem of Fast Wavelet Transform, precision. For low frequency values, it gives good estimations of their values but bad estimations of their physical location within the input signal domain. On the other hand, for high frequency values, it gives bad estimations of their values but good estimation of their physical location within the input signal domain. This phenomenon is not reduced to Wavelet Transform. It also appears in topics such quantum physics and it is named uncertainty principle [102].

On the other hand, the computational complexity of Fast Wavelet Transform is very low. It allows to obtain a complete multi-resolution analysis of a signal with a very reasonable computational load. Fast Wavelet Transform is performed in $O(n)$ operations.

In summary, we have that wavelet transform is a good tool to detect not only the values of the frequencies, but also the physical location where these frequencies occur. The computational complexity of the Fast Wavelet transform. However, it has several accuracy problems, as we have explained in this section.

2.3.2 Cross-correlation and Autocorrelation

Cross-correlation is a commonly used and well known signal processing technique. This tool and the methodologies derived from it are used in topics such us pattern recognition, cryptography, image matching, ... It is used to evaluate the degree of similarity between two signals. Another interesting feature of it is that it can be used to look for a known pattern within a large signal. From the point of view of statistics, cross-correlation has good properties because it allows calculations of distribution probability functions of random variables generated from the composition of other random variables with known distribution probability functions.

2.3.2.1 Definition

Cross-correlation functions of two signals is an operation that consists in shifting one of them and multiplying them. This operation is also known such us sliding dot product or inner-product. In terms of formulas, if f and g are two discrete signals, the discrete cross-correlation of them is:

$$(f \star g)[n] = \sum_{m=-\infty}^{\infty} f^*[m]\; g[n+m] \tag{2.8}$$

where the symbol $\star$ means cross-correlation and f^* means the conjugated function of f. Obviously, it f is a real function, $f = f^*$.

The underlying idea behind the formula is to detect the degree of similarity between two signals. For example, suppose that we have a discrete signal, g and that we shift this signal to the right a value k, obtaining a new signal, f, defined like this $f(t) = g(t-k)$. Suppose that we compute the cross-correlation between f and g. The formula switches g function along the x-axis and computes the adding of the multiplications of f and switched g. When $m = k$, the positive values of f are aligned with the positive ones of g and the same happen with the negatives and, therefore, the value of the cross-correlation function will be maximized in this point. Note that, from the information provided by cross-correlation function, we are able to detect that maximum degree of similarity between signals f and g occurs when the second one is shifted k times to the left.

In figure 2.5 we depict a sinusoid function and we show this function shifted to the right a value equal to $\frac{\pi}{2}$, to π and to $\frac{3\pi}{2}$. In figure 2.6 we have the cross-correlations between the initial sinusoid function and its right shiftings. As we can see, cross-correlation reach their maximal values at the same point where we switched the signal to the right. This is because cross-correlation slides to the right the initial function. Where this initial function is aligned with the shifted one, cross-correlations reach their maximal values.

2.3.2.2 Autocorrelation

Autocorrelation is a particular case of cross-correlation when $f = g$. It is specially useful for finding repetitive patterns within the input signal. Autocorrelation has always the global maxim value when $n = 0$ because in this point the signal f is correlating with himself. The interesting points of an Autocorrelation are the local maxim values because they are candidates to represent the value of an iterative pattern within the input signal. These local maxim values can represent the value of an iterative pattern or an harmonic of it, that is, a multiple of the real value of the iter-

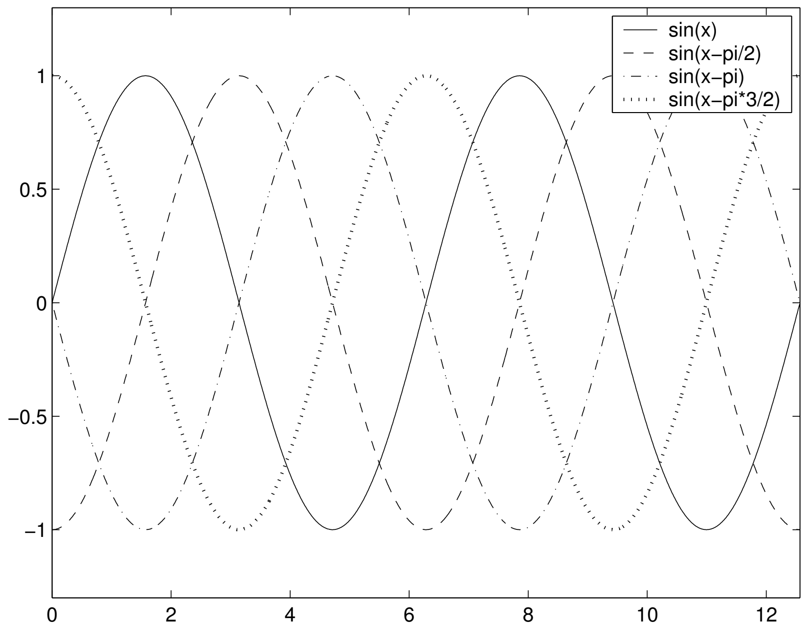


Figure 2.5: **Sinusoid and right-shiftings**

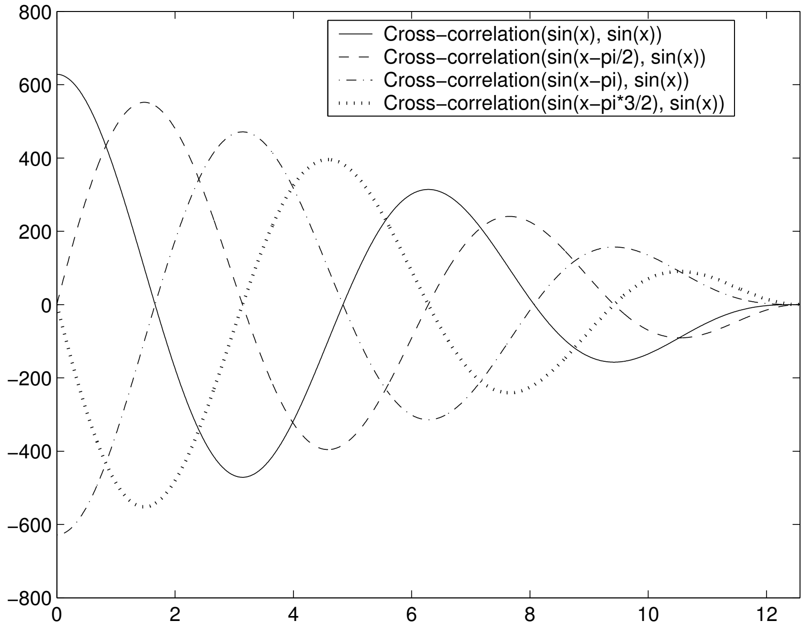


Figure 2.6: **Crosscorrelations**

ative pattern. The high importance and wide utilization of autocorrelation function, a special case of cross-correlation, remain on the properties we have described in this section.

2.3.2.3 Computation of Cross-correlation

The computation of cross-correlation directly from the formula 2.8 has a computational complexity of O(n^2), where n is the number of samples of the signals. This fact makes impossible a direct computation of cross-correlation function. We have to turn to some theory to find a solution to this problem. According to a theoretical mathematical result:

$$F\{f \star g\} = F\{f\}^* \cdot F\{g\} \tag{2.9}$$

Where $F\{f\}$ means the Fourier transform of signal f and f^* means again the conjugate of f. The interpretation of this formula is clear: It is possible to calculate the values of the cross-correlation using Fourier Transform, reducing the computational complexity dramatically because Fourier Transform can be obtained with only O($n{\cdot}\log(n)$) operations. To calculate the cross-correlation of two discrete signals f and g of n samples, we have to calculate, first, the Fourier Transform of both signals, second, multiply the coefficients of the two transforms, obtaining n coefficients from these multiplications and, finally, perform an inverse Fourier Transform to these n multiplicative coefficients.

At the end, the computation of cross-correlation is reduced, if we have two discrete signals of n samples, to two Fourier Transforms, n multiplications of complex numbers and an inverse Fourier Transform. All these calculations have a computational complexity equivalent to O($n{\cdot}\log(n)$) operations, better than the O(n^2) operations required if we calculate the cross-correlation using formula 2.8 directly.

2.3.2.4 General Considerations

In summary, since its special properties, cross-correlation function is a widely used tool in a very differently topics. It is used, essentially, to evaluate the similarity between two signals or to look for patterns within large signals. Autocorrelation function is a special case of cross-correlation function, specially useful for detect periodic behavior within the signals. These techniques are well known and their theoretical roots are rigorously established in mathematical bibliography [8].

Cross-correlation function has several well known advantages. First, we have to mention its usefulness in problems related to pattern recognition. Second, it is possible to reduce the computation of this function to a composition of Fourier transforms, reducing remarkably its computational complexity. Finally, we have to mention the simplicity of its implementation because only a library containing an implementation of the Fast Fourier Transform (FFT) is required.

However, this technique has several disadvantages. The first one is related with the search for periodic behavior within the signal. Autocorrelation offers several candidates to be the repetitive pattern. However, sometimes it is not clear which of these candidates are harmonics of the repetitive pattern or correspond to other smaller repetitive patterns. For this reason, it is not always easy to automatically extract the value of the main iterative pattern of a given signal from its autocorrelation function. On the other hand, cross-correlation has another disadvantage related to the search for a given pattern within the signal domain. To make this search more effective, we need very concrete information about the pattern we are looking for. For example, if we only know its length but we have no idea about its morphology, some problems will sometime appear in the detection of the pattern.

2.3.3 Mathematical Morphology

Mathematical Morphology is a branch of mathematics focused on the analysis of geometrical spaces. Its theoretical roots are set theory, topology random processes and geometry. Several geometrical and topological characteristics of continuous and discrete figures of the space can be summarized by mathematical morphology. Usually, it is used to study digital images, but it can be applied to signals, graphs and other spacial structures. The main focus of mathematical morphology is to extract automatically useful information from images or signals without supervision. For example, it is used to extract information about roads and railways from pictures made by planes or satellites.

Historically, the theory was developed by J. Serra and G. Matheron [78] in the School of Mines of Paris around 1964. After, several non-linear filters were defined by the same authors in the context of mathematical morphology, such us dilation, erosion, opening and closing. Around 1980, Mathematical Morphology gained international recognition and it started to be used in a larger set of problems.

2.3.3.1 Definitions

Suppose that we have a non-negative signal, $f(x)$, called input signal, and a real value k, called structuring value. For each structuring value, we can define a pair of operators that can be applied to $f(x)$: The Dilation of a function $f(x)$ by k is defined by:

$$[f \oplus k](x) = \max_{y \in [x-k, x+k]} \{f(y)\} \tag{2.10}$$

Note that, applying this operation, the new dilated signal has the maximum value of input signal $f(x)$ within a given interval, defined by the value k. On the other hand, the Erosion of a function $f(x)$ by k is defined by:

$$[f \ominus k](x) = \min_{y \in [x-k, x+k]} \{f(y)\} \tag{2.11}$$

In this case, the new dilated signal has the minimum value of the input signal, $f(x)$, within a given interval. The interests behind these two basic operators are the combination of them. On the one hand, opening is the operation consisting in apply, first, the erosion operator and, after, the dilation one. On the other hand, closing consists in apply, first, the dilation operator and, after, the dilation one. In terms of formulas:

$$f \circ k = (f \ominus k) \oplus k \tag{2.12}$$

$$f \bullet k = (f \oplus k) \ominus k \tag{2.13}$$

The underlying idea behind opening operator is to eliminate small regions where signal's value is different from zero. On the other hand, closing unifies non-zero regions of the signal separated by small environments where the signal is equal to 0. In the context of Mathematical Morphology, many other filters can be defined, such us granulometry, thinning, skeletonization, and others. However, in our work we use mainly Closings and Openings.

2.3.3.2 Computation

The computation of Erosion and Dilation filters is made directly from the definitions 2.10 and 2.11. These calculations are possible to perform because the signals we work with have a finite precision limit. Therefore, search for maximum become a linear search across the signal. Remember that Closing and Opening filters are compositions of Erosion and Dilation. For this reason, they can be obtained with the same methodology as well.

2.3.3.3 General Considerations

Mathematical morphology is a good tool to extract relevant information from signals. The analysis performed by its filters is based on the morphological properties of the signals. This is because is interesting for us. The analysis point of view of Mathematical Morphology is very different from other signal processing theories, for example Fourier Analysis. The combination of several different analysis is what enables us to perform a complete analysis, as we will see in the next sections.

Mathematical Morphology has several well known advantages. First, it allows an automatic analysis of several geometrical properties of signals, images, graphs, etc.. Second, it has a wide set of morphological filters which are easy to define, easy to implement and have a wide set of application possibilities. Finally, Mathematical Morphology is a solid theory with robust theoretical roots.

Chapter 3

Automatic Phase Detection and Structure Extraction of MPI Applications

The process of obtaining useful message-passing application tracefiles for performance analysis in supercomputers is a long and tedious task. When using hundreds or thousands of processors, the tracefile size can be as large as 20 GB. It is clear that analyzing, or even storing, these large traces is a problem. The methodology we have developed and implemented performs an automatic analysis that can be applied to huge tracefiles, by determining its internal structure and selecting meaningful parts of the tracefile. This chapter presents the methodology and also the results obtained from real applications.

3.1 Introduction

In this section, we explain the main characteristics of the automatic phase detection and structure extraction of MPI applications. First, we define what we mean by structure extraction and why it is possible to extract this structure from executions of MPI applications. Second, we make an overview of the automatic process and we explain its main characteristics. Finally, we indicate the main applications of structure extraction in performance analysis of high performance computing kernels.

3.1.1 Definition of Structure Extraction

Structure extraction is an automatic process which consists in detecting the different phases of real executions of high performance computing applications, ruling out the perturbed regions and, finally, providing good representative segments of the whole execution. This process takes advantage of the typical characteristics of high performance computing applications: Many repetitions of the same segment of code, due to the typical iterations which numerical kernels perform, and several well defined execution phases, due to the initialization of values, the processing of them, and the final output of the results.

This process is composed by several steps, each one of them is focused on one of the above specific tasks which the structure extraction has to carry out. Each step is automatic and, thus, can be carried out without human intervention. As we will see, spectral analysis of signals plays a key role in the structure extraction process. The signals are directly extracted from data derived from real executions. In the next sections, we explain with more precision which are the main characteristics of the whole process and how it is carried out.

3.1.2 Overview of the Structure Extraction Process

The first step of our approach is to characterize the execution of the application from a particular point of view. There are a lot of approaches to the characterization of the execution of applications, but in our work we use signals to do this. These signals are generated from data contained in the tracefile. The information contained in them is the time evolution of a given metric during the execution of the application. For example, we generate a signal that contains how many processes are executing a MPI [58] call in an instant of time. It is possible to generate this kind of signal because tracefiles are timestamped sequences of events. The characterization of executions of applications is done using signals for several reasons. The first reason is that it is possible to keep the highly-detailed information contained in tracefiles of the variations on space (set of processes) and time. Secondly, we use signals because it is possible to isolate a given performance metric and perform an analysis based only on it and, therefore, determine the impact of this parameter over the whole execution. Thirdly, the algorithms based on signal processing theory have a low computational complexity, are able to automatically provide relevant information and, finally, are based on a solid and powerful mathematical theory.

We use signal processing techniques (wavelet transform, cross-correlation, morphological filters, ...) in order to provide a very fast, automatic detection of the phases of MPI application execution. These signal processing techniques are interesting for our analysis because they have several important properties which allow us to build an automatic system based on them. First, morphological filters derived from the theory of Mathematical Morphology provide a signal processing framework which is able to detect regions that are affected by corrupting factors. Second, wavelet analysis provides a classification of the physical domains of signals according to the occurrence of frequencies on them, that is, the criterion of wavelet transform in order to perform the analysis is to separate regions according to their frequency behavior, i. e., a region with a small iteration which is repeated many times is separated from another region with no periodic behavior. Third, autocorrelation provides us with a means of detecting the value of the iterative period within the regions with strong periodical behavior, that is, it gives the exact value of the time span of the main iterative periods within the periodical regions, which have been detected by wavelet transform. Fourth, cross-correlation makes it possible to detect which of the iterations are the most representative of the whole periodic region. The criterion of cross-correlation to select the most representative iterations is to look for the iterations closest to a perfect periodical signal.

3.1.3 Applications of Structure Extraction in Performance Analysis

In recent years, parallel platforms have vastly increased in performance and in number of nodes and processors. For example, the most recent supercomputing facilities have more than 100.000 processors and many of them have a performance peak near or higher than the petaflop barrier [85]. Thus, the study of the execution of applications in these platforms has involved increasingly hard and tedious work. A complete timestamped sequence of events of an application, that is, a tracefile of the whole application, results in a huge file (10-20 GB). It is impossible to handle this amount of data with tools such as Paraver [52]. Also, it is often the case that some parts of the trace are perturbed, and the analysis of these parts can be misleading. A third problem is the identification of the most representative regions of the tracefile. To reduce tracefiles sizes, the process of application tracing must be carefully controlled, enabling the tracing in the interesting parts of the application and disabling otherwise. The number of events of the tracefile (hardware counters, instrumented routines, etc...) must be limited. This process is long and tedious and requires knowledge of the source code of the application.

For these reasons, several authors [63, 97] believe that the development and utilization of trace-based techniques has several flaws. However, techniques based on tracefiles allow a very detailed study of the variations on space (set of processes) and time that could crucially affect the performance of the application. Therefore, there is a need to develop techniques that allow us to handle large event traces.

Our approach is to start from very large tracefiles of the whole application, allowing simple tracing methodologies, and then analyze them automatically. The underlying philosophy is to use resources that are generally available (Disk, CPU, ...) in order to avoid the use of an expensive resource, namely analyst time. The tool we have developed first warns the analyst about those parts of the trace perturbed by an external factor not related to the application or to the machine itself, and subsequently gives a description of the internal structure of the application, which leads to the identification and extraction of the most relevant parts of the trace. Third, our automatic system can be applied not only on off-line tracefile but also on data extracted on-line from an execution of a parallel application.

In section 3.2 we discuss related studies which have been carried out in the last few years. In section 3.3 we explain the different metrics we have used and we give some examples. In section 3.4, we describe the typical execution phases of high performance computing applications and we explain how we automatically detect them. In section 3.5, we discuss several factors which can corrupt the information contained in tracefiles. In section 3.6, we describe the typical iterative structure within the computing phase of high performance computing applications and we explain the methodology we use to detect it. Finally, in section 3.7 we evaluate our automatic methodology.

3.2 Related Work

There are various approaches to tracefiles which attempt to either avoid or to handle large event traces. KOJAK [49], is a tool for the automatic detection of performance bottlenecks. It looks for patterns representing inefficient behavior and quantifies its influence over execution. It focuses on studying locally every communication contained in the tracefile. However, KOJAK does not distinguish between execution phases because it does not look for the internal tracefile structure. For that reason, it needs large traces in order to amortize the initialization phase or the cost of perturbations. Our automatic system can be used to prepare data that will be given to KOJAK as input. This previous step will help it to focus on the most relevant regions in the tracefile.

DeWiz performance tool [51] performs an automatic analysis. The idea of DeWiz stems from the fact that most tools rely on graph-based analysis methods or use space-time diagrams for visualization of program behavior. As a result, a unified directed graph may be used to capture different program properties and to represent the basis for analysis activities. This analysis is focused, on one hand, on the detection of communication failures by pairwise analyzing of communication patterns, and on the other hand, DeWiz identifies repeated patterns in the event graph. It is in this second point that the DeWiz approach and our own is similar. However, DeWiz is unable to find structure on the temporal evolution of the application's execution.

These two tools, KOJAK and DeWiz, also try to distribute data processing in order to analyze large tracefile. A parallel extension of KOJAK, SCALASCA [6], has been developed and DeWiz also has a modular design which enables it to be executed on distributed computing infrastructures.

VAMPIR Next Generation tool (VNG) [47, 9], an extension of the visualization tool Vampir [62] does not perform an automatic analysis but tries to explore, like SCALASCA and DeWiz, distributed data processing. Furthermore, VNG is composed of a parallel analysis server and a visualization client, where each analysis is executed on a different platform. Another important VNG feature is the utilization of a data structure called Complete Call Graph (CCG), which holds the full event stream, including time information, in a tree.

Previous studies have used signal processing techniques in order to detect program phases. In [81], wavelets are used as a time-frequency analysis method to identify behavior changes on the locality of a program. Next, phase markers are inserted in the program using binary rewriting. When the execution of the instrumented program begins, the first few executions of a phase are used to predict all subsequent executions. This approach uses signal processing in order to predict the locality of a program and, for that reason, it is different from ours. In contrast, in [43] the authors also use wavelet analysis to perform phase analysis. Wavelet coefficients are used as a similarity metric and k-means clustering is applied on them. The goal of this work is to capture the memory bus behavior of commercial applications. This approach is different to ours because, unlike our approach, it does not use wavelets as a time-frequency analysis method. In [27], wavelets are used to reduce the amount of data that has to be analyzed to study the load imbalance on high performance computing applications. This approach achieves several orders of magnitude of data reduction using compression techniques from signal processing and image analysis. Additionally, low error and high speed reductions are achieved by the approach. The main difference between this work and ours is that we use the information obtained from the wavelet transform to make an analysis about the general structure of executions while, in this work, the wavelet transform is just used to compress data, not to understand it.

On the other hand, Paradyn [75] is a non-trace-based performance tool which analyzes data in real-time, i.e., program instrumentation and performance evaluation are done during the execution. It performs a search which is focused on answering three questions: 1) Why is the application performing poorly?, 2) Where is the performance problem?, and 3) When does the problem occur?. To answer the first question, the system includes hypotheses about potential performance problems in parallel programs. It collects performance data to test whether these problems exist in the program. In answering the second question, it isolates a performance problem to a specific program resource (e.g., a disk system, a synchronization variable, or a procedure). To identify when a problem occurs, it tries to isolate a problem to a specific phase of the program's execution. Finding a performance problem is an iterative process of refining the answers to these three questions.

Finally, mention should be made of Periscope [30], an automatic performance analysis tool for large-scale parallel systems. Periscope applies an automatic distributed online search in order to detect performance bottlenecks. It consists of a graphical user interface, a hierarchy of analysis agents and two separate monitoring systems. The graphical interface allows the analyst to begin the analysis process and to study the results. The agent hierarchy performs the actual analysis. The node agents autonomously search for performance problems which have been previously specified. Typically, a node agent is started on each node of the target machine. It is responsible for the processes and threads on that node. Detected performance problems are reported to the master agent.

Several of the approaches discussed above perform an automatic performance analysis. However, most of them do not look for the internal structure of the tracefile (and thus they need large traces to amortize the cost of the initialization phases), and only a few look for repeated patterns not based on temporal evolution of the application's execution. Thus, we believe that our approach satisfies the need for an automatic performance tool based on the structure of the temporal evolution of the application. While other approaches have attempted to overcome the increasing size of tracefiles in recent years using distributed computing, our work tries to overcome that problem by using a shrewder methodology and a more sophisticated power analysis. Furthermore, unlike the other approaches, our methodology can be parallelized.

3.3 Metrics Generation

As we have said above, our approach consists in characterizing executions of parallel applications using signals. The main reasons are that, first, signals keep the highly-detailed information contained in tracefiles, second, signals make possible to isolate a given performance metric and, third, the algorithms based on signal processing theory have a low computational complexity.

In order to summarize important aspects of parallel program performance, we derive signals from data contained in the tracefile, capturing the temporal application behavior from a particular point of view. However, these signals can be generated from any kind of time-stamped data. For this reason, out approach is not limited to work with tracefiles. Each metric we generate summarizes one aspect of the parallel program's execution. In this section, we discuss a set of metrics. Some of the metrics put more emphasis on the execution phases than on those associated with communication, while others put more emphasis on communication. The automatic system takes these metrics as input and processes them. Depending on the situation, some metrics show phases of the program's execution more clearly than others.

3.3.1 Number of Processes Computing

This metric shows the number of processes which are performing computations at each instant of time. It indicates the location of intensive computing regions. Typically, its lower values are located on instants of time when the execution is performing communications. Otherwise, when the execution is performing the main computations, the value of the signal will reach its highest values. Another important qualitative characteristic of this metric is its low variance, explained by the fact that it can only reach a finite set of values, from 0 to P, where P is the number of processes. It is generated from the running states contained in the tracefile. For each thread, we generate a signal which contains, for each instant of time, t, value 0 if the thread is not in a running state and 1 otherwise. Finally, signals generated by threads are added, obtaining this metric.

3.3.2 Sum of Durations of Computing Bursts

This metric shows the instantaneous sum of computing bursts. It will reach its lower values when the execution is performing a few short calculations. Otherwise, high values will appear when a lot of long bursts are being computed by the processors. For this reason, this metric summarizes the kind of calculations we find in execution. Finally, The variance of the metric will be higher than that of the Number of Processes Computing metric because it can reach a wide set of parameters, depending on the duration of running bursts.

For each thread, we generate a signal which contains, for each instant of time t, the duration of the running burst that is being executed. If there is no running burst in this moment, i. e. the process is inside a MPI Call, the value assigned to t is 0. Finally, the signals generated for all the threads are added. Focusing on point to point MPI calls provides a way of looking at the application parallel structure from the point of view of communications between processes. This approach complements the metrics explained above because it takes into account communications performed by the application.

3.3.3 Number of Point to Point MPI Calls

This metric shows the number of processes which are executing a point to point MPI call at a given moment. Therefore, the range of its values is from 0 to P, P being the total number of processors. It reaches its minimum values when the application is in a calculation phase because these phases have very few communications. On the other hand, the maximum values of this metric are reached when the occupation of the interconnection network is at its maximum.

To generate this signal, we have to cover all the tracefile and, every time we find the beginning of a MPI call we add 1 to the signal. When we find the end of a MPI call, we subtract 1 to the signal. Obviously, the value of the signal at the beginning of the execution is 0.

3.3.4 Number of Collective MPI Calls

This metric shows the number of processes which are executing a collective MPI call. The range of its values is from 0 to P, P being the total number of processors. Typically, the execution of applications implies a lot of collective calls at the initialization phase in order to provide the information to start the computations. However in computation phases we can also find collective calls. The difference is that, on the one hand, the collective calls in the initialization phase do not appear following a periodic pattern and, on the other hand, the collective calls on the computation phase have a periodic pattern.

3.3.5 Instructions per Cycle

This metric shows the average rate between instructions and cycles of the set of processes in a given instant of time. These values are extracted from the hardware counters contained in the tracefile. The values of these counters are extracted at the beginning and at the end of computing bursts. For this reason, the granularity of the signal will be equal to the granularity of the signals derived directly from the computing bursts.

3.3.6 Communications

This metric shows the number of flying messages at a given moment. It is very close to the number of point to point MPI calls but is not exactly the same, for two reasons: first because the transmission of a messages involves two MPI calls and second because the execution of a MPI send or receive does not imply an instantaneous transmission of the message. Typically, if the intercommunication network is collapsed, these two metrics will show different behavior because the MPI calls need to wait in order to send or receive the message. However, if the network is not collapsed, the behavior of these two metrics is closer.

3.3.7 Examples

In figure 3.1, we show examples of the metrics we referred to above in this section. In this figure we show, for each instant of time of the execution, the value of the metrics. The figures were obtained from a tracefile that was generated executing WRF-NMM [64] application with 128 processors in MareNostrum Supercomputer. At the top of the figure we can see two signals: The sum of durations of computing bursts, expressed in milliseconds, and the number of processes computing. In the middle, there are two more metrics: The number of point-to-point MPI calls and the number of collectives. Finally, at the bottom, we have the IPC and the total number of flying communications.

It is important to state that we see the same kind of behavior in all of the signals. In general we see an initial phase of low activity, a phase in the middle with strong activity and, finally, a phase with low activity. The exception to this behavior is the collective communications signal. In this signal, we see a strong activity at the beginning of the execution. The reasons for the common behavior of the signals and the exception of the collectives are explained in the next section below.

3.4 Program Phases

Typically, high performance computing applications radically change their behavior when executed on parallel architectures. For example, in the case of MPI applications, there is little in common between the interval of time in which a MPI collective call is being performed, and the interval of time when all the processors are performing computations. The first is characterized by an intense utilization of the interconnection network and the lack of computations while the second is characterized by great activity focused on computations and a lack of communication between processes. This is only a very simple example of the wide diversity of situations that can be found within the execution of a high performance computing application. In the next section, we explain the typical structure of a high performance computing application and how we detect it.

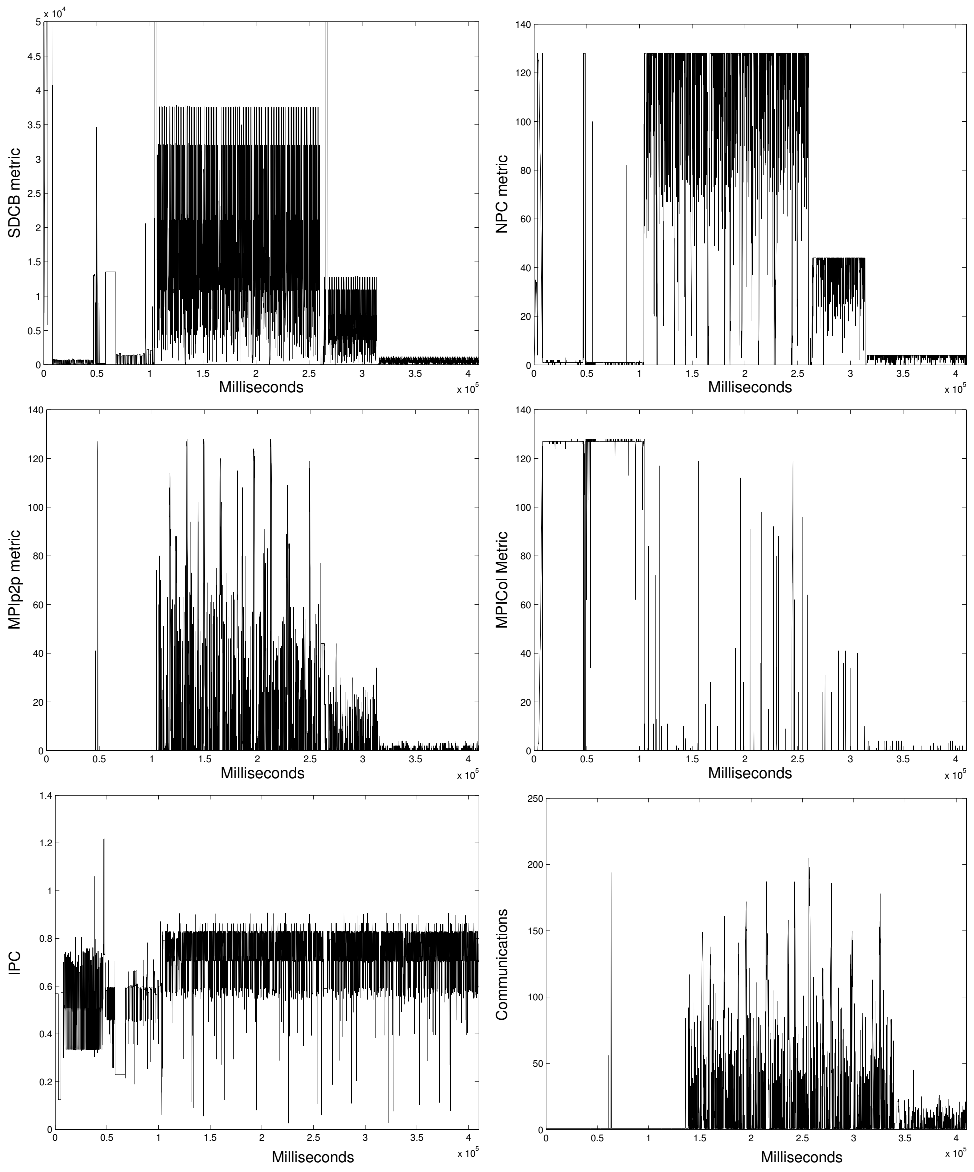


Figure 3.1: **Visualization of the metrics explained in section 3.3. At the top, we show the Sum of Duration of Computing Bursts and the Number of Processes Computing. In the middle, we show the number of Point to Point MPI Calls and the number of Collective Calls. Finally, at the bottom, we see the IPC and the number of Communications. These metrics have been generated from a tracefile of an execution of WRF-NMM application on 128 processors.**

3.4.1 Description of the Program Phases

The execution of scientific applications has several well-defined phases. Typically, the execution has an initialization phase, which consists of the initial communications in order to assign initial values to the variables, domain decomposition, etc... This initialization phase is often not targeted by the analysis since the high cost is typically amortized over a long run. The performance of this initialization phase is not related to the core scientific computation we want to study here. Also, many applications show a final phase, which consists, mainly, of output data operations. As with the initialization phase, this final phase is not important in studies of parallelization efficiency.

The core phase that interests us, and that is found in scientific parallel applications, is the computation phase. This phase is characterized by sets of computing bursts alternated by communications, and typically has a periodic behavior prompted by the application's structure, which performs iterations on space (physical domain of the problem) and time. Thus, there is a pattern that we can find repeated many times during this computation phase. Furthermore, it is in this computation phase when the application shows its parallel efficiency, and it is in this phase where it is interesting to study whether the application has a good parallelization or not, whether the load is balanced optimally, etc. Consequently, the parallelization study of the application has to be made here. Although this is the typical behavior found in the execution of many scientific applications, it is possible to find different behaviors. Nevertheless, the most important issue for us is whether our automatic system will always be able to detect the execution phases.

In the signals depicted in figure 3.1 we see this behavior clearly. The case of the collectives signals is specially interesting because it shows the strong collective communication pattern in the initialization phase in order to provide the tasks with the initial values to start the computations. Furthermore, the other signals have the typical behavior showing strong activity in the computation phase.

3.4.2 Detection of the Program Phases

As explained in the previous section, the initialization and final phases are typically not targets in performance analysis because, since they strongly depend on the input/output system of the machine, the interconnection network, the operating system and many other factors, they do not represent the application. These considerations explain why we are not so interested in these phases and why we concentrate on the detection of the computation phase in order to guide and focus performance analysis.

The computation phase typically comprises computation bursts followed by communications. And, typically, it has periodical behavior; that is, there is an iterative pattern repeated many times. For all of these reasons, we can expect the highest frequencies of the signals to appear in the computation phase. Hence, our criterion for automatically detecting the computation phase of applications is: the region where the highest frequencies occur.

The tool used to detect the higher frequencies is wavelet transform, several generalities of which were explained above in section 2.3.1. As has been explained, wavelet transform provides a multi-resolution analysis able to identify the locations where the frequencies occur. Given a signal, assume that we sample it in $N = 2^i$ samples, using a sampling frequency α. As was explained in section 2.3.1, we obtain from wavelet transform 2^{i-1} coefficients of the locations of frequencies belonging to $[\frac{\alpha}{4}, \frac{\alpha}{2}]$. In general, wavelet transform gives 2^{i-k} coefficients of the locations of frequencies belonging to $[\frac{\alpha}{2^{k+1}}, \frac{\alpha}{2^k}]$. Remember that, on the one hand, we are interested essentially in the detection of the highest frequency in order to detect the computation phase and, on the other hand, the precision of the location is best for the highest frequencies. For these reasons, we base our detection on the 2^{i-1} coefficients (level 1) of the frequencies belonging to $[\frac{\alpha}{4}, \frac{\alpha}{2}]$. If at this level there is no significant information, we can use the 2^{i-2} coefficients (level 2) whose frequencies belong to $[\frac{\alpha}{8}, \frac{\alpha}{8}]$, and so on. Remember that we have i levels of coefficients, where i is an initial parameter.

Once the wavelet transform has provided us with the coefficients, we make a selection of the high-frequency regions based on two parameters: λ and δ. The first represents the threshold from which we select the coefficients, while the second represents the granularity of the selection. More precisely, λ is a number between 0 and 1 that defines the minimum of the selected coefficients. It is defined in terms of the maximum, that is, $\lambda = 0.3$ means that we select all the coefficients equal or greater than the 30% of the maximum coefficient. The second number, δ, says, given a selected coefficient due to λ's criterion, how many neighbor coefficients we select. For example, if $\lambda = 0.3$ and $\delta = 10$, we will select all the coefficients equal or greater than 30% of the maximum and, after, that, for each selected coefficient, we take ten coefficients from the right neighborhood and ten more from the left neighborhood.

Figure 3.2 gives an example of a signal. It is clear that this signal has three very defined phases: first, a phase with low frequency behavior, then, a second phase with high frequency behavior and, finally, a low frequency phase. In figure 3.3 we can see how wavelet transform shows the presence of high frequencies. Finally, in figure 3.4 we show the selection we made from the wavelet transform. The parameters are $\lambda = 0.3$ and $\delta = 10$.

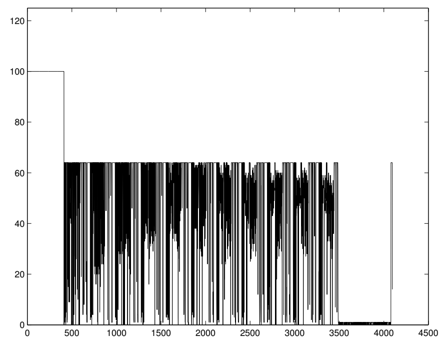


Figure 3.2: **Input Signal. Sampling Frequency =** α

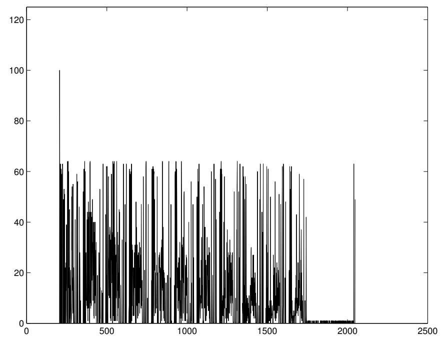


Figure 3.3: **Wavelet Output. Location of frequencies between** $[\frac{\alpha}{4}, \frac{\alpha}{2}]$

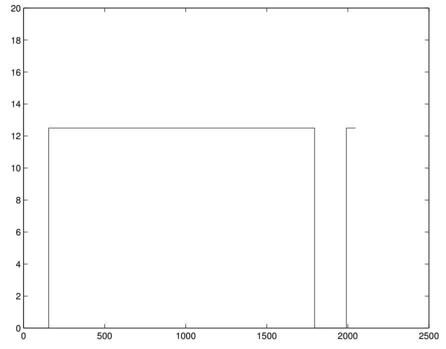


Figure 3.4: **Selection**

3.5 Corrupting Factors

By corrupted regions we mean those regions with distorted relative timing behavior of the application being analyzed. See [60] for a detailed discussion of these distortions. Furthermore, all these perturbations have a common characteristic: they are neither caused by the application nor by the architecture. They are caused by external factors such as tracing packages, unknown system activity, etc. Different phenomena or metrics can be identified as being the cause of a significant perturbation of the program behavior. An example of these phenomena is flushing, which is caused by the fact that tracing packages keep individual records in a buffer in memory during the tracing process. The problem is that when the buffer is full, these records have to be flushed to disk. This flushing takes so long that it affects the execution and the statistics derived from the tracefile. Also, flushing does not appear simultaneously in all the processes, although it is typical that the flushing of different processes occur in bursts. Other corrupting factors are preemptions, problems with the communication network of the machine, etc... In the next paragraphs, we describe the main corrupting factors and we explain how we automatically detect them.

3.5.1 Flushing

Tracing packages [67] keep individual records in a buffer in memory during the tracing process. The problem is that when the buffer is full, the records will have to be flushed to disk. This flushing takes so long that it can affect execution. Furthermore, it can affect other processes as well as the process that is currently flushing, thus generating further delays. If we do not rule out these delays, the statistics will be notably affected. Another important property of flushing is that it does not have to appear simultaneously in all processes, even if, in many applications, flushing of different processes typically occurs in bursts. In figure 3.5 we give an example. We can see the impact of flushing records to disk on normal execution behavior. The signal indicates how many processes are flushing data to disk. The application is WRF-NMM [64] executed on 128 processes.

The phenomenon of flushing is characterized by a signal indicating for each instant of time the number of processors flushing to disk. We derive this signal from a tracefile that contains flushing events, indicating when each process starts and finishes the flushing to disk. Figure 3.6 shows an example of a flushing signal. In this kind of signal we frequently observe interleaved small bursts with flushing peaks and periods without flushing. The tracefile is perturbed not only during flushing but also at instants right after flushing peaks. Therefore, we want to consider the bursts of flushing as a single perturbed region. With this objective, we use a set of morphological filters, defined in the context of Mathematical Morphology. These filters are non-linear and are based on the minimum and maximum operations, aiming at the study of structural properties of the signal. The two basic morphological filters are Erosion and Dilation. The first has the property of eroding those regions of the signal with values different to zero. The second filter has the property of dilating the regions of the signal different to zero. Both operators have an associated width that has to be specified before the filter is applied. If we combine the two operators doing a Dilation followed by an Erosion we obtain an interesting result: first, the Dilation will merge the small regions with their larger or nearby neighbors, and after that, the Erosion will allow us to return towards the initial signal, except in the cases that two different regions have been merged by the Dilation. With this combination, we obtain a new morphological operator called Closing. Figure 3.7 shows the result of performing a Closing on the signal represented in figure 3.6. Note that the small regions that appear in figure 3.6 have been merged into a larger region. This scheme is specially useful in analyzing flushing signals because it merges several close regions where some processes are flushing data to disk. As we have said, the execution is perturbed not only during flushing but also at instants right after flushing. The closing operator merges the bursts of flushing into single perturbed regions.

3.5.2 Preemptions

Preemptions by daemons or other system activity are another cause of performance degradation in large scale applications. Identifying preemptions is important in order to calculate their impact and to try to reduce it. Otherwise, if the analyst suspects that a given trace file is affected by preemptions, he will drop all the file. However, there may be parts of the trace without perturbation and, therefore, good for the analysis. In conclusion, we can say that the identification of regions with preemptions allows the analyst to improve the confidence of the tests. An easy way to identify regions with preemptions is based on the cycles counter available in the trace. The ratio of cycles to time should be equal to the processor clock frequency. If we obtain a value less than the clock frequency, we will infer that for some time the process did not get CPU and, therefore, the cycles counter did not increase.

To automatically detect regions perturbed by preemptions, we want to eliminate small oscillations of the signal, because they are not significant, and keep the large intervals where the value of the signal is lower than the CPU frequency. The solution is to perform a closing to the signal because it eliminates small oscillations under the

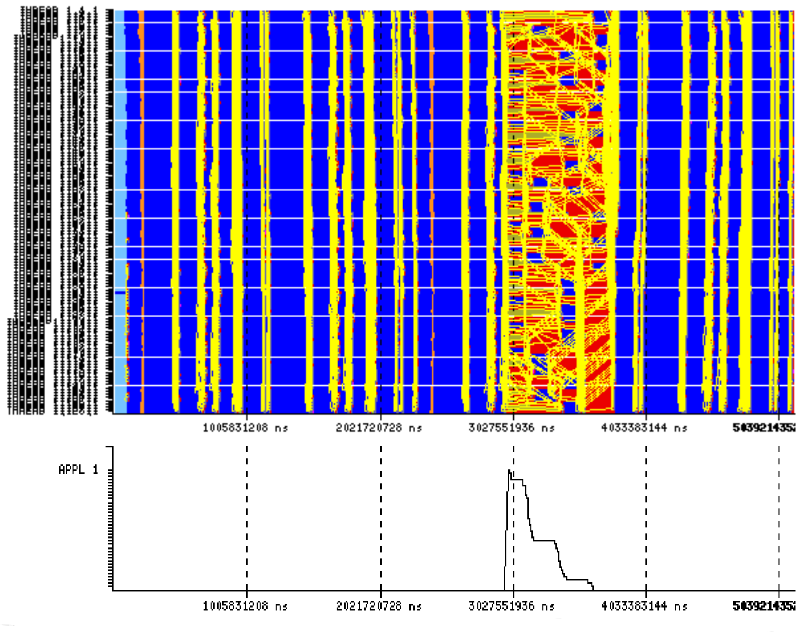


Figure 3.5: **An example of flushing. Processes are flushing records to disk.**

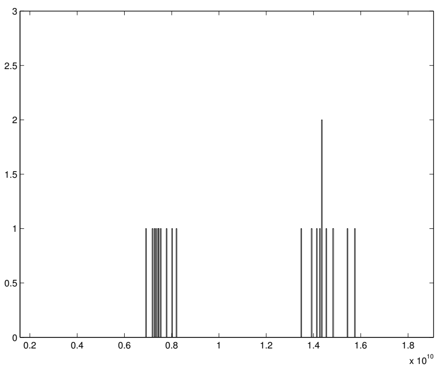


Figure 3.6: **Signal**

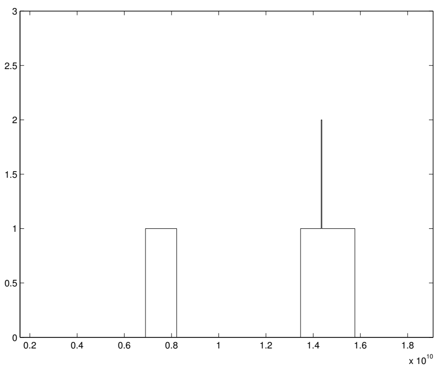


Figure 3.7: **Closing of the Signal**

CPU frequency and keeps the significant ones. After that, we consider regions with low values as corrupted regions due to preemptions.

3.5.3 Clogged System

This term is applied to the regions with a large number of messages in transit while a low system bandwidth is achieved. There are several interpretations of this behavior. It may be caused by the preemption of one thread that plays a fundamental role in the critical path of the region where the behavior takes place. Another possible reason is that another system activity stole bandwidth while the execution of the application was running. As we have said, the metric that will identify this behavior is the ratio between the messages in transit and the system bandwidth achieved at a given moment. This metric can be easily derived from the records contained in the trace file. Figure 3.8 gives an example. The application is NAS-SP [3] executed on 121 processors. At the top we show a typical Paraver visualization. In it we can see initially normal execution behavior followed by a region where the transmission of messages is extremely low. Finally, normal execution behavior is resumed by the application. At the bottom, we can see the ratio between the messages in transit and the system bandwidth. This metric is able to characterize the communication problems that the execution is suffering. This metric is directly derived from the tracefile of the application. Typically, it is characterized by small bursts in which the signal has low values. This is the normal behavior, indicating that there are no problems with the communications. However, if the interconnection network is collapsed and many messages are waiting to be sent, or if the bandwidth is remarkably low, the signal will reach high values in large bursts.

To automatically detect regions perturbed by clogged system effect we need a filter capable of eliminating the sets of short, low bursts and of keeping back long, high ones. The Opening filter is useful for this operation. It is composed of, first, an Erosion, which eliminates short bursts, and, second, Dilation, which restores the remaining bursts to their original size. Thanks to Opening, we are able to automatically detect the location of long and high bursts and eliminate the low values of the signal.

In summary, following the methodology described in this section, we make three steps: First, we generate a signal from the initial tracefile, for example the number of processes flushing to disk. Second, we apply a morphological filter to extract the relevant information from the signals. The choice of the morphological filter depends on the characteristics of each signal. For example, in the case of a flushing signal, the Closing filter is used in order to merge the pulses of the signal that are too close

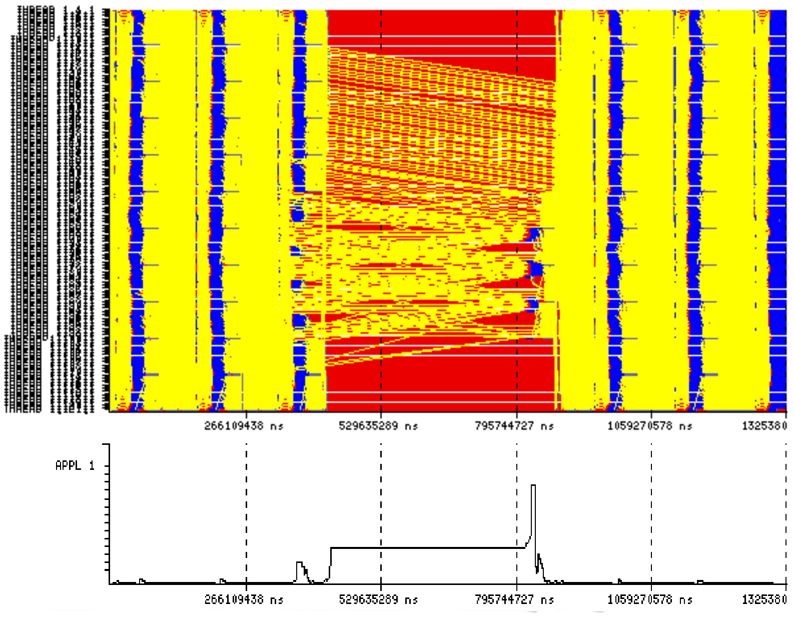


Figure 3.8: **An example of clogged system. Many communications are being transmitted with an extremely low bandwidth.**

in a unified burst. The width associated to each filter is based on the minimum span of time we consider useful for the analysis. Third, we rule out the perturbed regions, taking into account the pulses of the resulting signal. They indicate which are the perturbed (thus, useless) regions of the tracefile and the areas of the signal which are the non-perturbed regions. Finally, the process of identification of structure explained in the next section is applied to these non-perturbed regions of the trace.

3.6 Structure

Once we have detected, on the one hand, the regions with relevant information, that is, with high frequency behavior and, on the other hand, we have detected and ruled out the regions corrupted by factors not related to the application or the architecture, we perform an analysis focused on the detection of the iterative behavior of the largest non-perturbed region with high-frequencies.

There are two main characteristics of the structure that we look for in a region. First, the periodic structure is based on the identification of several different smaller regions that are very similar. We say that two execution regions are very similar if the signal that represents a metric is very similar in the two regions. The second main characteristic we look for is the hierarchy of the periodicities. The periodic structure is expressed in different levels: the first level is the structure within the original trace, the second level of periodicities is the structure within one period of the first level, and so on. In order to obtain that hierarchical structure, our algorithm is recursive, that is, when the internal structure of one level is detected, we apply again the algorithm within one period of that level.

The information derived from this hierarchical structure based on periodicities is useful in at least two ways. First, our tool shows the analyst the structure as a first approach to the execution of the application under analysis. Second, the tool provides the user with chunks of traces which are cut from the original trace. These small traces are representative parts of the original trace at different levels of the structure. These chops of the original tracefile allow an accurate analysis, but the tool also reports several metrics (percentage of time in MPI, ...) for each of the regions to give a first approach to the performance obtained by the application.

The analysis is based on signals derived from the region we are studying. As indicated in section 3.3, it is possible to select many signals to perform the study of the periodicity behavior of the application. In applications with strong communication activity, signals related to MPI calls can give good information. On the other hand, in applications with strong computations activity, signals derived from computing bursts are a better option. In general, signals related to computing bursts give better results, as we will see in section 3.7.

3.6.1 Iterative Pattern Detection

To find the internal structure of the application we apply the autocorrelation function to the signal generated from the tracefile. Below is a simple definition of autocorrelation function:

$$A(k) = \sum_{i=0}^{N-1} (x_i)(x_{i+k}) \tag{3.1}$$

where the set $\{x_i\}$ is generated sampling the signal obtained from the tracefile. The higher values of the function $A(k)$ will be reached when k is equal to one of the main periods of $\{x_i\}$. However, for accuracy reasons [73], the numerical values of $A(k)$ are not obtained following (3.1). It is possible to obtain the value of $A(k)$ function performing, first, a Discrete Fourier Transform (DFT) and, after that, an Inverse Discrete Fourier Transform (IDFT) taking the square of the modulus of each spectral coefficient obtained with the DFT [73]. This method can be implemented using an FFT library. An important feature of FFT is its computational complexity, O($n\ log(n)$), which allows us to calculate the values of $A(k)$ in reasonable time. Once we have the values of the Autocorrelation function, the principal periodicities are selected [24]. We will select the maximum of the relatives maximums. In other words, the period we select, T, will satisfy the following:

$$A(T) = Max\{A(k)|A(k-1) < A(k) > A(k+1), k > 0\} \tag{3.2}$$

Here, we should point out that the signal may not have meaningful periods, or that has 2 or 3 significant periods. Therefore, there is a need for a method to estimate the correctness of the period obtained. The approach taken is the following: assuming that T is the period identified and that M is the set of those k where $A(k)$ has a relative maximum.

$$\forall k, (k \in M \wedge k \neq T) \Rightarrow 0.9 > \frac{A(k)}{A(T)} \tag{3.3}$$

If the above formula holds, the 90 % of the value of $A(T)$ is higher than all the values in the rest of the maximum values. In that case, we will assume that T is a good approximation to the main period. We will check the logical formula (3.3) every time we perform an autocorrelation. If the formula is true, we will assume the correctness of the results. If not, we will perform a filtering to the original signal in order to filter the small oscillations that can perturb the results and repeat the process. We perform the filtering in order to obtain a coarse-grained description of the signal. This replacement of a fine-grained description with a lower-resolution coarse-grained model will outline the global signal behavior.

Another good criterion to estimate the correctness of the period obtained is the detection of its harmonics. By harmonics we mean multiples of the maximum of the relative maximums. For example, if the maximum of relative maximums is T, its first harmonic is $2T$. If this first harmonic is the second maximum of the relative maximums, the correctness of the period obtained will be remarkably high because the existence of harmonics indicates a strong iterative behavior whose value is T.

Once a "good" period is identified, we select a region of the signal containing an iteration of this period and apply the methodology again to look for inner structure. At the same time, we cut the original tracefile in order to provide the analyst with one period on every periodic zone we found. Finally, this methodology needs the execution of intense processes. In order to perform these executions and take advantage of several concurrent processes, we have implemented the methodology with GRID Superscalar [2], a grid programming environment developed at BSC.

3.6.2 Representative Iterations

Once we have extracted the value of the iterative pattern, T, we have to select n iterations from the execution. This is not a trivial issue, because we can often find iterations which are not good representatives of the whole execution. To detect the best iteration, we perform a cross-correlation. In signal processing, cross-correlation is a measure of similarity of two signals. It reaches its maximum where the first signal is closer to the other one. In this work, we consider the cross-correlation between the input signal (extracted from the execution of the application) and a test signal. This test signal contains n sinusoid periods of value T. Below is the definition of the test signal:

$$f(x) = \begin{cases} sin(\frac{2\pi x}{T}) & \text{if } x \in [0, nT] \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

The underlying idea here is to detect where the input signal is closer to a sinusoid function of period T. To detect this, we select the point where the cross-correlation between the input signal and the test one is higher. This point, t is the beginning of the representative segment.

In summary, at the end of the spectral analysis, a representative subset which contains one iteration is selected. This representative subset starts at time t and finish at time $t + nT$. If it was possible, we applied the methodology recursively over the subset. The extraction of hierarchical patterns is based on spectral analysis. This detection and the subsequent hierarchical search are detailed in [12].

3.6.3 Hierarchy detection

As seen above, we perform a hierarchical detection of the structure of executions of applications. The objective of this hierarchical detection is to identify the minimal representative region of the execution. Many high performance computing codes base the computing phase on a succession of nested loops. It is possible to find internal levels of these nested loops that are representative of the whole computing phase because the main calculations are carried out within these internal levels. However, the initial structure detection finds the most external iterative structure, that is, the iterations of the most external loop. If we want to detect the structure related to internal nested loops, we will apply the methodology recursively over the selected representative region.

The structure found within the whole execution of the application is called Level 1 structure. As we have said, it is the structure related to the most external levels of the nested loops. Subsequently, the iterative structure found within one iteration of

Level 1 is called Level 2 structure. It is related to the second level of the nested loops. This terminology continues as we apply the searching methodology recursively. The search for hierarchy finishes when no iterative structure is found.

Finally, it is important to state that this internal iterative behavior within the Level 1 structure exists only if most of the computations are performed within the internal nested loops. Therefore, the existence of nested loops in the source code does not warrant the detection of Level 2 structure, since it is possible that the application performs the main calculation outside the most internal loops.

3.7 Evaluation

This section is divided into four subsections. In the first one, we provide an evaluation of the detection of the complete internal structure of several applications. This subsection is focused on the evaluation of the hierarchical search for structure of applications and on the detection of the internal structure among them. In the second subsection, we provide an evaluation of the automatic analysis on several applications using the metrics explained in section 3.3. This subsection is focused on the evaluation of the efficiency of several metrics in detecting internal structure. For this reason, the results depicted in this section are more focused on the differences of the results obtained with the different metrics than on the complete internal structure of the applications. The third subsection is focused on evaluating the size reductions achieved using our technique, that is, we compare the size of the original tracefile with respect to the size of two iterations extracted from the original data. Finally, in the fourth subsection we show that the automatic methodology scales linearly with respect to the size of the input.

3.7.1 Evaluation of Complete Internal Structure and Hierarchical Searching

In this section, we evaluate the complete internal structure detected by spectral analysis and its hierarchical search. In order to do so, we apply the methodology explained in this chapter to four real applications: Liso [41] with 74 processors, Idris [95] with 200 processors, Gadget [91] with 256 processors and Linpack[54] with 2048 processors. These have been executed and traced in MareNostrum [57]. The structure found in these applications is based on one of the metrics explained in section 3.3, the MPI Point to Point calls.

In figure 3.9 we show graphically a part of the structure of the Liso tracefile. This structure is shown with a Paraver visualization. In this visualization, the horizontal

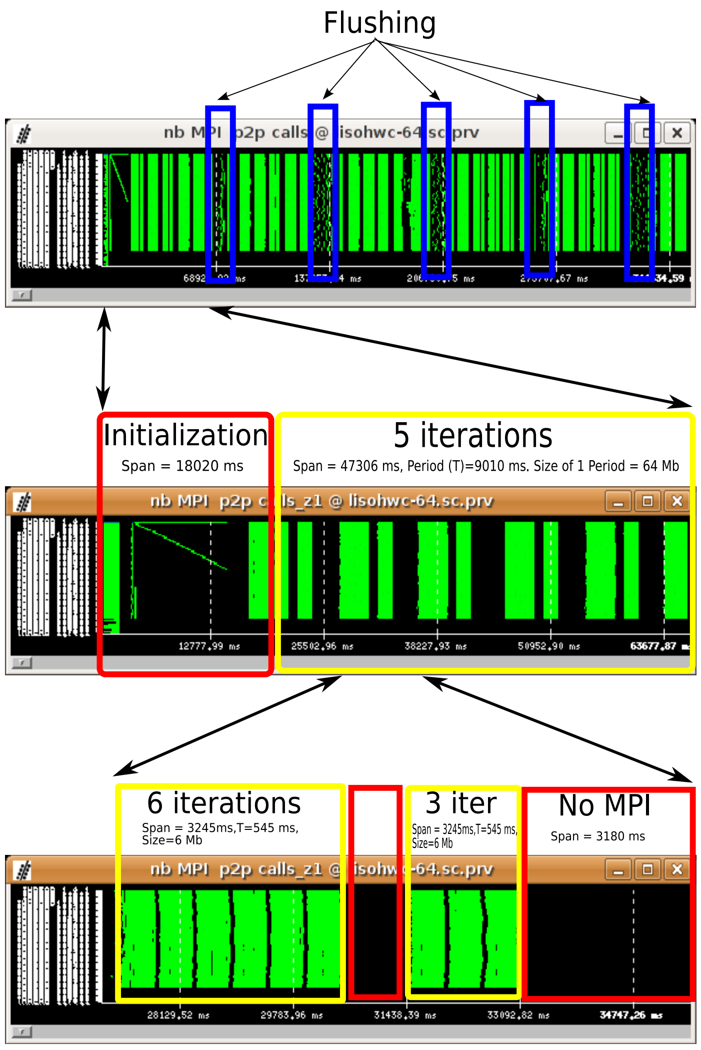

Flushing
nb MPI p2p calls @ lisohwc-64.sc.prv
Initialization
Span = 18020 ms
5 iterations
Span = 47306 ms, Period (T)=9010 ms. Size of 1 Period = 64 Mb
nb MPI p2p calls_z1 @ lisohwc-64.sc.prv
12777,99 ms
25502,96 ms
38227,93 ms
50952,90 ms
63677,87 ms
6 iterations
Span = 3245ms,T=545 ms, Size=6 Mb
3 iter
Span = 3245ms,T=545 ms, Size=6 Mb
No MPI
Span = 3180 ms
28129,52 ms
29783,96 ms
31438,39 ms
33092,82 ms
34747,26 ms

Figure 3.9: **On top, visualization of the whole Liso tracefile and the flushing zones . In the middle, the first region without flushing is shown. We highlight a region without periodic structure and a region with 5 iterations. At the bottom, we show one of these iterations.**

Span/#it/T			Flushing
Level 0	Level 1	Level 2	
356352/1/-	18020/1/-		
	47306/5/9010	3245/6/545	
		970/1/-	
		1615/3/550	
		3180/1/-	
	10273/1/-		X
	49166/5/9105	1490/3/545	
		1140/1/-	
		1585/3/550	
		3030/1/-	
		1860/3/535	
	11880/1/-		X
	60773/7/9100	3215/6/540	
		1004/1/-	
		1615/3/550	
		3266/1/-	
	11576/1/-		X
	49253/5/9145	1650/3/550	
		2825/1/-	
		3400/6/550	
		1270/1/-	
	12269/1/-		X
	48347/5/9045	3185/6/550	
		1015/1/-	
		1625/3/560	
		3220/1/-	
	13312/1/-		X
	24177/3/8905	1875/1/-	
		3425/6/550	
		1010/1/-	
		1635/3/555	
		960/1/-	

Table 3.1: **Liso application structure detected by our system. The time units are milliseconds. The first three columns show the hierarchical levels of periodicity. Level 0 column shows the total elapsed time, Level 1 column shows the different phases detected by the automatic system and Level 2 shows the internal structure of one of the periods of Level 1. In the first three columns, each cell contains three numbers: The total span of the region, the number of the periods found in that region and the duration of each period.**

Span/#it/T			Flushing
Level 0	Level 1	Level 2	
1091695/1/-	205740/1		
	885955/9/102870	550/5/110	
		590/1/-	
		540/7/80	
		101190/1/-	

Table 3.2: **Idris. Table Representation**

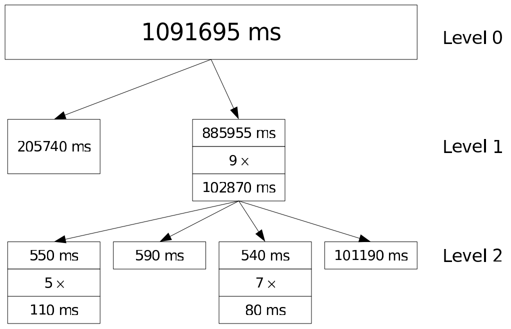


Table 3.3: **Idris. Tree Representation**

Span/#it/T		Flushing
Level 0	Level 1	
1097460/1/-	55000/1/-	
	811126/23/35095	
	286334/1/-	X

Table 3.4: **Gadget**

Span/#it/T		Flushing
Level 0	Level 1	
223168/1/-	82850/1/-	
	140318/130/1130	

Table 3.5: **Linpack**

axis represents the time and the vertical axis the different processes. The color black means that a given process in a given instant of time is not executing any MPI Call. On the other hand, a light colored point (green when printing or visualizing in color) indicates that an MPI call is being executed by the process. The figure first shows a visualization of the whole tracefile. The flushing regions are also outlined. In the second part of figure 3.9 we show the structure of the first region without flushing. In this case, we show, first, a region with a non-periodic structure that corresponds to the initialization phase of the application. The span of the initialization phase is 18029 ms. After that, there is periodic region with 5 iterations. The span is 47306 ms and the period shown is 9010 ms. Finally, in figure 3.9 we show one of the iterations of the periodic zone.

Table 3.1 shows that the automatic system was able to detect the structure shown in figure 3.9. Furthermore, we can see the results of the automatic analysis for the whole Liso application. In this table we show two characteristics of the execution: first, from left to right we show the hierarchy. Second, from top to bottom we show the temporal sequence.

In the first column, we show the duration of the whole execution in milliseconds. Next, in the second column of table 3.1 there is a decomposition of the total elapsed time of execution. In this second column, we show a set of numbers in each cell. The first number is the total time span of the region, the second is the number of periods found in that region and, finally, the third is the duration of each period. In the regions where no periodicity was found a dash line is written.

The second column refers to the first level of hierarchical structure, i. e., is the structure over the original trace. For example, the MPI Point to Point calls distribution of the first (18020 ms) and second (47306 ms) regions of the tracefile shown in figure 3.9 are represented in the first and second cells of the second column.

The third column contains the second level of the structure, i.e, the structure that can be found in one of the periods of the first level. For example, the second, third, fourth and fifth cells of the third column are the decomposition of one of the periods of level 1. The MPI Point to Point distribution is shown on figure 3.9. Here, the second level of structure can be identified, with 6 periods (of 545 ms) of communication, a computation period (of 1005 ms), 3 more periods of communication (each of 550 ms) and a final computation period (of 3180 ms).

Finally, the fourth column shows the existence of flushing events in a given region of the tracefile.

Note that the first five rows correspond to the structure represented in figure 3.9

The output of the tool is basically the information contained in this table plus the names of the files where the chops of the original tracefile can be found.

In table 3.2 we show the structure detected in the Idris application. In table 3.3 we show another possible representation of the same information. In this representation, the hierarchical structure is depicted from top to bottom and the temporal sequence is drawn from left to right. Finally, in tables 3.4 and 3.5 the structure found in Gadget and Linpack applications is shown.

3.7.2 Evaluation of Metrics

In this section, we evaluate the automatic system using several applications as input and changing the metrics. The applications are ALYA [40] executed on 64 processors, some of the NAS benchmarks [3]: NAS-BT class C executed on 16, 36, 64, 121 and 256 processors, NAS-CG class C executed on 16, 32, 64 and 128 processors, CPMD [18] on 64 processors, NAS-LU class C on 16, 32 and 64 processors, NAS-MG class C on 16, 32 and 64 processors, NAS-SP class C on 16, 36 and 64 processors, RTM [5] on 32 processors, SPECFEM3d [48] on 1944 processors, VAC [76] on 128 processors and WRF-NMM [64] on 128, 256 and 512 processors.

We analyzed these applications using many metrics: Sum of Duration of Computing Bursts (SDCB), Number of Processes Computing (NPC), Number of Processes in a MPI point-to-point call (MPIp2p), Ratio between the Cycles and the Instructions (CPI), Number of Flying Communications (Communications), Number of Floating

Name	Processors	SDCB	NPC	MPIp2p	CPI
		Main Period	Main Period	Main Period	Main Period
ALYA	64	11132	Not Found	Not found	Not found
BT	16	1277	1277	1277	1276
	36	639	639	639	639
	64	357	357	357	357
	121	197	197	197	197
	256	115	115	114	Not found
CG	16	39	39	39	39
	32	23	23	23	Not Found
	64	17	17	17	Not Found
	128	11	11	Not Found	Not Found
CPMD	64	229393	Not Found	Not Found	Not Found
LU	16	1441	1439	1439	Not Found
	32	530	539	539	Not Found
	64	227	227	209	Not Found
MG	16	1777	1775	1773	Not Found
	32	982	981	981	Not Found
	64	372	372	372	372
RTM	32	1175	1175	1170	1170
SP	16	1341	1341	1340	1341
	36	841	840	840	840
	64	330	330	330	330
SPECFEM3D	1944	10626	10356	Not Found	Not Found
VAC	128	4644	4667	4631	Not Found
WRF	128	2194	2194	2194	Not Found
	256	1193	Not Found	Not Found	Not Found
	512	792	Not Found	Not Found	Not Found

Table 3.6: **Evaluation of Spectral Analysis. Metrics used are SDCB (Sum of Duration of Computing Bursts), NPC (Number of Processes Computing), MPIp2p (Number of Processes in a MPI point-to-point call) and CPI (Cycles per Instruction). Results are depicted in milliseconds.**

Name	Processors	Communications	FLOPS	MPICol	Instructions
		Main Period	Main Period	Main Period	Main Period
ALYA	64	Not Found	Not Found	11395	Not Found
BT	16	1277	1277	Not Found	1277
	36	639	639	Not Found	639
	64	357	357	Not Found	357
	121	197	197	Not Found	197
	256	115	114	Not Found	116
CG	16	39	Not Found	Not Found	Not Found
	32	23	Not Found	Not Found	Not Found
	64	17	Not Found	Not Found	Not Found
	128	Not Found	Not Found	Not Found	Not Found
CPMD	64	Not Found	Not Found	229386	Not Found
LU	16	1442	Not Found	Not Found	1441
	32	529	Not Found	Not Found	Not Found
	64	227	210	Not Found	Not Found
MG	16	1774	1778	Not Found	1778
	32	982	982	Not Found	Not Found
	64	372	372	Not Found	Not Found
RTM	32	1170	Not Found	Not Found	Not Found
SP	16	1341	1341	Not Found	Not Found
	36	841	841	Not Found	Not Found
	64	330	330	Not Found	Not Found
SPECFEM3D	1944	Not Found	Not Found	Not Found	Not Found
VAC	128	4638	Not Found	4626	Not Found
WRF	128	2194	Not Found	2193	Not Found
	256	Not Found	Not Found	Not Found	Not Found
	512	Not Found	Not Found	Not Found	Not Found

Table 3.7: **Evaluation of Spectral Analysis. Metrics used are Communications (number of flying communications), FLOPS (number of floating point instructions executed), MPI-col (number of processes in a MPI collective call) and Instructions (number of instructions performed). Results are depicted in milliseconds.**

Point Instructions Executed (FLOPS), Number of Processes in a MPI Collective Call (MPICol) and Number of Instructions Performed (Instructions).

Tables 3.6 and 3.7 depict the results. In the first column of these tables, we show the name of the application and in the second column we specify the number of processors used to execute the application. In the third column of table 3.6 we show results using the Sum of Duration of Computing Bursts metric. In each cell of this column, we depict the duration in milliseconds of the main periodicity detected within the execution of the application. We do not show information about the whole structure of the application, as was done in table 3.1, in order to provide a more friendly representation of data. As we have said, in this section we are more interested in comparing the results of the automatic analysis using many metrics than in a complete description of the results. Then, in the fourth column of table 3.6 we show the results using the Number of Processes Running metric, in the fifth we depict the results taking the Number of Processes in a MPI point-to-point call and, finally in the sixth column we show the results using the Ratio of Instructions per Cycle.

In table 3.7 we show, in the third column, results taking the Number of Flying Communications as input metric, and, in the fourth column, we depict results using the number of floating point instructions executed. In the fifth column we show results obtained using Number of Processes in MPI Collective Calls as metric. Finally, in the sixth column, we show some results taking the Number of Instructions Performed.

As we can see in tables 3.6 and 3.7 the best results, understanding best result as a correct structure detection, are achieved using the two metrics related to computing bursts (Sum of Duration of Computing Bursts and Number of Processes Computing). Metrics such as the Number of Processes in a MPI Point to Point call and Number of Flying Communications also provide good results in general. On the other hand, metrics such as Number of Instructions Performed, IPC and Number of Floating Point Instructions Executed provide poorer results. The justification of these differences is that the regularity of the first set of metrics highlights the iterative structure of executions of applications. Metrics related to hardware counters are subjugated to a lot of irregular oscillations which can cloud the iterative structure. The metric generated from the collective call is also very regular, but the problems is that sometimes there are no collective calls, or they do not have iterative behavior because they only initialize data. Finally, it is important to state that the results are sound, that is, if the system successfully detects the value of the main periodicity of an execution using different metrics as input, the values obtained are very close.

Application	Total Trace Size	Level 1 Size	Level 2 Size
Liso	2.02 GB	64 MB	6 MB
Idris	2.7 GB	250 MB	25 MB
Gadget	2.7 GB	53 MB	
Linpack	6.7 GB	46 MB	

Table 3.8: **Sizes of all the representative traces of each level.**

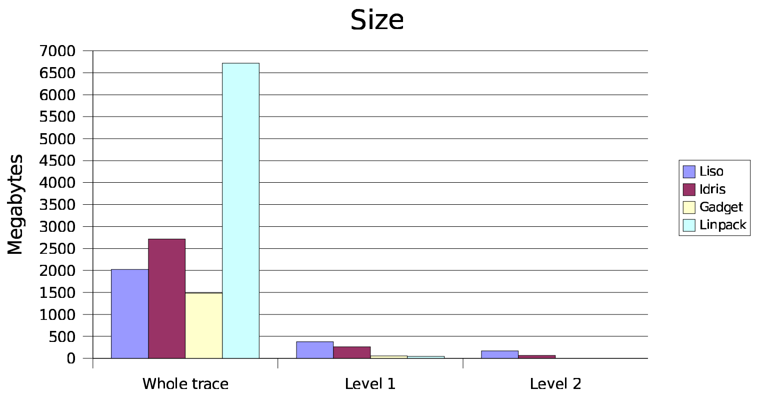


Figure 3.10: **Sum of the sizes of all the representative traces of each level.**

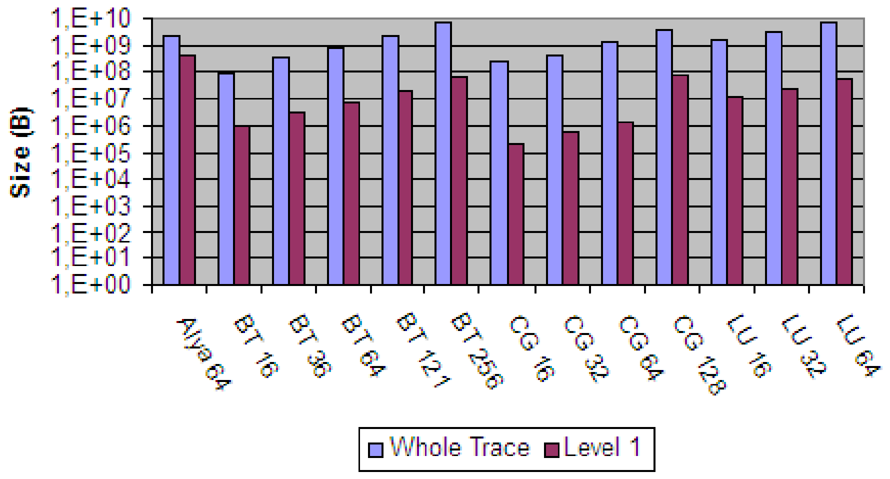


Figure 3.11: **Representation, in logarithmic scale, of size reductions of the applications depicted in tables 3.6 and 3.7. Results are depicted in Bytes. The metric used to perform the analysis is the Sum of Durations of Computing Bursts.**

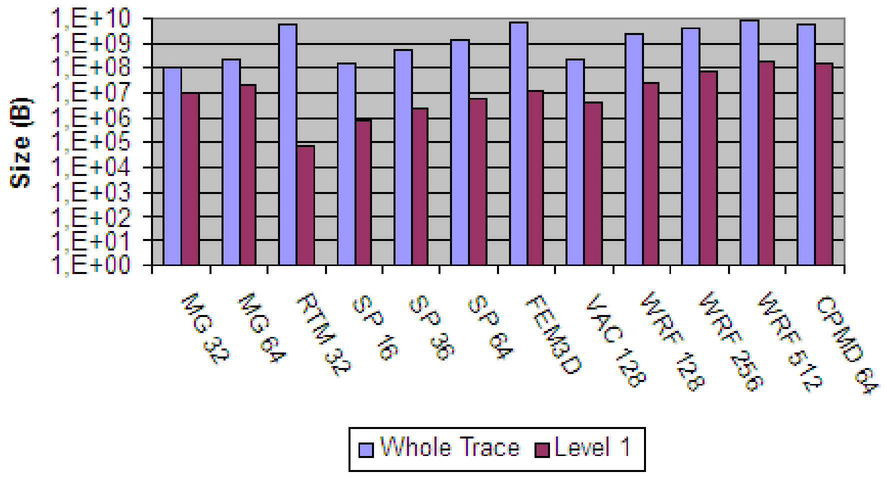


Figure 3.12: **Representation, in logarithmic scale, of size reductions of the applications depicted in tables 3.6 and 3.7. Results are depicted in Bytes. The metric used to perform the analysis is the Sum of Durations of Computing Bursts.**

3.7.3 Evaluation of Tracefile Size Reductions

This section evaluates the tracefile size reductions achieved using the automatic methodology explained in this chapter. First, we provide data about the applications used in section 3.7.1. In this data we see that strong reductions are achieved when we pass from the original tracefile to the first level chops. The hierarchical search reduces the size even more, but this last reduction is weaker than the first.

In table 3.8 we show the average size of the chops of the original trace. As observed in section 3.6.1, every time the system finds a periodic region, it selects one of the periods of that region and cuts the tracefile to provide the analyst with representative chops of the original tracefile. The sizes shown in table 3.8 are, first, the size of the total tracefile and, second, the average size of the periods of the first level. For example, the Liso tracefile has 6 periodic regions, one of these regions in every non-flushing zones. If we take one period of every periodic region and then we cut the tracefile, we will obtain 6 small tracefiles. The average size is the value we show. Finally, the third column is the average size of the second level periods. Note the large reduction in the amount of data to study.

In figure 3.10 we represent, first, the size of the whole trace. Next, we show the total sum of the sizes of the first and second level chops. Obviously, if there is only one periodic region, the value shown in figure 3.10 is the same as the value represented in table 3.8. The first level of Idris application is an example. The most important

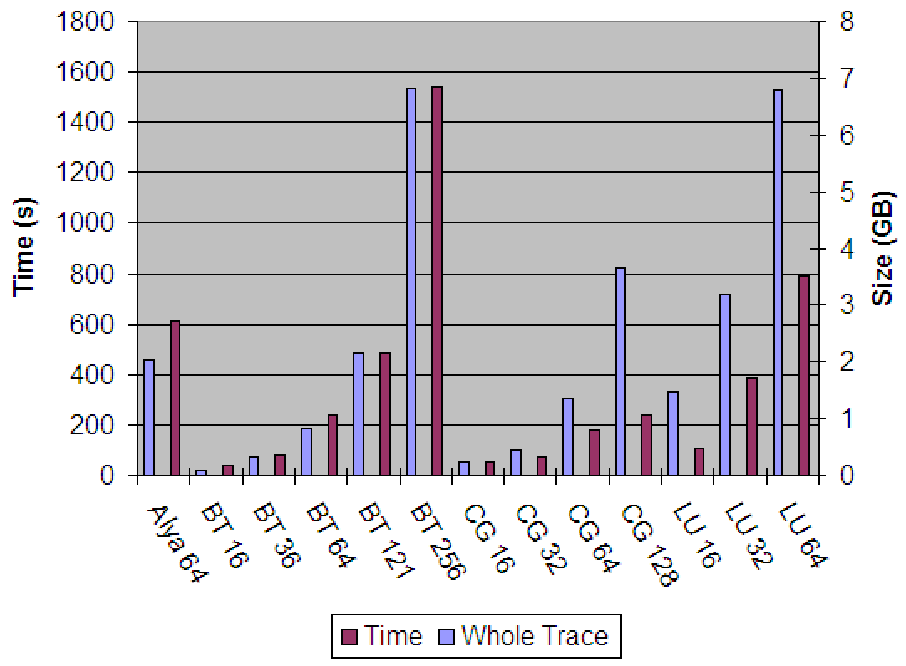


Figure 3.13: **Scalability of the system**

thing, however, is that the global behavior of the applications, with the exception of the flushing and initialization regions, is contained in Level 1 tracefiles. We have reduced notably, from GB to a few MB, the amount of data to be analyzed in order to study the performance of the applications.

We also provide the size reductions of the applications used in section 3.7.2 as shown in tables 3.6 and 3.7. Remember that these applications were used to check the behavior of several metrics with respect to spectral analysis. Strong reductions on them are also achieved by using the metric generated from durations of computing bursts.

In figures 3.11 and 3.12 we show the results in terms of the size of the tracefiles. For each execution of each application on different number of processors, we show the size of the whole tracefile and the size of the reduced one. This reduced trace contains two iterations. In these cases, the reduced trace is derived from the structure found within the whole execution of the application, that is, Level 1 structure. As we can see, we achieve remarkable reductions in all the cases. The results shown have been obtained using the Sum of Duration of Computing Bursts as input metric.

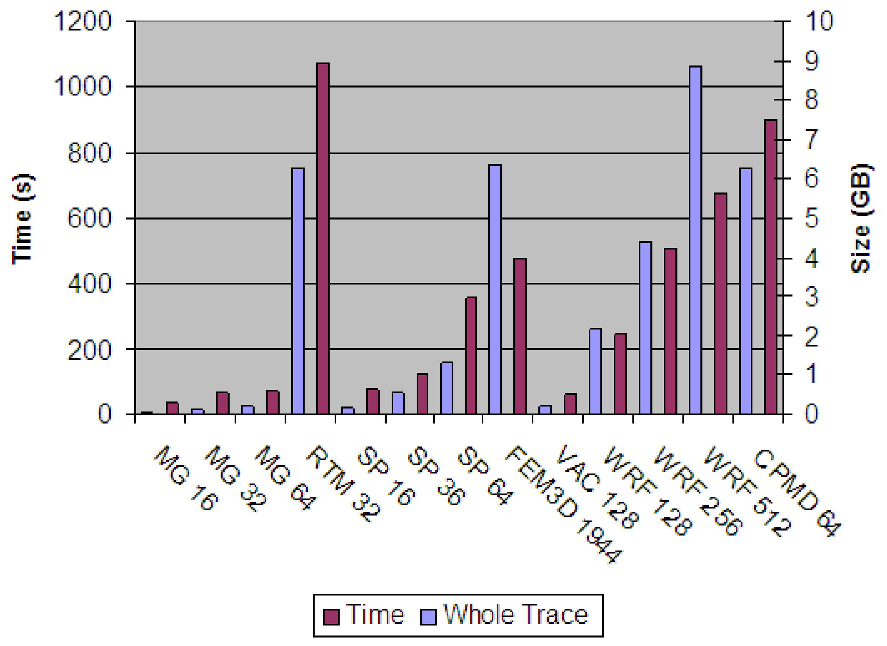


Figure 3.14: **Scalability of the system**

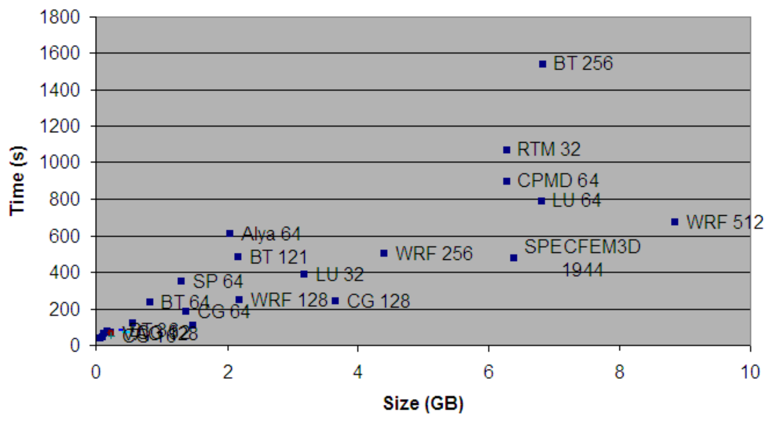


Figure 3.15: **Scalability of the system**

3.7.4 Scalability of the Analysis

As has been pointed out in this chapter, the analysis we apply to executions of applications consists mainly in applying signal processing algorithms. It is widely reported that these algorithms have very low computational complexity. Therefore, our automatic methodology also has low complexity. In order to prove this statement, in this section we perform a detailed study of the scalability of our analysis with respect to input data size.

In figures 3.13 and 3.14 we depict, for each application, the input data size and the execution time needed to perform the reduction of size. As we can see, the relationship between these two magnitudes is almost linear. This fact becomes clear looking to the results of BT or LU applications. As we increased the number of processors where we executed these applications, the size of the generated data is increased and the reduction time is also increased. The relationship between the input data size and reduction time is clearly linear.

In figure 3.15 we depict another view of the scalability of the system. In the x-axis, we put the size of the tracefiles. In the y-axis, we put the time needed by the automatic system to perform the reductions. Each point represents an execution of the automatic system. Each point is labeled with the name of the application tracefile used as input data. The same conclusions indicated above can be extracted from this figure.

It should be noted that the input data size is higher in the case of WRF executed on 512 processors than in the case of RTM executed on 32 processors. However, the required reduction time is higher in the case of RTM 32 than in the case of WRF 512. The explanation for this is that data extracted from WRF execution on 512 processors has more perturbed regions (flushing, clogged system) than RTM. Since the analysis is applied on non-perturbed regions, the study of RTM requires the analysis of larger data sets and, therefore, the required time to perform the reduction is higher. The same can be said comparing WRF 512 against CPMD 64. In general, input data sets with very few perturbed regions require more analysis time, but they provide more possibilities of finding significant structure within them.

It is important to state that the most remarkable size reductions are the largest reduction processes. Two clear examples of this fact are, first, RTM 32 and CPMD 64. On both cases, we achieve significant reductions, ruling out several Gigabytes of redundant or damaged data but we need more than 15 minutes to perform the reductions. Even these largest reduction processes can be performed in a few minutes.

Chapter 4

Speedup Analysis of MPI Applications

The intricacy of high performance computing applications has been growing very fast in the last years. Only skilled analysts are able to determine the factors that are undermining the performance of up-to-date applications. Analyst time is a very expensive resource and, for that reason, a strong effort to develop automatic performance analysis methodologies has been made by the scientific community. In this chapter, we propose a methodology that is able to automatically detect the main performance problems of applications. This methodology is based on, first, a size reduction of the performance data obtained from the executions and, second, an analytical model obtained from this performance data which fits the speedup of the applications in terms of several parameters related to several performance issues. The chapter also shows results obtained from real up-to-date applications and validates the conclusions automatically derived from the methodology.

4.1 Introduction

In the last years, supercomputing has consolidated as a fundamental tool for do research in many topics (computational biology, mechanics engineering, meteorology, ...). A wide set of scientific applications has been developed to give answers to questions arising in scientific research on those topics. Improving application performance is critical for scientific progress.

Performance analysis of parallel applications based on event tracing is a well accepted technique since it provides a very detailed study of the variations on space (set of processes) and time that do appear in parallel program executions and could affect notably the performance of the applications. A complete time-stamped sequence of events is analyzed afterwards with the help of visualization tools like Vampir [62] or

Paraver [52], which allow a fine-grained study of execution behavior. However, these tools become almost useless when large amounts of data, generated from executions of massive parallel applications, have to be analyzed. In this context, the process of tracing an application becomes a tedious and expensive task because the analyst is forced to apply techniques to reduce the total size of the tracefile. Once the reduced tracefile is generated, the analyst can start the performance analysis without any general guidelines of the execution of the application. This is also a hard task that can only be carried out by a specialist. The aim of our approach is to provide solutions to these problems automatizing the process of performance analysis.

We propose a system able to detect the main factors that are undermining the application's scalability. In chapter 3, we have already presented the first step of this detection that is based on a previous analysis of the application's execution phases and on an identification of the most relevant regions of the tracefile. These regions summarize the application's behavior during the execution. The system generates subtraces from the original tracefile. In this chapter we go one step further, by proposing a methodology that applies an analytical model to several tracefiles, each obtained varying the number of processors. This makes possible the generation of a scalability model of the application which explains the execution's evolution on an increasing number of processors. The combination of the automatic structure extraction of the tracefile and the automatic fitting of an analytical model of the speedup automatically and quickly provides an accurate performance analysis of the application.

The analytical model is also used as a basis for a prediction framework. This framework provides an estimation of the main performance undermining factor of the application if it is executed on a high number of processors. It also provides an estimation of the speedup achieved by the application. These predictions are useful to avoid real executions on high number of processors which are very expensive and, very often, they do not provide significant performance improvements.

The chapter is organized as follows: First an overview of the related work in section 4.2. After, the automatic speedup analysis is shown in section 4.3. An evaluation of the methodology is shown in section 4.4.

4.2 Related Work

There are other approaches to model the behavior of applications from some point of view. In [20], the authors explain a parallel machine model that highlights the critical technology trends underlying parallel computers. It is intended to serve as a basis for developing fast and portable parallel algorithms and to provide guidelines to machine designers. Such a model maintains a balance between detail and simplicity in order to reveal important bottlenecks without making analysis of interesting problems intractable. The model uses four factors. They model the computing bandwidth, the communication bandwidth, the communication delay, and the efficiency of coupling communication and computation. According to the authors, parallel programs typically adapt to parallel machines configurations in terms of these factors.

In [87] a modeling of an application's use of interconnect network and capturing factors related to scalability is shown. The method is based on a relaxed task graph model, a queuing model, and a memory hierarchy model. The relaxed task graph is a compact representation of communicating processes of an application mapped onto the target machine. Simultaneous accesses to the resources of a multi-processor node are modeled by a queuing network. The execution time of the application is computed by an evaluation algorithm.

In [32], the authors discuss issues related to the accuracy of performance prediction methods for MPI applications. They present the results of two sets of experiments to quantify the effect of the instrumentation overhead and variance in the accuracy of Dimemas. The results show that this performance prediction tool can be used with a high level of confidence as the effect of instrumentation overhead on the predicted performance is minimal. The authors show that it is possible to carry out instrumentation runs in highly loaded multi-user environments and still be able to accurately analyze the performance of the application as if it had run alone.

In [89, 11], a performance modeling methodology that is faster than the traditional cycle-accurate simulation and more sophisticated than the performance estimation based on system peak-performance metrics is shown. It combines machine signatures, that is, characterizations of the rates at which a machine can (or is projected to) carry out fundamental operations and application profiles, that is, detailed summaries of the fundamental operations to be carried out by the application independent of any particular machine. Algebraic mappings of the application profiles on to the machine signatures are used to arrive at a performance prediction. These mappings are called convolutions.

In [34] there is another approach. It uses high-fidelity simulation, to observe component and system characteristics (e.g. performance and power) in order to make vital design decisions. To reduce the simulation time, the system supports large-scale system simulation by using models to capture the behavior of only the performance-critical component. To achieve a balance of accuracy and simulation time, the authors provide a methodology and associated toolset to evaluate numerous architectural options. This approach allows users to make system design decisions based on quantifiable demands of their key applications rather than using manual analysis which can be error prone and impractical for large systems.

In [33], there is a comparison of two models used for the analysis, design and prediction of asynchronous message passing programs. The authors compare the accuracy of the models focused on the prediction of message passing programs based on pairwise synchronization against the predicting models focused on barrier synchronization. They consider three test cases to evaluate the models and find that the accuracy of the prediction models of these two message passing paradigms are equivalent. On the other hand, the authors see a better performance in message passing programs based on barrier synchronization.

In [46], analytical models are derived from the examination of the key characteristics of an application. These models are based in computational and communication performances of an individual system and can be used to predict the performance of an application prior to system availability. Two applications are considered to evaluate the methodology: an adaptive mesh refinement code on structured meshes, and an Sn transport code on unstructured meshes. The interesting point in this work is see how the models are used to explore the achievable performance on hypothesized future systems with increased peak computation and communication performance.

In [105], a method approaching cross-platform performance translation based on relative performance between two platforms is shown. According to the authors, relative performance can be observed without running a parallel application in full. We show that it suffices to observe very short partial executions of an application since most parallel codes are iterative and behave predictably manner after a minimal startup period The prediction approach is based in very short partial executions of an application. The authors claim that prediction derived from partial executions can yield high accuracy at a low cost.

In [74], an approach focused on models which are able to describe the performance of parallel applications without looking at the source code is shown. Linear models are extracted from a tracefile by fitting the outcome of a set of simulations. The simplicity of linear models allows an easy interpretation of the impact of the main performance parameters. This approach also takes into account the need for performance prediction on parallel machines which are not the machine where the execution have been done.

In [21], a method focused on address the problem of simultaneous runtime optimization of high performance computing systems is shown. In this work, the authors combine two software tools for reducing the dynamic power consumption of parallel systems. The first tool is called dynamic concurrency throttling (DCT). It adapts the level of concurrency at runtime based on execution properties. The second tool is called dynamic voltage and frequency scaling (DVFS). The combination of these two approaches has a huge search space which it is not possible to explore, particularly at runtime. To overcome this problem, this work proposes a dynamic multi-dimensional predictor which shows the impact of any thread mapping and any DCT and DVFS levels available on the parallel system. The prediction is based on a a regression model.

4.3 Parallelization Efficiency

As we have seen, our automatic system provides (for each periodic phase), first, a small subtrace which contains 2 iterations of the periodic phase of the application. This subtrace, smaller than the whole trace, allows the analyst to perform a detailed analysis of the execution of the application that cannot be made using other analysis approaches such as profiling. Second, the system gives several general indicators about the performance of the execution which are useful during the analysis. These indicators are automatically extracted from the 2-iterations subtrace and based on a model of the speedup. This section is focused on this second kind of analysis that our system performs.

4.3.1 Speedup Extraction

In order to extract the real speedup achieved by the application we will base our analysis on the results obtained from the size reduction explained on chapter 3. We compute how many times the execution of a fixed number of iterations with p processors is faster than the execution of the same number of iterations with p_0 (we take p_0 as reference) processors:

$$S_p = \frac{T_{p_0}}{T_p} \tag{4.1}$$

where T_p and T_{p_0} are the repeated pattern's period found by the algorithm under p and p_0 processors, respectively.

4.3.2 Speedup Model

Meaningful information about the speedup behavior will be very useful for the analyst in order to achieve a deep understanding of the application's execution. Since speedup's observed value does not provide enough in-depth information, we have developed an analytical approach, able to give automatically an explanation of the speedup's evolution when the number of processors is increased due to a set of indicators such as load balance or communication efficiency.

4.3.2.1 Initial Parameters

Given an application, for each execution with a different number of processors P, a set of indicators or parameters from the 2-iterations subtrace are extracted. Each parameter is defined in the period of time which contains these 2 iterations. The time span of this period is called T. The composition of these parameters defines a model of the speedup. These parameters have a clear physical meaning:

Communication Efficiency (CommEff) is defined as the maximum computing time, normalized to T, performed by a single process:

$$CommEff = \frac{Max(t_i)}{T} \tag{4.2}$$

It shows the real weight of the computation phase within the region we are studying. If the computation phase covers all the region, this number will be close to 1. Otherwise, if there is a high proportion of communications, the value will be close to 0.

Second, *Load Balance (LB)* that is defined as the ratio between the sum of computing time performed by all the processes and $P \cdot Max(t_i)$:

$$LB = \frac{\sum_i t_i}{P \cdot Max(t_i)} \tag{4.3}$$

It shows the differences between the computing times of each thread. If these values are similar, this number will be close to 1. Otherwise, if there are significant differences between each thread's computing time, the value will be far from 1.

Therefore, *CommEff* and *LB* parameters give information about parallelization efficiency. However, while *CommEff* evaluates the weight of MPI time due to actual communications, *LB* evaluates MPI time due to load imbalance. In both cases, the values of the parameters are closer to 1 for low values of MPI time. In figure 4.1, we show a representation of these parameters. In the x-axis, there is the normalized time. In the y-axis, there is the set of processes. For each process we show, first, its computing time and, second, its communication time. In this picture, we highlight the maximum of the computing times, that is, the *CommEff* factor. On the other hand, the area drawn with the darkest grey (blue in colors) is the sum of computing time performed by all the processes. *LB* is the quotient between this area and $P \cdot Max(t_i)$. The communication time (time in MPI) has been divided in two subsets: Communication time due to load imbalance and actual communication time.

The third factor is the *Average Instructions per Cycle* (IPC) which the application has achieved within the computation time. Applications may experience different IPC's when varying the number of processors, for example due to different problem

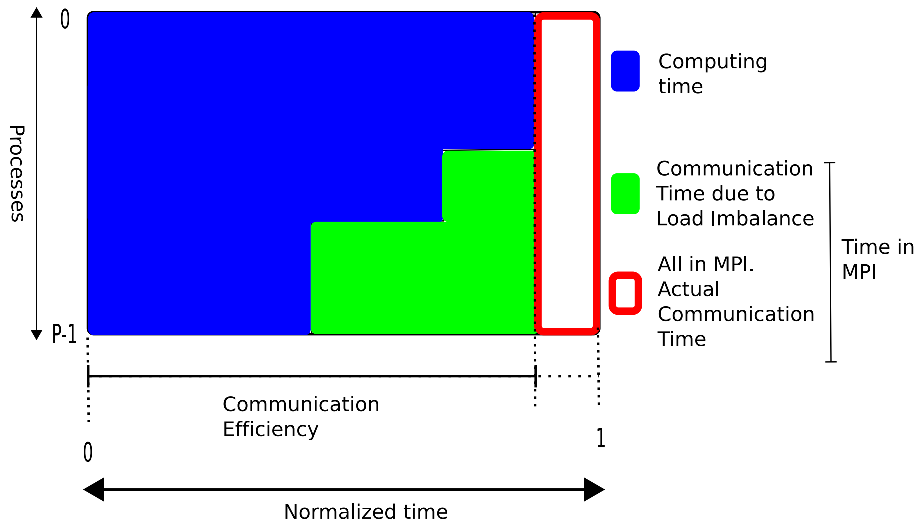


Figure 4.1: **Representation of *CommEff* and *LB* factors. In the x-axis, there is the normalized time. In the y-axis, there is the set of processes. For each process we show, first, its computing time and, second, its communication time. The communication time (time in MPI) has been divided in two subsets: Communication time due to load imbalance and actual communication time.**

size per processor and cache size. These factors could bring superlinear behavior of speedup. This circumstance is considered by our analytical speedup model, being an example of its generality. IPC is extracted from the hardware counters contained in the tracefile.

Finally, the last factor is the *Number of Instructions (#Instr)* performed by the application during the computation time. Ideally, this parameter will remain constant as we increase the number of processors of the execution. If it grows, it will mean that the application is adding load when we increase the number of processors, i. e., it replicates calculations. This replication can have an important impact on the application's performance.

4.3.2.2 Derivation of a Partial Model

In this section we derive an equation which relates the speedup of the application with the parameters explained above. The aim of this equation is to explain the evolution of the speedup in terms of parameters chosen to have real physical meaning. To detect quickly the real factor that is undermining the speedup of the application is the final goal and the motivation of this model.

Let's assume that we have executed a given application using P_0 and P processors and that we have extracted the 2-iterations subtrace in both executions. The spans of these subtraces are T and T_p. Let's consider here several issues:

The first one is that the total execution time in the case of P processors, that is, the sum of all processes execution times, is equal to $P \cdot T_p$ since each process' average execution time is T_p.

The second one is that total execution time can be decomposed in total communication time, T_{cm} and total computation time, T_{cp}. In terms of equations, these two considerations can be expressed like this:

$$\text{Total Execution Time} = P \cdot T_p = T_{cm} + T_{cp} \tag{4.4}$$

The third is that the total computation time, T_{cp}, is equal to the sum of all processes computation time:

$$T_{cp} = \sum_i t_i \tag{4.5}$$

The fourth one is that the total computation time, T_{cp}, is equal to the number of cycles performed by the application during the computation time, $\#cycles$, divided by the processor's clock frequency, α. We assume that this frequency is the same for all the processors where the parallel application is run:

$$T_{cp} = \frac{\#cycles}{\alpha} = \frac{\frac{\#cycles}{\#Instr}\#Instr}{\alpha} = \frac{\#Instr}{IPC} \cdot \frac{1}{\alpha} \tag{4.6}$$

Finally, taking into account all this considerations, we can find the following expression of the total execution time:

$$PT_p = T_{cm} + T_{cp} = \frac{T_{cp}}{\frac{T_{cp}}{T_{cm}+T_{cp}}} = \frac{\frac{\#Instr}{IPC} \cdot \frac{1}{\alpha}}{\frac{T_{cp}}{T_{cm}+T_{cp}}} = \tag{4.7}$$

$$= \frac{\frac{\#Instr}{IPC} \cdot \frac{1}{\alpha}}{\frac{T_{cp}}{P \cdot T_p}} = \frac{\frac{\#Instr}{IPC} \cdot \frac{1}{\alpha}}{\frac{\sum_i t_i}{P \cdot Max(t_i)} \frac{Max(t_i)}{T_p}} \tag{4.8}$$

In section 4.3.2.1, we have defined ***CommEff*** as $\frac{Max(t_i)}{T_p}$. On the other hand, we have that the quotient $\frac{\sum_i t_i}{P \cdot Max(t_i)}$ is ***LB***. Finally, we obtain:

$$PT_P = \frac{\#Instr}{LB \cdot CommEff \cdot \alpha \cdot IPC} \tag{4.9}$$

And putting this expression into the speedup definition in 4.1:

$$S_p = \frac{T_{p0}}{T_p} = \frac{P}{P_0} \frac{CommEff}{CommEff_0} \frac{LB}{LB_0} \frac{IPC}{IPC_0} \frac{\#Instr_0}{\#Instr} \frac{\alpha}{\alpha} = \tag{4.10}$$

$$= \frac{P}{P_0} \frac{CommEff}{CommEff_0} \frac{LB}{LB_0} \frac{IPC}{IPC_0} \frac{\#Instr_0}{\#Instr} \tag{4.11}$$

4.3.2.3 Derivation of the Final Model

In the previous section, we consider that $CommEff$ parameter shows the real weight of the computation phase within the region we are studying. However, sometimes communications overlap computations and sometimes not. It is useful to separate these two effects, that is, to decompose $CommEff$ parameter in two parameters. The first one measures the weight of overlapped communications and the second one measures the weight of the communications that are not overlapped with computations. The first parameter is called Micro Load Balance (μLB) and the second one Real Communication Efficiency ($RealCommEff$).

To determine the value of these two parameters, we perform a simulation of the 2-iterations subtraces with ideal communications, that is, $Latency = 0$, $Bandwidth = \infty$ and $\#links = \infty$. Performing this simulation, the time spent on those regions of the subtrace where only communications are performed is not considered, since those communications are instantaneous. Simulating the 2-iterations subtrace, we obtain a simulation time which is an estimation of the execution time under ideal communications, T_{ideal} which value is less or equal to the 2-iterations execution time, T. To carry out the simulations, we use Dimemas [31], a simulator of message-passing applications on a configurable parallel platform. The simulation of the small 2-iterations subtraces requires a few seconds.

Parameter $RealCommEff$ is defined as $\frac{T_{ideal}}{T}$. It computes the relative time when at least one of the processes is computing. For the reasons we explained in the above paragraph, $0 \leq RealCommEff \leq 1$. μLB is defined as $\frac{Max(t_i)}{T_{ideal}}$ where t_i is the computing time of the process i. Now we have:

$$\mu LB \cdot RealCommEff = \frac{Max(t_i)}{T} = CommEff \tag{4.12}$$

Combining formulas 4.12 and 4.11, we easily obtain the following:

$$S_p = \frac{P}{P_0} \frac{RealCommEff}{RealCommEff_0} \frac{\mu LB}{\mu LB_0} \frac{LB}{LB_0} \frac{IPC}{IPC_0} \frac{\#Instr_0}{\#Instr} \tag{4.13}$$

Summarizing, we have obtained an analytical model that shows the impact of the non-overlapped communications ($RealCommEff$), of the communications overlapped with computations (μLB and LB) and of the computations (IPC and $\#Instr$) over the speedup. It provides a first diagnostic of the application that guides the analyst in the analysis of the 2-iterations subtrace. It is important to highlight that, in order to obtain $RealCommEff$ and μLB parameters, we need a complete timestamped sequence of events, i. e., a tracefile of the execution we are studying. This is due to the fact that we have to make a distinction between the communications overlapped with computations and the non-overlapped ones. This distinction can not be made without a temporal sequence of events. Other techniques of analysis of applications, such us profiling, do not allow us to separate the impact of communications overlapped with computations to the impact of the non-overlapped communications since they do not provide a complete timestamped sequence of events.

The model decomposes the speedup in six multiplicative quotients. The first one is the linear speedup (ideal) and the other ones are applied to this. If the theoretical model speedup is far from the linear one, we will quickly detect the reason by looking at the quotient that is far from one, that is, the problem that is preventing an ideal speedup of the application.

The utility of the model remains on, first, the intuitive physical meaning which everyone of the factors has, second, on the fact that together are able to fit the real speedup extracted value and, finally, on the circumstance that is possible to automatically extract the parameters from the subtraces automatically generated.

4.4 Evaluation

In this section, we perform an evaluation of the analytical speedup model in order to show its usefulness. As we have said above in this chapter, the input of the speedup model are the trace chops generated using the methodology explained in chapter 3. In order to provide a general overview of our approach, we briefly explain several characteristics of the size reduction and, after, we make a detailed exposition of the results obtained using the speedup model.

4.4.1 Size Reduction

We have applied the size reduction explained in chapter 3 to the Weather Research and Forecasting (WRF) system. We have analyzed the Nonhydrostatic Mesoscale Model (NMM) [64] core and the Applied Research Weather (ARW) [1] one. We executed each core on 128, 256 and 512 processors. These executions have been made using, first, a 4 kilometers grid of the Iberian Peninsula of 401x401x35 points and, second, a 12 kilometers grid of Europe of 401x401x35 points. The executions have been performed on the MareNostrum supercomputer [57].

In figure 4.2 there is a graphical representation of the results of the analysis of the WRF-NMM execution run on 128 processors using the Iberia's grid as input. At the top there is a global visualization of the tracefile of one execution. Below, we have a graphical representation of the signal generated from the tracefile. In chapter 3 there is more information about how is this signal obtained and what kind of information contains. Below this signal, we have a zoom of the signal focused on the periodical region. At the bottom, we have a visualization of the generated subtrace, which contains 2 iterations of the periodical region.

In figure 4.3 the execution times needed for the automatic system to find the structure of the tracefile are shown jointly with the size of the original tracefiles analyzed by the system. As we can see, the execution time grows almost linearly with the size of the original tracefiles. In figure 4.4 we show the size reduction achieved in logarithmic scale. As we can see, the volume of data to study is dramatically decreased after the reduction process.

4.4.2 Results Obtained with the Performance Model

We have applied the analytical model to the set of 2-iterations subtraces obtained following the methodology explained in chapter 3. The executions have been performed on MareNostrum supercomputer.

After the application of the size reduction, the automatic system generated several subtraces (one for each execution). Each subtrace contains two iterations of the computation phase of the application. From them, the system obtains the parameters explained on section 4.3.2 and has fitted the model. These results are shown in figure 4.6. In order to scale the value of the parameter $\#instr$, we draw the ratio $\frac{\#instr_0}{\#instr}$. The value $\#instr_0$ is the number of instructions performed by the application in the execution that we take as reference to perform the speedup analysis.

The results of WRF-ARW application are shown in figures 4.5 and 4.6. First, in figure 4.5 we can see that when we use the Iberian's grid as input and using 128 processors as reference, the measured speedup is close to 1.5 executing on 256 processors and close to 2.8 executing on 512 processors. It is remarkable that the speedup extracted from the model is almost identical to the real measured speedup, which is an empirical proof that the analytical proof of the model provided above is right. On the other hand, as we can see in figure 4.6, the factor that has a largest impact on the speedup evolution is the decrement of μLB and LB parameters. These decrements are indicators of a remarkable impact of the communications overlapped with computations on application's performance.

In figures 4.5 and 4.6 we also show the results of WRF-NMM. When we use the Iberian's grid as input, the measured speedup is close to 1.8 executing with 256 processors and close to 3.2 executing with 512 processors. Again, the directly extracted

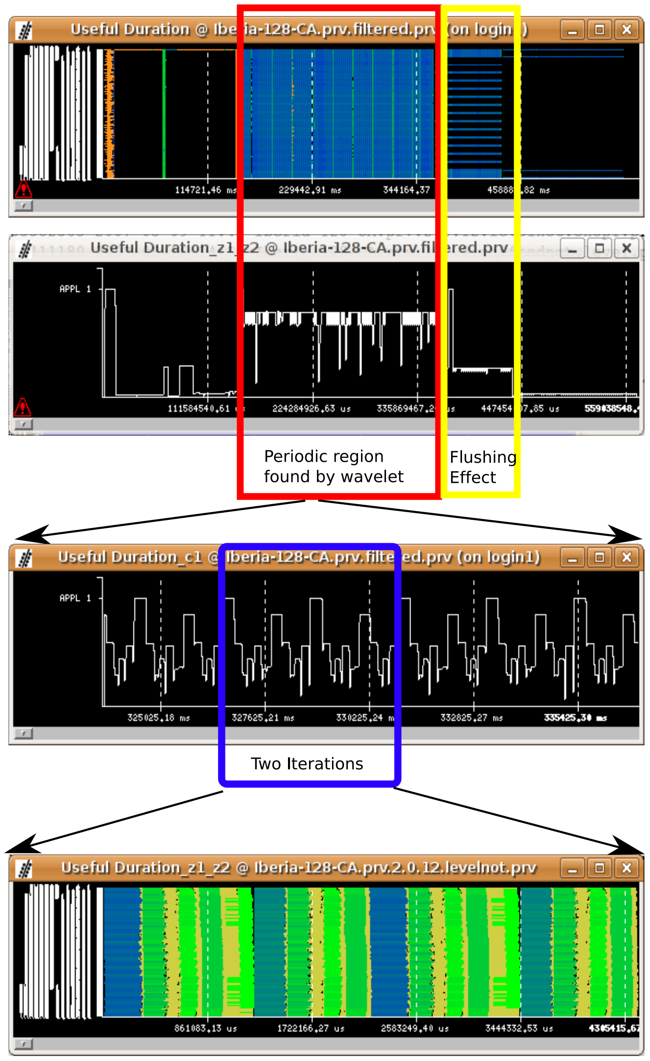

Useful Duration @ Iberia-128-CA.prv.filtered.prv (on login1)
Useful Duration_z1_z2 @ Iberia-128-CA.prv.filtered.prv
APPL 1
Periodic region
found by wavelet
Flushing
Effect
Useful Duration_c1 @ Iberia-128-CA.prv.filtered.prv (on login1)
Two Iterations
Useful Duration_z1_z2 @ Iberia-128-CA.prv.2.0.12.levelnot.prv

Figure 4.2: **Application of the size reduction. The input is a tracefile obtained from the execution of WRF-NMM application on 128 processors in MareNostrum supercomputer. First, the automatic system derives a signal from the tracefile. After that, Wavelet Analysis finds the periodic phase and, finally, a subtrace that contains two iterations is generated.**

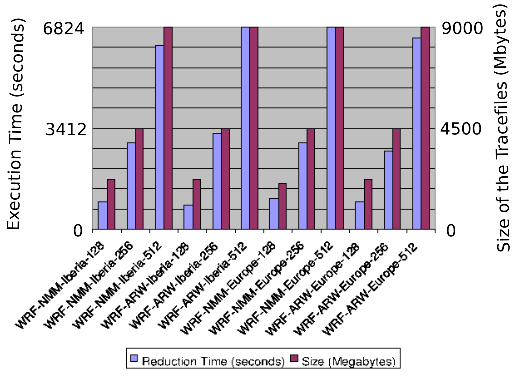


Figure 4.3: **Execution Times needed by the automatic system to perform the reduction from the whole trace to the 2-iterations subtrace. Sizes of the whole traces are also shown.**

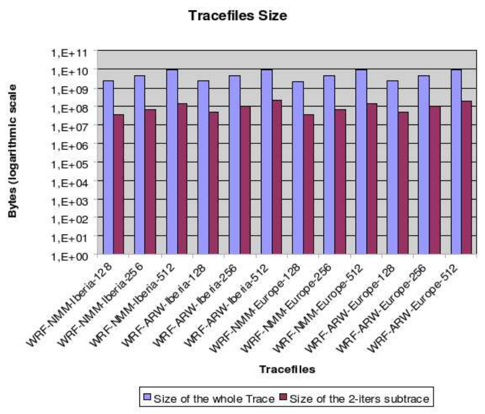


Figure 4.4: **Size of the whole trace and size of the 2-iterations subtrace generated by the automatic system.**

values of the speedup are almost equal to the speedup extracted from the model. The main problem of WRF-NMM is the increasing number of instructions executed when the number of processors is increased. This fact is derived from the decrement of $\frac{\#instr_0}{\#instr}$, that we detect in figure 4.6. For that reason, we can conclude that WRF-NMM replicates calculations. Replication of calculations means that, due to an inefficient domain decomposition, several regions of the original data are mapped to more than one thread and, therefore, are processed more than one time. This replication of computations undermines the scalability of the application.

Looking at figure 4.5, we state that the speedup of the WRF-NMM code is higher than the WRF-ARW. The model is able to automatically find the explanation of this fact: In figure 4.6 we see that the only factor that is undermining the scalability in WRF-NMM is the increase of the number of instructions executed, since the rest of the factors remain unchanged as we increase the number of threads. However, in the case of WRF-ARW, we find not only an increase of the number of instructions executed, but also a decrement of μLB and LB. It means that the communications overlapped with computations scale well in WRF-NMM but do not scale well in WRF-ARW. In figure 4.7 there is an empirical proof of this different communication behavior. A Paraver visualization of communications overlapped with computations, that is, a visualization of the simulation under ideal communications ($Latency = 0$, $Bandwidth = \infty$ and $\#links = \infty$) is shown. In this kind of simulations, communications which are not overlapped with computations are performed instantaneously. For that reason, if we visualize this simulation, we only will see overlapped communications. In this visualization, the horizontal axis represents the time and the vertical axis the different processes. Black color means that a given process in a given time instant of time is not communicating. On the other hand, light colored points (green when printing or visualizing in color) represent communications. The applications are WRF-ARW executed on 128 and 512 (using an Europe's Grid as input) and WRF-NMM executed on 128 and 512 processors (using an Iberia's Grid). These two applications show different behavior in the overlapped communications' pattern. The relative weight of the communications increases notably in WRF-ARW-Iberia. This increase is almost zero in WRF-NMM-Europe. This fact is captured by the analytical model in terms of the evolution of μLB and LB parameters. This is a practical case that demonstrates the usefulness of this automatic methodology: It is able to automatically detect performance problems without the need of any kind of visualization or performance analysis made by analysts.

We have also applied the size reduction and extracted the analytical model to the CPMD [18] code, which is particularly designed for molecular dynamics. According to the documentation of CPMD code [19], its main parallelization problem in distributed memory is load-balancing. This documentation explains that a partial solution to this problem is a technique which allows to scale the application in modern supercomputers by avoiding load imbalance. It is applied when executing CPMD with the parameter *Taskgroups*. We have executed CPMD using this parameter and without using it. Results are shown in figure 4.6. They correctly show the impact of the technique which avoids load imbalance. In the two executions that we have made without the technique, the parameters LB and μLB are smaller than the same parameters extracted from the executions that use the technique. This is another empirical proof of the usefulness and robustness of our model: It is able to automatically detect non-trivial execution characteristics without visualization or another kind of human activity.

It is interesting to state that, when CPMD uses *Tasksgroups* parameter, the load imbalance and the replication of calculations have almost disappeared. However, on the other hand, the $RealCommEff$ parameter is higher in the execution that does not use *Tasksgroups* parameter. It means that the application, when it uses the technique designed to avoid load imbalance, performs more non-overlapped communications to balance correctly the load. Finally, the speedups achieved are the same on both executions. The speedups directly extracted are, again, almost equal to the speedups extracted from the model.

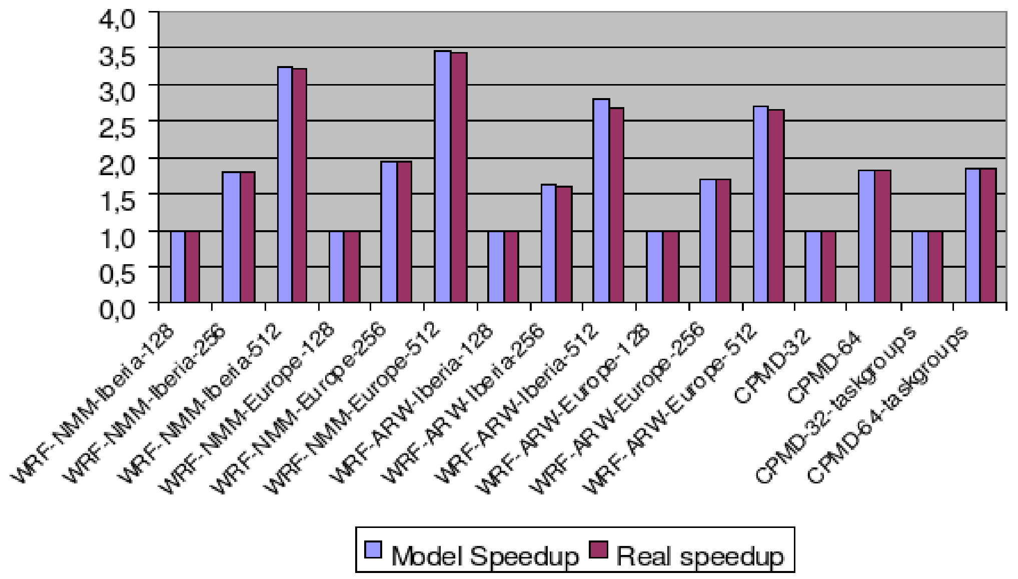


Figure 4.5: **Values of the speedups obtained from the executions of WRF-NMM and WRF-ARW on 128, 256 and 512 processors. Input data are grids of Iberia and Europe. Each grid has 401 x 401 x 35 points. The figure also shows these values for CPMD application executed on 32 and 64 processors. CPMD has also been executed on 32 and 64 processors using a technique which avoids load imbalance (Taskgroups). The figure also shows the speedups derived from the model. In all the cases, the model fits correctly the values.**

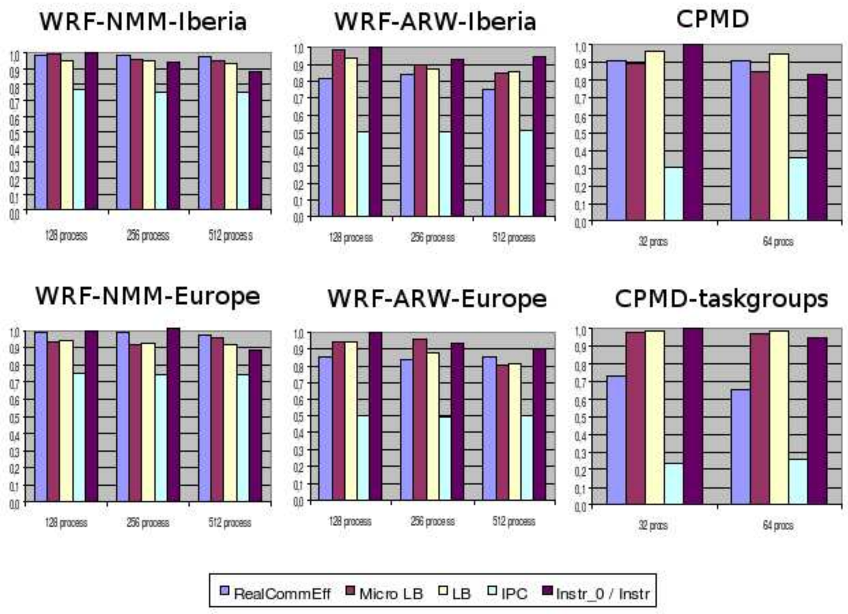


Figure 4.6: **Values of the parameters of the model obtained from the executions of WRF-NMM and WRF-ARW on 128, 256 and 512 processors. Input data are grids of Iberia and Europe. Each grid has 401 x 401 x 35 points. The figure also shows these values for CPMD application executed on 32 and 64 processors. CPMD has also been executed on 32 and 64 processors using a technique which avoids load imbalance (Taskgroups).**

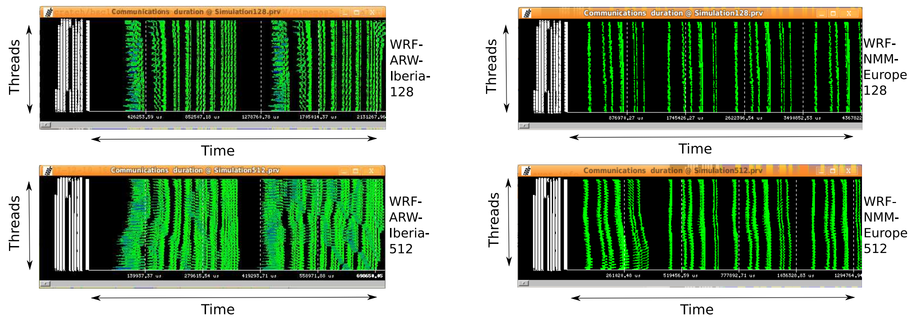


Figure 4.7: **Representation of communications overlapped with computations, that is, a visualization of a simulation under ideal communications. The applications are WRF-ARW executed on 128 and 512 (using an Europe's Grid as input) and WRF-NMM executed on 128 and 512 processors (using an Iberia's Grid).**

4.5 Prediction

This section is focused on the prediction of, first, applications speedup and, second, the main performance undermining factor as we increase the number of processors where the executions are performed. We take executions on a few processes as reference and we predict the execution of the application on a larger number of processes. We assume that processor architecture and interconnection network remain unchanged.

The prediction scheme is based on several executions of the application on a few processors. For each execution, a set of parameters is extracted. These sets are characterizations of the executions. The prediction is based on them. For each parameter, we perform a log-linear fitting [39], capturing its evolution as we increase the number of processors and obtaining a function of these parameter against the number of processors. The reason why we use log-linear fitting instead of linear one to predict the parameters is because this kind of fitting provides better numerical results in terms of prediction's accuracy. For example, if we execute a given application on 16, 32 and 64 processors, we extract the *Average Instructions per Cycle* (IPC) for each execution and we fit a log-linear model to the data, we will obtain a function of the IPC against the number of processors. From this function, we will predict the IPC of an hypothetic execution on 1024 processors.

The last step of the prediction scheme is the derivation of the speedup from the analytical model explained above. The set of predicted parameters is used as input. In the next sections, there are numerical evaluations of the prediction errors applying this methodology to several applications. We also compare real values of the model parameters with respect to predictions of these values. In figure 4.8 there is a graphical representation of the scheme of the prediction of executions increasing the number of processors.

This scheme has several strengths and also several weaknesses. Essentially, its basic strength is that it provides fast predictions without the need of expensive simulations or long executions. On the other hand, its basic weakness is that, since this method is based on the information extracted from executions performed on very few processors, it is unable to predict performance undermining factors that only appear on executions carried out on many processors.

4.5.1 Prediction of the Execution Time

In this section, we use real executions on MareNostrum (MG on 16, 32 and 64 processors and, on the other hand, BT and SP on 16, 36 and 64 processors) as reference

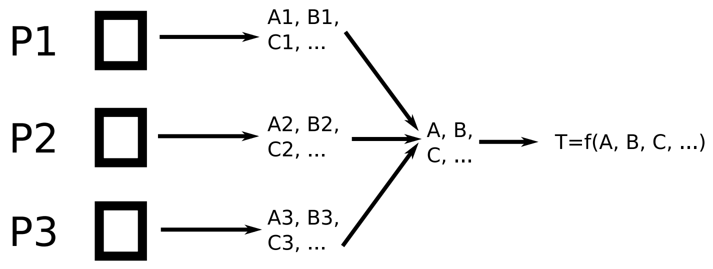


Figure 4.8: **Graphical representation of the scheme of the prediction increasing the number of processors: First, we perform executions of the application on P1, P2 and P3 processors. For each execution, we extract a set of parameters. From these sets, we predict a new set of parameters. This new set belongs to a predicted execution of the application on P4 processors. Finally, we extract the execution time from the analytical model applied to the predicted parameters.**

to predict executions on MareNostrum increasing the number of processors. We predict the elapsed time of the executions on 121 (128 in the case of MG), 256 and 1024 processors. To do this, we follow the methodology explained in section 4.5, that is, we extract real values of model parameters from a few of real executions, we perform log-linear fittings, one fitting for each parameter, we obtain the predicted parameters and, finally, we derive the predicted speedup. After, we extract the relative error of the predicted speedup, comparing its value with respect to the real speedup value, which has been obtained from real executions of the applications on 128, 256 and 1024 processors.

Obviously, there are many numerical techniques to perform this prediction. In this work, we compare the results provided by the analytical model with the values provided by several classical prediction techniques: linear, quadratic and log-linear extrapolations. Linear extrapolation consists in to perform a linear fitting on real speedups and, after that, extrapolate the values increasing the number of processors. This scheme is not only used on linear fitting but also on quadratic and log-linear ones. In figures 4.9, 4.11 and 4.13 we have drawn predicted speedups provided by the 4 extrapolation techniques we have used (analytical model and linear, quadratic and log-linear fittings). The quadratic fitting is always very far from the real speedup value in the case of 1024 processors. On the other hand, log-linear fitting is also far from the real values. Only the predictions based on linear fitting and analytical model are reasonably near from the real values. However, the prediction based on analytical model is always more accurate than the based on linear fitting.

Relative errors of the predictions are shown in figures 4.10, 4.12 and 4.14. On the one hand, we have drawn the value of the relative error of the prediction based on analytical model. On the other hand, we have drawn the relative error of the predic-

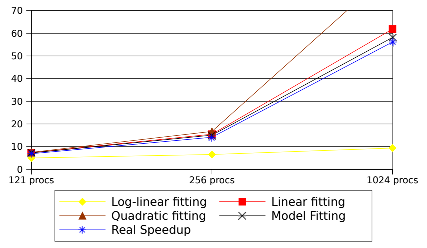


Figure 4.9: **Predicted NAS-SP speedups using several techniques. Real speedup on MareNostrum is also shown in the picture.**

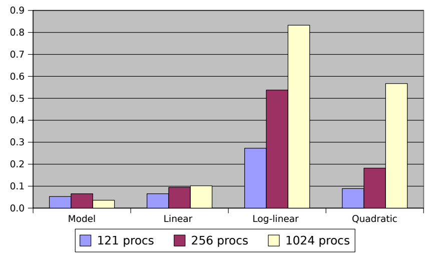


Figure 4.10: **Relative error predicting NAS-SP speedup on MareNostrum using several techniques. Results are shown for 121, 256 and 1024 processors.**

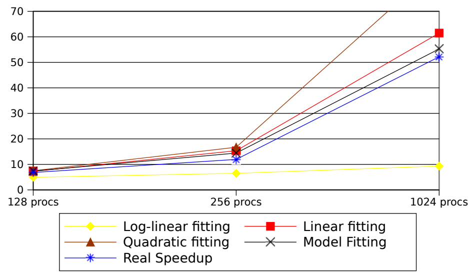


Figure 4.11: **Predicted NAS-MG speedups using several techniques. Real execution on MareNostrum speedup is also shown.**

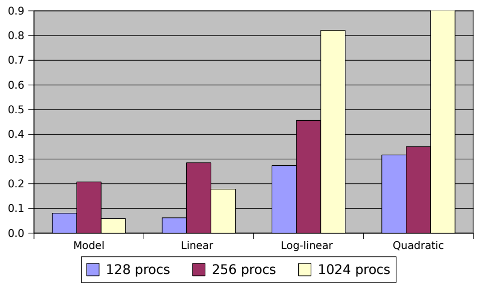


Figure 4.12: **Relative error predicting NAS-MG speedup on MareNostrum using several techniques. Results are shown for 128, 256 and 1024 processors.**

tions using the classical methodologies: Remember that the first one is a linear fitting of the speedup on 128, 256 and 1024 processors taking the execution time on 16, 32, 64 as references, the second is a log-linear fitting of the speedup and the third is a quadratic one. As we can see, the minimum relative error is always achieved using our technique. It means that we have improved the classical approach of performing a simple fitting from the data.

4.5.2 Prediction of the Performance Undermining Factors

In the last section we have shown that our methodology has successfully predicted the speedup of the applications we are studying. Remember that, in order to predict the elapsed time, we obtain estimated values of the performance factors using log-linear fittings. These values are not only useful in order to predict the speedup, but also to predict the performance evolution of the application.

In section 4.4 we have shown that, using the analytical model explained there, it is

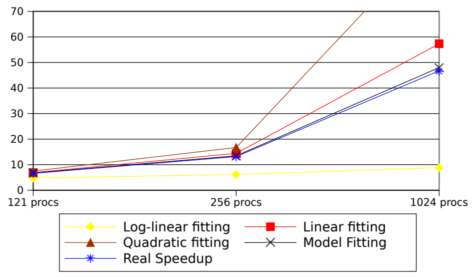


Figure 4.13: **Predicted NAS-BT speedups using several techniques. Real speedup on MareNostrum is also shown.**

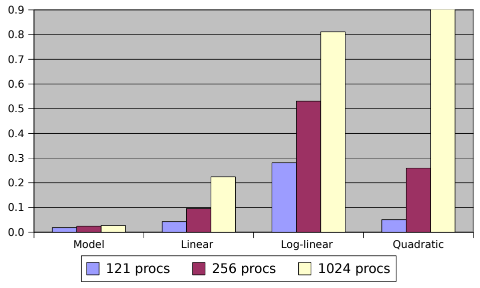


Figure 4.14: **Relative error predicting NAS-BT speedup on MareNostrum using several techniques. Results are shown for 121, 256 and 1024 processors.**

possible to detect automatically the performance undermining factor. In this section we are focused on show how is it possible to to predict this performance undermining factor. The information obtained it is not trivial because, thanks to this information, we obtain a first understanding of the reasons of the performance problems of the application.

In figure 4.15 we can see the real parameters of the analytical model extracted from real executions of NAS-SP application. It is clear that the two parameters which are undermining the performance of the application are LB and μLB. The decreases of LB and μLB parameters imply that the differences between the computing times of each process increase as we increase the number of processors where the application is executed. These are the two factors which mainly undermine the speedup of the applications. In figure 4.16 we can see the predicted values of the model using a log-linear fitting and taking the real executions on 16, 36 and 64 as input. The predicted values are quite similar to the real ones. It means that, using our prediction scheme, we are able to predict not only the speedup but also the factors which undermine this speedup.

In figures 4.17 and 4.18 we have the results for NAS-MG application. In this case, the real values show decreases of LB and μLB parameters. Again, the predictions of the model fit correctly the real values and, therefore, they predict successfully the main performance undermining factors of NAS-MG application.

In figure 4.19 we can see the values extracted from real executions of NAS-BT applications. The performance undermining factors are $RealCommEff$ and μLB. The decrease of $RealCommEff$ implies that the weight of communications non-overlapped with computations increases as we increase the number of processors. Predictions shown in 4.20 have the same kind of behavior, that is, decreases of $RealCommEff$ and μLB factors.

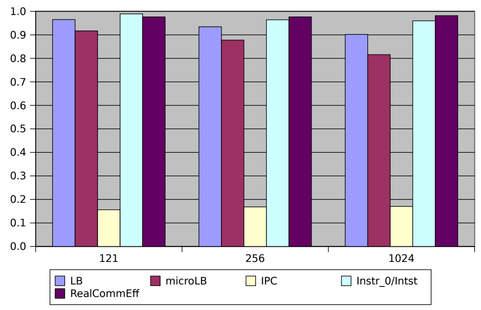


Figure 4.15: **NAS-SP. Parameters of the Analytical Model explained above. They have been extracted from real executions on MareNostrum.**

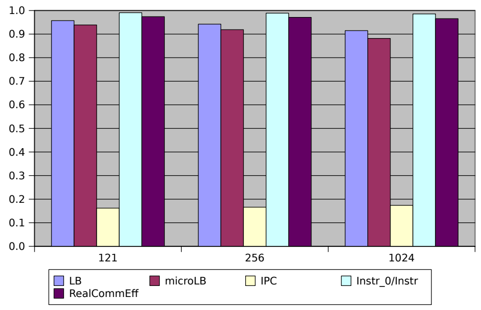


Figure 4.16: **NAS-SP. Parameters of the Analytical Model predicted from real executions on 16, 36 and 64 processors on Marenostrum.**

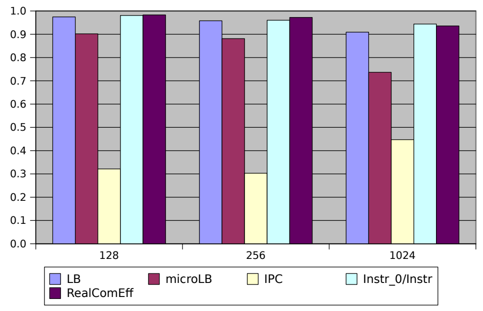


Figure 4.17: **NAS-MG. Parameters of the Analytical Model explained above. They have been extracted from real executions on MareNostrum.**

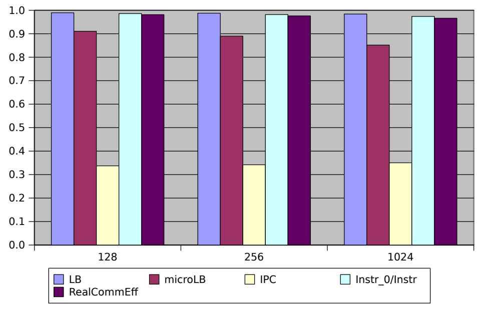


Figure 4.18: **NAS-MG. Parameters of the Analytical Model predicted from real executions on 16, 32 and 64 processors on Marenostrum.**

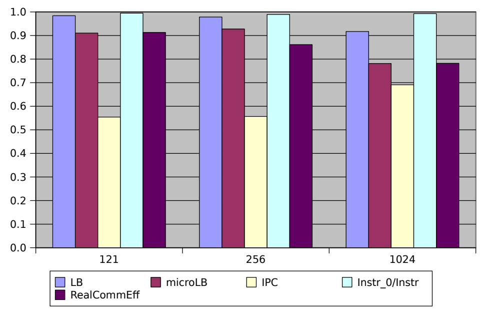


Figure 4.19: **NAS-BT. Parameters of the Analytical Model explained above. They have been extracted from real executions on MareNostrum.**

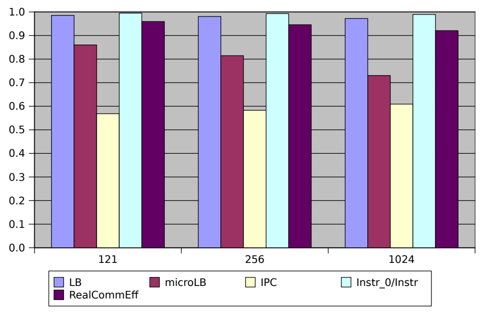


Figure 4.20: **NAS-BT. Parameters of the Analytical Model predicted from real executions on 16, 36 and 64 processors on Marenostrum.**

Chapter 5
Optimal Frequency Extraction

Several applications have been proposed as benchmarks to evaluate the adequateness of architectures and high performance computing infrastructures. The performance of these benchmarks is used to determine the strength and the weaknesses of novel designs. Therefore, the performance evaluation of benchmarks is a key factor in the process of designing new architectures. In this chapter, we propose an analysis of executions of benchmarks based on spectral analysis. This analysis provides a representative segment of the executions. This segment is useful because it provides the optimal sampling interval length of applications. This optimal length complements and improves existing techniques focused on the reduction of the application's instruction execution stream of sequential benchmarks and enables the extraction of significant performance information of parallel benchmarks without executing the whole application.

5.1 Introduction

The behavior of programs is a key issue in computer architecture and high performance computing. The development and design of novel architectures and supercomputing infrastructures is based on the performance results achieved by sets of benchmarks. Performance metrics are extracted from real executions to analyze the main characteristics of applications [4]. However, several problems arise from this situation: First, it is very expensive to extract complete performance data of executions. The reason is that it requires intensive use of computational power because many performance factors have to be taken into account and it is not always possible to extract all the required information from a single execution. Second, we have that a fine-grain performance analysis of real applications requires the generation of huge amounts of data. This data requires an intensive use of storage power and many hours of analysis time.

In this work, we show an approach able to overcome these problems. We collect performance data of applications using hardware counters. They contain values of a performance metrics during a given interval of execution. From the values of the hardware counters, we derive signals which summarize the behavior of the application. Each signal is related to a given performance metric of the application. For example, we generate a signal which summarizes unified L1 misses during the whole execution of the application.

We apply spectral analysis techniques (wavelet analysis [23] and cross-correlation function [73]) to those signals in order to detect the periodic behavior on them. Once the periodic behavior is detected, it is possible to elect a minimal region which is representative of the whole execution. This election opens a wide range of options. In the case that we are analyzing parallel benchmarks, our approach accelerates the process of performance analysis because it reduces the amount of data that has to be analyzed. The reason is that performance data is summarized by spectral analysis techniques, eliminating the redundant data and electing the minimal subset which represents the whole execution. Second, our approach reduces dramatically the computing time required to extract information about performance. The reason is that our approach detects an optimal interval length that summarizes the whole execution. Therefore, extracting performance data from intervals whose length is the optimal, we can obtain significant information of the whole execution.

In the case of sequential benchmarks, our approach improves existing tools focused on the reduction of the application's instruction execution stream. This improvement is based on the detection of the optimal sampling frequency. This point has not been addressed by previous works. In these works, the sampling frequency is an arbitrary and fixed election [83] or it is simply modified to reduce the error [103]. As we have said, the extraction of the optimal frequency improves remarkably the results of these approaches without an increase of computational load.

It is important to state that algorithms which come from the spectral analysis theory have in general a reasonably low computational complexity. Since our analysis is based on spectral analysis, it has a good behavior in terms of scalability. Therefore, the analysis can be applied in a wide range of data and results can be obtained in a reasonable span of time.

In section 5.2, we make an overview of other approaches and we discuss the interactions between them and our approach. In section 5.3 we explain how we elect the minimal representative subset of the whole execution and we show results. In sections 5.4 and 5.5 we show two important applications of spectral analysis.

5.2 Related Work

In the last years, several approaches have been made in order to detect representative and small regions of executions of applications:

The Sampling Microarchitecture Simulation (SMARTS) [103] framework applies statistical sampling to microarchitecture simulation. It provides a procedure for sampling a minimal subset of a benchmark's instruction execution stream to estimate the performance of the complete benchmark with quantifiable confidence. The sampling frequency and the length of each sampling unit are used to control the overall simulation time. SMARTS uses a statistical sampling theory to estimate the CPI error of the reduced simulation versus the complete one. If the estimated error is higher than the user specified confidence interval, it recommends a higher sampling frequency. However, its minimal representative subset is non-connected and, therefore, requires a warm-up every time we have to simulate one of the regions of the subset and it does not allow an early termination of the simulation because its minimal representative subset is spread along the execution.

Another interesting approach is MinneSPEC [50]. In this work, the authors develop new benchmarks workloads for use in simulation-base studies. The behavior of the SPEC CPU 2000 benchmark programs when executed with these small workloads is compared with the reference workloads. Our approach could be applied to executions with small workloads, reducing even more the data generated and the simulation subset.

SimPoint technique [83] chooses a number of simulation points to be the representative of the behavior of the entire program. To determine the simulation points, they first perform a profiling of the benchmark to identify the candidate simulation points and then they use clustering to elect a set that is representative of the entire program. After simulation, the results from each simulation point are weighted to compute the final simulation results. The number of simulation points and the length of each determine the overall simulation time. SimPoint determines the number of simulation points using machine learning techniques. However, the length of the simulation points is equal to the sampling interval chosen for the profiling. As we demonstrate in the following sections, the election of the sampling interval length is not a trivial issue for the posterior election of representatives. As we will see in section 5.4, if we elect a length close to the minimal representative length, the results of the clustering will improve notably.

In [17] the authors obtain multiple simulation points by randomly picking intervals of execution, and then examining how these fit to the overall execution of the program for several architecture metrics (IPC, branch and cache statistics). Our work could be used as input because it can determine the optimal sampling interval of the random analysis.

In [106], the authors perform an evaluation to determine the accuracy of the techniques we have explained above. In these evaluations, they arrive to the conclusion that SimPoint is the best technique because it has a quite accurate representative election and a fast obtaining of them. SMARTS is slightly more accurate than SimPoint but it is remarkably slower.

In [25], the authors reduce simulation time using statistical techniques: Given a benchmark program, they generate a synthetic trace from statistics of branch, cache and program. After, they perform simulations using this small synthetic trace as input. This methodology enables quick and accurate design decisions in the early stages of computer design, at the processor and system levels. Spectral analysis can be used in the early stages of the process focused to obtain synthetic traces, that is, spectral analysis can accelerate the generation of these traces.

In [101] the authors show a simulation tool infrastructure that uses statistical sampling theory to reduce the simulation effort. The goal is maintain the accuracy while the simulation time is reduced. The main problem they address is the dependency of a sample of the execution time on memory states. For example, a sample of 100 executed instructions could be meaningless if the cache and the branch predictor were full of invalid entries.

In [72] there is an approach focused on the detection of patterns within the execution on programs. The authors look for these patterns in executions of message passing programs.

In [53], the authors use a wide range of techniques derived from statistics and derive models based on piece-wise polynomial regression and neural networks to model performance of parallel applications. This work is related with our in the sense that the authors apply analytical techniques to study executions of programs. The difference is that we apply signal processing techniques.

As we have seen, many efforts have been dedicated to reduce the performance analysis effort from many points view. However, the question of determine the optimal sampling frequency and apply it in performance analysis has not been addressed. In this chapter, we handle these questions with the aim of complementing and improving the current work in this field.

5.3 Minimal Representative Subset Detection

As we explained in chapter 3, we use auto-correlation functions to detect periodicities within executions. The analysis is improved by the use of Discrete Wavelet Transform.

In this chapter, we generate signals from hardware counters extracted from real executions. The values of these counters are extracted performing a sampling during the execution. Each time that a fixed number of instructions is executed, we extract values of several hardware counters. Finally, we obtain a timestamped sequence of values of hardware counters. Each member of this sequence characterizes an interval of the benchmark's execution. Values of hardware counters of a given interval indicate several performance metrics within this interval. Signals are generated from these values and the analysis is applied to them.

The extraction of information requires the election of a minimal interval length from which we extract the values of counters. This election implies that there is maximum frequency in our studies whose value supposes a threshold which can not be trespassed, that is, there is a maximum frequency which delimits the maximum granularity which can be studied. According to Nyquist-Shannon theorem [65, 80], the sampling frequency limits the minimal period that can be detected. Of course, this frequency can be fixed to a high value that ensures the detection of significant subsets. As we can see in section 3.7, the minimal iterative periods detected are remarkably higher than the minimal sampling periods. This fact ensures that we are not losing significant information in the extraction of the values of counters.

5.3.1 Results

This section is divided in two parts: In the first one, we report the results obtained applying the methodology explained above to SPEC CPU 2000 (SPEC) Benchmarks [93]. In the second part, we study NAS Parallel Benchmarks (NPB) [3]. SPEC benchmarks are widely used in computer architecture to study the impact of hypothetical architectural changes using simulation. Simulated executions are the basis to evaluate novel ideas an approaches. NPB are used to measure performance of high performance computing machines. As we can see in this section, our methodology performs well in both sets of benchmarks. This is a proof of its generality and applicability. In the next sections, we provide errors to demonstrate the representativity of the selected execution segments.

5.3.1.1 Sequential Benchmarks

In table 5.1 and figure 5.1 we show the results obtained applying the methodology explained in chapter 3 to SPEC CPU 2000 benchmarks [93]. They have been executed on one core of a PowerPC 970MP processor at 2.3 GHz. The name of each benchmark is shown in the first column. After, we show the length of the representative subset.

It is expressed in #instructions. In the third column, we show the total length of the application. After, in the fourth, we show the average CPI value of the representative subset. Average CPI value is obtained from hardware counters extracted each 10^7 instructions executed. Fifth column contains the global average CPI value of the whole application. Finally, we show the relative error of the CPI's representative subset in the sixth column. In the seventh, eighth, and ninth columns we show the average unified L1 cache misses every 10^7 instructions in the case of the representative subset, the same value in the case of the whole application, and the relative error of the representative subsets.

The spectral analysis (wavelet transform and cross-correlation analysis) has been made taking the CPI signal as input. For this reason, it is logical that the relative error calculated using CPI is, in general, lesser than the error calculated using L1 misses. As we can see in table 5.1, the errors are in general very low, except in a few of applications, for example galgel. The reason is that these applications do not have a strong periodic behavior and it is not possible to find representative subsets which fit exactly the global behavior of the application. On the other hand, applications with a strong periodic behavior, for example applu, have representative subsets which fit almost exactly the global behavior of the application. It's important to state that errors calculated using the average unified L1 cache misses are in general low. Since average unified L1 cache misses have not been used in the representative subset's detection, low errors demonstrates that the representative subsets have physical meaning, that is, they really summarize the behavior of the whole application. The mean error of prediction of L1 cache misses of SimPoint is reported to be equal to 20 %, in the case of instruction cache, and 5 %, in the case of data cache [83]. Using our technique, the mean error of prediction of unified L1 cache misses is equal to 4.6 %.

Another important issue is the length of the representative subsets. We can see clearly that the applications are divided in two groups. The first one is composed by the applications with a length of the representative subset equal or lesser than 10^9 instructions. The second one is composed by the applications which have a length of the representative subset higher than 10^9 instructions. The applications of the first group have periodic behavior within the iteration initially elected as representative. For this reason, it has been possible to reduce even more the length of the representative subset. On the other hand, the applications of the second groups do not have a strong periodic behavior within the iteration initially elected as representative subset. The length of the representative subsets obtained using SimPoint is reported to be between 10^9 and 10^8 [83]. The same order of magnitude as ours.

In figure 5.1, we show a summary of the evaluations we have made. For each representative subset, we have obtained the relative error a set of metrics comparing their average value within the representative segment against the global average value. Results shown in table 5.1 are summarized in the average errors depicted in the first two columns of the figure. We also show the average errors obtained taking into account instructions dispatched, branch mispredictions and unified L2 cache misses. As we can see, average errors are low in the case of instructions dispatched and branch mispredictions. In case of unified L2 cache misses, the average error is higher because several benchmarks, for example gzip, have a small number of L2 misses. Due to this fact, relative errors can be high in these benchmarks. Since a few of programs have high error rates, the final arithmetic mean is remarkably increased.

Gap, Perlbmk and Sixtrack do not have periodic behavior and, therefore, the reduction cannot be made. In table 5.1 we have depicted this fact. In order to show graphically this non-periodic behavior in contrast with the periodic one, in figures 5.2 and 5.3 we depict CPI measured from executions of one typical periodic application, applu, and a non-periodic one, perlbmk. On the left of figure 5.2 we have global view of the evolution of the CPI during the whole execution of applu. We can see a large number of oscillations during the execution. On the right, we see a zoom of these oscillations. It is clear that they are periodical. In figure 5.3 we can see the same representations in the case of perlbmk application. It is clear that, in this case, the execution has mainly a non-periodic behavior. For this reason, it has not been possible to find a representative subset using our methodology.

Instead of this, our methodology is able to find good representative subsets in most of the SPEC CPU 2000 benchmarks. If it detects a representative segment of the execution, it means that an iteration of the main loop of the program has been detected. And the same detection can be obtained from an analysis which uses different metrics as input. In this section, we have made an evaluation taking into account several totally independent metrics. That is, if the execution of a given application is based on the iterative execution of the same segment of source code, we detect this behavior and we select one of these executions of the same segment of code as a representative subset using any kind of metrics as input. This approach is general and it works in most of the cases, as we demonstrate in the results shown in this section. This generality is based on the iterative character of the loops contained in many source codes. However, as we have seen, there is a minority of programs whose executions do not have this iterative properties because these executions are not based on iterations contained in loops. In this case, our approach can not select a minimal representative subsets of benchmarks' instruction execution stream. There are also mixed cases, that is, applications whose executions are strongly dominated by the iterative execution of a loop but they also have non-periodic regions. In these applications, we can select a representative subset but the relative errors extracted

Name	Representative's Length (#Ins)	Application's Length (#Ins)	Representative's CPI	Global CPI	Error CPI	Representative's L1 misses	Global L1 misses	Error L1 misses
Ammp	$63 \cdot 10^7$	$22611 \cdot 10^7$	3.693445	3.680016	0.36%	0.028336	0.028967	2.22%
Applu	$51 \cdot 10^7$	$18020 \cdot 10^7$	2.515707	2.514651	0.04%	0.010121	0.010081	0.40%
Apsi	$386 \cdot 10^7$	$27013 \cdot 10^7$	1.993946	1.993527	0.02%	0.017020	0.017064	0.26%
Art	$4 \cdot 10^7$	$3090 \cdot 10^7$	6.321399	6.320715	0.01%	0.095121	0.093114	2.11%
Bzip2	$247 \cdot 10^7$	$11418 \cdot 10^7$	1.015490	1.049764	3.37%	0.006498	0.006785	4.41%
Crafty	$2 \cdot 10^7$	$3809 \cdot 10^7$	0.850096	0.848195	0.22%	0.014319	0.015180	6.01%
Equake	$17 \cdot 10^7$	$10604 \cdot 10^7$	2.349061	2.347667	0.06%	0.019959	0.019048	4.56%
Facerec	$702 \cdot 10^7$	$22394 \cdot 10^7$	1.173704	1.178809	0.43%	0.011775	0.011840	0.55%
Fma3d	$61 \cdot 10^7$	$23885 \cdot 10^7$	1.640557	1.640441	0.00%	0.011932	0.012175	1.99%
Galgel	$267 \cdot 10^7$	$20125 \cdot 10^7$	1.446805	1.444753	0.14%	0.011812	0.014225	20.43%
Gap	Not Found	$22296 \cdot 10^7$	Not Found	Not Found	Not Found	Not Found	Not Found	Not Found
Gcc	$32 \cdot 10^7$	$4599 \cdot 10^7$	1.023815	1.023999	0.02%	0.009547	0.008462	11.36%
Gzip	$290 \cdot 10^7$	$1538 \cdot 10^7$	0.870649	0.889749	2.15%	0.012350	0.011667	5.53%
Lucas	$102 \cdot 10^7$	$12834 \cdot 10^7$	2.614271	2.614617	0.01%	0.002158	0.002390	10.75%
Mcf	$1018 \cdot 10^7$	$5126 \cdot 10^7$	20.072752	20.050443	0.11%	0.167533	0.150976	9.88%
Mesa	$28 \cdot 10^7$	$29072 \cdot 10^7$	0.958044	0.957993	0.00%	0.007190	0.007666	6.62%
Mgrid	$48 \cdot 10^7$	$48088 \cdot 10^7$	0.997486	0.997603	0.01%	0.002439	0.002418	0.86%
Parser	$3 \cdot 10^7$	$31619 \cdot 10^7$	1.654040	1.668365	0.86%	0.016293	0.015256	6.36%
Perlbmk	Not Found	$1167 \cdot 10^7$	Not Found	Not Found	Not Found	Not Found	Not Found	Not Found
Sixtrack	Not Found	$31906 \cdot 10^7$	Not Found	Not Found	Not Found	Not Found	Not Found	Not Found
Swim	$26 \cdot 10^7$	$20152 \cdot 10^7$	1.744981	1.744852	0.01%	0.011389	0.011186	1.78%
Twolf	$29 \cdot 10^7$	$25279 \cdot 10^7$	2.971909	2.971368	0.02%	0.046011	0.045594	0.91%
Vpr	$8 \cdot 10^7$	$7875 \cdot 10^7$	3.158843	3.160272	0.04%	0.044318	0.043834	1.10%
Wupwise	$420 \cdot 10^7$	$31935 \cdot 10^7$	0.950760	0.950736	0.00%	0.003213	0.003225	0.37%

Table 5.1: **Minimal Representative Subsets of SPEC CPU 2000 benchmarks.**

from it will be slightly higher than a typical periodic application. For example, bzip2 is one of this mixed cases.

It is important to state that, using this methodology, we obtain minimal subsets of a benchmark's instruction execution stream which are consecutive, that is, the program executes all the instructions of the minimal subset without executing an instruction which do not belong to the representative. It provides the optimal sampling length of SPEC CPU 2000 benchmarks because, due to iterative behavior, each execution segment whose length is equal to the optimal one summarizes the whole execution.

5.3.1.2 Taking the Length Arbitrarily

In the past section, we have detected the length and the localization of the minimal representative subset of a benchmark's instruction execution stream. Also, we have determined the correctness of these representative subsets showing the relative errors of the values of several metrics within the subsets. In table 5.1 we have shown that these relative errors are in general low and, therefore, we have demonstrated the correctness of the subset's election.

In this section, we want to emphasize that the length is crucial in order to obtain good representative subsets. To do so, we consider arbitrary lengths (20%, 40%, 60%, 80%, 100%, 120%, 140%, 160%, 180% and 200% of the optimum length detected using spectral analysis). After, we extract the values of several metrics within segments of the execution whose length is one of the previously described. These segments start

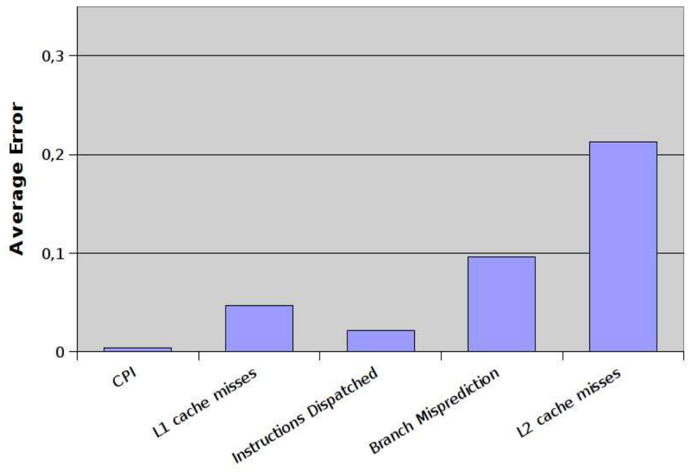


Figure 5.1: **Average relative errors of the minimal representative subsets of SPEC CPU 2000 benchmarks. Five different metrics have been used to compute errors.**

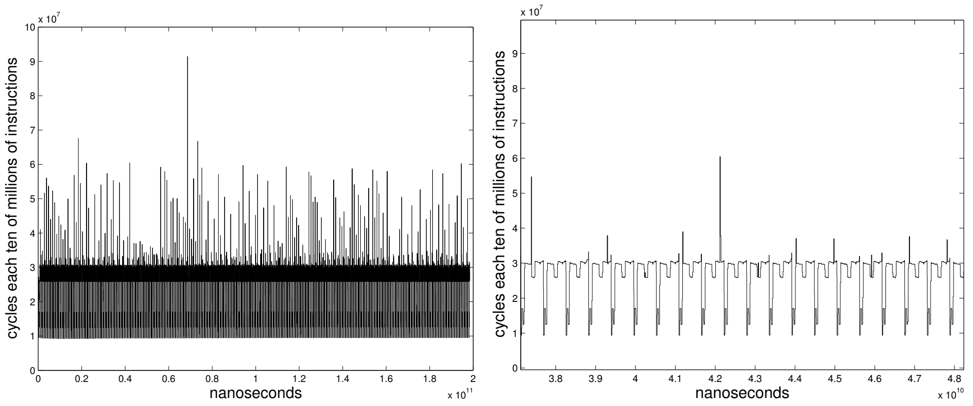


Figure 5.2: **On the left we have a global view of the evolution of the CPI during the whole execution of applu application. We can see a large number of oscillations during the execution. On the right, we see a zoom of these oscillations. They are periodical.**

Name	Metric	20% of R. L.	40% of R. L.	60% of R. L.	80% of R. L.	**100% of R. L.**	120% of R. L.	140% of R. L.	160% of R. L.	180% of R. L.	**200% of R. L.**
Ammp	CPI	**8.07**	30.72	20.04	**12.96**	**0.36**	25.47	41.10	56.23	77.16	9.80
	L1 miss	**8.46**	8.81	6.11	**4.59**	**2.18**	3.94	28.76	33.18	37.47	17.10
Applu	CPI	17.18	21.50	7.21	2.49	**0.04**	8.53	**1.95**	3.36	9.51	**0.21**
	L1 miss	53.65	2.82	31.01	7.13	**0.40**	7.49	**0.11**	12.23	3.56	**0.17**
Apsi	CPI	26.80	55.32	55.05	21.29	**0.02**	6.02	7.89	8.41	**4.27**	**0.07**
	L1 miss	36.73	20.04	15.79	1.74	**0.26**	11.00	7.28	2.65	**2.60**	**1.82**
Art	CPI	7.69	5.08	**4.64**	5.39	**0.01**	5.95	10.30	5.14	9.26	**5.46**
	L1 miss	18.52	15.94	**1.66**	7.67	**2.16**	7.29	12.41	7.47	7.47	**7.09**
Bzip2	CPI	10.35	16.23	12.26	8.05	**3.26**	**1.39**	**5.83**	8.40	11.20	13.55
	L1 miss	9.06	14.47	5.47	7.95	**4.23**	**0.23**	**5.25**	7.21	9.61	12.44
Crafty	CPI	79.91	13.19	39.38	24.58	**0.22**	20.18	**8.35**	18.74	10.85	**1.00**
	L1 miss	36.43	18.30	42.90	29.55	**5.67**	24.09	**12.86**	22.34	14.83	**0.08**
Equake	CPI	8.08	4.90	**1.33**	3.09	**0.06**	2.55	4.73	2.35	3.98	**2.30**
	L1 miss	1.92	11.38	**4.31**	9.98	**4.78**	8.45	10.05	7.67	9.80	**7.22**
Facerec	CPI	36.40	10.45	3.98	**3.44**	**0.43**	2.27	11.33	8.98	4.99	**2.48**
	L1 miss	15.28	9.16	13.12	**1.18**	**0.55**	10.58	7.07	8.61	11.54	**8.31**
Fma3d	CPI	116.34	63.32	10.36	2.61	**0.00**	10.19	13.76	5.41	**0.86**	**0.07**
	L1 miss	191.83	102.21	22.66	2.26	**1.99**	13.91	17.87	7.93	**0.39**	**2.44**
Galgel	CPI	**16.95**	21.17	15.82	12.56	**0.14**	3.53	**0.41**	0.38	0.80	3.67
	L1 miss	**1.84**	6.61	11.21	13.95	**16.96**	18.98	**18.30**	18.42	18.72	19.53
Gap	CPI L1 miss	Not Found	Not Found	Not Found	Not Found	Not Found	Not Found	Not Found	Not Found	Not Found	Not Found
Gcc	CPI	1.43	3.90	9.65	9.57	0.02	6.66	2.93	**1.33**	**0.14**	**1.48**
	L1 miss	16.05	13.82	9.31	9.52	12.82	10.63	9.73	**7.57**	**6.05**	**6.60**
Gzip	CPI	3.30	3.42	2.95	2.97	**2.15**	**3.85**	**4.09**	4.70	4.94	4.94
	L1 miss	17.97	17.89	19.35	19.35	**5.85**	**11.40**	**23.12**	32.07	35.92	35.92
Lucas	CPI	4.32	7.43	7.43	2.54	**0.01**	1.63	1.94	**2.28**	1.60	**0.51**
	L1 miss	31.96	39.75	15.90	15.90	**9.71**	13.91	21.21	**11.84**	12.10	**10.49**
Mcf	CPI	**3.88**	**5.30**	**6.78**	14.58	0.11	0.61	2.15	2.27	2.27	2.27
	L1 miss	**0.04**	**5.45**	**4.77**	7.81	10.97	12.41	12.89	12.87	12.87	12.87
Mesa	CPI	5.57	**1.52**	**0.96**	**1.72**	0.00	1.62	0.43	1.64	0.77	0.00
	L1 miss	7.82	**1.80**	**1.29**	**0.43**	6.21	3.71	3.73	2.05	3.19	5.44
Mgrid	CPI	20.69	21.87	8.24	1.04	**0.01**	4.46	5.88	3.47	**0.93**	**0.41**
	L1 miss	54.84	118.20	44.45	12.32	**0.87**	11.08	28.10	15.63	**4.80**	**0.25**
Parser	CPI	39.89	390.40	7.92	41.08	**0.86**	15.48	**4.81**	5.84	19.52	**0.01**
	L1 miss	53.99	439.87	18.45	59.42	**6.80**	24.58	**2.08**	13.51	28.20	**8.40**
Perlbmk	CPI L1 miss	Not Found	Not Found	Not Found	Not Found	Not Found	Not Found	Not Found	Not Found	Not Found	Not Found
Sixtrack	CPI L1 miss	Not Found	Not Found	Not Found	Not Found	Not Found	Not Found	Not Found	Not Found	Not Found	Not Found
Swim	CPI	39.69	24.19	11.25	6.91	**0.01**	10.89	8.90	5.12	**3.78**	**0.06**
	L1 miss	30.19	22.18	4.18	4.69	**1.81**	7.78	7.74	1.79	**2.26**	**1.00**
Twolf	CPI	4.05	0.84	**0.42**	0.99	**0.02**	0.49	**0.10**	0.50	0.04	0.32
	L1 miss	0.17	1.45	**0.50**	1.56	**0.91**	1.59	**1.17**	1.64	1.32	1.07
Vpr	CPI	22.52	**2.64**	9.20	1.22	**0.04**	4.36	7.08	**2.50**	7.08	3.51
	L1 miss	16.10	**1.39**	6.10	6.10	**1.10**	4.37	9.47	**5.30**	9.58	6.34
Wupwise	CPI	8.00	5.22	1.44	1.91	**0.00**	1.52	1.42	**0.39**	0.73	**0.05**
	L1 miss	26.82	16.03	3.38	2.69	**0.37**	5.28	4.09	**0.96**	0.68	**0.43**
Mean Error	CPI	22.91	33.74	11.25	8.62	**0.37**	6.55	6.92	7.02	8.31	**2.48**
	L1 miss	29.98	42.26	13.42	10.75	**4.60**	10.12	11.58	11.09	11.09	**7.85**

Table 5.2: **Values of CPI and L1 miss' relative errors taking representatives whose length is different from the optimal one. For each application, the three best lengths are highlighted. We consider a length better than another if its adding of the two errors is lower than the other one. It is important to state that, in general, 100 % and 200 % are two of the best lengths. Their mean errors are the best.**

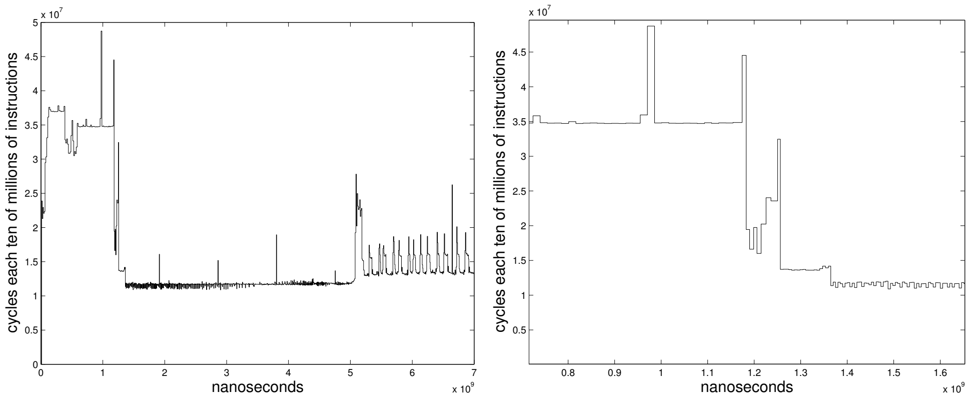


Figure 5.3: **On the left we have a global view of the evolution of the CPI during the whole execution of perlbmk application. On the right, we see a zoom. In this case, the whole execution is mainly under a non-periodic behavior.**

in the same point which is the beginning of the representative subsets detected by the spectral analysis.

In table 3.1 we depict the experiments. In the first column, there are the names of the applications. After, in the following columns we show the results obtained using several length's segments. Note that the results taking 100% of the representative's length have the same values than the results in table 5.1. For each application, we highlight the three best lengths. We consider a length better than other if the addition of CPI and L1 misses errors is lower than the other one.

We can see clearly that, in general, the best results are obtained taking 100% of the representative's length or 200% of it because the mean errors of these two lengths are the lowest. The interpretation of this fact is clear: the representative detected using spectral analysis really summarizes the execution of the application. Adding more instructions to the representative segment makes worse the results, except in the case that we take exactly two times the representative segment. On the other hand, it is important to state that this behavior is detected looking not only the CPI metric, but also the metric derived from L1 Cache Misses. Remember that the spectral analysis is performed taking the CPI metric. It means that the representatives are useful from the CPI point of view and from other metrics point of view.

5.3.1.3 Parallel Benchmarks

We have applied spectral analysis techniques to Class B NAS Parallel Benchmarks [3]. We executed BT, CG, FT, MG and SP on 16 and 64 processors. They have been executed on MareNostrum supercomputer [57]. In table 5.3 we depict the results obtained. Representative segment and whole application sizes are depicted in milliseconds because time it is a clear metric in parallel applications since it has a common value for all the processes. In the second, third and fourth columns we show the durations of initialization phase, of representative segment and of whole execution. In the fourth and fifth columns we depict the average value of CPI within the representative segment and within whole execution. This average value is obtained taking into account all the processes involved in the execution. In the sixth column there is the relative error of the average CPI extracted from the representative segment in contrast to the same value extracted from the whole execution. In the seventh and eight columns we depict the average value of the unified L1 cache misses per millisecond within the representative segment and within the whole application. Finally, we show the relative error of this average value. These data have been obtained extracting hardware counters each nanosecond of execution.

It is important to state that the main difference between results obtained from parallel and sequential benchmarks is the initialization phase. In parallel cases, wavelet analysis detects an initial non-periodic phase which consists in communications in order to assign initial values to the variables, domain decomposition, etc... In the sequential benchmarks we have analyzed, this initial phase is negligible because it is very short. This is the reason why we show the initialization phase in parallel benchmarks and we omit this value in sequential benchmarks.

In general, we achieve significant reductions, that is, the length of representatives segments are remarkably smaller than whole application lengths. Relative errors of CPI and L1 cache misses metrics are quite low in general. There is only one case that shows a significant error: L1 cache misses extracted from the representative segment of FT application are not a good approximation of the real value. The reason is that there are very few L1 cache misses during the execution of this application and, therefore, L1 cache misses are not a significant metric in the case of FT. Many executions on 64 processors do not have significant L1 misses because the whole data can be stored in the L1 cache memories of 64 CPU's.

Any kind of performance analysis can be performed taking the data contained within the representative segment of execution selected using spectral analysis. This is an automatic method to remarkably reduce the amount of data which has to be considered to extract conclusions about the performance of applications. The length of the representative segment is a useful value to efficiently extract information from executions, as we can see in section 5.5.

Name	Initialization's Length (ms)	Representative's Length (ms)	Application's Length (ms)	Representative's CPI	Global CPI	Error CPI	Representative's Ratio L1 misses / ms	Global Ratio L1 misses /ms	Error Ratio L1 misses / ms
BT 16	339	335	69426	0.806	0.815	1.10%	461.3	466.3	0.21%
BT 64	1102	95	30096	1.515	1.587	4.53%	Negligible	Negligible	–
CG 16	718	15	37743	2.272	2.325	2.28%	2527.9	2269.11	11.40%
CG 64	1616	8	32448	1.613	1.723	6.38%	2359.70	2132.34	10.66%
FT 16	2995	1079	25349	1.754	1.515	15.77%	5.551	3.779	46.89%
FT 64	1963	313	8686	1.613	1.515	6.47%	Negligible	Negligible	–
LU 16	256	203	58419	1.695	1.666	1.74%	2328.73	2566.67	9.27%
LU 64	1920	61	36587	1.449	1.515	4.35%	4861.08	4786.55	1.56%
MG 16	270	199	5709	2.128	2.083	2.16%	3402.79	4451.13	23.55%
MG 64	140	55	2590	1.724	1.5151	13.78%	Negligible	Negligible	–
SP 16	198	329	135291	5.010	5.172	3.13 %	311.488	307.19	1.39%
SP 64	4051	71	43897	2.857	2.778	3.49%	Negligible	Negligible	–

Table 5.3: **Minimal Representative Subsets of Class B NAS parallel benchmarks.**

5.4 Clustering

There are several approaches to analyze data. In this section, we explore interactions between spectral analysis techniques (wavelet analysis, Fourier transform, ...) and clustering techniques (k-means algorithm, ...). On the one hand, information derived from spectral analysis does not classify the intervals but it gives information about the iterative behavior of the execution. On the other hand, information derived from clustering techniques provides a classification of intervals according to the values of the counters. Intervals with low values of the counters will be classified in the same cluster and the same will happen with intervals with high values. The key point to obtain a successful clustering of the intervals is to have a good initial set of data. And this is where spectral analysis and clustering interact: spectral analysis can provide the value of the optimal sampling interval length which generates a good initial set of data for clustering.

As we have seen in section 5.2, the most recognized methodology to detect minimal subset of a sequential benchmark's instruction execution stream to estimate the performance of the complete benchmark, SimPoint, is based on clustering techniques. These techniques have several advantages. For example, the computational complexity of clustering algorithms is low and they achieve strong reductions of the benchmark's instruction execution stream. However, clustering techniques have several faults, for example, strong sensitivity to initial conditions.

In this section, we focus on the fact that clustering techniques do not provide insights about the optimal sampling interval length and we demonstrate that to perform a clustering on data which have been obtained using this optimal value is better than to use data obtained from an arbitrary sampling length. For example, in [83], the sampling interval length has a fixed value of 10^8 instructions. This is an arbitrary election, forced by the fact that it is not possible to estimate the value of the optimal length without spectral analysis. As we demonstrate in section 5.4.2, the election of this length is important in terms of accuracy. Therefore, spectral analysis can be applied as input of clustering based methodologies in order to improve their accuracy and obtain better minimal subsets of benchmarks' instruction execution stream.

To illustrate the impact that the detection of the optimal sampling length has, we depict several information extracted from an execution segment of the applu application in figure 5.4. We show the CPI signal. As we have said in section 5.3.1.1, this signal has a minimal granularity of 10^7 instructions. We also depict three examples of three different samplings. If a sampling is performed using a length of, for example, 10^8 instructions, each sampling point will provide the average value of the cycles performed each ten of million of instructions within an interval of 10^8 instructions. The first one, sampling A, is performed each $3 \cdot 10^8$, the second, sampling B, is performed each $5.1 \cdot 10^8$ millions of instructions and the third is done each $8 \cdot 10^8$. As we can see, the variance of data in the case of sampling B is lower than in the other two samplings. The explanation is that each point of sampling B represents one minimal iterative region of the execution. Remember that the length of the minimal representative subset in the case of applu is, according to table 5.1, $5.1 \cdot 10^8$. It means that the length of the minimal representative subset of a benchmark's instruction execution stream shows a way to generate a good initial set of points. In the next sections, we evaluate the errors of clustering techniques applied to many sets of points.

5.4.1 Description of the Experiments

To demonstrate the limitations that arise from the fact that the sampling interval length is arbitrarily chosen, we perform the following experiment: Taking the SPEC CPU 2000 benchmarks, we extract two performance metrics from each application. On the one hand, we extract the number of cycles performed by the processor from each sampling interval. Remember that the sampling interval length is measured in #instructions. Therefore, if we divide the number of cycles by the sampling interval length, we will obtain the CPI within this interval. On the other hand, we extract L1 cache misses from each sampling interval. Remember that, as we have said in section 5.1, the signals are generated from hardware counters. For example, suppose that we extract a hardware counter that gives the number of L1 cache misses. If the sampling interval length is equal to 10^7 instructions, the extraction of the counter will be made each time 10^7 instructions are executed. A signal is generated from this information. That is, for each application and each sampling interval length, we generate two signals: One signal contains information derived from the counter of L1 cache misses and the other one contains data derived from the counter of cycles.

For each application, each sampling interval length and each metric we have a

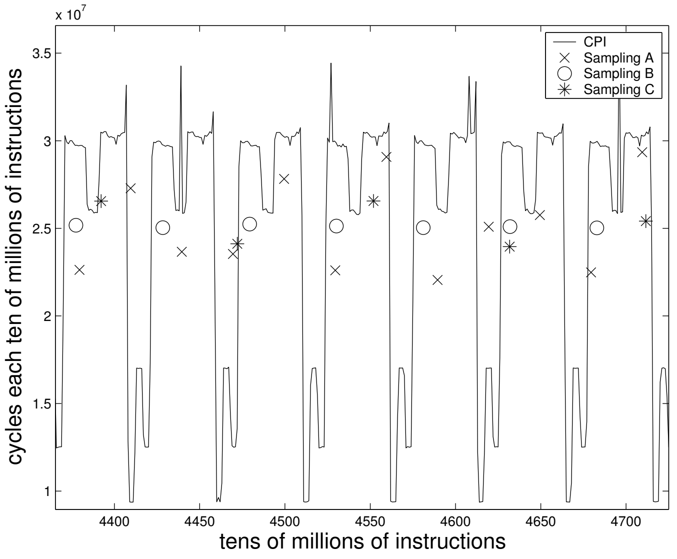


Figure 5.4: **CPI signal extracted from an execution segment of the applu application. We also show samplings using 3 different lengths. The first one, *Sampling A*, is performed each $3{\cdot}10^8$ millions of instructions, the second, *Sampling B*, is performed each $5.1{\cdot}10^8$ and the third is done each $8{\cdot}10^8$. If a sampling point has the coordinates (x, y), it means that y is the average value of cycles executed each ten million of instructions within the execution interval $[x - length, x]$. For example, if (x, y) belongs to *Sampling A*, it summarizes the execution interval $[x - 3 \cdot 10^8, x]$.**

signal. This signal contains values associated to intervals. We perform a clustering of the intervals based on these values using k-means [55] algorithm. After, we elect a representative interval for each cluster. It is the nearest point of each cluster to cluster's centroid. This method could be used as a predictor, that is, if we perform the clustering and we find the representatives taking the L1 cache misses as input, we can predict the global value of CPI taking its values within the representatives and multiplying it for the weight of the centroid. In terms of formulas, if we want to predict the global value of a given performance factor X_{global}, we do the following:

$$X_{global} = \sum_{i=1}^{n_{clusters}} w_i \cdot x_i \tag{5.1}$$

Where w_i is the weight of each cluster, that is, the ratio between the number of points contained in it and the total number of points, x_i is the value of the performance factor X of the representative of cluster i and $n_{clusters}$ is the total number of clusters of the division performed by k-means.

The value of $n_{clusters}$ is a value that has to be provided to k-means algorithm as an input parameter. In our experiments, we provide a value of 4. Obviously, a value greater than 4 will provide better results because clusterings will be more accurate and, therefore, the representative subset will be greater. In any case, the goal of this work is to explore the sensitivity of the sampling interval length and how this length can be provided by spectral analysis. Results shown in next section are focused to this point. Obviously, the value of $n_{clusters}$ affects the results of the experiments, reducing or increasing the errors. However, the qualitative behavior of the results remains untouchable, that is, if the sampling interval length provided by spectral analysis improves clustering with $n_{clusters} = 4$, it will also improve clusterings with $n_{clusters} \geq 4$. It is important to state that improvements with low values of $n_{clusters}$ are important because they provide small and accurate representative subsets.

In addition, we perform two predictions more using two more techniques. On the one hand, we chose as representative the interval that has the closest value of a given metric to the global mean of this metric within the whole execution. For example, if we are analyzing an execution from the L1 cache misses point of view, we will elect the interval which has a L1 cache misses value closer the global mean of L1 cache misses. This global mean is extracted from the values of all of the intervals. On the other hand, we chose as representative a set of consecutive intervals. The elected set of consecutive intervals has the closest mean value of a given metric to the global mean of this metric within the whole execution. The number of intervals of each set is equal to the parameter $n_{clusters}$.

The three techniques we have explained in this section are inspired to the tech-

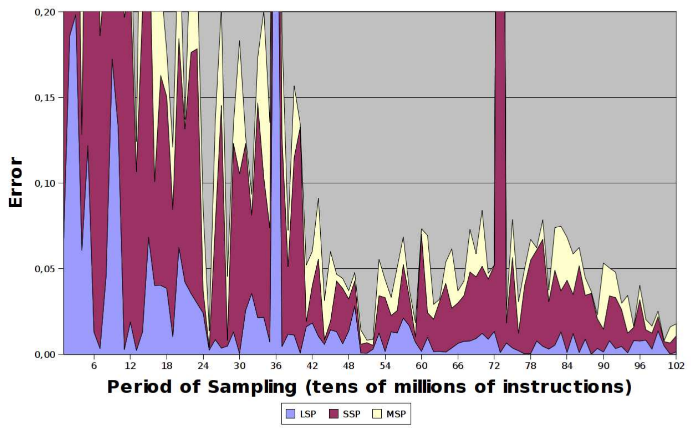


Figure 5.5: **Applu application: Errors obtained using techniques inspired on Multiple SimPoints, Single SimPoints and Large SimPoints taking several sampling interval lengths. Best results are obtained near $5.1 \cdot 10^8$ and $1.02 \cdot 10^9$. These are the values we showed in table 5.1 when we were studying the length of the minimal representative subset and two times this value.**

niques explained in [83] and called Multiple Simulation Point (MSP), Single Simulation Point (SSP) and Long Simulation Point (LSP). The first one, MSP, is comparable to the clusterization explained above using k-means algorithm. The second, SSP, is comparable to chose as representative subset the interval that has the closest value of a given metric to the global mean of this metric. Finally, the third, is equivalent to elect several consecutive intervals as representative subset.

5.4.2 Evaluation

In this section, we evaluate the sensitivity of changing the length of sampling intervals. In order to do so, we have applied several times the techniques explained above to the same set of applications but changing the sampling interval length. This set of is composed by the applications applu, fma3d, lucas, mgrid and swim. They are 5 typical applications with iterative behavior. They represent most of the SPEC CPU 2000 benchmarks.

We extract data taking a sampling interval length equal, first, to 10^7 instructions, after $2 \cdot 10^7$ until two times the value of the representative subset's length depicted in table 5.1. We want to study what happens when the sampling interval length is near to the representative's length or a multiple of it.

In figures 5.5, we can see the results of experiments using applu application. To calculate the values of Multiple Simulation Point technique, we have performed a clusterization using k-means algorithm. To highlight the independence of metric election, we use unified L1 cache misses as input. The errors are obtained comparing

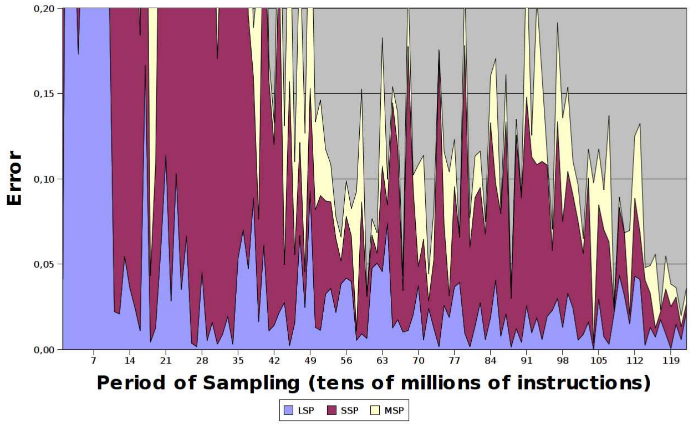


Figure 5.6: **Fma3d application: Errors obtained using techniques inspired on Multiple SimPoints, Single SimPoints and Large SimPoints taking several sampling interval lengths. Best results are obtained near $6.1 \cdot 10^8$ and $1.22 \cdot 10^9$. These are the values we showed in table 5.1 when we were studying the length of the minimal representative subset and two times this value.**

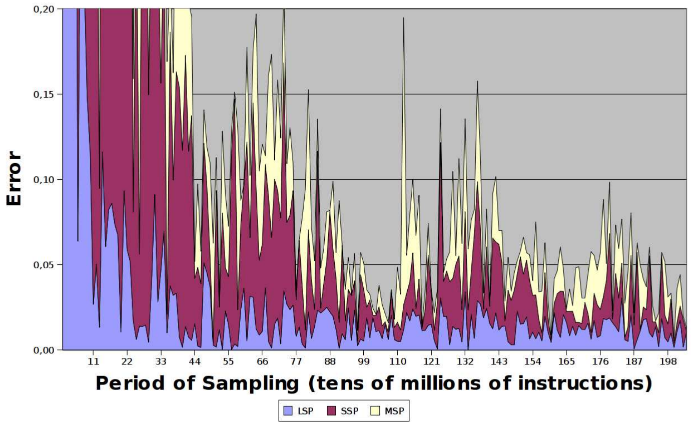


Figure 5.7: **Lucas application: Errors obtained using techniques inspired on Multiple SimPoints, Single SimPoints and Large SimPoints on Lucas application and taking several sampling interval lengths. Best results are obtained near $1.02 \cdot 10^9$ and $2.04 \cdot 10^9$. These are the values we showed in table 5.1 when we were studying the length of the minimal representative subset and two times this value.**

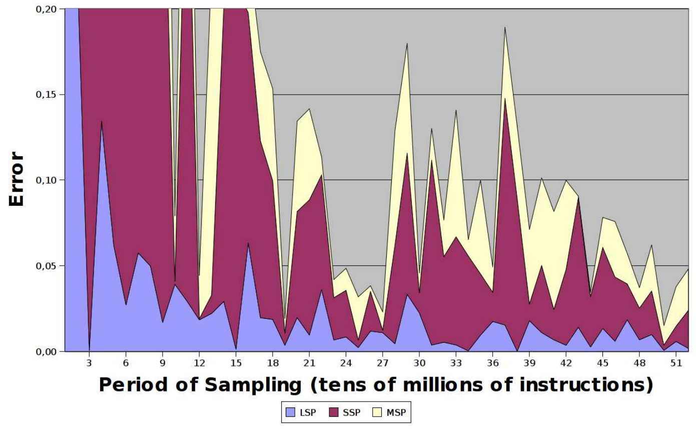


Figure 5.8: **Swim application: Errors obtained using techniques inspired on Multiple SimPoints, Single SimPoints and Large SimPoints on Swim application and taking several sampling interval lengths. Best results are obtained near $2.6 \cdot 10^8$ and $5.2 \cdot 10^9$. These are the values we showed in table 5.1 when we were studying the length of the minimal representative subset and two times this value.**

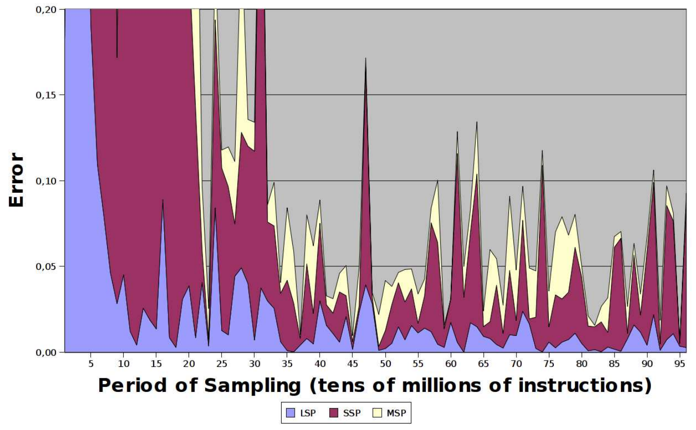


Figure 5.9: **Mgrid application: Results using techniques inspired on Multiple SimPoints, Single SimPoints and Large SimPoints techniques taking several sampling interval lengths. Best results are obtained near $4.8 \cdot 10^8$ and $9.6 \cdot 10^8$. These are the values we showed in table 5.1 when we were studying the length of the minimal representative subset and two times this value.**

the predicted CPI with the real one. Parameter $n_{clusters}$ is equal to 4. We also show the predictions of the two other techniques. As we can see in the figures, we first take sampling interval length equal to 10^7 instructions, after $2 \cdot 10^7$,..., until $1.02 \cdot 10^9$. In figure 5.5 we can see that the best values for sampling are, for the three techniques, included in two intervals. The first interval is a neighborhood of the representative's length depicted in table 5.1, $5.1 \cdot 10^8$. The second interval is neighborhood of two times this value. The conclusion we extract is that, if we sample using the length of the minimal representative subset, we obtain the best predictions. Furthermore, it does not matter the technique we use because the results obtained using Long Simulation Point, Single Simulation Point and Multiple Simulation Point are excellent. This fact highlights the importance of the detection of the optimal sampling interval.

In figure 5.6 we can see the experiments using fma3d application. As we can see, we take sampling interval lengths between 10^7 until $1.22 \cdot 10^9$. Again, we can see how the techniques are not very accurate for low values of sampling interval length, specially in the case of Single Simulation Point. In this case we can see how the best values are around $6 \cdot 10^8$ and $1.2 \cdot 10^9$. The first point is equal to value of the length of the representative subset, shown in table 5.1. The second point is equal to two times the length of the representative segment.

In figures 5.7, 5.8 and 5.9 we show the same kind of results for lucas, swim and mgrid. In all cases, we can see how the best results are obtained in neighborhoods whose center is equal to the value of the representatives' length of table 5.1 or equal to two times this value.

In all these examples we have shown several facts which are important to state: The first one is that an optimal election of sampling interval length always improves the accuracy of the results, that is, it does not matter to use Long Simulation Point (LSP), Multiple Simulation Point (MSP) or Single Simulation Point (SSP) if we use the optimal sampling interval length because the results will be accurate. However, the length of the representative subset will be different depending on the technique we use, that is, SSP technique provide smaller representative subsets than MSP and LSP. Therefore, it seems more reasonable to use SSP technique if we are able to detect the optimal sampling interval length because this technique provides smaller representative subsets than the other two. However, if we have no idea of the optimal sampling length, it is more reasonable to use MSP or LSP techniques because they provide more accurate results than SSP.

In conclusion, it is demonstrated that a previous detection of the optimal sampling interval length improves the results of clustering techniques focused on the detection of minimal representative subsets of applications' instruction execution stream. An important issue is the independence of the representative subset with respect to architectural parameters. Note that if we use results from our technique to provide SimPoint with the optimal sampling interval length, we will obtain more accurate results than SimPoint alone and the representative subset will be independent of the architecture because it is reported that SimPoint detects representative subsets independently of the architecture [71].

5.5 Optimal Information Extraction

In order to study the performance problems of parallel applications, it is mandatory to extract information from their executions. It is not an easy work. Many times, the amount of information extracted is very large and it is not trivial to select significant data. In our previous work, we addressed this problem and we proposed methods to automatically elect significant data [12] and to extract general but non-trivial performance properties of the applications [14].

In this section, we use information extracted from spectral analysis to develop a methodology able to optimally extract relevant data. It is well known that, if we want to extract values of several hardware counters, we have to address several problems: First, we have to perform a complete execution of the application if we do not know insights about the representative execution segments. Second, we have the problem that many combinations of hardware counters are incompatible and, therefore, we have to execute the application several times, one time for each incompatible counter. Remember that these executions are, in general, computationally expensive.

The information extracted from spectral analysis can overcome this problems. As we have said, our methodology can identify, first, the initialization phase of the execution and, second, the computation phase and its main iterative period. Remember that this iterative behavior of the computing phase is due to the loops contained in the source code. The length of iterations is the optimal sampling length because to sample using optimal sampling length supposes extract data from each iteration of the computing phase. As we have seen, these iterations summarize the behavior of the application. Therefore, if we want to extract the values of three incompatible hardware counters, we only have to execute the application during the initialization phase, which is not representative, and during three iterations of the computation phase. These three iterations are the three sampling intervals from which we extract the prediction of the global value of each incompatible hardware counter.

Our concrete experiment consists in predict the number of translation lookaside buffer (TLB) misses, the number of floating point instructions and the number of branch mispredictions performed during the whole execution of an application. These hardware counters are incompatible [66], that is, if we are extracting data about the

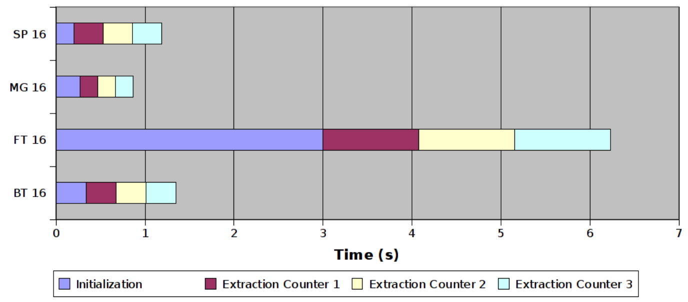
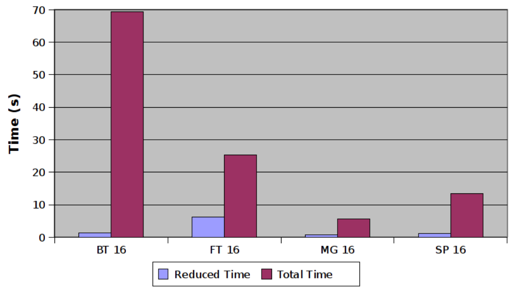


Figure 5.10: **Extraction of the predictions of global values of hardware counters. Counter 1 is the number of floating point operations performed, counter 2 is the number of TLB misses and the third is the number branch mispredictions.**



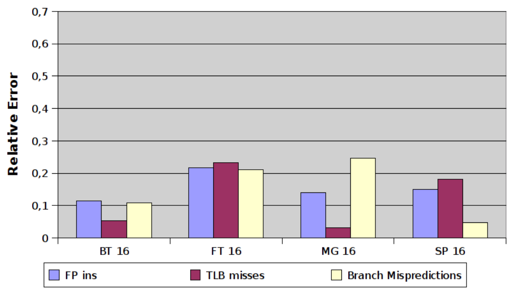


Figure 5.11: **Time focused on extract predictions of three independent hardware counters (reduced time) and total execution time of applications. To extract the global value of each incompatible hardware counter we need one complete execution.**

Figure 5.12: **Relative errors of the predictions of global values of hardware counters obtained using our methodology based on spectral analysis (wavelet transform, ...). Note that the average error is around 15% and the maximal error is around 25%.**

number of floating point instructions in a given moment, we can not extract data about TLB misses or branch mispredictions. We validate the results comparing the values obtained using our methodology against values extracted from complete executions.

The set of applications is composed by several representative applications of NAS Parallel Benchmarks: BT class B, FT class B, SP class B and MG class B. All of them have been executed on 16 processors of MareNostrum supercomputer [57]. Similar studies can be performed with the other applications represented in section 5.3.1.3. The reason why we limited the range of study to executions on 16 processors is that the experiments explained in this section are very expensive from the computational point of view because the validation requires many complete executions of each parallel application. Since hardware counters are incompatible, we need to execute the applications as many times as the number of incompatible counters we are interested to.

In figure 5.10, we have depicted graphically how we obtain the predictions of global hardware counters. For example, in the case of FT, we obtained the prediction of the global floating point instructions from the execution segment which starts just after the initialization phase, this is 2995 ms after the beginning of the execution, and ends 1079 ms after. The prediction of TLB misses is obtained from the following execution segment of 1079 ms and, finally, the number of branch mispredictions from the next one. Remember that this optimal sampling length has been obtained from the spectral analysis and that it is depicted in table 5.3. In figure 5.10 we also show the same kind of information in the cases of SP, MG and BT applications.

In figure 5.11 we compare the execution time needed to extract the predictions of the three hardware counters against tot total execution time of each application. Remember that, in order to extract the global values of three incompatible hardware counters, it is mandatory to execute three times the whole application, that is, three times the total time value depicted in figure 5.11.

In figure 5.12 we show the relative errors of our predictions. In general, relative errors are higher in those application whose initialization phase has more relative weight within the execution. A clear example of this effect is FT. On the other hand, it is also possible to obtain relative errors higher than 10% if predicted metrics have low values during the execution. Branch Mispredictions in the case of MG is an example. The average error is around 15% and the highest one is around 25%. Thanks to our methodology, we can estimate significant hardware factors with a very reduced computational effort.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis we have applied spectral analysis to information derived from real executions of applications. On the one hand, these applications are message passing programs. In these cases, spectral analysis is useful in order to reduce the amount of information and in order to study performance properties of message passing codes. On the other hand, these applications are SPEC2000 benchmarks, programs widely used in computer architecture. In these cases, spectral analysis is useful in order to detect the optimal sampling frequency, value that remarkably improves existing analysis tools focused on detect representative subsets of whole executions. We can extract several conclusions, related to several topics:

6.1.1 Automatic Phase Detection and Structure Extraction of MPI Applications

This topic consists of explore the possibility of automatically derive the internal structure of a tracefile. This structure has two main properties: First, it is based on periodicities and, second, it is hierarchical. We have shown that it is possible to, first, detect the perturbed regions of the tracefile and, second, derive the internal structure of non-perturbed regions. It is useful in many aspects: It makes easier the process of tracing the application, it avoids the spending of time studying perturbed zones, it gives the internal structure of the tracefile and, finally, gives the most representative regions of it. In conclusion, we have reduced the problem of analyzing a huge tracefile (10 or 20 GB) to the study of several hundreds of MB.

Other important aspect to take into account is the flexibility of the methodology. We have demonstrated that it is possible to apply the automatic analysis to any kind of metric we can generate from data contained in tracefiles. This flexibility opens the possibility to study the application from many points of view, ruling out the non-significant metrics and taking into account the useful ones.

The automatic detection of structure of MPI applications tracefiles is the first step to automatize, as much as possible, the process of performance analysis of executions of applications in order to reduce the time required to analyze data generated from executions. Automatic structure detections using spectral analysis opens a wide range of possibilities, as we can see in the other contributions of the thesis. It enables the possibility to fit an analytical speedup model which is able to detect performance problems and warn about it. This fitting can not be made without the size reduction because the size of the data would be too large.

6.1.2 Speedup Analysis of MPI Applications

This topic consists in the proposal of a model for the automatic detection of the factors which undermine the scalability of applications. First, it is mandatory to apply the size reduction explained above in order to detect the execution phases and rule out the non-significant ones. The reduction also generates subtraces which contain several repetitions of the periodic pattern detected on the computation phase. After that, it is possible to automatically extract information from the subtraces and to fit an analytical model of the speedup. This model is useful to explain the behavior of the speedup, extracting conclusions about the main problems of the performance of the application.

The performance analysis of applications is based on the 2-iterations subtrace. This trace contains with a very high detail the information of the execution of the application but is smaller than the original trace since the damaged regions of it and the redundant information have been eliminated. The information derived from our speedup model provides a very good general guideline for the analyst. The combination of these two factors, the automatic reduction of the size of the tracefile without lose of high detail information and the conclusions derived automatically from the model, makes easier the process of performance analysis of an application and reduces dramatically the required time. The potential of this combination cannot be achieved by other performance analysis techniques such us profiling.

The model can easily be extended in order to automatically study performance properties of applications which are related to memory. With this extension, it is possible to automatically perform a numerical analysis of the impact of memory behavior of applications. Furthermore, note that our approach is as flexible as the application requires, i. e., it is possible to study any factor which can undermine the speedup only extending the speedup model.

On the other hand, an automatic analysis done in a streaming manner, that is, an analysis which do not requires the storage of the whole trace, is being implemented to extend as far as possible the functionalities and features of the approach shown in this thesis. This kind of analysis will give us the possibility of automatically obtain a speedup decomposition and, therefore, an explanation of the behavior of the speedup before than the complete execution of applications.

We have also shown an approach focused to the prediction of the execution time of parallel applications. Taking into account several executions of a given application, it is possible to extrapolate this executions on a higher number of processors. This prediction scheme is able to provide an estimation of the execution time of the application increasing the number of processors which is better than the classical linear interpolation. It is also possible to obtain information about the main performance undermining factor using the analytical model we propose.

The possibility to obtain fast predictions of executions of applications gives a first qualitative idea of suitability of increasing the number of processors if we want to improve the performance of the application. Since there is an increasing need of computing power in many scientific and engineering areas, this topic will become more and more important in the next years. For this reason, it is mandatory to do more research to improve the current prediction methods.

In conclusion, we provide a methodology which is able to automatically detect general but non-trivial performance characteristics of applications. Since the methodology is based on, first, a size reduction performed using fast signal processing algorithms and, second, on a direct extraction of parameters, the automatic analysis is obtained very quickly. On the other hand, our analysis scheme is flexible in the sense that any performance property can be taken into account and its impact in the speedup can be evaluated numerically. Finally, this methodology can be applied either after the execution of the application or during the execution.

6.1.3 Extraction of Optimal Sampling Frequency of Applications

In this topic, we have explored the possibility of extract the optimal sampling frequency of applications using spectral analysis. We have applied the methodology to SPEC CPU 2000 and NAS Parallel benchmarks. In general, minimal representative subsets automatically found summarize the whole execution of the application. An interesting feature of these subsets is that they are consecutive. This point is important because it supposes the detection of an optimal sampling length which shows which is the size of minimal representative execution segments. It has many applications.

On the one hand, we have shown how the length of these representative subsets can be used as input by clustering based tools such us SimPoint. We have shown that, if we use a sampling interval length equal to the length of the representative subset (or a multiple of it), the results of clusterings will be more accurate. The interaction between techniques derived from spectral analysis and currently existing tools focused on electing a minimal subset of a benchmark's instruction execution stream to estimate the performance of the complete benchmark is a very productive topic because spectral analysis can give valuable insights to existing tools. This point is very important and has not been taken into account in previous works.

On the other hand, we have used the optimal sampling length to predict global performance properties of applications. In the case of parallel applications, it is very relevant because it significantly reduces the computational load required to extract performance data. Prediction errors are reasonably low.

In summary, spectral analysis is a powerful tool that can be used in the context of analysis of applications because it improves the current methodologies if it is used as input of the currently existing tools, opens new perspectives in this topic if it is used alone and, finally, does not increase significantly the overhead of the analysis due to its low computational complexity.

6.2 Future Work

The work performed in this thesis opens several perspectives that can be addressed in the future:

Run time analysis of executions of applications is a perspective that is currently being implemented. On the one hand, this run-time analysis provide a remarkable reduction of the storage requirements of our methodology because it provides the possibility to perform the analysis without generating a whole tracefile. On the other hand, the computing time required to execute the whole application can also be remarkably reduced because it is not mandatory to execute the application entirely. It is important to state that, from the conceptual point of view, the analysis methodology applied on run-time is the same as the applied one to postmortem tracefiles. That is, the application of the methodology is only an implementation issue, it is not a fundamental issue of our approach. This fact highlights the generality of our work.

The integration to Paraver of the source code implemented to evaluate the approach of this thesis is being made. This integration can provide to Paraver with an automatic analysis module can satisfy the need for simplification and automatization of the analysis in order to help unfamiliarized users to work with Paraver.

An extension of the performance properties of our model will provide more generality and flexibility to our approach. For the moment, issues related to the memory or the operating system are not directly taken into account. Indirectly, these issues can impact performance factors which are eventually related to them. However a direct expression of such performance issues can be easily derived from the theoretical point of view and can provide a clearer performance analysis.

The integration of the spectral analysis methodology as a pre-process step in order to improve the analysis performed by tools such as SimPoint or Smarts can be fruitful to improve these tools and search for more complete, general, and accurate methodologies to select the minimal representative subset of an application's instruction execution stream.

As you have seen, in this work we have studied executions of parallel applications implemented using the message passing paradigm. We have not paired attention to shared memory applications. However, several performance properties of these applications, such us the efficiency of the scheduling or the load balance can be studied using our approach, that is, generating signals from the executions and, after that, analyzing them. The perspective of study shared memory applications can give more generality and flexibility to our approach.

List of Tables

3.1 **Liso application structure detected by our system. The time units are milliseconds. The first three columns show the hierarchical levels of periodicity. Level 0 column shows the total elapsed time, Level 1 column shows the different phases detected by the automatic system and Level 2 shows the internal structure of one of the periods of Level 1. In the first three columns, each cell contains three numbers: The total span of the region, the number of the periods found in that region and the duration of each period.** 66

3.2 **Idris. Table Representation** 67

3.3 **Idris. Tree Representation** 67

3.4 **Gadget** 67

3.5 **Linpack** 67

3.6 **Evaluation of Spectral Analysis. Metrics used are SDCB (Sum of Duration of Computing Bursts), NPC (Number of Processes Computing), MPIp2p (Number of Processes in a MPI point-to-point call) and CPI (Cycles per Instruction). Results are depicted in milliseconds.** 69

3.7 **Evaluation of Spectral Analysis. Metrics used are Communications (number of flying communications), FLOPS (number of floating point instructions executed), MPIcol (number of processes in a MPI collective call) and Instructions (number of instructions performed). Results are depicted in milliseconds.** . . 70

3.8 **Sizes of all the representative traces of each level.** 72

5.1 **Minimal Representative Subsets of SPEC CPU 2000 benchmarks.** 106

5.2 **Values of CPI and L1 miss' relative errors taking representatives whose length is different from the optimal one. For each application, the three best lengths are highlighted. We consider a length better than another if its adding of the two errors is lower than the other one. It is important to state that, in general, 100 % and 200 % are two of the best lengths. Their mean errors are the best.** 108

5.3 **Minimal Representative Subsets of Class B NAS parallel benchmarks.** 111

List of Figures

2.1 **Paraver Tracefile Example** 27

2.2 **Paraver views of two iterations of NAS-BT class A executed on 4 processors.** . . 30

2.3 **Haar Scaling Function** 33

2.4 **Haar Wavelet** 34

2.5 **Sinusoid and right-shiftings** 37

2.6 **Crosscorrelations** 37

3.1 **Visualization of the metrics explained in section 3.3. At the top, we show the Sum of Duration of Computing Bursts and the Number of Processes Computing. In the middle, we show the number of Point to Point MPI Calls and the number of Collective Calls. Finally, at the bottom, we see the IPC and the number of Communications. These metrics have been generated from a tracefile of an execution of WRF-NMM application on 128 processors.** 52

3.2 **Input Signal. Sampling Frequency =** α 55

3.3 **Wavelet Output. Location of frequencies between** $[\frac{\alpha}{4}, \frac{\alpha}{2}]$ 55

3.4 **Selection** 55

3.5 **An example of flushing. Processes are flushing records to disk.** 58

3.6 **Signal** 58

3.7 **Closing of the Signal** 58

3.8 **An example of clogged system. Many communications are being transmitted with an extremely low bandwidth.** 60

3.9 **On top, visualization of the whole Liso tracefile and the flushing zones . In the middle, the first region without flushing is shown. We highlight a region without periodic structure and a region with 5 iterations. At the bottom, we show one of these iterations.** 65

3.10 **Sum of the sizes of all the representative traces of each level.** 72

3.11 Representation, in logarithmic scale, of size reductions of the applications depicted in tables 3.6 and 3.7. Results are depicted in Bytes. The metric used to perform the analysis is the Sum of Durations of Computing Bursts. 72

3.12 Representation, in logarithmic scale, of size reductions of the applications depicted in tables 3.6 and 3.7. Results are depicted in Bytes. The metric used to perform the analysis is the Sum of Durations of Computing Bursts. 73

3.13 Scalability of the system 74

3.14 Scalability of the system 75

3.15 Scalability of the system 75

4.1 Representation of *CommEff* and *LB* factors. In the x-axis, there is the normalized time. In the y-axis, there is the set of processes. For each process we show, first, its computing time and, second, its communication time. The communication time (time in MPI) has been divided in two subsets: Communication time due to load imbalance and actual communication time. 83

4.2 Application of the size reduction. The input is a tracefile obtained from the execution of WRF-NMM application on 128 processors in MareNostrum supercomputer. First, the automatic system derives a signal from the tracefile. After that, Wavelet Analysis finds the periodic phase and, finally, a subtrace that contains two iterations is generated. 88

4.3 Execution Times needed by the automatic system to perform the reduction from the whole trace to the 2-iterations subtrace. Sizes of the whole traces are also shown. 89

4.4 Size of the whole trace and size of the 2-iterations subtrace generated by the automatic system. 89

4.5 Values of the speedups obtained from the executions of WRF-NMM and WRF-ARW on 128, 256 and 512 processors. Input data are grids of Iberia and Europe. Each grid has 401 x 401 x 35 points. The figure also shows these values for CPMD application executed on 32 and 64 processors. CPMD has also been executed on 32 and 64 processors using a technique which avoids load imbalance (Taskgroups). The figure also shows the speedups derived from the model. In all the cases, the model fits correctly the values. 91

4.6 Values of the parameters of the model obtained from the executions of WRF-NMM and WRF-ARW on 128, 256 and 512 processors. Input data are grids of Iberia and Europe. Each grid has 401 x 401 x 35 points. The figure also shows these values for CPMD application executed on 32 and 64 processors. CPMD has also been executed on 32 and 64 processors using a technique which avoids load imbalance (Taskgroups). 92

4.7 Representation of communications overlapped with computations, that is, a visualization of a simulation under ideal communications. The applications are WRF-ARW executed on 128 and 512 (using an Europe's Grid as input) and WRF-NMM executed on 128 and 512 processors (using an Iberia's Grid). 92

4.8 Graphical representation of the scheme of the prediction increasing the number of processors: First, we perform executions of the application on P1, P2 and P3 processors. For each execution, we extract a set of parameters. From these sets, we predict a new set of parameters. This new set belongs to a predicted execution of the application on P4 processors. Finally, we extract the execution time from the analytical model applied to the predicted parameters. 94

4.9 Predicted NAS-SP speedups using several techniques. Real speedup on MareNostrum is also shown in the picture. 95

4.10 Relative error predicting NAS-SP speedup on MareNostrum using several techniques. Results are shown for 121, 256 and 1024 processors. 95

4.11 Predicted NAS-MG speedups using several techniques. Real execution on MareNostrum speedup is also shown. 95

4.12 Relative error predicting NAS-MG speedup on MareNostrum using several techniques. Results are shown for 128, 256 and 1024 processors. 95

4.13 Predicted NAS-BT speedups using several techniques. Real speedup on MareNostrum is also shown. 96

4.14 Relative error predicting NAS-BT speedup on MareNostrum using several techniques. Results are shown for 121, 256 and 1024 processors. 96

4.15 NAS-SP. Parameters of the Analytical Model explained above. They have been extracted from real executions on MareNostrum. 97

4.16 NAS-SP. Parameters of the Analytical Model predicted from real executions on 16, 36 and 64 processors on Marenostrum. 97

4.17 NAS-MG. Parameters of the Analytical Model explained above. They have been extracted from real executions on MareNostrum. 97

4.18 NAS-MG. Parameters of the Analytical Model predicted from real executions on 16, 32 and 64 processors on Marenostrum. 97

4.19 NAS-BT. Parameters of the Analytical Model explained above. They have been extracted from real executions on MareNostrum. 97

4.20 NAS-BT. Parameters of the Analytical Model predicted from real executions on 16, 36 and 64 processors on Marenostrum. 97

5.1 Average relative errors of the minimal representative subsets of SPEC CPU 2000 benchmarks. Five different metrics have been used to compute errors. . 107

5.2 **On the left we have a global view of the evolution of the CPI during the whole execution of applu application. We can see a large number of oscillations during the execution. On the right, we see a zoom of these oscillations. They are periodical.** 107

5.3 **On the left we have a global view of the evolution of the CPI during the whole execution of perlbmk application. On the right, we see a zoom. In this case, the whole execution is mainly under a non-periodic behavior.** 109

5.4 **CPI signal extracted from an execution segment of the applu application. We also show samplings using 3 different lengths. The first one, *Sampling A*, is performed each $3 \cdot 10^8$ millions of instructions, the second, *Sampling B*, is performed each $5.1 \cdot 10^8$ and the third is done each $8 \cdot 10^8$. If a sampling point has the coordinates (x, y), it means that y is the average value of cycles executed each ten million of instructions within the execution interval $[x - length, x]$. For example, if (x, y) belongs to *Sampling A*, it summarizes the execution interval $[x - 3 \cdot 10^8, x]$.** 113

5.5 **Applu application: Errors obtained using techniques inspired on Multiple SimPoints, Single SimPoints and Large SimPoints taking several sampling interval lengths. Best results are obtained near $5.1 \cdot 10^8$ and $1.02 \cdot 10^9$. These are the values we showed in table 5.1 when we were studying the length of the minimal representative subset and two times this value.** 115

5.6 **Fma3d application: Errors obtained using techniques inspired on Multiple SimPoints, Single SimPoints and Large SimPoints taking several sampling interval lengths. Best results are obtained near $6.1 \cdot 10^8$ and $1.22 \cdot 10^9$. These are the values we showed in table 5.1 when we were studying the length of the minimal representative subset and two times this value.** 116

5.7 **Lucas application: Errors obtained using techniques inspired on Multiple SimPoints, Single SimPoints and Large SimPoints on Lucas application and taking several sampling interval lengths. Best results are obtained near $1.02 \cdot 10^9$ and $2.04 \cdot 10^9$. These are the values we showed in table 5.1 when we were studying the length of the minimal representative subset and two times this value.** 116

5.8 **Swim application: Errors obtained using techniques inspired on Multiple SimPoints, Single SimPoints and Large SimPoints on Swim application and taking several sampling interval lengths. Best results are obtained near $2.6 \cdot 10^8$ and $5.2 \cdot 10^9$. These are the values we showed in table 5.1 when we were studying the length of the minimal representative subset and two times this value.** 117

5.9 **Mgrid application: Results using techniques inspired on Multiple SimPoints, Single SimPoints and Large SimPoints techniques taking several sampling interval lengths. Best results are obtained near $4.8 \cdot 10^8$ and $9.6 \cdot 10^8$. These are the values we showed in table 5.1 when we were studying the length of the minimal representative subset and two times this value.** 117

5.10 **Extraction of the predictions of global values of hardware counters. Counter 1 is the number of floating point operations performed, counter 2 is the number of TLB misses and the third is the number branch mispredictions.** 120

5.11 **Time focused on extract predictions of three independent hardware counters (reduced time) and total execution time of applications. To extract the global value of each incompatible hardware counter we need one complete execution.**120

5.12 **Relative errors of the predictions of global values of hardware counters obtained using our methodology based on spectral analysis (wavelet transform, ...). Note that the average error is around 15% and the maximal error is around 25%.** 120

Bibliography

[1] Advanced Research WRF (ARW) core of the Weather Research and Forecasting system. *http://www.mmm.ucar.edu/wrf/users/*

[2] Badia, R. M.; Labarta, J.; Sirvent, R.; Perez, J. M.; Cela, J. M.; Grima, R.; "Programming grid applications with GRID Superscalar", *Journal of Grid Computing*, Volume 1, Issue 2, pages 151-170, 2003.

[3] Bailey, D.; Barszcz, E.; Barton, J.; Browning, D.; Carter, R.; Dagum, L.; Fatoohi, R.; Fineberg, S.; Frederickson, P.; Lasinski, T.; Schreiber, R.; Simon, H.; Venkatakrishnan, V.; , Weeratunga, S.; "The NAS Parallel Benchmarks", *RNR Technical Report RNR-94-007*, 1994.

[4] Barker, K.; Davis, K.; Hoisie, A.; Kerbyson, D.; Lang, M.; Pakin, S.; Sancho, J.C.; "Experiences in scaling scientific applications on current-generation quad-core processors" *Proceedings of IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1-8, 2008.

[5] Baysal, E.; Kosloff, D. D.; Sherwood, J. W. C. "Reverse time migration", *Geophysics* vol 48, pages 1514-1524, 1983.

[6] Becker, D.; Wolf, F.; Frings, W.; Geimer, M.; Wylie, B.J.N; Mohr, B. "Automatic Trace-Based Performance Analysis of Metacomputing Applications" *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1-10, 2007.

[7] Bell, G.; Gray, J.; Szalay, A.; "Petascale computational systems", *Computer* Volume 39, Issue 1, pages 110-112, 2006.

[8] Bracewell, R; Kahn, P. B.; "The Fourier Transform and Its Applications" *American Journal of Physics* Volume 34, Issue 8, pages 712-712, 1966.

[9] Brunst, H.; Kranzlmuller, D.; Nagel, W. E.; ”Tools for Scalable Parallel Program Analysis - Vampir VNG and DeWiz”, *Distributed and Parallel Systems*, pages 93-102, 2004.

[10] Burrus, C. S.; Gopinath, R. A.; Guo, H.; *Introduction to Wavelets and Wavelet Transform* Prentice-Hall, 1998.

[11] Carrington, L.; Wolter, N.; Snavely, A.; ”A Framework to For Application Performance Prediction to Enable Scalability Understanding”, *Scaling to New Heights Workshop*, 2002.

[12] Casas, M.; Badia, R. M.; Labarta, J.; ”Automatic Extraction of Structure of MPI Applications Tracefiles”, *Proceedings of the European Conference on Parallel Computing (Euro-Par)*, pages 3-12, 2007.

[13] Casas, M.; Badia, R. M.; Labarta, J.; ” Automatic Phase Detection of MPI Applications”,*Parallel Computing: Architectures, Algorithms and Applications (ParCo)*, Volume 15, pages 129-136, 2007.

[14] Casas, M; Badia, R. M.; Labarta, J.; ”Automatic analysis of speedup of MPI applications”, *Proceedings of the 22nd ACM International Conference on Supercomputing (ICS)*, pages 349-358, 2008.

[15] Casas, M; Badia, R. M.; Labarta, J.; ”Prediction of Behavior of MPI Applications”, *Proceedings of the 2008 IEEE International Conference on Cluster Computing (Cluster)*, pages 242-251, 2008.

[16] Caubet, J.; Gimenez, J.; Labarta, J.; DeRose, L.; Vetter, J.; ”A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications” *OpenMP Shared Memory Parallel Programming*, pages 53-67, 2001.

[17] Conte, T. M.; Hirsch M. A.; Menezes K. M.; ”Reducing State Loss for Effective Trace Sampling of Superscalar Processors”, *Proceedings of the 1996 International Conference on Computer Design (ICCD)*, pages 468-477, 1996.

[18] CPMD code: A parallelized plane wave/pseudopotential implementation of Density Functional Theory, particularly designed for ab-initio molecular dynamics. *http://www.cpmd.org*

[19] CPMD manual, page number 137. *http://www.cpmd.org/cpmd_manual.html*

[20] Culler, D.; Karp, R.; Patterson, D.; Sahay, A.; Schauser, K. E.; Santos, E.; Subramonian, R.; Eicken, T. V.; ”LogP: Towards a Realistic Model of Parallel Computation”, *Proceedings of 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 1-12, 1993.

[21] Curtis-Maury, M.; Shah, A,; Blagojevic, F.; Nikolopoulos, D. S.; De Supinsky, B. R.; Schulz, M; ”Prediction Models for Multi-dimensional Power-Performance Optimization on Many Cores”, *Proceedings of Parallel Architectures and Compilation Techniques (PACT)*, pages 250-259, 2008.

[22] Daubechies, I.; ”The Wavelet Transform, time-frequency localization and signal analysis”, *IEEE Transactions on Information Theory*, Vol. 36, No. 5, pages 961-1005, 1990

[23] Daubechies, I.; *Ten Lectures on Waveletes*, SIAM: Society for Industrial and Applied Mathematics, 1992.

[24] De Chevigne, A.; Kawahara, H.; ”YIN, a fundamental frequency estimator for speech and music”, *Journal of Acoustical Society of America*, volume 111, pages 1917-1930, 2002.

[25] Eeckhout, L.; Nussbaum, S.; Smith, J. E.; De Bosschere, K.; ”Statistical Simulation: Adding Efficiency to the Computer Designer’s Toolbox”, *Proceedings IEEE Micro*, volume 23, number 5, pages 26-38, 2003.

[26] Freitag, F.; Corbalán, J.; Labarta, J.; ”A Dynamic Periodicity Detector: Application to Speedup Computation”. *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 6-15, 2001.

[27] Gamblin, T.; De Supinski, B. R.; Schulz, M.; Fowler, R. J.; Reed, D. A.; ”Scalable load-balance measurement for SPMD codes”, *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC 2008.

[28] Geer, D. ”Chip makers turn to multicore processors”, *IEEE Computer*, Vol. 38, No. 5., pages 11-13, 2005

[29] Geimer, M.; Wolf, F.; Wylie, B. J. N.; Mohr, B. ”Scalable Parallel Trace-Based Performance Analysis” *Proceedings of the 13th European Parallel Virtual Machine and Message Passing Interface Conference (Euro PVM/MPI)*, pages 303-312, 2006.

[30] Gerndt, M., Kereku, E.; ”Automatic Memory Access Analysis with Periscope”, *Proceedings of the International Conference on Computational Science (ICCS)*, LNCS 4488, pages 847-854, 2007.

[31] Girona, S., Labarta, J., Badia, R. M.; ”Validation of Dimemas communication model for MPI collective operations”, *Proceedings of the 7th European Parallel Virtual Machine and Message Passing Interface Conference (Euro PVM/MPI)*, pages 39-46, 2000.

[32] Girona, S.; Labarta, J.; ”Sensitivity of Performance Prediction of Message Passing Programs”. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 620-626, 1999.

[33] González, J. A.; Rodríguez, C.; Roda, J. L.; González-Morales, D.; Sande, F.; Almeida, F.; Léon, C.; ”Performance and Predictability of MPI and BSP Programs on the CRAY T3E”, *Proceedings of the 6th European Parallel Virtual Machine and Message Passing Interface Conference (Euro PVM/MPI)*, pages 27-34, 1999.

[34] Grobelny, E.; Bueno, D.; Troxel, I.; George, A.; Vetter, J.; ”FASE: A Framework for Scalable Performance Prediction of HPC Systems and Applications”, *Simulation: Trans. Soc. Model. Simul. Int. 83, 10*, pages 721-745, 2007.

[35] Grossmann, A.; Morlet, J.; ”Decomposition of Hardy functions into square-integrable wavelets of constant shape”, *SIAM Journal of Mathematical Analysis*, 723-736, 1984.

[36] Halfhill, T. T.; ”Parallel Processing with CUDA”, *Nvidia's High-Performance Computing Platform Uses Massive Multithreading*. Microprocessor Report. Reed Electronics Group, 2008

[37] Hastie, T.; Tibshirani, R.; Friedman, J.; *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Springer Series in Statistics, 2001.

[38] Hillis, W. D.; Steele, G., L.; *Data Parallel Algorithms* Communications of the ACM, 1986.

[39] Hogg, R.; *Probability and Statistical Inference*, 7th ed., Pearson, 2006.

[40] Houzeaux, G.; Eguzkitza, B.; Vázquez, M; ”A variational multiscale model for the advection-diffusion-reaction equation” *Communications in Numerical Methods in Engineering*, 2007.

[41] Hoyas, S.; Jimeńez, J. ”Scaling of velocity fluctuations in turbulent channels up to Re=2003”, *Physics of Fluids*, volume 18, pages 351-356, 2006.

[42] Huang, M.; Renau, J.; Torrellas, J.; ”Positional adaptation of processors: application to energy reduction” *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 157-168, 2003.

[43] Huffmire, T.; Sherwood, T.; ”Wavelet-Based Phase Classification”, *Proceedings of Parallel Architectures and Compilation Techniques (PACT)*, pages 95-104, 2006.

[44] Jost, G.; Labarta, J.; Gimenez, J.; ”Paramedir: A Tool for Programmable Performance Analysis” *Lecture Notes in Computer Science*, Vol.3036, pages. 466-469, Springer, 2004.

[45] Kaiser, G.; *A Friendly Guide to Wavelets*, Birkhauser, 1994.

[46] Kerbyson, D. J.; Wasserman, H. J.; Hoisie, A.; ”Exploring Advanced Architectures Using Performance Prediction”, *Proceedings of the 2002 International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA)*, p.27, 2002.

[47] Knuepfer, A.; Brunst, H.; Nagel, W. E.; ”High Performance Event Trace Visualization”, *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 258-263, 2005.

[48] Komatitsch, D.; Ritsema, J.; and Tromp, J.; ”The spectral-element method, Beowulf computing and global seismology”, *Science, 298*, pages 1737-1742, 2002.

[49] KOJAK: Kit for Objective Judgment and Knowledge-based Detection of Performance Bottlenecks, *http://www.fz-juelich.de/zam/kojak/*

[50] KleinOsowski, A.; Lilja D. J., ”MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research”, *Computer Architecture Letters*, Volume 1, page 7, 2002.

[51] Kranzlmuller, D.; Scarpa, M.; Volkert, J.; ”DeWiz - A Modular tool Architecture for Parallel Program Analysis”, *Proceedings of the European Conference on Parallel Computing (Euro-Par)*, pages 74-80, 2003.

[52] Labarta, J., Girona, S., Pillet, V., Cortes, T., Gregoris, L.; ”DiP : A Parallel Program Development Environment”, *Proceedings of the European Conference on Parallel Computing (Euro-Par)*, pages 665-674, 1996.

[53] Lee, B. C.; Brooks, D. M.; De Supinski, B. R.; Schulz, M.; Singh, K.; McKee, S.; ”Methods of inference and learning for performance modeling of parallel applications”, *Proceedings of 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 249-258, 2007.

[54] Linpack benchmark: *http://www.netlib.org/linpack/*

[55] MacQueen, J. B.; "Some Methods for classification and Analysis of Multivariate Observations", *Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281-298, 1967.

[56] Mallat, S., G.; *A Wavelet Tour on Signal Processing*, Elsevier, 1999.

[57] Marenostrum Supercomputer: *http://www.bsc.es*

[58] Message Passing Interface (MPI) standard, *http://www-unix.mcs.anl.gov/mpi/*

[59] Mpitrace libraries:
http://www.hpcx.ac.uk/support/documentation/IBMdocuments/mpitrace

[60] Mohr, B.; Traff, J. L.; "Initial Design of a Test Suite for Automatic Performance Analysis Tools". *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 131-140, 2003.

[61] Muller, M.S., Knupfer, A., Jurenz, M., Lieber, M., Brunst, H., Mix, H., Nagel, W.E. "Developing scalable applications with Vampir, VampirServer and VampirTrace" *Parallel Computing: Architectures, Algorithms and Applications (ParCo)* vol. 15, pages 637-644, 2007.

[62] Nagel, W., Weber, M., Hoppe, H.C., Solchenbach, K. "VAMPIR: Visualization and Analysis of MPI Resources." *Supercomputer* volume 63, pages 69-80, 1996.

[63] Nataraj, A.; Malony A.; Shende, S.; Morris, A.; "Kernel-Level Measurement for Integrated Parallel Performance Views: the KTAU Project", *Proceedings of the 2006 IEEE International Conference on Cluster Computing (Cluster)*, pages 1-12, 2006.

[64] Nonhydrostatic Mesoscale Model (NMM) core of the Weather Research and Forecasting (WRF) system. *http://www.dtcenter.org/wrf-nmm/users/*

[65] Nyquist, H.; "Certain topics in telegraph transmission theory", *AIEE Transactions,* vol. 47, pages 617-. 644, 1928

[66] Papi User's Guide
http://icl.cs.utk.edu/projects/papi/files/documentation/PAPI_USER_GUIDE.pdf

[67] OMPItrace manual, *http:www.cepba.upc.es/paraver/docs/OMPItrace.pdf*

[68] OpenMP Application Program Interface: *http://openmp.org/wp/*

[69] Papoulis, A. *The Fourier Integral and Its Applications*. New York: McGraw-Hill, pages 244-245 and 252-253, 1962.

[70] Pencek, B.; ”Reconfigurable Application-Specific Computing: How Hybrid Computer Systems using FPGAs are Changing Signal Processing”, *Industrial Embedded Systems*, 2009

[71] Perelman E.; Hamerly, G.; Calder, B. ”Picking Statistically Valid and Early Simulation Points” *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques, (PACT)*, pages 244-255, 2003.

[72] Preissl, R.; Kockerbauer, T.; Schulz, M.; Kranzlmuller, D.; Supinski, B.; Quinlan, D.J.; ”Detecting Patterns in MPI Communication Traces”, *Proceedings of the 37th International Conference on Parallel Processing, (ICPP)*, pages 230-237, 2008.

[73] Press, W. H.; Flannery, B. P.; Teukolsky, S. A.; Vetterling, W. T.; *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, 2nd ed. Cambridge, England: Cambridge University Press, pages 538-539, 1992.

[74] Rodríguez, G.; Badia, R. M.; Labarta, J.; ”Generation of Simple Analytical Models for Message Passing Applications” *Proceedings of European Conference on Parallel Processing (Euro-Par)*, pages 183-188, 2004.

[75] Roth, P.C., Miller, B.P.: ”On-line automated performance diagnosis on thousands of processes” *Proceedings of 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (PPoPP)*, pages 69-80, 2006.

[76] Saxena, V.; Hwang, C. W.; Huang, S.; Eichbaum, Q., Ingber, D., Orgill, Dennis P. ”Vacuum-Assisted Closure: Microdeformations of Wounds and Cell Proliferation”, *Plastic and Reconstructive Surgery*, Volume 114, pages 1086-1096, 2004.

[77] SCPUs User’s Guide *http://www.bsc.es/media/1388.pdf*

[78] Serra, J. *Image Analysis and Mathematical Morphology*, Academic Press, 1982.

[79] Servat, H.; Llort, G.; Gimnez, J.; Labarta, J.; ”Detailed performance analysis using coarse grain sampling”, *Research report UPC-DAC-RR-2008-34*

[80] Shannon, C. E.; ”Communication in the presence of noise”, *Proc. Institute of Radio Engineers*, vol. 37, no.1, pages 10-21, 1949

[81] Shen, X.; Zhong, Y.; Ding, C.; ”Locality Phase Prediction”, *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, (ASPLOS),*, pages 165-176, 2004.

[82] Shende, S. S.; Malony, A. D.; ”The Tau Parallel Performance System”, *International Journal of High Performance Computing Applications*, Vol. 20, No. 2, pages 287-311, 2006.

[83] Sherwood, T.; Perelman, E.; Hamerly, G.; Calder, B.; ”Automatically Characterizing Large Scale Program Behavior” *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, (ASPLOS)*, pages 45-57, 2002.

[84] Sherwood, T.; Perelman, E.; Hamerly, G.; Sair, S; Calder, B.; ”Discovering and Exploiting Program Phases” *IEEE Micro: Micro's Top Picks from Computer Architecture Conferences*, pages 84-93, 2003.

[85] Roadrunner *http://www.lanl.gov/roadrunner/rrposterslanlbooth.shtml*

[86] Simon, B.; Odom, J.; DeRose, L.; Ekanadham, K.; Hollingsworth, J. K.; Sbaraglia, S.; ”Using Dynamic Tracing Sampling to Measure Long Running Programs”, *Proceedings of the ACM/IEEE International Conference on Supercomputing (SC)*, pages 59-68, 2005.

[87] Simon, J.; Wierum, J. M. ”Accurate performance prediction for massively parallel systems and its application”, *Proceedings of European Conference on Parallel Processing (Euro-Par)* pages 675-688, 1996.

[88] Smith, J. E.; Dhodapkar, A. S., ”Dynamic microarchtecture adaptation via co-designed virtual machines” *Proceedings of the International Solid State Circuits Conference (ISSC)*, 2002.

[89] Snavely, A.; Wolter, N.; Carrington, L.; ”Modeling Application Performance by convolving Machine Signatures with Application Profiles”, *Proceedings of the IEEE 4th International Workshop on Workload Characterization (WWC)*, pages 149-156, 2001.

[90] Song, F.; Wolf, F.; ”CUBE User Manual” I*CL Technical Report. ICL-UT- 04-01.* 2004.

[91] Springel V., Yoshida N., White S. D. M., ”Gadget: a code for collisionless and gas-dynamical cosmological simulations” *New Astronomy*, Volume 6, Issue 2, pages 79-117, 2001.

[92] Srivastava, M. S.; Khatri, C. G.; *An introduction to multivariate statistics*, Elsevier, 1979.

[93] Standard Performance Evaluation Corporation *http://www.spec.org/*

[94] Stromberg, J. O.; "Wavelet transforms with the Franklin system and application in image processing", *Proceedings of 21th Nordic Conference on Mathematics*, 1992.

[95] Teysser, R.; "Cosmological hydrodynamics with adaptive mesh refinement - A new high resolution code called RAMSES", *Astronomy & Astrophysics*, volume 385, pages 337-364, 2002.

[96] Trace generation: just some examples *www.bsc.es/plantillaA.php?cat_id=492&pdf=1*

[97] Vetter, J.S.; Worley, P.H.; "Asserting Performance Expectations", *Proceedings of the ACM/IEEE International Conference on Supercomputing, (SC)*, 1-13, 2002.

[98] Viega, J.; Voas, j.; "Can Aspect-Oriented Programming Lead to More Reliable Software", *IEEE Software*, pages 19-21, 2000.

[99] Wagner, R. S.; Baraniuk, R. G.; Du, S.; D. B. Johnson and A. Cohen, "An Architecture for Distributed Wavelet Analysis and Processing in Sensor Networks", *Proceedings of the 5th International Conference on Information Processing in Sensor Networks (IPSN)*, pages 243-250, 2006.

[100] Walnut, D. F.; *An Introduction to Wavelet Analysis*, Birkhauser Boston, 2004.

[101] Wenisch, T. F.; Wunderlich, R. E.; Ferdman, M.; Ailamaki, A.; Falsafi, B.; Hoe, J. E.; "SimFlex: Statistical Sampling of Computer System Simulation," *IEEE Micro*, vol. 26, no. 4, pages 18-31, 2006.

[102] Wheeler, J. A.; Zurek, H.; *Quantum Theory and Measurement*, Princeton University Press, pages 62-84, 1983.

[103] Wunderlich, R. E.; Wenisch, T. F.; Falsafi, B.; Hoe, J. C. "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling", *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 84-95, 2003.

[104] Xprofiler
http://www16.boulder.ibm.com/doc link/enUS/adoclib/aixbman/prftools/xprofiler.htm

[105] Yang, L. T.; Ma, X.; Mueller, F.; "Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution", *Proceedings of the ACM/IEEE International Conference on Supercomputing (SC)*, pages 40-49, 2005.

[106] Yi, J. J;, Sendag, R.; Lilja, D. J.; Hawkins, D. M.; ”Speed versus Accuracy Trade-Offs in Microarchitectural Simulations”, *IEEE Transactions on Computers*, pages 1549-1563, 2007.

[107] Zaki, O.; Lusk, E.; Gropp, W.; Swider, D.; ”Toward Scalable Performance Visualization with Jumpshot” *High-Performance Computing Applications*, volume 13, number 2, pages 277-288, 1999.