Universitat Autònoma de Barcelona

Departament d'Arquitectura de Computadors i Sistemes Operatius

Performance Evaluation of Applications for Heterogeneous Systems by means of Performance Probes

Thesis submitted by **Alexandre Otto Strube** for the degree of Philosophiae Doctor by the Universitat Autònoma de Barcelona, under the supervision of Dr. Emilio Luque

July 15, 2011

Preface

Memoria presentada por Alexandre Otto Strube para optar al grado de Doctor por la Universidad Autónoma de Barcelona. Trabajo realizado en el Departamento de Arquitectura de Computadores y Sistemas Operativos de la Universidad Autonoma de Barcelona dentro del programa de Doctorado en Computación de Altas Prestaciones bajo la dirección del Dr. Emilio Luque Fadón.

"I almost wish I hadn't gone down that rabbit-hole
-and yet-and yet-
it's rather curious, you know, this sort of life!"

— Alice in Wonderland

Acknowledgments

This is a strange part of this work. One is always afraid of saying the old clichès, or of forgetting to mention some person and then, unwittingly, hurt someone's feelings. At least I don't care about the clichès.

Before anything, some unusual acknowledgments: mr Carlito Jensen, a good friend of my dad. Why? Because I have stolen I book of COBOL from his house when I was less than five years old. That book shaped my life like nothing else did. My cousin Ricardo, who started teaching me BASIC when I was 6, and d.Neda, who kept on my education on computers during primary school, when my colleagues were trying to learn how to subtract, and who hired me as a teacher for the computing science classes when I was 12.

Let's get started. First and foremost, Andrea "Dolly" Couto, my old friend, versed in pretty much anything, from languages to art, cooking, photography, sailing, beer and wines to software system analysis. She started the chain of events that made me start in a Ph.D., and without her, I would not be here at all. I don't know if I should thank you or smack you, but I prefer the former. So I pay for the beer!

Then Eduardo, who was of more importance than he probably knows about. Besides his help on my first steps as a researcher, his thesis was the whole base of my work. I try to mimic - mostly unsuccessfully - his discipline and writing style, in the sense of completeness and cohesion. Thanks Edu.

In Computer Science, every work tends to be radically different from the previous one. My personal guru from the very first moment I landed in our lab, Genaro helped me with his seemingly infinite knowledge of everything related to computers and other subjects, and with his good will to everyone. Again, I could not have done this without you, my friend.

Of course I cannot forget to thank Emilio Luque and Dolores "Lola" Rexachs. Your wisdom, patience and care go far beyond research. You are not only my role model as researchers, but as human beings. One day I aspire to be like you.

I cannot forget my sister Fernanda. She has a heart as big as my parents, and

a smile as beautiful as the sky. And she gave me my niece, Gabi, which I see during these lonely and long nights working. She is the piece of hope that this world can be a better place.

As we are in the subject of people I aspire to be like, I cannot forget my parents, Ralf and Nilce. I have no words to explain how I love and admire you. If one day I manage to be half the human being you are, I will be a great person already. Thanks for everything, always.

Pretty much no one is quite sure about the existence of a god, but one thing I know for sure: guardian angels exist. Some of them helped me directly. See, my health has not been the best during these crazy years. But when my heart decided to give a break to tell me to go slower, Hisham was on my side all the way to the hospital. Would I be here if it were not for him? Probably no.

Likewise, when the anaphylaxis hit me like a tsunami, letting me blind and with a start of a cardiac arrest, Katia literally dragged me to the doctors. She helped in so many ways throughout the years that I can only say thank you. I hope you are happy, wherever you are.

In a large journey like this, from an arrogant grad student to a humble Ph.D, you never go alone. Dario, the brother I discovered in this side of the world, always had the shoulder and the ears available when I needed, and for bringing to the world the lovely Aramí. Thanks, thanks a lot. Thanks also Gustavo and Alexandra, my great friends, thanks for your patience with this crazy friend. And thanks Daniel and Jamile. I spent almost as much time in their house as in mine. They made the distance from my family a little more bearable, by putting me into theirs.

Being this a small world, I have had the opportunity to make a friend in so many different places and circles that it is hard to be just coincidence. From the times when we both worked on internet providers, then to the motorcycling world, then to the sailing world, and finally, working side by side 10.000km far from where we met, I must thank Leonardo for being such an example of worker I should be, and of living to the full and surviving to it.

My second family supported me and has set the example of how someone from my generation can be happy and good. Thanks Kathrine, thanks Charles, thanks my godson Oscar, thanks Ricardo.

Thanks Carla, and thanks Luciano, for the wii matches, for all the fun and joy.

I must say thanks, or “multumesc”, to my great friend Alex Prunean. We both have been through difficult times, and you helped me a lot. I hope I did, too.

I cannot forget my colleagues students, some who started with me, some who

were there already, and some who came later: Alvaro Wong, Chris, John, Alex, Pilar, Claudia, Gonzalo, Moni, Ihab, Jairo, Alvaro Chalar, Ronal, Andrès, Tharso and others which deserve to be here as well.

After I left Barcelona, life turned another way, and new people, new experiences entered into my life. I must thanks all the support from my new bosses, Felix and Markus, and from my colleagues and now friends, Pavel, Micha, Claudia, David, and Peter.

Last, but not least, the person who is saw how hard these last steps of the Ph.D. have been, and who is making my days sunnier. Olga, thanks for your patience, your care and your love.

Contents

Abstract

This doctoral Thesis describes a novel way to select the best computer node out of a pool of available potentially heterogeneous computing nodes for the execution of computational tasks. This is a very basic and difficult problem of computer science and computing centres tried to get around it by using only homogeneous compute clusters. Usually this fails as like any technical equipment, clusters get extended, adapted or repaired over time, and you end up with a heterogeneous configuration.

So far, the solution for this, was:

- To leave it to the computer users to choose the right node(s) for execution, or
- To make extensive tests by executing and measuring all tasks on every type of computing node available in the pool. In the typical case, where a large number of tasks would need to be tested on many different types of nodes, this could use a lot of computing resources, sometimes even more than the actual execution one wants to optimize.

In a specific situation (hierarchical multi-clusters), the situation is worse, as the configuration of the cluster changes over time, so that the execution tests would have to be done over and over, every time the configuration of the cluster is changed.

I developed a novel and elegant solution for this problem, named "Performance Probe", or just "Probe", for short. A probe is a striped-down version of a computational task which includes all important characteristics of the original task, but can be executed in a much shorter time (seconds, instead of hours), is much smaller than the original task (about 5% of the original size in the worst cases), but allows to predict the execution time of the original within reasonable bounds (around 90% accuracy).

These results are very important: as scheduling is a basic problem of computer science, these results cannot only be used in the setting described by the thesis (of setting the right compute node for tasks in a hierarchical multi-cluster), but can also be applied in many different contexts every time scheduling and/or selection

decisions have to be made: selecting where a computational task would run most efficiently (which cluster at which centre); picking the right execution nodes in a large complex (grid, cloud), workflows and many more.

Chapter 1

Introduction

Read the directions and directly you will be directed in the right direction.— Doorknob

Applications are the reason people use computers. Different applications use the computers in different ways, and might be better suited to run in one type of computer than another. And yet, as obvious as it should be, to determine the best match between a kind of computer and an application is neither easy nor a solved problem at all.

When applications started to become so demanding that a single computer would not be able to run them in reasonable time, one of the solutions was to join several computers together and divide the problem tackled by these application into smaller pieces, enough for each computer to process them in a reasonable time. The cluster was born.

The end of speed improvements in processor cores by simple clock speed increase made the evolution go towards adding more computers together to address the increasing demand in computing power first, and then adding more processors into a single computer, and subsequently adding more cores into a single dye. A personal computer of today is not so different from a cluster from yesterday.

Increasing the number of computing elements does not bring automatic improvements to applications. Applications must be either adapted to conform to this new model or rewritten.

One of the most popular models to make an application avail the extra computing power provided by multiple computers running together in a network is the Master/Worker paradigm. A simple definition of it is that a "master" machine assigns tasks to "workers", which receives them, compute and return the final result of this task back to the master. The master will give the workers more tasks until the complete problem is solved, and then the master aggregates the results of all

completed tasks in form of the final result. It is widely used because of its simplicity, reliability and elasticity.

An economic-wise form of having huge computing power is that of clusters of commodity workstations. They are cheap and in case of failures or upgrades, it is easy to replace them individually. It is a system widely used in academia, for instance, when laboratories might provide the workstations for normal use during the day for students and let them available for parallel applications during non-working hours.

The replacement cycle of these clusters makes them heterogeneous throughout time. And as already said, different applications perform differently according to the computer where they are running. The heterogeneity also introduces inefficiencies into the scheduling, which can be seen as times when the workers stay idle waiting for more data.

This makes the decision of choosing which machine should run a given task more difficult.

Clusters of workstations cannot grow indefinitely, be it because of space constraints, network addressing or even power consumption. Therefore, in order to keep increasing the computing power to handle ever-increasing problems, one of the solutions is to add another level in the distribution, by joining multiple clusters of workstations together. We call this extra level "multi-cluster". The Grid and the Cloud are specializations of this concept.

The extra level of organization leads to differences in network performances: each cluster has its own high-speed network, but communication between them is not necessarily (and probably never is) the same. The heterogeneity between machines is even more probable.

It is worth noting that we call heterogeneous machines and clusters in terms of internal architecture or computational power. It does *not* mean heterogeneity such as that mentioned in other papers related to GPU or accelerators, for instance.

One form of organizing a multi-cluster running a master-worker application is hierarchically. Similarly as a single cluster running a master-worker application, one element acts as a master, while others act as workers. On this case, a master cluster has not only its own machines to send tasks, but also other clusters. These clusters can be seen by the master as machines capable of processing a lot of tasks, but with a higher time required to send tasks to them. The master cluster sends the tasks to a worker, or sub-cluster through an entity called Communication Manager, responsible only for taking care of the communication between a master cluster and its sub-clusters. On the sub-cluster there is another Communication Manager, which receives the tasks and send to a local master, which finally assigns them to

individual computing units or even to sub-clusters. This sub-master joins together the results of the tasks under its responsibility, and sends them back to the master cluster, again using the Communication Manager.

The hierarchical multi-cluster is a powerful tool to handle big problems with advantages akin to that of a cluster of workstations: it is relatively cheap, simple to implement, easy to grow, and simple to replace or disable individual elements (or even whole clusters).

Ideally, machines should be computing as much as they possibly could, with no time lost by waiting between the end of one task and the beginning of the next one. In order to achieve this, the communications should occur while the workers are computing. So while the worker is computing its task, it should be already receiving the next one, effectively "hiding" or "masking" the communication time.

Easier said than done. To be able to mask communications, the master machine assigning all tasks must have knowledge of two things:

- The time required to send a given task to each of them. The master should send tasks early enough that a worker wastes no time receiving it, but not so early that in the case of a malfunction or a slow machine that unbalances this scheduling, and, of course,
- the computing power of each machine while running this specific application.

These two parameters, network and machine performance are enough for a scheduler to optimize the execution of an application in this hierarchical multi-cluster environment. The relationship CPU power/Network bandwidth gives the scheduler knowledge enough to send more tasks to a given machine than another, less tasks, or even discard a machine (or a whole cluster) completely, either because the machines are so slow that they does not help at all, or does not help on the efficiency threshold set, or because the time required to send tasks to them might delay the whole execution. The model is able to adapt to changing network conditions throughout executions, and with our Probes, also is able to quickly characterize nodes that might become available during the execution.

It is trivial to calculate network performance by just sending data from one machine to another and calculate the time required to send it. But to determine the performance of a computer while running an application is a harder task.

Performance characterization and prediction has been a subject in Computer Science since its inception, and it is an issue far from solved, given the very nature of applications and how they run on different computers. Compilers, run-time environments, abstraction layers and virtual machines (such as 1960's VM, 1970's CP/M, 1990's Java and modern full-OS VMs) made the matters even more difficult.

One of the issues of performance characterization has to do with the time taken to run an application. If an application is important enough to be characterized, it probably also tends to have a large execution time, with orders of magnitude from hours to months, and will be run thousands of times, and these are applications that run several times.

The other issue is that different machines behave differently, even though they are computationally equivalent (or ISA-equivalent) - that is, with the same inputs and the same application, they give the same results, even though the internals are not identical. Even when machines are apparently the same, smaller revisions of processors or memory controllers can add to heterogeneity.

1.1 Some history

To deal with the issue of execution time, computer scientists came up with the idea of benchmarks. Some of them were totally artificial, created to stress specific components of the computers, while others are kernels of "real" applications with some fixed inputs, intended reproduce this application's behavior during a short time.

Both kinds of benchmarks can be useful in some contexts - they give people a way to compare different computers. The Linpack benchmark [18], for instance, is the official standard of the Top500 supercomputer ranking [1], and has been used for decades as such.

However, benchmarks, as any computer program, only do what they are told to, and nothing more. The aforementioned Linpack, for instance, criticized for not fulfilling its basic purpose of real comparisons between modern supercomputers, as it does not takes into account the ever-increasing problem of memory-bound codes [90], [77], not being representative even of its own field of dense linear algebra applications [26], does not represent behavior of applications in general [75], and have a much higher peak efficiency than real applications [76]. In fact, doubling the Linpack index can give real applications a speedup of only around 10% [53]. So we can affirm that benchmarks are not useful, *per se*, as valid tools for performance characterization of real applications on computers.

There is research about using micro benchmarks to stress specific computer components [21], as they are relatively simple and can be highly optimized for specific uses. Some researchers also join multiple ones together in order to try to correlate their performances to that of real applications [39], [43], [64], [40], or even general characteristics of programs [36].

If a computer system is going to be used for a very specific application, there

is no benchmark better than running the application itself. However, suppose the following scenario: a company has this very specific application, that takes weeks to execute every single time. The process of evaluating new equipment to run this program should evidently stress the characteristics necessary for this specific execution. With multiple options available, the production of performance indexes can take a time so long that it might be impossible to do it.

There are other scenarios where a precise and quick knowledge of one application's performance is desired, as the multi-cluster mentioned before, and that is why we pursue the goal of characterizing the performance of a program quickly.

1.2 Objectives

The objective of this work is to create a way to characterize and predict the performance of one specific part of a parallel master-worker application: the worker, as accurately and quickly as possible.

The quality of this prediction must be as close to the application itself as possible, to better inform the scheduler about the performance of each machine available in the multi-cluster system. A scheduler must be fast, which means that this characterization must be quick, on the order of seconds.

As the multi-cluster environment usually uses slow networks as its interconnect, amount of data required to perform this characterization must be reasonable, to be transferred quickly.

These requirements made us create a methodology to analyze an application's execution, extract its relevant behavior, shrink it to its bare minimum and build a Probe from it to be sent to the remote machines.

1.3 Proposal and Outcomes

We proposed the creation of a methodology that is able to take a parallel application's worker - that is, its serial component, extract its performance characteristics - which means, the phases able to characterize performance, and create a Probe, able to be sent to remote clusters and quickly characterize the performance of every computing node available.

This Probe must have a high prediction quality related to the execution time of the program it originated from, but with a much shorter execution time than it.

The Probe must also be small enough to be transported through slower non-

local networks, such as the internet, in a reasonable time.

Our method basically analyzes the behavior of an execution of the program to be characterized, extracts the non-repetitive behavior from the application and is able to run only the unique behavior, with the repetitions being extrapolatable from them.

I think this proposal was reasonably fulfilled. Our Probe methodology is able to:

- run entirely in three orders of magnitude less time than running the application itself,
- predict execution time of the application with an average precision of over 90%, and
- be transported through the internet in matter of seconds.

1.4 Related Work

The work of Sherwood, Sair and Calder [66] is one of the foundations of this research. They created a methodology to find repetitive behavior on applications so they are able to reduce run-time for benchmarking future processors on functional simulators.

Further, Sherwood, Sair and Calder [66] proposed a profiling technique able to understand an execution as a series of different phases that may repeat.

A work related to ours is that of Sodhi and Subhlok [73], where their performance skeletons intend to mimic application behavior in a shorter execution time for evaluating shared resources using a node selection algorithm [81]. While they focus on shared resources and network usage, we focus this work on computation for master/worker applications by characterizing the worker. This happens for two reasons: (a) it is the model most widely used by cloud and grid environments because of the easy 'elasticity' of it, and (b) our research group already have ongoing work on characterizing communication patterns, such as the study conducted by Wong, Luque and Rexachs [91]. Besides, the general workflow paradigm is composed of nodes in graphs that only communicates in the beginning and the end. In this sense, a node in a directed acyclic graph (DAG) is no different from a worker. [72] [74]

The work of Weaver and McKee [87] developed a tool that runs under the QEMU emulator or the Valgrind as a method to gather multi-platform basic block vectors faster than when using functional simulators, but to use them inside those

functional simulators. Our work instruments applications in real machines, for characterization of the applications running on those real environments, not under simulators, where some facilities for characterization are present that we do not have, as instrumentation that does not change the execution, instruction counters external to the environments and so on.

The absence of good, fast data about execution time of applications on specific computing nodes makes experiments on real environments difficult, and as consequence, several experiments on grid scheduling policies are made by simulating the grid environment instead of performing the experiments on real ones. For instance, [42] performed simulated experiments where individual nodes' performance is a random value in a range about the average node. It suggests that in real environments, these values would be given by historical data, also

The work of Wanek, Schikuta and Ul Haq [85] states that their performance values are "declared by the service providers", but also suggests that they are probably based on historical executions.

The analytical model of Topcuouglu, Hariri and Wu [84] mentions (and uses) an estimated execution time, but does not state how this estimation is obtained.

There are frameworks for performance modelling and prediction, but they rely on characteristics of simulators, like the research of Snavely, et. al. [70], [71].

The work of Reis et. al. [62] could benefit of our work as well. When artificially injecting transitory faults in applications during fault detection experiments, the number of such experiments may reach the number of millions. And that means running an application thoroughly on each of those experiments. Instead of running the application to the end, one could inject transitory faults (bit flip, register value change, etc) only on those parts of the application considered to be the most relevant to those experiments and cut the experimentation time. Our proposal of a Software Probe is able to run only a relevant phase of an application, which can be adapted to that purpose.

Some of the works that can directly benefit from our research, for instance, is a system called Adaps, by Glasner and Volkert [23]. It states that it is possible to use our probe methodology to create a predictor for their system. The probes are able to provide a prediction of application execution time for their general scheme of run-time application forecasts.

1.5 Work Organization

This manuscript is organized as follows. In chapter 2, the environment and programming model is described. Chapter 3 sets the theoretical basis for the model. In 4 the application characterization method and the Probe construction is described, where chapter 5 goes a step further, into the description of methods to reduce Probes' sizes. Chapter 6 is the experiments and results chapter, and this work is finished in chapter 7.

Chapter 2

The multi-cluster environment

Let me see: four times five is twelve, and four times six is thirteen, and four times seven is – oh dear! I shall never get to twenty at that rate! — Alice

2.1 Introduction

For decades, the advances in raw computing processing power has been given by faster chips, with more transistors per chip, and faster clock speeds. The number of transistors per chip approximately doubled every 18 months, which was predicted by Intel's co-founder Gordon Moore in 1965. Since then, this computing power exponential increase has been known as Moore's Law.

2.2 The free ride

Programmers, however, called it differently: "the free ride" [82]. The simplest way to make your program faster was to simply wait for the next generation of processors, and with no additional effort from the programmer, her application would run faster, "for free".

Even though algorithms able to process data in parallel exists for years [69], programmers kept taking advantage of the free ride for as long as possible - which makes sense. Why bother in learning new (and recognizedly difficult) techniques that most computers would not be able to extract any advantage at all?

The exponential growth of computing power, however, reached two related obstacles: power consumption and heat dissipation. The more power a processor uses, the more heat it generates. The added heat needs more dissipation, which in

turn also consumes more power.

2.3 Clusters of workstations

With the advent of local-area networks, a form of computing became readily and, most important, cheaply available: the clusters of workstations (COWs). Linking together several computers into a network and making them work towards the solution of a single problem is one of the most cost-effective ways to tackle problems bigger than the capacity of a single computer.

Scientific computing has changed a lot in the last decades. From the vector computers of the past, to personal academic workstations, the current trend is to aggregate a big number of processing units by means of one or more interconnects, in the form of clusters. Today, every supercomputer is built around the principle of aggregating a number of CPUs ranging from the dozens to hundreds of thousands, each with relatively small computing power, where applications run in parallel in all these processing units, in order to achieve a single goal, be them highly specialized machines, or if they are small clusters of standard workstations interconnected by ethernet networks.

Most of highly specialized machines, such as the modern supercomputers, uses proprietary interconnects, homogeneous CPUs and its own I/O, cooling systems, etc. On the other hand, clusters of standard workstations tend to be heterogeneous by nature, even when the machines share a common ISA and operating system - be it by the multiplicity of vendors and platforms, be it by the natural replacement cycle of machines. Heterogeneity, even in single clusters, brings hindrances to parallel programming: scheduling, load balancing, domain decomposition, processor selection, as mentioned by [15].

2.4 The master/worker paradigm

The textbook example of parallel programming is the master/worker paradigm. On it, the computation and the data are divided in tasks that can be executed independently and simultaneously on different computing elements.

On it, two distinct entities exist: one master and some workers. The master is responsible for decomposing a problem into tasks, and for distributing these tasks among its workers. Also of masters's responsibility is to gather the results from these workers and from them, generate the program's final output.

The workers operate cyclicly: they receive the task, process it and the result

back to the master, and wait for the next task.

The great advantage of the master/worker paradigm is that it is well-suited for dynamic, heterogeneous environments, where adaptability, reliability, capability and efficiency are required [25].

- Adaptability, or elasticity: the same program can be solved in different environments, with different number of workers, and even with changing number of workers available, when allowed by the runtime. The performance increase can be made by adding more workers;
- Reliability: the loss of one of more workers during an execution affects only the execution of the task being computed by the element lost. The overall functioning is not affected;
- Capability: more powerful computing elements will finish performing their tasks before other elements; In the master/worker model, it will simply "ask" for more work, instead of waiting for some synchronization barrier, for instance;
- Efficiency: computing elements must not be idle waiting for others. Idle times in single units add to the overall execution time. We understand efficiency here as the time the computing elements are effectively working in proportion to the total execution time - that is, excluding the moments they are idle.

These factors are essential, for instance, in cloud computing environments, where the number of computing elements and their computing power is highly variable among executions and possibly even during one specific execution. It is also essential for grid environments, where the available computing elements for performing a computation are not usually known beforehand and might have some dynamic changes in the environment throughout the execution.

2.5 Multi-clusters

A natural extension of the act of gathering machines together in the form of the cluster is that of joining clusters of workstations together. The ample availability of network connectivity across organizations and the pervasiveness of the Internet made possible the creation of these multi-clusters.

There are several forms of multi-cluster, from the simple act of using a set of geographically distributed machines that can communicate directly to each other, to grids, most present in scientific research, and cloud environments, popular among enterprises.

However, the usage of multi-clusters to increase computing power brings with itself added levels of heterogeneity, and with it, further difficulties in scheduling and processor selection [50]. Now, not only the heterogeneity between machines is important, but the heterogeneity in terms of interconnect of each cluster, the interconnect between clusters and heterogeneity in computing power among individual nodes are important. The communication between each pair of clusters might be different, and this poses a challenge for masking the communication time throughout the computation.

Heterogeneity between machines is a well-studied problem[84] [57], but in multi-clusters, efficient executions of applications is field less touched.

These factors are not linear and worse, not uniform across applications. What is perceived as an inferior CPU for one application might not be the case for another one. Differences in cache usage, for instance, account for big differences in application performance, and the same happens for every other parameter that influences in performance.

2.6 The hierarchical multi-cluster

The hierarchical multi-cluster is a extension of both concepts of geographically distributed multi-cluster and the master/worker paradigm. In this architecture, each cluster is a master/worker itself. The cluster where the application is launched is considered then the master cluster. The other ones are called sub-clusters, and inside them there are the figures of the sub-master, the sub-workers. On all clusters, the element responsible for managing WAN communication is the Communication Manager (CM), as seen in figure 2.1

In this scheme, what the master node sees is a set of machines where it can send tasks to. The CM then is responsible to send tasks to the sub-clusters under its responsibility and receive computation results from them. In that sense, what the master perceives is that the CM behaves like a powerful machine (given it is the computing power of its sub-clusters) with a higher interconnect latency (because it is sending tasks to its sub-clusters through the internet).

This architecture allows great flexibility and scalability to run master/worker applications, as to increase computing power is a matter of adding more clusters, as long as the interconnect allows it.

As the sub-clusters are seen by the master node of the master cluster as a powerful node, it might send tasks of either coarse-grained or more tasks, in the case of a fixed-sized problem. To optimize communication, the CM might make use of a pipeline scheme in order to send and receive a higher amount of work to

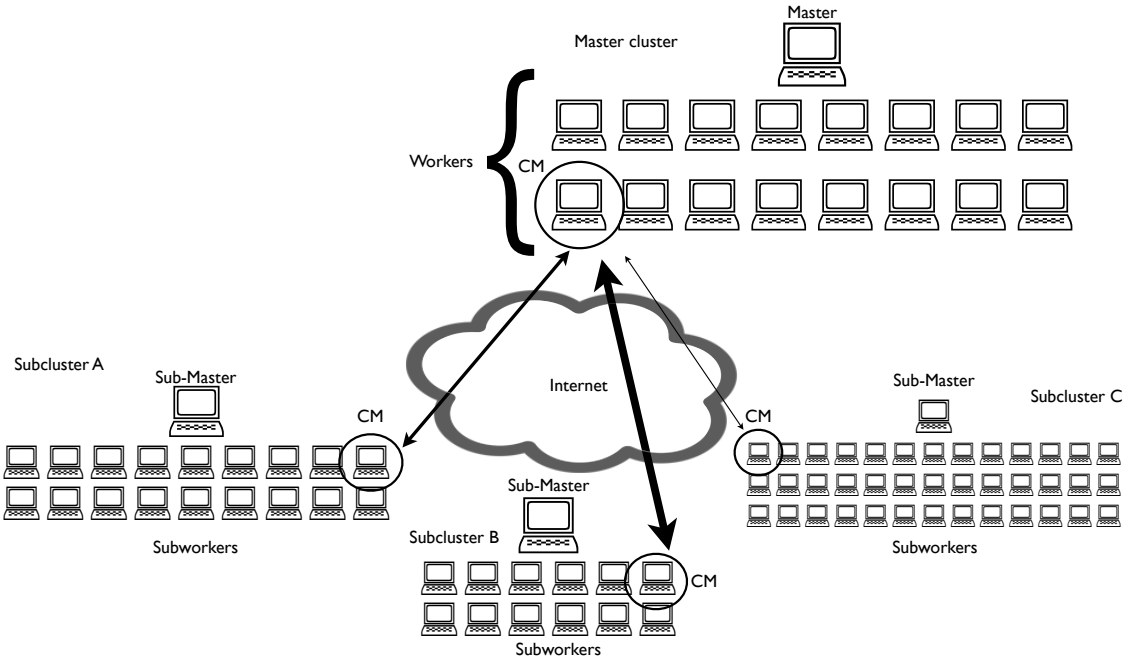

Figure 2.1: A hierarchical multi-cluster environment. Different machine sizes represent different processing power. Thickness of the arrows represent different network bandwidths.

a sub-cluster than that which would be sent to a single worker, and make use of buffers/caches to communicate only when necessary.

Sub-masters receive work from the master cluster via communication manager. They then behave as regular masters in single clusters. They can schedule their tasks and grain sizes among their workers, and join back the results to send back to the master cluster.

Workers and sub-workers work identically - they receive tasks from a master - be it the root master or from their own master, compute it, send the results back to their master, and request new data. In order to mask communication time, they might start receiving the next task in background before they finish their current task, and can send their results back to their master while already processing its next task.

On the application level, the communication manager and the sub-clusters are transparent, so it can be run unchanged in a single cluster or in a hierarchical multi-cluster, and even with failures in communicating with the sub-clusters, as the CM is able to detect it and redistribute the work accordingly, without stopping the application.

The work of [9] sets the basic ground for this work. On it, an analytical model for execution of master/worker applications in hierarchical multi-clusters is defined, based on the CPU and network performances between the different elements and the computation/communication ratios.

This model is able to run master/worker applications on a set of aggregate clusters working together hierarchically, and the cluster where the application was launched sees the other clusters as powerful machines, with usually slower networks than its own nodes. According to this power to bandwidth ratio, this model can decide on the number of tasks to send to the sub-clusters, to use only parts of them or slash clusters entirely, if they will not be able to reduce this application computing time over a user-specified efficiency threshold.

This model is also able to predict a master/worker application execution time. But both for scheduling the number of tasks and for predicting total execution time, two kinds of information are required:

- The network throughput between all the clusters, and
- Computational capacity of every node available, in every cluster, for that specific application.

To calculate network throughput is trivial. To calculate computational capacity, however, is not straightforward. The model of [9] actually ran at least one

task of the application on every unique available node, during the first step of its execution, to then decide where to run.

This, of course, brings its own sort of problems. Very early in my research, I realized two problems with this model:

- Some nodes (in one of the Universities' heterogeneous cluster) were so slow that this performance determination would take an unfeasible time to run, sometimes even longer than the rest of the application.
- The performance results were not valid for the next executions, as it was really common to have different sets of computing nodes in some clusters beyond our control.

So, basically, the performance estimation was taking an unrealistic time.

Our goal is to enhance the efficiency of master/worker applications on heterogeneous environments, where the available machines - and, of course, their performance - is not known until the time to run this application comes.

We focus our studies on master/worker applications, but any environment with heterogeneous machines that needs quick and precise information about how an application will perform on a given machine may benefit from our method. That is, queueing systems that selects resources based on availability, performance and efficiency, such as grids and clouds, can take advantage of this research.

Chapter 3

Basic Block Vector Distribution Analysis

Speak English! I don't know the meaning of half those long words, and I don't believe you do either! — Eaglet

3.1 Introduction

This chapter explains the base methodology used to find repetitive behavior on an application's execution, made independent from any hardware characteristics. It is related to how the code performs, although it is totally independent from the programming language used and even from the presence of the source code at all. Instead, if focus on how the code's flow, which is highly correlated to the source code. This methodology is called basic block vector analysis.

3.1.1 Metrics

In order to characterize a program's execution, metrics are needed. Given we want to predict execution time, the first metric is, evidently, execution time.

Nevertheless, just execution time is not enough for characterization - and more important, it does not help us find repetitive behavior, which we could trim out from the Probes.

The first metric we analyzed was the cache hits/misses of the applications. Considering the ever-growing gap between the processor speeds and memory latencies, one access to a memory position not replicated in the first-level cache (a *cache*

miss) might stall the processor for hundreds of clock cycles. So the cache miss/hit ratio during an execution, if replicated, could mimic a great part of an application program.

The obvious issue with the cache hit/miss metric is that it is completely dependent on the underlying hardware. Even in hardwares of the same architecture line, different cache sizes or configurations might produce unbeknownst consequences.

Moreover, the cache miss ratio is a metric derived both from the hardware and from the behavior of the memory access pattern.

The memory access pattern alone can represent a significant part of the performance characteristics of an application [10] , but it is not the panacea that can give an optimal representation of the execution time, as issues such as address space randomization and allocation above the available capacity - which might make the system start to make use of virtual memory - makes the memory access pattern a an information of little use for our needs when taken into account individually.

Furthermore - the memory access itself is derived from a more fundamental behavior - and this one independent of the specifics of the underlying hardware: the code itself.

Other researchers arrived to the same conclusion - which bolstered our decision into investigating the path of mimicking the program control flow itself.

The program control flow can be described as a series of decisions, jumps and loops. And the structure of programs makes that some behavior repeats itself, such as a calculation done over a large amount of data, for instance, or a function being called several times.

Although programs have repetitive behavior, on the same time, throughout execution they can have wildly different behavior. During one part of an execution, a program can be cpu-bounded, and in a further moment, it can turn to memory-bound as a consequence of a different part of it being run.

The changes into a program's execution from one state to another tend to repeat. We call these repetitions *phases*, and that is one of our basic elements. Phases are sets of intervals within a program's execution that present similar behavior, regardless of temporal adjacency [48].

This work classify phases breaking a program into intervals of execution, and finding similarity between them. Similarity is how close an interval of the program execution is close to another in some metric. In our case, we classify similarity by the number of executions of each basic block into each interval - that is, each vector, with the other vectors. Vectors where the more or less same basic blocks are executed more or less the same amount of time tend to take the same time to run

in the large scale.

For classifying similarity, we do not use any architecture, but as already stated, the program's control flow is used to classify similarity, as what the code is doing at a particular moment determines the program's behavior.

With this insight, it is possible to find phases in programs without hardware-dependent metrics at all.

The metric used to classify the code traversal throughout the execution is the Basic Block Vector Analysis, explained below.

3.2 Basic Block Vector Analysis

In a well-designed master/worker application, the determinant in execution time is that of the workers computing their tasks. So we focus our efforts on understanding the performance of the worker.

Most code that does not depend on constant user iteration (such as majority of scientific high-performance applications) presents repetitive behavior of some sort - methods and functions calls, and loops, are all repetition of the same code with some different parameters. This is especially true in number-crunching scientific code. According to Sherwood, Perelman and Calder [65], "the large scale of programs is cyclic in nature". On their research, they measured under simulation that for the SPEC95 benchmarks, the hardware statistics in fixed periods of time, and noted that these statistics (1) repeat from time to time, or (2) have a repeatable cyclic behavior until the end of their execution. It is demonstrated that periodicity of the basic block frequency profile "reflects the periodicity of detailed simulation across several different architectural metrics (e.g., IPC, branch miss rate, cache miss rate, value misprediction, address misprediction, and reorder buffer occupancy)".

Periodic behavior is defined as a repeatable pattern seen for a given architecture metric. On Sherwood's breakthrough article, it is possible to notice that no matter what the hardware characteristic is chosen, they change at the same time, i.e., there are distinct phases, and to discover them, they used a metric independent of any architectural parameter, but highly correlated with their performance. Their intuition was that what is executed in a given moment determines program behavior, and it reflects on architectural metrics.

This metric is then given by counting the number of times a piece of code - defined here as basic block - is executed under several contiguous periods of time. This is the most efficient technique for phase detection, according to Dhodapkar and Smith [17]. In this work, fixed-sized phases are used, as they work well enough, but

there is research on phases with variable sizes and even hiearchical ones, such as in the work of Lau[47]. The size of the phase means the number of instructions of one or more basic block vectors.

A basic block is classically defined as a piece of code with one point of entry and one exit - i.e. no control flow. They called this metric "basic block distribution analysis". A key point is that the phase behavior seen in any program metric is directly a function of the code being executed [49, 68] . Because of this, a metric that is related to the code can describe phase behavior [67].

Sherwood's goal was to reduce run-time of functional simulators for next-generation processors. With a functional simulator, it was possible to count the instructions and have precise information of which basic block of a given code this simulated processor is running.

The biggest insight was the selection of an approach that does not use any knowledge of the architectural state of a program, but yet is highly correlated with the performance of such metrics.

His method consisted in counting the number of times a basic block was executed during a fixed number of instructions (called the basic block vector - BBV [65]), and compare the similarity of these vectors.

Their goal with such method, for the field of computer architecture simulation, was to find:

- The end of the initialization part of the program, and the start of the cyclic part of the program.
- The period of the program. The period is the length of the cyclic nature found during a programs execution.
- The ideal place to simulate given a specific number of instructions one has time to simulate.
- An accurate confidence estimation of the simulation point.

Simulation point here means the exact point in the execution where one could start the simulation (recovering from an architecture checkpoint, complete with cache/register/memory state, straightforward in a simulator), thus reducing simulation time.

It is common that applications pass through a initialization period, where its data structures are created, files are created and read and so on, before proceeding to the actual processing phase. The processing phase usually consists of periodic

behavior, alternating between completely different sections of code (functions, procedures, loops, methods).

The idea is that similar vectors were running more or less the same code, and, on large scale, they tend to take the same execution time [61].

By having the same execution time, it was possible to run only a reduced number of occurrences of one phase, skipping repetitive behavior [59]. This greatly reduced simulation time. In figure 3.1, for instance, the three dominant functions, x_solve, y_solve and z_solve took the wide majority of this execution's time, but by no means the functions themselves took a long time, but instead, the sheer number of calls to these functions.

Their work resulted in a tool, called Simpoint [27]. Basically a K-means clustering algorithm that reads a basic block vectors' file and points out the relevant phases found, to drive the simulation to these specific points. The basic blocks were collected either during functional simulation or in real machines through code instrumentation [58] in order to feed the simulator later.

Once an application is profiled, for instance from an execution trace of an architecture simulator such as simpleScalar [14], the basic block profile is then fed into their *SimPoint* [28] tool.

Our Probe departed from the concept of phase classification and simulation points and implemented these in a tool that is able to run in real computers, running real code, instead of simulators. The specifics of the Probe are going to be discussed on the next chapters.

Simpoint is a tool that uses the k-means algorithm from machine learning to group code signatures into clusters based on signature similarity. The single most representative code signature from each group is selected for execution, the importance of this selected phase is weighted, and the results of each execution are extrapolated to estimate the program's overall behavior, according with each phase's weight.

Each execution phase has a typical set of instructions, that is, under a given period of time the application will use only a subset of the total architecture instruction set. Dhodapkar and Smith found that exists a relationship between phases and architecture instruction working sets, and that those phase changes tend to occur when the instruction set changes. In [17] they found that BBV analysis is accurate. Their approach focused on online phase classification. On the other hand, Simpoint is an offline tool.

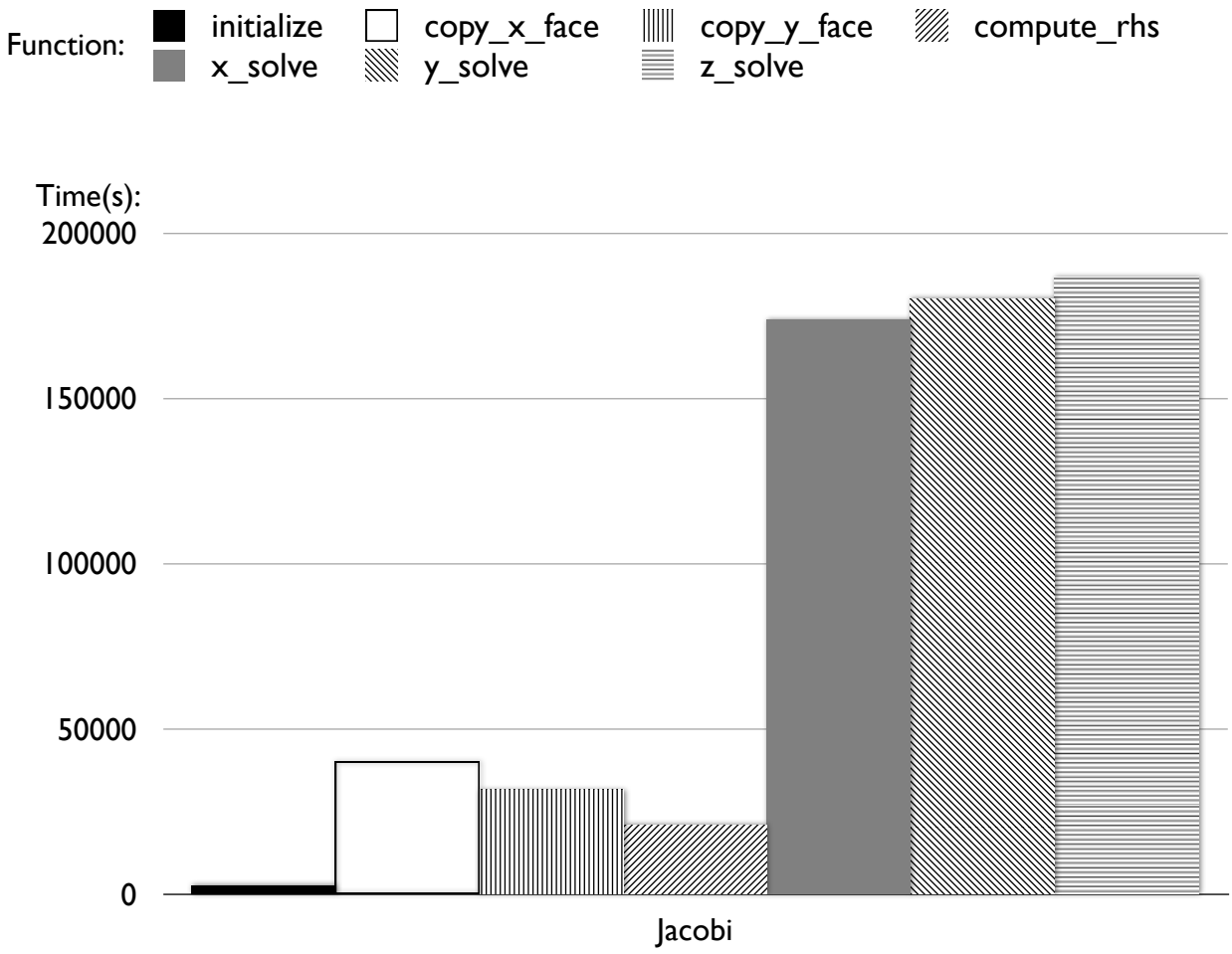

Figure 3.1: Total time spent for the main functions in the Jacobi Relaxation Benchmark as measured by the Scalasca toolset in 1024 processors.

According to [48], the representatives for each phase selected by Simpoint based on full-BBV data not only represent the whole program well, but also represents each phases well. Their results shown an average error rate about 2% on the CPI metric when compared with full executions across the SPEC2000 benchmarks.

Their first attempt was to find a single continuous window of executed instructions that matched the whole program's execution, so that this smaller window of execution can be used for simulation instead of executing the program to completion. In [67], Sherwoord, et. al found out that more sophisticated applications cannot be represented using a small contiguous section of execution.

Applications do have regular behavior, however not in a single window of execution, coming from the beginning to some specific point. Instead, the repetitive phases are found along the whole execution of some program. To find them, Simpoint examines the similarity between different phases, grouping the ones which are similar together, in a method called clustering. The goal of clustering is to divide a set of points into groups such that points within each group are similar to one another, and points in different groups are different from one another.

One problem with the clustering algorithm used in simpoint, the k-means, is that its performance suffers from the excess of dimensions on the basic block vectors, as every basic block is one dimension across the different BBVs. So the target basic block vector from the full execution can have millions of dimensions on one fairly complex application.

To address this problem, it was used an algorithm dimension reduction, by creating a new lower-dimensional space and then projecting each data point into the new space. The algorithm is the random linear projection, that reduces the amount of dimensions while retaining the properties of the data.

In [67], it was found out that 15 dimensions are enough to correctly cluster the phases, and that increasing this number gives little for improvement on finding clusters. Once the clusters are found, it is necessary to run then, in a noncontiguous way, as they can be spread among the whole execution. That is, the execution can be broken down into N executions, where N is the number of clusters found through analysis, and each execution is run separately. On homogeneous clusters of workstations, this can be used to break the execution into parallel components that can be distributed across the nodes. In any case, results from the separate clusters of phases needs to be weighed and combine to arrive at the overall performance of the program.

Chapter 4

Probe

What is the use of a book, without pictures or conversations?— Alice

4.1 Introduction

The general idea is that if we are able to know how long will an application take to run on a machine without taking the time that would be required to run it thoroughly previous to an execution, we will be able to decide if this machine fits our needs or not.

For that, we came with the concept of a software Probe. We define a Probe as a piece of code that is able to quickly extract useful information about the relationship between the application and the system we want to know about. This means that a Probe from a given application will run fast and will return us precise information about how the application we are analyzing is going to run in there.

The work in multi-clusters already introduced at chapter 2, and described in more detail in [8] , [9] deals with the issue of efficiency of Master/Worker applications in those heterogeneous and distributed environments. For it, efficiency is defined as the ratio between the best possible execution time T_{best} and the total execution time T_{ex}, as in equation 4.1:

$$Efficiency = \frac{T_{best}}{T_{ex}} \tag{4.1}$$

Basically, that means the proportion between the time taken actually doing useful computational work over the total execution time, with delays.

Several works on the literature uses simulated environments, usually because of the long times required to characterize applications and test their policies, as noted in chapter 1. The work of Kim, Rho, Lee and Ko [42] is typical: they performed simulated experiments where individual nodes' performance is given by a random value in a range around the average node's one. It suggests that in real environments, these values could be given by execution historial.

Therefore, the absence of a good and fast way to determine execution time has hampered experiments on real environments for a long time. Mostly, either historical data or simulation was used. Until now.

4.2 The alternatives

Before explaining our method, we briefly introduce two of the alternatives, why they could be useful or otherwise: Thorough executions and Benchmarks.

4.2.1 Thorough executions

The explanation for this one is straightforward: there is no better method to predict the execution time of an application running in a machine than running this application itself. The obvious problem with that is the execution time required to do this makes it of little use for the selection of sub-clusters or sets of machines according to the communication/computation rates. Actually, this was the method used on the multi-cluster environment already which is the base of our research.

The problem is not isolated to the fact that execution time of one worker doing this first task thoroughly may delay the overall execution, but furthermore, as at least one execution of a task must be executed on each unique kind of machine on each sub-cluster, which creates a cascade effect on delay. This delays the decision of discarding a cluster completely, for example.

4.2.2 Comparison of hardware characteristics

Modern computers have special registers called "performance counters" that store the counts of hardware-related activities, such as instruction counters, cache faults, memory access and so on. While they are very useful for low-level application performance tuning, its difficulty to relate them back to the code running [86] and micro-architecture differences between CPUs of the same families (which themselves lead to wildly different results of the same application on compatible but not identical machines) make the question of performance determination of little usage in our case.

4.2.3 Benchmarks

The other alternative is using benchmarks to characterize machines [45]. The use of benchmarks for determining performance of machines is only useful to have an index to compare machines between themselves, but even then, only under some circumstances. Benchmarks are hardly able to represent an application's performance [53].

There are several different kinds of benchmarks. Here are some examples:

Synthetic Benchmarks

Benchmarks created to exercise all aspects of the machine, or some specific, are not new. Curnow created the Whetstone synthetic benchmark on the 1970's [16], with the explicit objective of "breaking" compiler optimizations, doing operations found in common programs from that time.

Another classic synthetic benchmark is the one of Weicker, known as Dhrystone, a pun to the previous one [89]. This one tries to be general for the applications of its time, in terms of the distribution of statement types, data types, and data locality.

None of them try to reflect any specific application, quite on the contrary - they tend to be means to compare CPUs in a general manner.

Joshi, Eeckhout and John created a tool, called BenchMaker [38], [40] which from a set of program characteristics related to the instruction mix, instruction-level parallelism, control flow behavior, and memory access patterns [39], generates a synthetic benchmark whose performance relates to that of a real-world application, with mixed results.

It is unclear how to transport the instruction mix and instruction-level parallelism from one application to a benchmark to a different cpu with the same ISA, but different internal structure.

Another approach is that of Strohmaier and Shan [75], [76], [77]. Their Apex-Map tool is able to create different types of memory access pattern in parallel applications, in order to compare architectures. No claim of similarity with any applications is made.

Kernels of widely-used algorithms, Microbenchmarks

These benchmarks take the main loop of well-known algorithms and encase them into smaller packages with some default input data. Perhaps the most omnipresent

of these is the High-Performance Linpack [60] that solves a (random) dense linear system in double precision (64 bits) arithmetic on distributed-memory computers.

The research of Hoisie [32] uses an application itself as a benchmark. This application is the Sweep3d, a "particle transport code taken from the United States Department of Energy (DOE)'s Accelerated Strategic Computing Initiative (ASCI) workload, SWEEP3D represents the core of a widely utilized method of solving the Boltzmann transport equation". As such, it is a really good benchmark for testing the scalability of extremely parallel systems, but no correlation with other programs exist. We used, however, Sweep3D as an application in and on itself, and characterized its behavior, and created Probes from it.

Some works try to be more thorough in covering the different characteristics of different application for pure benchmarking purposes, i.e. for comparing machines. Such examples are the SPEC [37],Phansalkar:2006p1558 and NAS Parallel benchmarks [11]. Another use of these benchmark suites is to use themselves as the application, as is commonly done in the field of performance measurement, exactly because they are so thorough in general characteristics.

Mashups of different benchmarks to try to reproduce application behavior

Some works try to analyze characteristics of several different benchmarks, and, according to their indexes, create a "mix" of benchmarks that can correlate to the performance of some application, such as the work of Murphy and Kogge [55], where they found a high rate of discrepancy between benchmarks and real applications.

4.3 Our alternative: the Probes

We focused in transporting the knowledge and techniques used in the world of simulation to real applications, running on real computers.

The idea is to be able to give the multi-cluster scheduler (and possibly other schedulers) information in order to be able to select the computing nodes best suited for running a specific application, according to an efficiency threshold.

For that, a Probe must be sent to the remote node(s) to be characterized, where it runs for a short time and returns the predicted execution time of the application it was based upon.

The goals set for the Probe implementation were the following:

- Be able to correctly predict the execution time of an application (that is, with

a high quality of prediction);

- This prediction must be done in a fraction of the time required to run the application itself;
- The amount of data to be sent for this prediction be small enough that it can be sent to the remote clusters in a reasonable time.

I believe these objectives were achieved. The Probe is able to reach a prediction quality higher than expected. In most cases, prediction quality stays way above 90%.

Regarding the time the Probes takes to run, it stays in most of the experiments under 15 seconds, with most of them revolving around 3 and 16 seconds.

And finally, about the transmission size, the use of the reduction techniques described in the next chapter made the probes significantly smaller, enough to be transmitted through the internet to the remote clusters in matters of minutes.

The time reduction that our Probe methodology provides for predicting execution time is not only useful for master/worker applications. It is also useful for performance prediction of highly parallel codes, such as in the work of Wong, Luque and Rexachs [91], where the computation part between communication events of its characterization methodology can be reduced to individual Probes, enhancing its prediction time, and any other scheduler that needs performance information. Grid and workflow engines are obvious examples.

So we define the concept of a software Probe as a program which was generated using parts of the original program based on its execution behavior that can reproduce the performance behavior of a given application/input data pair on a machine in a fraction of time required to run this application itself. That is, the small relevant phases we execute are representative of the whole execution by extrapolation of their importance to the total execution.

We can determine, just by knowing how the code is exercised, the repetitive performance behavior, that is, parts of the program which behaves similarly in terms of performance.

As stated in chapter 3, when the code is doing the same operations, its time to perform them tends to be similar, in the large scale, as there is a strong relation between the code being executed and performance predictability. [7]

Therefore, to know the characteristics of an application on a given machine, only the relevant parts of this application must be ran, a very reduced number of times. The rest of the program's execution is mostly different iterations of the same phases, and can be represented by these relevant parts and their weights.

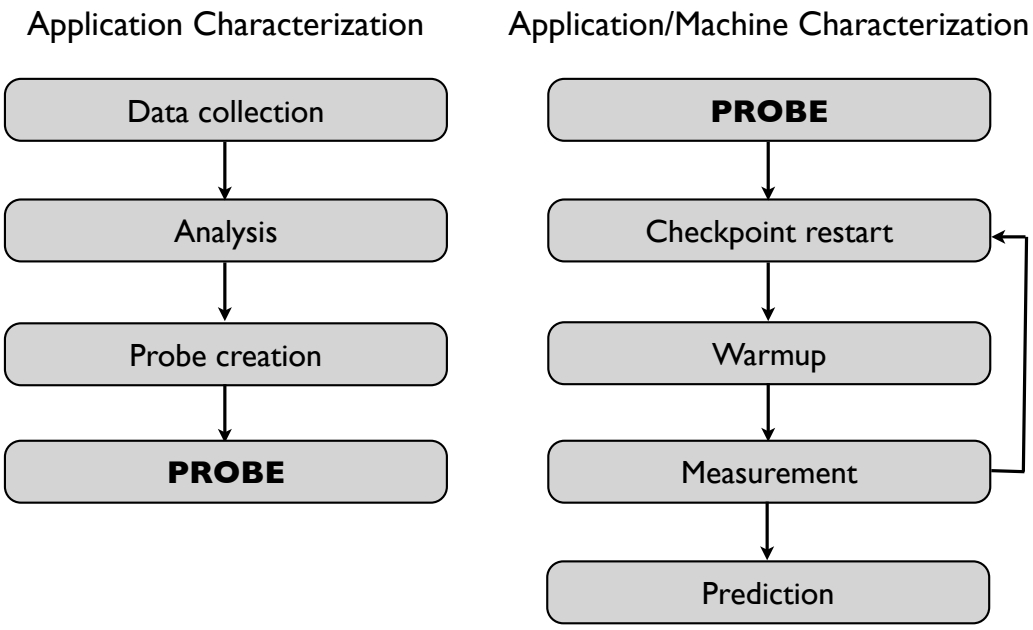

Figure 4.1: General Scheme.

Phases with less than a minimal participation in total execution time - for instance, 1% of it - can be discarded. With a reduced number of executions of each phase it is possible to extrapolate the full execution time.

Our software probe aims to give us performance information about a machine we know nothing about in this aspect. This information can then be useful for determining if this machine is worthy to run this given application completely, or if it's best to let it to run somewhere else.

4.4 Creation methodology

Before being used, a Probe must evidently be created. The next sections describe the steps required for creating a probe, starting from application characterization, state saving, Probe build, with the tools and techniques required to perform such

actions.

In general terms, we must:

- Run the program, in a controlled environment, to monitor its execution. No source code is required, as we use instrumentation directly on the binary;
- Analyze this program's execution, in order to find the repetitive behavior. With this we can discover the relevant phases and its weights;
- Capture this behavior. Given the Probes are for real computers and not simulators, the only way of doing this is by the use of checkpoints. We also take into account issues such as architectural warmup;
- Create additional support structures to run the checkpoints for a specific amount of time
- The Probes are ready. New they need to be sent to remote machines run, and predict the original application's execution time.

We want to predict the execution time of an application or worker task while performing on a machine, without spending too much time in the process. However, this fast characterization must be kept accurate regarding to the performance characteristics of the application itself, that is, it must have a good prediction.

To be able to keep a good balance between these antagonistic goals, we use the basic block vector analysis previously described in chapter 3 order to run as little of the original program, but running only the pieces of code necessary for execution time prediction, that is, performance-wise. By avoiding repetitions, we are able to capture meaningful behavior and reduce time altogether. According to the basic block vector analysis, it is possible to define the proportion of the total execution time taken by each phase. By extrapolating the time taken by the execution of each phase to its proportion of the total execution time, it is possible to predict the application's total execution time.

4.4.1 Overview

The generation of a Probe for an application comprises the following steps: data gathering, phase discovery, phase save, Probe construction. With them it is possible to perform remote machine's probe execution and measure this Probe's performance.

In essence, what happens is:

- In our reference machine, the application is ran to the end, and the program's basic block vectors are collected;
- The *Simpoint* utility reads the basic block vectors input file and discovers the relevant parts of our applications - the phases - and its weights. The weights are the importance, or the proportion that each relevant phase has over the entire execution;
- With the beginning of the relevant phases known (in number of instructions), we run the application again, instrumented. This instrumentation counts the number of instructions, and when the beginning of a phase is reached (minus a number of instructions for warmup), it saves a checkpoint, with special instrumentation that will be engaged when this checkpoint is restored.
- The execution proceeds saving as much checkpoints as phases found by *Simpoint*.

Essentially, the first version of our Probes is done after this point. One can send these special checkpoints to a remote machine and restore from them. What this instrumentation code does is:

- Wait for the warmup. At this point, instrumentation is kept to a minimum inlined instruction counter, to not interfere with the execution. After the warmup, it saves the time and resets the instruction counter.
- The instruction counter is inlined again, in order to interfere as little as possible with the execution. When the number of instructions set for the basic block vector size is reached, this phase's execution time is calculated and execution is interrupted.

The Probe runtime then proceeds to the next phase, and so on, until all the phases have been executed. This happens in matter of seconds.

Our software Probe will be created by modifying the application we want to know about dynamically. In a first step, to acquire the basic block vectors, we instrument the application with gathering instructions. Later, to save the application's phases beginnings, we will instrument these places with checkpoint commands. And finally, the probe will be ran as the application dynamically modified to restore from the checkpoints, run each phase, compute its time, and then jump to the next relevant phase, by means of restoring from the next checkpoint.

These steps will now be described in detail.

4.5 Data Collection

The first step of our methodology is to acquire knowledge about the application behavior. We use instrumentation code to gather the basic blocks [88]. It inserts code at the program we want to characterize, counting the times each basic block is run during a period measured in number of instructions committed.

At the beginning of our research we used a static instrumentation toolkit called Atom, from Intel. It was a very straightforward tool to create instrumented binaries. However, it was being phased out by Intel because of its numerous problems in favor of their new tool for Dynamic instrumentation, then still in development, called Pin [31]. Atom's problems and total lack of support made impossible to use it.

The trend in academia seemed to be the usage of DynInst, which has a similar goal of dynamically instrumenting programs [34], so we followed suit. Using DynInst, we created a tool to gather the basic block vectors. However, several compatibility issues with then GNU/Linux kernel versions, and the fact that it requires a previous environment setup which is hardly done automatically, is a big drawback when characterizing unknown machines - something that needs to be automatic, so we ditched this tool and started the search again for one better suited for our needs.

As Intel was putting heavy effort on their new tool, we tried using it, and it suited our needs much better than the DynInst. it requires only that its binary and some helper files on the remote machines, which lessens the requisites on the environment we want to characterize when compared to DynInst, as we can send the runtime together with the Probes with little trouble.

Together with the fact that the overhead imposed by DynInst was, in our observations, somewhat bigger than the one imposed by Pin, we decided in favor of the Pin toolkit.

Pin is able to instrument at different levels of granularity, thus being able to minimize the intrusion according to the user's needs, ranging from the full program image up to the instruction level. That is, a program that does not need to be instrumented at instruction level will not be, thus avoiding unnecessary intrusion.

Our instrumentation code uses, evidently, Basic Block granularity level. It counts the number of times each basic block was executed on each interval of N instructions. On our experiments, we used N as 100 million instructions, which is big enough to capture performance parameters and big enough to make the cache warmup effect not significant, however being small enough to reduce on orders of thousands the time required for characterization on an unknown machine. It then generates a basic block file to feed Simpoint. All machines we tried executed 100 millions instructions under a second even in cases of low cache hit ratios.

```
MAX_INSTRUCTIONS=100000000; // This is actually an argument
inscounter=0;               // Instruction counter
interval=0;                 // Numer of this BBV
map <address, long> BBV;    // Basic blocks vector

while(true):
    if(inscounter != MAX_INSTRUCTIONS):
        BBV(address)++;     // Uses the address of the basic block as key
        inscounter++;
    else
        dump(BBV, interval, output_file); // Dumps data in Simpoint format
        empty_map(BBV);                   // Resets the structures and
        inscounter=0;                     // stays running the application
        interval++;
```

Figure 4.2: Pseudocode of the instrumentation for capturing basic blocks

One characteristic of Pin for acquiring basic block vectors is that, since it runs on user level, it is unable to instrument operating system activity. This turned out to be an advantage in our case. The idea is that our instrumentation code only captures the application activity, because the original application, when used on other machines, will use the libraries of that machine, so any differences in execution time given by different libraries will be reflected in our Probes in the same way they would on the original application.

Another apparent limitation is that instrumentation might slow down an application's execution in orders of magnitude, when instrumentation is not done right. For the basic block vector acquisition of this stage of our process, time is not relevant, however. In further stages, instrumentation is kept to a bare minimum and inlined, in order to affect as little as possible the execution.

The instrumentation code consists of an instruction counter and a map, a histogram-type data structure, where the key is the number of the basic blocks, and the value is a counter of the number of executions of this basic block. The pseudocode for the data acquisition instrumentation routine can be seen in Figure 4.2:

4.6 Phase discovery

Once the basic block vectors were gathered, the Simpoint tool is then used to discover the application phases and weights.

Although its original purpose is to discover simulation points for running short executions of applications in functional computer architecture simulators, we can use with no modifications to discover the most relevant phases of real applications, in real machines.

Besides the underneath complexity of Simpoint, its a straightforward tool to use. Once one provides it with the basic blocks vectors and the max K to be used on the k-means algorithm, its operation is automatic. Simpoint's output consists of two files:

- one containing the initial point of each relevant phase: this means, the committed interval since the beginning of an execution;
- one containing its weights, that is, what portion of the whole execution this phase represents.

An interval, as defined in the pseudo-code, is the number of a specific basic block vector throughout the execution. The initial interval that SimPoint gives us must be multiplied by the interval size used when gathering the basic blocks - in our case, 100 million. For instance, a interval of 34 means that this phase starts at 34 * 100 million = 3400000000 committed instructions after the begin of the execution.

This means that the granularity of the phase discovery method - and the phases' size itself - is given by the size of the basic block vectors. There is research in using BBVs of variable sizes [47], but the potential gain in precision did not justify the extra overhead.

The weights of each phase are the proportions of this phase's number of executions in relation to the full execution. It means that if a phase has a weight of 50%, for instance, means that this specific basic block vector (or those similar to it) accounts for half of the collected BBVs throughout the execution.

To this point, we know the relevant phases and their weights on the overall execution. With the initial instruction of each relevant phase known it is possible to proceed to the next step of actually generating the probes from them. On the original work of Sherwood, this would be a matter of saving a snapshot of the application on these moments, and simply use them with the instruction counter of the simulator running with an alarm. On real machines, this is not so trivial, and that is what we explain in the further sections.

4.7 Binary generation

Our method of generating the Probes needs:

- The application/data set being characterized;
- The interval size;
- The interval numbers of each relevant phase.

These requirements are rather obvious - there are also additional requirements for the specific prototype implementation, which are:

- The instrumentation toolkit;
- System-wide checkpointing library;
- Our instrumentation code to generate the Probes;

It was coded in C++, C, Assembly and Shell Script and developed as a pin tool. Pin tool is the name of instrumentation code that uses the Pin toolkit.

From the previous steps of application behavior analysis and phase discovery, we have the relevant phases initial instruction as given by Simpoint. This is the input for the further proceedings.

The application is ran monitoring the total instructions committed. When this instruction counter reaches the point of one relevant phase minus a number of instructions for warmup, it creates a checkpoint. Execution is resumed and this procedure goes on until the last phase was checkpointed.

One of the characteristics of the Pin toolkit is that the instrumentation code lies in the same address space of the application. Actually, it creates a copy of the whole executable, instrumented, and jumps the execution to this new code, keeping the original code unaffected in memory. This means that when checkpointing, it also keeps the instrumentation code on the checkpoint itself.

This is important to us. It means that the same instrumentation code used for creating the checkpoint snapshots of the execution is going to be used when measuring the phase execution time on remote machines.

The checkpointing library provides undocumented callbacks for a code to know either it is being checkpointed or execution is being restored/restarted from a checkpoint. As we cannot touch the program itself, this code is embedded into the instrumentation. So we have two instrumentation modes:

- Code for counting instructions and generate the checkpoints, and
- Code for time measurement and execution interruption.

Both codes are totally different, but they need to be enclosed in the same instrumentation because of the characteristics cited above. See figure 4.3.

When we are actually saving the phases for further executions, we call it "monitoring" mode. In this mode, its working is straightforward. Its input is the Simpoint output, where there are the number of each relevant phases on the execution. This instrumentation code then sort Simpoint's output and runs over the application, counting instructions. When the point

$$save\ point = (Interval * Interval\ size) - warmup \tag{4.2}$$

is reached, the instrumentation pauses the execution and checkpoints the application altogether with the instrumentation. The instrumentation code has a special callback which says that this pause was generated by the procedure of creating a checkpoint, so it tells the application to keep on with the execution, and for the instrumentation code to keep on with the instruction count until the next phase is reached.

The outcome of this procedure is a set of checkpoints that can be run on remote machines. These checkpoints carry not only the original application, but also special instrumentation code to count the instructions and measure the phase execution time.

4.8 Probe execution

The previous steps gives us enough data to generate a Probe. This Probe is then composed by:

- A shell script to restart from the checkpoints;
- The Pin runtime and the application files, if necessary;
- The checkpoints themselves.

As previously stated, the instrumentation code when the application is generating the checkpoints behaves differently that when it is restarting from them. On

```
phases=[simpoint output];
MAX_INSTRUCTIONS=100000000; // This is actually an argument
WARMUP=7000;                // Instructions for warmup
inscounter=0;               // Instruction counter
counter=MAX_INSTRUCTIONS;   // Negative counter of one phase
if(mode==MONITORING):
    if(inscounter==phases[x]*MAX_INSTRUCTIONS - WARMUP):
        make checkpoint;
        x++;
    inscounter++;
else if(mode==RESTART_FROM_CHECKPOINT):
    start_time=time.now();
    loop: if(counter--==0):         // JE in asm
        SIGSTOP;
        end_time=time.now();
        printf(start_time, end_time);
        exit(0);
```

Figure 4.3: Pseudocode of the instrumentation both phase capture and measurement

the previous step, the only thing it needed to do was to wait for the save point, and when reached, call the checkpointing routine and go on as long as necessary.

This is not the case when running the Probe. Now it needs to

- restart the application and detect this state,
- warm up the architecture,
- measure start time of the phase,
- count the instructions until the phase's end according to the interval size,
- interrupt the execution and count time again.

4.8.1 Phase save

Since there's no way to fast-forward an execution up to a point we want - in our case, to the beginning of a representative phase - we propose implementing the saving of checkpoints of the execution while gathering their phases. The checkpoint brings

the memory content along with execution context, thus is serves as a fast-forward to the points in execution we want.

While clusters of workstations using the Linux operating systems are increasingly common, many aspects of the software environments are steps behind those provided by commercial supercomputer systems. One of its deficiencies is the lack of an implementation of a checkpoint and restart implementation general enough to support a variety of parallel scientific applications. Duell defines checkpoint as the process of saving the entire state of a job to disk, than later restore it [19], and has proposed an implementation, known as Berkeley Lab Checkpoint/Restart, or BLCR [29].

The advantages of using BLCR over other checkpoint/restart implementations is that its actively maintained, well documented and widely tested and available as source code for GNU/Linux systems [20]. It is general enough that its requirements must not restrict too much the kind of applications we are able to characterize.

As our probe will run only the relevant phases of an application, we need a way to do this. Those relevant phases are spread across the whole application, and there's no way to simply jump to some point of a large execution. That's why we use checkpoints - while we are characterizing a machine, we will save checkpoints at the beginning of each relevant phase. Each saved checkpoint will be used as the starting point of its corresponding relevant phase on the probe.

The current limitations of BLCR is that TCP/UDP sockets are not restored, and appears at restart time as closed. MPI implementations such as MPICH-V [13], that use BLCR as its underlying checkpointing system, implement the communication primitives themselves, for instance. Therefore, the applications we characterize cannot communicate during its steady period.

This can limit the possibilities of the applications which we are able to characterize. However, the master-worker programming model, for instance, where each task is sent to a worker, is computed and it returns its results at the end of its computation, so it can receive another task [44], is a good example of an application model that fits well for characterization. In this model, we can characterize a worker's task phases, allowing us to better select the machines for this master/worker application.

4.9 Measurement

All these tasks are performed by the instrumentation code when in "restart" mode. Here, timing is crucial, so the instrumentation noise must be kept as low as possible. We achieved that by writing it partially in inlined assembly, with as little memory accesses as possible. Another strategy to minimize noise was to make the instruction

count updated only at the end of each basic block and inlined, instead of after each instruction. This made the code almost 50 times faster (thus less intrusive) than before. The instrumentation noise is one inlined branch instruction (the JE) and one operator decrement (count–) per basic block, which is around 1% of it in our experiments.

When restoring execution from a checkpoint, a side effect will happen. The caches and predictors will not be in a consistent state with what would be done in a full execution that happened from the beginning up to that point. The process of putting the machine in a state similar of that from a full execution is called warmup [30]. We use a warmup size of 7,000 instructions, as stated as enough by Laha [46]. On that time, caches were much smaller than today's multi-megabyte caches of modern processors, but still, we did not have any observable difference in prediction precision when changing this value from 7,000 up to 50,000 instructions. In our experiments, we used a phase length interval of 100 million instructions, which was noted big enough to mitigate cache warmup effects.

After the number of instructions executed is the size of our interval (the instrumentation itself is not counted), instrumentation sends a signal (the Unix's SIGSTOP) to pause the application execution and measures the time, then it stops the application execution (With SIGKILL). The reason for using SIGSTOP first is that it pauses immediately, without taking any further action, so it's immediate. After measuring the time, the instrumentation sends a SIGKILL, which frees memory, sockets, handlers and so on, which takes a short, but perceptible and undetermined time. When the process is terminated, the shell script regains control and can proceed to restart/resume from the next phase.

In order to extrapolate the execution time, it is necessary to have a base value for comparison. So, with the execution time of the full application uninstrumented when compared to the instrumented one in the reference machine, we are able to calculate instrumentation noise (see Figure 4.4), which tends to be a fixed portion of execution time - i.e., it tends to keep the same amount of noise regardless of the machine we are running our Probes at, so we consider it constant. So on our reference machine we calculate the noise as the proportion between the time it takes to run the whole application on the reference machine with the phase instrumentation $ETRef$ and the uninstrumented execution time $ETuninstRef$.

$$Noise = \frac{ETRef}{ETuninstRef} \tag{4.3}$$

We calculate the proportion of each phase time and the total execution time, instrumented, on the reference machine. This is given by the formula in equation

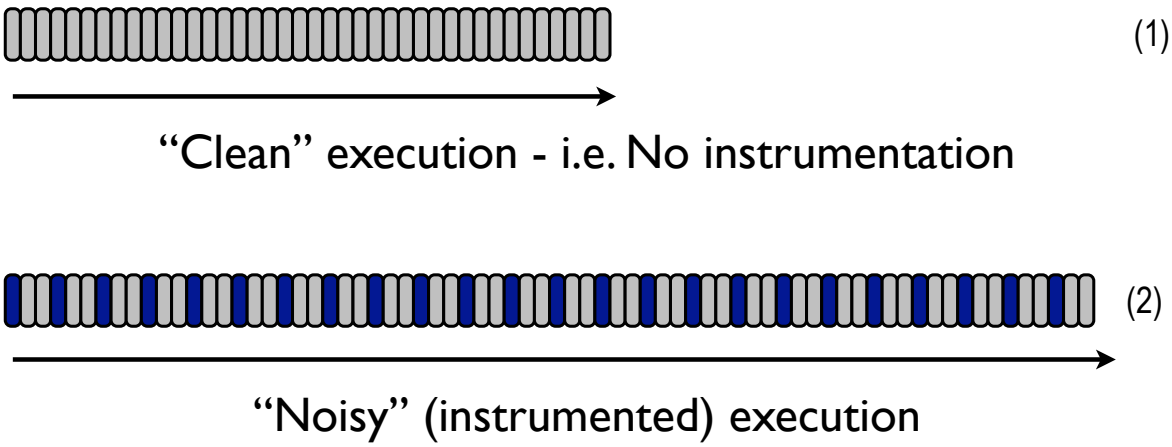

Figure 4.4: Instrumentation noise. The dark dots are instrumentation instructions inserted in the execution during runtime.

4.4, where $ETRef$ is the full execution time with instrumentation in the reference machine, $Weight$ is this phase's weight, and $TPhaseRef$ is the time this phase took to run on the reference machine.

$$TimeUnitValue = \frac{(ETRef \ * \ Weight)}{TPhaseRef} \tag{4.4}$$

For example, if the application takes 120 seconds to run, and one phase takes 3 seconds, but this phase's weight is 50% of the total execution, by substitution, each second of this phase is worth $(120 \cdot 0,5)/3 = 20$ seconds of the full execution time.

With the time unit value, it is easy to extrapolate the execution time of the phase probed regarding to the full execution

$$PhasePredicted = TimeUnitValue * TProbed \tag{4.5}$$

With all the data, we are able to predict our execution time as can be seen in equation 4.6

$$ETPredicted = \frac{(\sum_{i=1}^{numPhases} PhasePredicted)}{Noise} \tag{4.6}$$

(4.9)
$$CPerf_{avail} = \sum_{workers} Perf_{Avail}$$

As Noise itself is a fraction, we multiply the predicted execution time by its inverse. The final result is given by equation 4.7:

(4.7)
$$ETPredicted = (\sum_{i=1}^{numPhases} \frac{(ETRef * Weight_i)}{TPhase\mathrm{Ref}_i} * T_iProbed) * \frac{ETuninstRef}{ETRef}$$

4.10 How it fits in the multi-cluster model

In the multi-cluster performance model developed by [9], a methodology to improve the execution time of master/worker applications in multi-clusters was developed. The performance model can predict the execution of the whole applications provided it knows beforehand the network throughput and the time it will take for completing a task in each and every of the available computing elements. The steady performance model (that is, after initialization of all workers) is given by $Perf_{Ste}$ in equation 4.8.

(4.8)
$$Perf_{Ste} = min\left(CPerf_{Avail}, Oper * \frac{TPut_{EC}}{S_{Comm}}, Oper * \frac{N * TPut_{InetIN}}{S_{TaskInter}}, Oper * \frac{N * TPut_{InetOUT}}{S_{ResultInter}}\right)$$

On this equation, $CPerf_{Avail}$ is given by equation 4.9, $Oper$ is the number of basic operations of a task, $TPut_{EC}$ is the external cluster LAN average throughput, S_{Comm} is the total communication of a task, N is the number of the tasks of this execution, $TPut_{InetIN}$ is the external cluster's internet incoming throughput, $S_{TaskInter}$ is the size of an inter-cluster task, $TPut_{InetOUT}$ is the the external cluster's outgoing throughput, and $S_{ResultInter}$ is the inter-cluster's task size.

Every parameter of this equation is either a factor of the network throughputs, which are straightforward to discover and problem size, which is known beforehand, and that sub-cluster available performance, described in equation 4.9.

On it, $Perf_{Avail}$ is the relative computation performance of a machine while running a worker task. While every other parameter of these equations is discoverable or already known, this one is not, neither is trivial to be discovered. In this

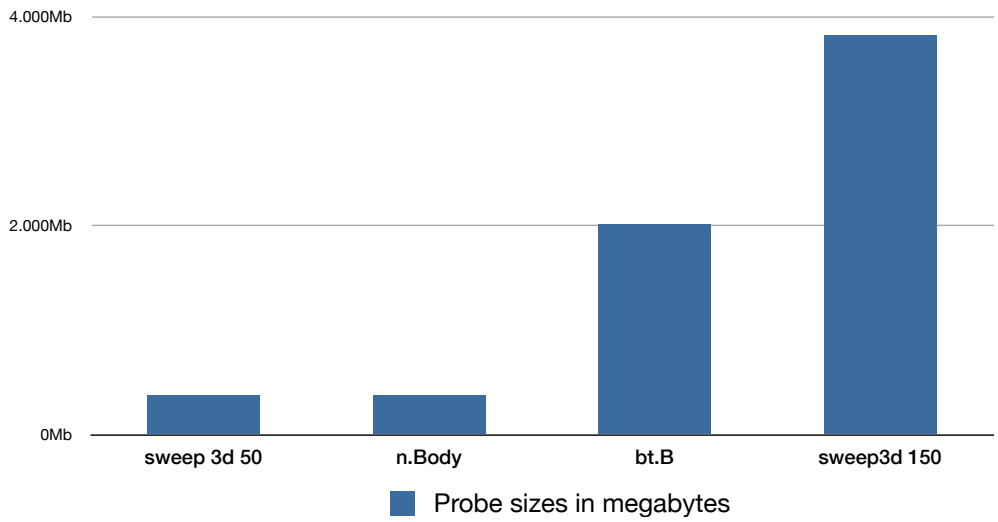

Figure 4.5: Probe sizes.

methodology, to determine the performance of the application, a whole long-running task was sent to *every* available node on the whole multi-cluster. That may take *long* times, according to the time of a single worker task.

What our Probe does is to fill this gap of the $Perf_{Avail}$ value in Argollo's research in a way fast enough that the model can determine the value for this variable to make dynamic adaptations to the performance of the multi-cluster. This way it is possible to reschedule tasks and discard those computing elements not fast enough to help with the computation efficiently during execution, with no need for a previous setup stage for performance discovery, which greatly enhances the usefulness of his model.

Chapter 5

Reduced Probe

Off with her head!— The Queen

5.1 Introduction

Chapter 4 described the basic concept of the probe, how to build it and what it does. One issue with this methodology is that by generating a number of checkpoints in order to characterize an application means that the Probes themselves can be pretty large.

In previous experiments, we found Probes in order of gigabytes. In chapter 6, one of the Probes is around 1.3 gigabyte. This means a transmission time of around three hours in a 1Mbps connection, something not totally unusual on the Internet. A strategy to reduce probe size was essential to make it useful in the multi-cluster environment.

In order to make the program mimic the functionality of the original application, two parameters are essential: memory access pattern and program flow.

5.1.1 Memory access pattern

Some researchers, as Toomula and Subhlok, for instance, go as far as to say that replicating only the memory access pattern is enough to predict performance [83], as access times for modern memories are orders of magnitude longer than computing time, and a cache failure can account for hundreds of clock cycles spend with the CPU being idle.

Other researchers, such as Weinberg et. al. [90], try to match the temporal and

spatial locality of memory access patterns of HPC applications with those of some benchmarks. Their insight is that it is possible to derive application performance in a machine from metrics given by execution of such benchmarks and its correspondency.

Andrade et.al. even developed a method to guide the compiler optimization process in respect to cache miss probability [4], [3], [6], [5].

Hollingsworth created a tool which is able to model memory patterns to predict the performance of future computing systems [35]. So we can safely assert that the memory access pattern is one of the important factors for performance prediction.

5.1.2 Program control flow

The other parameter relevant for performance prediction is the program control flow. Modern processor cores have multiple pipelines and duplicated functional units for thread parallelism, among other characteristics. To fully mimic a program behavior, computation is important as well.

The control flow is closely associated to the memory access pattern, as the decision flow is usually related to the contents of the memory in some moment.

Our initial research on mimicking only the memory access patterns to the exclusion of all other factors did not produce the performance prediction precision required to feed the multi-cluster scheduler with any useful informatio, so this line was promptly abandoned. One outcome of the need to keep the computation and the call path and decision flow of the application to be characterized is that the memory contents must also be kept in the Probe. Which, as already stated, leads to sizes big enough to diminish the usefulness of the probe on our target environment, the Internet.

As our research advanced through the use of checkpoints, it became evident that the sizes of the Probes were getting bigger with every iteration of applications. Memory capacity is increasing steadily, so are the data sets of scientific applications.

However, our empirical observations suggests that the average internet speed is not following suit. So these two factors: data sets already big (and therefore, Probes) getting even bigger, and the network not becoming faster on the same proportion, urged us to look to this issue of Probe size as a priority.

So we started to seek ways of reducing the Probe size, and this is further explained in this chapter.

5.2 Reduction

We investigated multiple ways to reduce Probe size. Currently, we use three different methods.

They are:

- Removal of less-important phases,
- Compression, and
- Touched set approach.

They will be detailed on the following sections.

5.3 Removal of less-important phases

This approach is straightforward - it consists of selectively not carrying the checkpoints of the less relevant phases. The tradeoff here lies between a possibly dramatic reduction of the Probe's size, with little complexity, as it is just the removal of checkpoints of phases with lesser weights against the loss of precision.

With the weights in hand, we know exactly the remainder percentage of our prediction stays. With a rule of three, we extrapolate that to the full execution.

This method is particularly useful in programs whose behavior is dominated by a small number of functions. This seem to be the case in several scientific applications, which perform transformations in data in very specific patterns.

Some example of extreme cases are those of workflow applications, where the individual tasks that transform the data were already separated in the workflow's nodes. As current implementations of workflows' nodes create separate executable files for each node, so we must create separate probes for each of them, and experiments shown us that these nodes are mostly dominated by one or two single phases, so it is difficult to remove any of those phases from the Probe.

In all our experiments, we set Simpoint to output a maximum of 30 phases. This is a number high enough to represent the whole application's behavior in practically every application we tested, even those with varying behavior during execution. The disadvantage is that it is also possible to come with a high number of phases, meaning big amount of checkpoints, up to a maximum of 30. We also tell Simpoint to automatically discard phases beforehand - those whose sum accounts for a maximum 1% of the overall execution are automatically discarded.

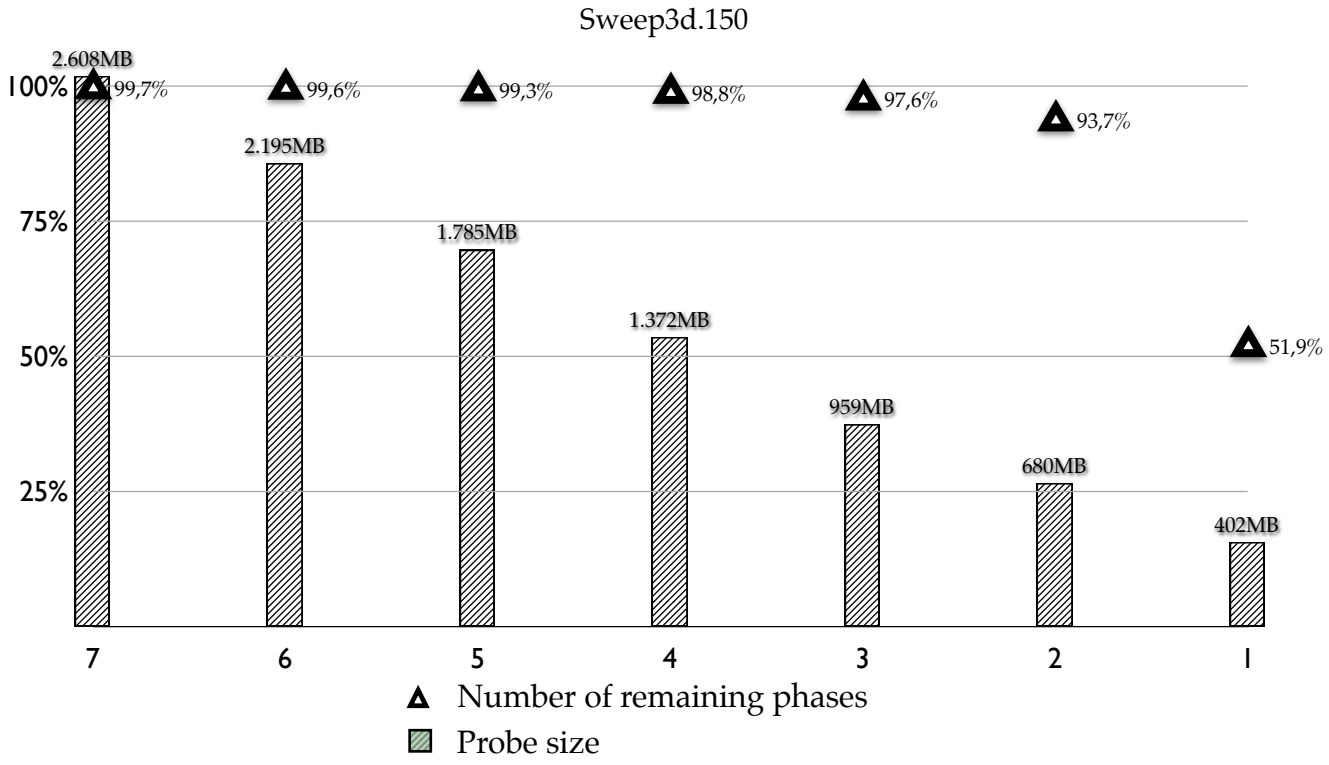

Figure 5.1: Prediction Quality x Probe Size.

In these cases, the Simpoint algorithm can output a number of phases that are little to no relevant to the overall execution, such as initialization and end phases, and during changes from important parts of the overall execution. Several of these phases can be discarded with little loss of prediction.

One example of phase removal is that of Figure 5.1. It shows the checkpoint sizes (thus, the phases) of the SWEEP3D application and the overall prediction quality and how this quality degrades whenthe number of phases gets reduced. It can also be seen how the Probe's size is also reduced.

This example shows that SWEEP3D's execution is dominated by two distinct phases - and keeping only those two phases and discarding the remaining, less-important ones, still gives us a prediction accuracy of more than 93%.

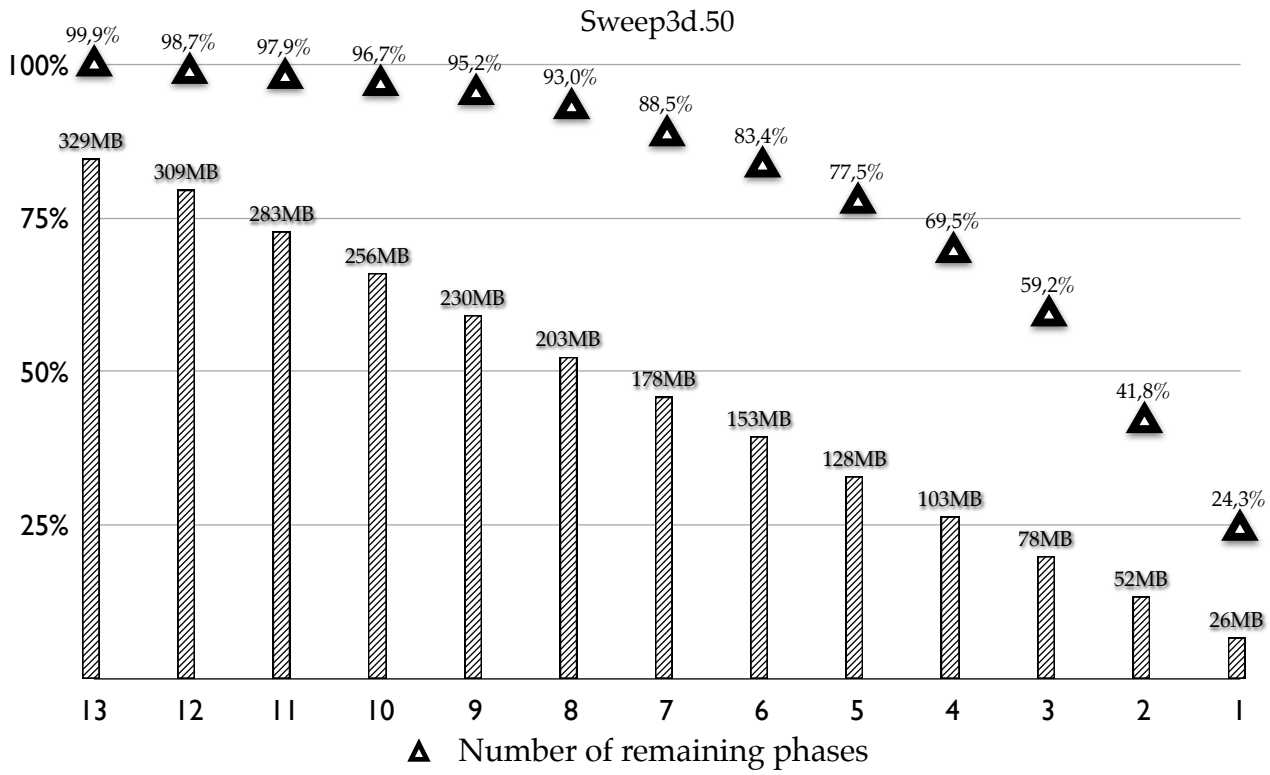

Figure 5.2: Prediction Quality x Probe Size in a smaller experiment.

The removal of little-relevant phases is valid, and in fact is necessary, as keeping phases with little importance can greatly increase the total Probe size with little gain in quality.

There is a tradeoff, however. The same application, but with a smaller problem size, yielded a different result, as can be seen at Figure 5.2. As the problem size is so much smaller, the two dominant functions have a smaller proportion on the execution, as they are essentially what increase with data size, while the other parts of the program are more or less static. So, in this case, keeping only the two most important phases would let us with a degree of precision of less than 42% - even if the Probe size was reduced in 83%. Therefore, how many phases are to be be kept is up to the user.

5.4 Compression

Compression itself is a widely studied subject in computer science for decades and alien to the scope of this work, so it used only as a tool, as is. We use the gzip compression tool, given its fast compression/decompression times, and its ubiquitous presence in UNIX systems.

Compressing checkpoints as they were yielded different results according to the workload and application, ranging from 20% to 35% of size reduction. So it helps us achieve our goal of reducing the total Probe size, and it can - and is - used altogether with other approaches.

Slightly higher rates were obtained using other compression algorithms, as bzip2 and 7-zip, but the increase in compression/decompression times is disproportional to the reductions in size. In essence, the time taken to compress a file using, say, bzip2, sending it to a remote cluster and decompressing it there is bigger than gzip in all our tests except with pathologically slow networks under 30kbps, the average speed of a dial-up connection in 1995. So we kept gzip for its simplicity, availability and speed.

5.5 Touched set approach

The touched set approach is based on two simple ideas:

- A Probe's phase runs for a very limited amount of time, and is improbable that in this time the program will access all its memory contents, and
- Every compression algorithm can benefit of large sequences of repeated characters, as all of them possess some form of run-length encoding [24].

A program's phase basically consists in the application's checkpoint in a specific moment of time, that will have its execution time measured after a given number of instructions is ran. In our experiments, this number is whether 10, 50 or 100 million instructions, but mostly 100 millions. In that interval - also known as *tracking window* [52], it is probable that only a subset of the application's memory contents its accessed. If a considerable amount of memory is not going to be used during a phase's execution, this memory contents does not need to exist in the checkpoint. That is what we call *touched set approach*.

Touched set, as defined by Yawei and Zhiling [51] is the memory contents accessed during the interval - in our case, during a phase's execution. All memory not touched during the interval does not need to be present in the checkpoint.

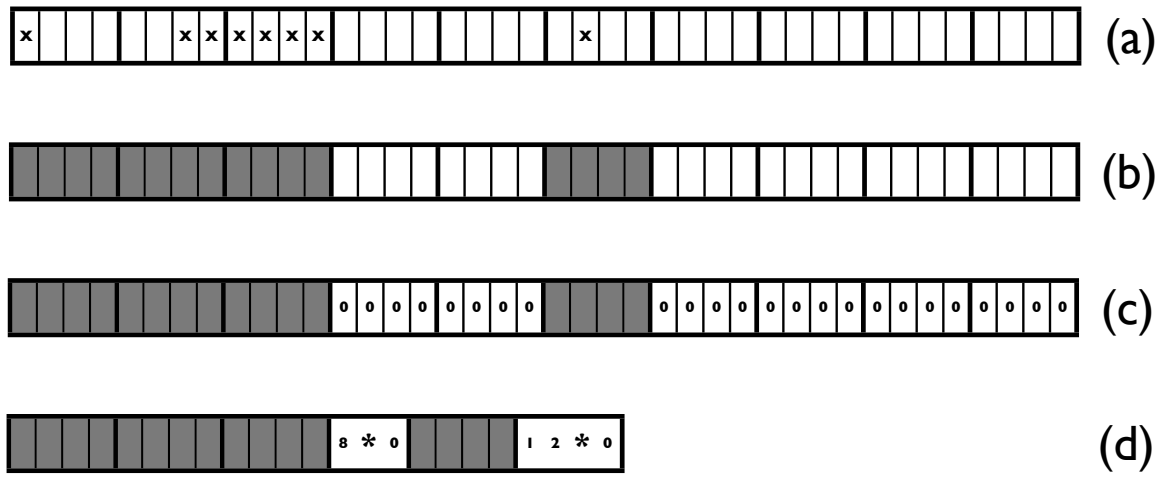

Figure 5.3: Touched set approach

This means that the whole checkpoint is not necessary for running an interval, but instead, only the memory contents effectively accessed during this interval, as exemplified in Figure 5.3(a). On it, the memory accessed during the execution of a given phase is marked with an *"X"*. As checkpoints have memory page granularity level, on Figure 5.3(b) we mark the pages to which these contents belong to as used, and on Figure 5.3(c), we fill the remaining contents with zeros, with can be compressed, as seen in Figure fig:touched-set(d). This means we can greatly trim the checkpoints to carry as little information as possible, only the memory contents that are going to be required for reproducing the phase.

Our idea was that this approach could reduce the Probe sizes drastically.

We implemented the touched set approach in our probe system by intersecting information by modifying the code both for generating our Probes and that of the checkpointing library kernel module, in the following way:

5.5.1 Probe generation

In the previous chapter, where we outlined the Probe generation process, the last step was run the application to the end in our characterization machine, in order to save the instrumented checkpoints.

The novelty we created to reduce the Probe size then is that now, in the moment right after saving a checkpoint, a trace of every memory page accessed

during its interval is recorded. That means, we still run the program thoroughly and save the checkpoints, but during what would be the interval of this phase's execution, the address every single memory page accessed is recorded.

The reviewed algorithm of the probe becomes more or less like this:

The algorithm shown at figure 5.4 saves the checkpoint just at the beginning of the warmup period, as usual. However, the change comes when the warmup interval has passed and the real phase point begins. On it, we dump the accesses to every memory page. Pin gives us instrumentation for memory accesses at byte level, but this precision is not necessary, and makes the trace files of memory accesses unreasonably large without any real need for it, as the checkpointing library maps memory pages and not individual bytes.

Which means that we either keep or remove one entire page at a time. If a single byte in a page was touched, the whole page will be kept on the checkpoint. This reduces the possible compression gains, but it was necessary because not doing it made the instrumentation and the trace so heavy it did not fit the machines tested (some traces were estimated in the regions of dozens of terabytes for each phase).

So, in order to reduce trace file granularity, we perform the following binary shift:

$$memory\_page = memory\_access >> 12 << 12; \tag{5.1}$$

This effectively zeroes the twelve less-significant bits of the address, giving us a granularity of 4096 bytes, the standard page size of the Linux operating system. This greatly reduces the size of the trace, as an array transposition, for instance, would generate a single line for every 4096 positions, instead of one for each element accessed.

Besides this trace, additional information is required, which is discussed on the next session:

5.5.2 Checkpointing library kernel module

The memory access trace must be matched with the contents on the checkpoints. The problem was that BLCR's checkpoint file structure is not documented, and the headers are of variable size.

In order to know the offset where a given memory page is recorded on this file, we had to modify the checkpoint kernel module to also create a trace of the tuple [memory page address / file offset].

```
phases=[simpoint output];
MAX_INSTRUCTIONS=100000000; // This is actually an argument
WARMUP=7000;                // Instructions for warmup
inscounter=0;               // Instruction counter
counter=MAX_INSTRUCTIONS;   // Negative counter of one phase
if(mode==MONITORING):
    switch(inscounter):
        case phases[x]*MAX_INSTRUCTIONS - WARMUP):  // Point to checkpoint
            make checkpoint;
            x++;
            break;
        case phases[x]:     // Point to start monitorin memory accesses:

            // Instrument only the instructions that touch main memory
            mem_address=instrument(instr_mem_access, memory_effective_address);

            // Granularity is at byte level, but checkpoint works at
            //  memory pages level, so only stores page addresses.
            mem_page=mem_address >> 12 << 12;
            break;

            default:
                inscounter++;

else if(mode==RESTART_FROM_CHECKPOINT):
    start_time=time.now();
    loop: if(counter--==0):         // JE in asm
        stop_execution;
        end_time=time.now();
        printf(start_time, end_time);
        exit(0);
```

Figure 5.4: Pseudocode for monitoring memory page addresses

With the touched set and the information given by the kernel module in hands, the only data required to stay in the checkpoint file is:

- File header,
- Process information required for restoring (pid, uid, gid, etc.),
- Part of the code segment (the one being used in this phase),
- The touched set.

The memory not accessed during the execution of the Phase interval is effectively discarded.

The touched set ξ in the checkpoint file is given by the set definition 5.2:

$$\xi = (\theta \cap \varphi) \tag{5.2}$$

And the set of pages ζ to be removed from the checkpoint file is in definition 5.3:

$$\zeta = (\theta \cup \varphi) - \xi \tag{5.3}$$

Where θ is the touched set as recorded when running the phase, and φ is the offset of those pages in the checkpoint file. Everything else can then be removed from the checkpoint file, thus reducing its size drastically.

Our first approach was to simply trim ζ from the checkpoint file, and change the headers accordingly, in order to denote where each memory page part of ξ is, directly reducing checkpoint's file size, as the non-used portions were not recorded at all, making it a densely-populated file. This presented some major drawbacks:

- Implementation cost: the checkpointing library needed to be extensively changed in the restart code, in order to accommodate the "jumps" we created on the pages' sequence, and
- Binary compatibility: as the checkpoint restart code was changed, we had a checkpointing library kernel module that needed to be installed in every system we would test our probe, which would break with the premise for the election of BLCR as the checkpointing library of choice given its availability in several different clusters, and we would not be able to restore "regular" checkpoint files on that, either, breaking functionality for other users.

For that reason, we created a simpler approach:

The memory pages present in the set ζ, that, for our purposes, can be discarded are instead filled with zeros, so the file can be now considered sparse.

The implementation of that is a Python program that reads all addresses of memory pages from the kernel module output for this specific checkpoint and transforms it into a vector of tuples containing the memory page and file offset. Then, it reads the trace of memory pages accessed during the execution of this interval, and excludes the pages read from the trace file from its vector. The positions that remain on the vector are those which were not accessed during this phase, and are therefore zeroed out in the checkpoint file.

The biggest advantages are:

- It keeps binary compatibility with standard BLCR installations in other clusters - which was really welcome by these clusters' administrators, our checkpoints are binary-compatible with the standard BLCR;
- Although the checkpoint file itself does not change, compression ratios increase greatly, as the memory pages that were discarded now are just big sequences of zeroes, which even the most primitive run-length encoding compression algorithm is able to take advantage of.

In resume, the steps are:

- The program is executed, instrumented;
- Simpoint runs over the BBV file and discovers the phases;
- The program is run again, instrumented, and makes checkpoints at the moment of a phase warmup with information about the memory pages' offset in the file dumped from the kernel module;
- The program keeps running. When it passes the warmup phase, it starts dumping the memory page addresses until the end of the phase;
- The program keeps running and doing the two previous points until all phases are checkpointed and traced;
- With the memory trace and checkpoint offsets matched, the memory pages ζ are zeroed from the checkpoint by a special utility;
- Previous point is made for all checkpoints;

- Probe can be compressed now, with the sparse checkpoint files, which reduces them significantly.

So, after the program was characterized, its phases discovered, the phases saved, a special program gathers the data coming from the kernel and that from the memory trace for each phase.

Then it creates the set ζ and zeroes it from that checkpoint's phase and compresses this now sparse file. Then it proceeds to the next phase, up to when there are no more phases to reduce.

The final output is a set of gzipped files. One further idea to reduce probe size is one characteristic of gzip compression: it can be decompressed before the whole file is present. So a probe could be sent through the network and uncompressed on the fly, with a pipe from the process receiving the byte stream to the *gunzip* utility. This eliminates the need to receive the whole file and then start the uncompression process, masking the time required for it, as it is done while receiving the file.

The resulting file on the other end is a checkpoint file containing only the memory contents required for running the original application's phase from the warmup to the phase's size plus a very light instrumentation required to measure phase execution time. After the phase reaches its designated number of instructions, time is registered and the execution is interrupted, allowing the execution of the next phase.

The combination of the aforementioned methods: removal of little-relevant phases, touched set approach and compression is what allows us to achieve great reductions in probe size, up to 95% smaller when compared to the original set of checkpoints, as we will show in the next chapter.

Several characteristics that can be seen in our method are not from the method *per se*, but instead it reflects those from the tools used to build this prototype. Some of them are:

- The application's binary and its open files must be present in the same locations on the machine to be characterized: This is a characteristic of BLCR, not of our method.

- Instrumentation library's *.so* files must be present on the machine to be characterized, on the same location: same as again, a requirement from BLCR. As BLCR does not "see" the instrumentation library, but instead thinks that it is part of the application itself, it requires them to be on the same location on the file system.

- Sockets are not restored by default: special support from, for instance, MPI libraries, must be present for restoring network connections. As we are characterizing the worker during its steady state, this is not really important at this moment.

Chapter 6

Experimental study

Begin at the beginning and go on till you come to the end: then stop.— The King

6.1 Introduction

As we seen on the previous chapters, this work studies the problem of characterizing the performance of a given application running on a machine in a fraction of the time required to run this application thoroughly.

The goal is to create knowledge of this application performance, so Argollo's method can use it to decide if this machine is worthy or not for execution according to an efficient threshold.

To reach this objective, we adopted the concept of a software probe that will run only representative parts of an application. Those parts will provide us with performance knowledge similar to the one which would be possible running the same application entirely on the given machine and we can then predict execution time.

To prove the concept of the Probe, we did some experiments, both regarding the accuracy of the probe as well as the possible gains in reductions. We characterized the applications and created the Probes in our reference machine. Then we ran these Probes on another machine or machines, predicted the execution time and compared with the real execution time on this machine.

On them, phases' times were compared, and weights were applied to each of those phases. The weights are given to us by *Simpoint*, and they mean the number of executions of that phase over the number of executions of all phases, i.e. it describes how important a phase is. This comparison made possible to extrapolate the execution time on the probed machine.

The first experiment was done with a double-nested loop matrix multiplication. Being a program with a well known behavior, we could verify if the method and our tool worked as expected. Being so simple and with a known memory access pattern (*stride*), *Simpoint* correctly found only one phase representative of 99,9% of program's behavior. and the Phase found was close to the middle of the execution. This proved that the general system works and that we might proceed to test execution time of more complex applications.

6.2 Experimental results

We realized a series experiments to verify the quality of our prediction.

We ran and characterized the worker in Master/Worker versions of the ASC's Sweep3d [33], a Matrix Multiplication, a n-Body problem, the NAS Benchmarks [11] BT and DC, class B. We show these specific experiments because these specific problem sizes took roughly similar execution times, although providing very distinct phase behavior. These are average results for the execution of a single worker. This probing would be done on each and every node available for execution in the parallel environment.

The testbed consisted of one Intel Pentium4 2.6ghz"Northwood" with 1 gigabyte of RAM as reference machine, and the Probes were sent to a cluster of seven Intel Pentium4 "Prescott" 2.88ghz machines with 512 megabytes of RAM, and a cluster of Intel Pentium4 "Cedar Mill" 3.0ghz machines with 1024 megabytes of RAM, all in a switched Fast Ethernet network.

Although Pentium 4 indicates the same technology, they are different processors, with different manufacturing methods and transistor sizes. It is worth noting that the comparisons is not only among CPUs, but instead the whole computer - a small amount of memory may affect an application's performance as much (or more, in extreme cases) than the number of pipeline stages or cache sizes.

In this work's experiments, each cluster was homogeneous, although they were heterogeneous among themselves. It means that we would be able to further speed characterization time by running different phases in different machines in parallel. If the clusters were heterogeneous, we also would gain time, as the Probe would be sent just once for the whole cluster, and each machine would be able to run it without having to download it again from our location.

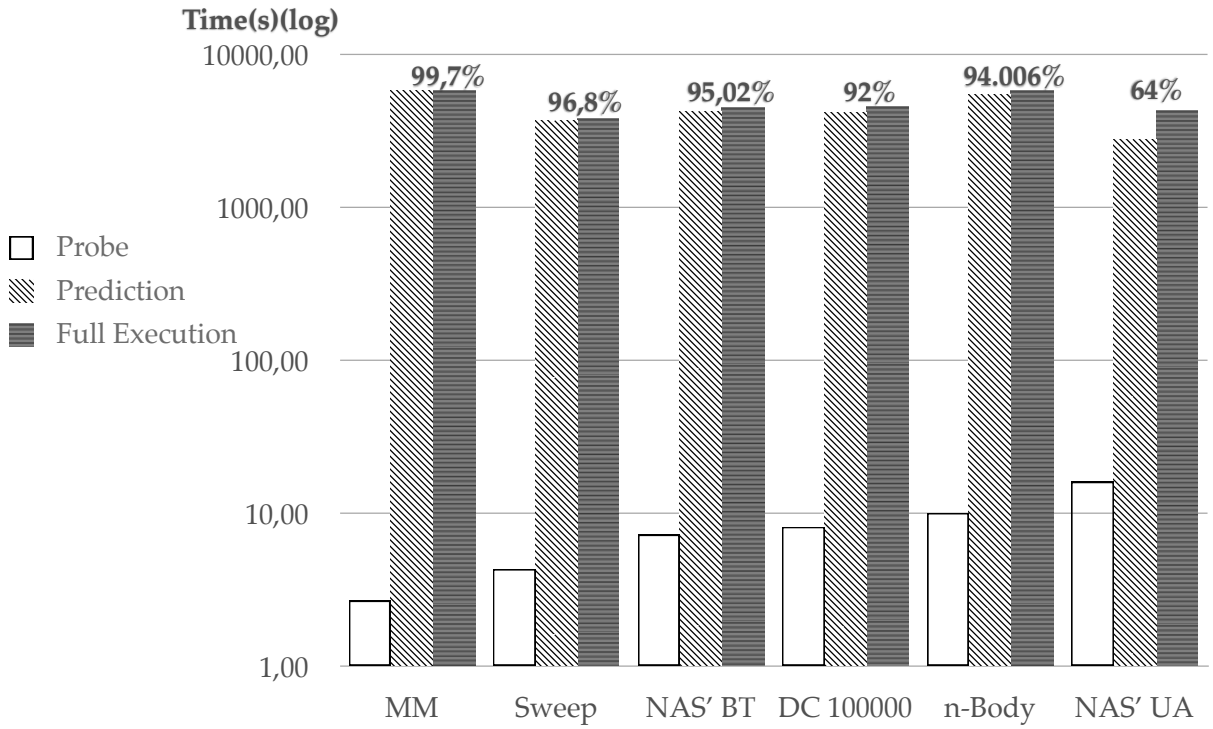

Figure 6.1: Execution time, prediction and probing time in log scale.

6.2.1 Precision

Figure 6.1 summarizes the prediction quality of our Probes. The bar in white is the execution time taken by the Probe to run. The bars in light grey are the predicted execution time calculated from these Probes' executions, and the dark grey bars are real executions of the same applications on these machines. The graph is in log scale, because the Probes took orders of magnitude less time to run than the actual application and would not appear there at all otherwise.

We chose problem sizes who took similar execution times for the sake of presentation. For the tested applications, the precision stood above 92%, while the probing time was around 0.2% of the original application's execution time for this set of experiments, with the exception of NAS' UA (more on that later). Longer applications in general yields even better reductions, consequence of a higher number of repetitions of the same phase. The probing time doesn't take into account checkpoint transmission and checkpoint loading, instead focus on its phase execution time.

The Matrix Multiplication is an obvious case, with a single phase that correctly represented the execution time.

The Sweep3d application, using a grid of 150x150x150 units, possesses a memory usage around 436 megabytes in most of its execution, except for the startup. Simpoint detects between 7 and 13 phases for it (according to the random dimension projection, the results can vary), with the behavior dominated in more than 90% by two phases, as shown in the previous chapter. In the case of ten phases, the Probe able to represent 99,7% of its total behavior actually predicted the execution time with 96% accuracy with all phases. On the next section, we show the reductions in size and prediction for this experiment.

The NAS' BT has three main phases that dominate more than 60% of the execution time, but in total, it produced 17 different phases for the problem size category B. The phases ran in less than ten seconds, but were able to predict the execution time of the full execution, which took around two hours, with an error of less than 5%.

The NAS' DC had our worst prediction quality with the exception of UA. The time predicted from its Probe was 8% less than the real application. Yet, this is still enough to show differences between execution times between different machines in orders of magnitude faster than the executions themselves.

The n-Body Probe was able to predict the execution time of the original application, with a correctness of 94%. Interestingly, this Probe had the smallest size of them all, keeping the state of the program around four megabytes throughout the whole execution. However, it was the probe that took longer to execute, close

to ten seconds for its ten phases. I suppose this has to do with a badly projected application with a pathological number of cache misses. More studies on the implementation and with other tools could give us some answers on this case. Still, the prediction information is enough for our purposes.

The UA case is special. It has a much worse prediction quality than any other experiment. Turned out that the Simpoint found 16 phases for it, but our reduced probes from phases 11 to 16 failed to execute, as the checkpoint segfaults immediately after restarting on such phases, so it was a forced reduction in quality which accounts to 38% of the execution, so our method was able to measure only about 62% of the program's behavior.

Upon further investigation, we realized that it might be a bug in the Pin toolkit itself or in our instrumentation code: specifically where we trace individual memory accesses. It seems that some bytes accessed were not recorded by the trace, which then reported them as not used, prone to further trimming.

We suspect that it can be some data type very close to the border of a memory page, which means that only the parts pertaining to the page where this data began were recorded, but not the bytes on the next memory page. Further research is necessary.

However, when we took into account that the 62% was the remaining behavior and extrapolated from it, the prediction still was over 91%, which indicates that the computation behavior, time-wise, of these phases was not that different from the previous ones.

If we considered only the first ten phases as being representative of the whole program and proceeded to reduce it using the techniques explained in the previous chapter, we would end up with a prediction quality of 53,4%, which, when extrapolated to the full execution, led us to predict the execution time of the UA benchmark on this machine with an accuracy of 89,5%. This reduced probe was 87% percent smaller than the one comprised by the original ten checkpoints, being reduced from around 221 megabytes to 27 megabytes.

6.2.2 Probe Transmission Time

As different programs have distinct number of phases, the number of checkpoints is evidently different. For the sake of illustration, we ran the Sweep3d application with a smaller problem size. The first group used a 50-unit cube, which took about 40 seconds to run. Our characterization method found 19 distinct phases in this execution, and the Probe comprised the application and these 19 checkpoints. The sum of checkpoints' sizes was more than 372 megabytes. Consequently, a set of files

this big would take a little bit over an hour to transmit over a 1 mbps network link, in the case the transmission is perfect. Also this specific case, our Probe would not worth the effort, as this is just an example of a very short execution where it took longer to be produced and sent than the execution time of the application itself. The obvious conclusion is that the longer the application's execution time, the more useful the Probes become.

However, when looking at the n-Body example, its state was kept constant around 4.2 megabytes, and the whole Probe had approximately 43 megabytes. This program took more than two hours to run, but the Probe's transmission time would be of only 5 minutes in the same hypothetic 1mbps network link. The time in our multi-cluster, with a fast network, is negligible.

When Sweep 3d used the 150-unit cube, the total Probe size was around 2.8 gigabytes, while BT.B was around 1.8 gigabytes. Sending both of them would be overkill, so now we experiment with reduced probes while trying to maintain prediction quality with the techniques explained on the previous chapter.

6.2.3 Reducing Probe size

Are those checkpoints really necessary? We set our characterization to reflect 99% of the execution, because we noted that this extra one percent gave us nothing in prediction quality while increasing enormously the number of checkpoints. But can we go even further? Can we discard phases on purpose, while maintaining quality on our precision? How much would the prediction quality loose while reducing the Probes to their bare minimum?

To answer to this question, we selectively verified our prediction quality with less and less phases. Figure 6.2 shows some of those results, We now proceed to discuss them individually.

Sweep3d.50

Figure 6.3 shows us a short execution, therefore there's no dominant behavior. This means that phases are, in general, equally important, and the curve is smooth.

Although this makes easier to choose the prediction precision, it makes bigger Probe size reductions more difficult. We experimented with keeping the five most relevant phases. Before extrapolation with a rule of three, this kept the prediction quality in about 78% of the execution. But in this experiment, the original Probe had more than 400 megabytes; Reducing it with keeping only the five most relevant phases, trimming the probes using the touched-set approach and compressing the

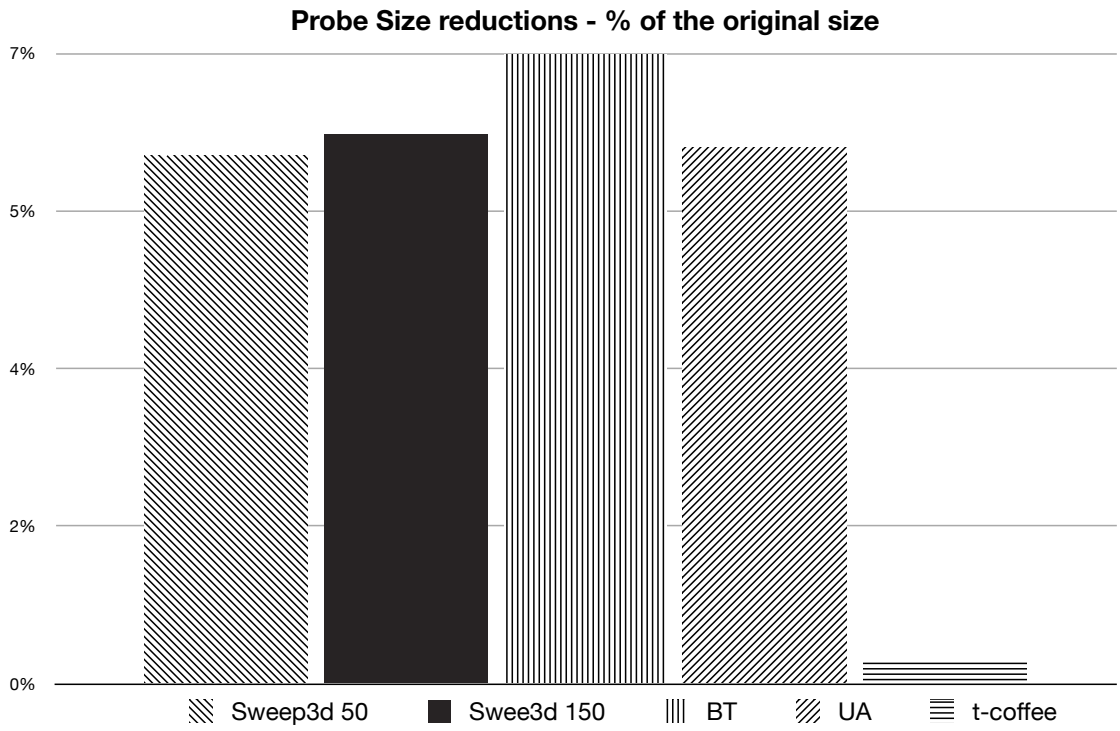

Figure 6.2: Reduction

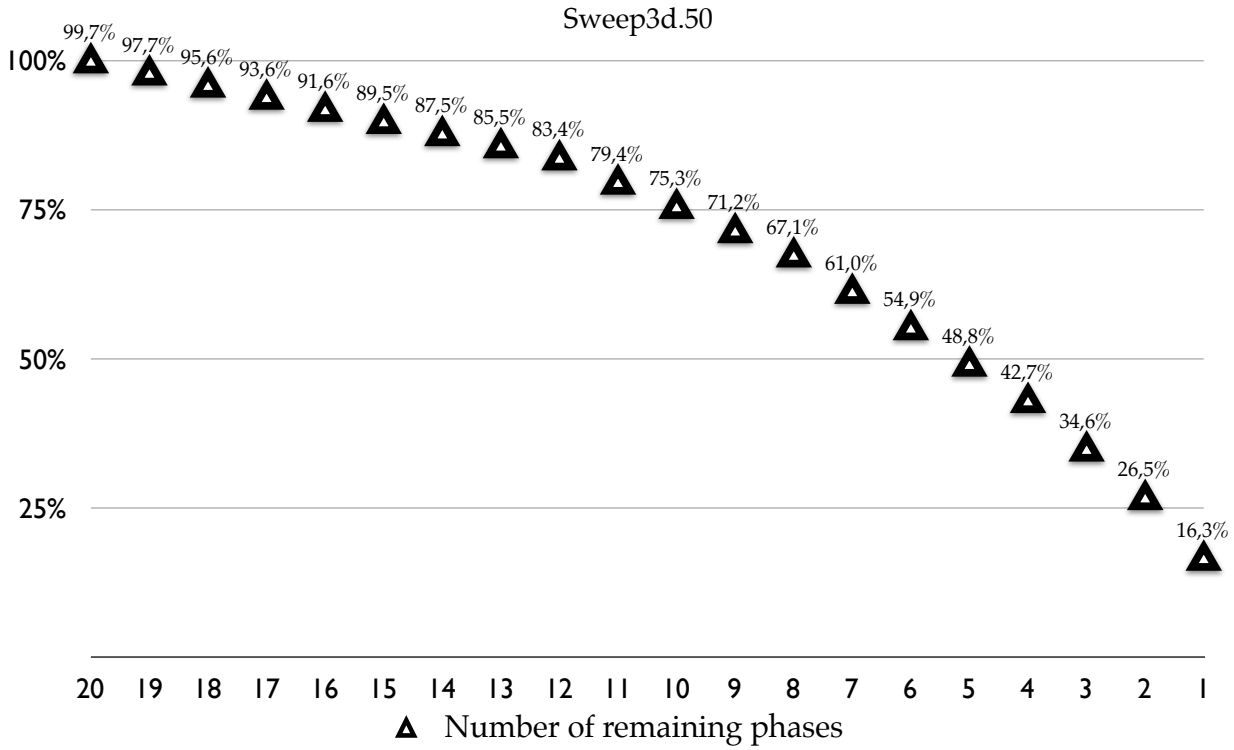

Figure 6.3: Prediction quality (without extrapolation) of the Sweep3D application with the cube sized as 50 units

resulting files let the probe with 19,2 megabytes. Still, this execution is so short that there is little point in making this process for this application/data size pair.

Sweep3d.150

Interestingly, we had two wildly different executions of the Sweep3d during our tests. In Figure 6.4, we see that *Simpoint* gave us seven phases, where execution is highly dominated by two of them, being others of little relevance. With this data in hand, we can selectively remove the less important phases while maintaining the desired prediction quality.

We decided to keep only the three most relevant phases in this experiment. Its quality should be around 97%, which held true by the experiments. However, even after trimmed down to one third of its original size by discarding the less-relevant phases, the Probe is still a behemoth of 959 megabytes. It would take about 2h20m to be transferred in a 1mbps network link. The communication time is almost as

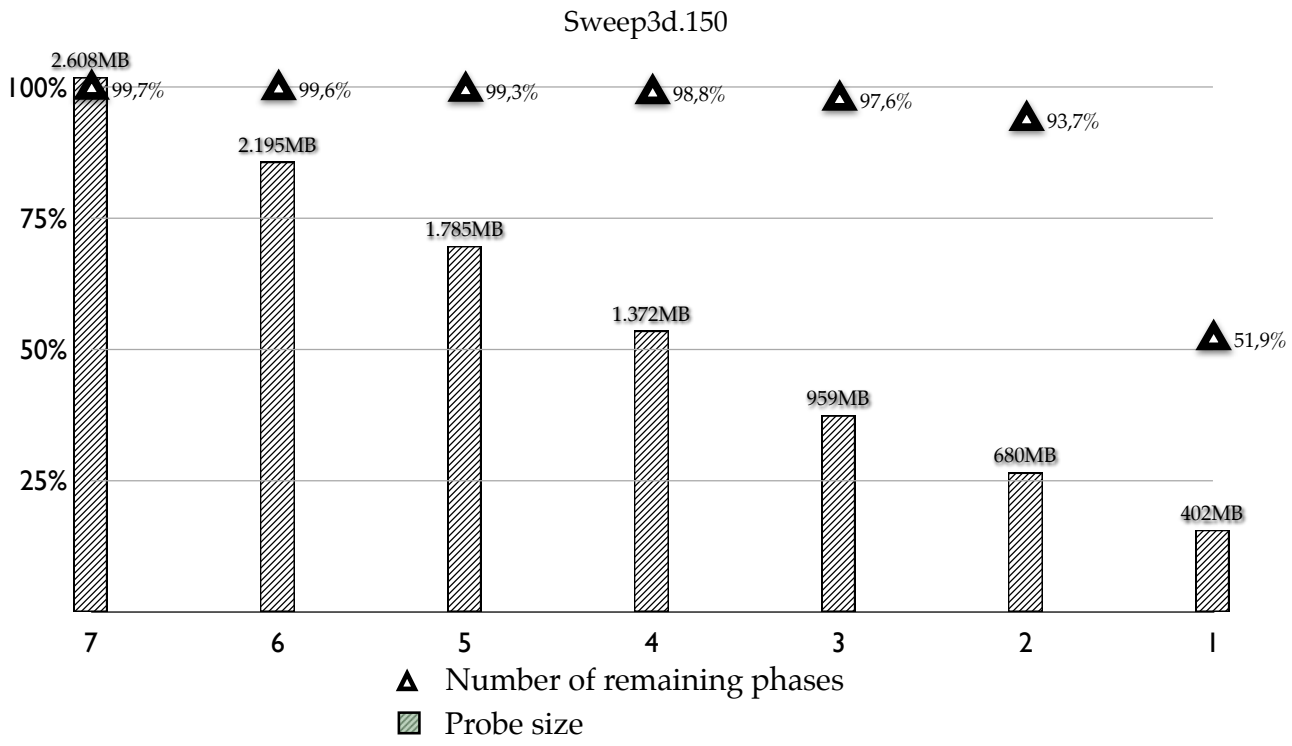

Figure 6.4: Prediction quality (without extrapolation, touched-set or compression) of the Sweep3D application with the cube sized as 150 units, versus probe size

big as the execution itself.

After using the touched-set approach and standard gzip compression, the Probe had 151 megabytes, a reduction of 94% from its original size of 2608 megabytes. In the 1mbps network link, this would take circa 20 minutes to be transferred.

A different execution of the same test gave us a totally different result. In Figure 6.5, the same application when run under our characterization method yielded 14 phases, with around 87% of its behavior in the first nine phases, with no dominant behavior.

This is caused by the random dimension projection algorithm in Simpoint. As the number of Basic Block Vectors and of Basic Blocks itself is huge, Simpoint projects those dimensions in a matrix of 15 dimensions in order to be computationally feasible. As the selection algorithm uses a random seed, behavior like that can appear. We noticed that several of those phases with a weight about 10% of the execution are in the same steady state of the execution, which means that they are actually similar. Their measured execution times and the checkpoint sizes was similar as well.

In this experiment, we kept the six most significant phases, which by itself would give us a quality of 70% of prediction of the total execution time. With extrapolation, this value was 93% of the actual execution time on the probed machines, so the methodology worked, even in this case. From the 5440 megabytes of the original Probe, the reduced version ended up with 330 megabytes, which is about 6% of its original size. In a slow network of 1mbps, it would take close to 40 minutes to send this Probe, so perhaps, a more drastic reduction is required in the presence of such a slow network, sacrificing prediction.

BT.B

The case shown in Figure 6.6 is a typical case of a benchmark, whose behavior is dominated by a small number of phases. In this case, 3. With these three phases, after extrapolating the behavior from 61% to the entirety of this execution, we arrived to a prediction quality of 91% of the original execution time, in average, with a reduction of 93% of the Probe size.

t-Coffee

T-Coffee is a multiple sequence alignment package, used to align Protein, DNA and RNA sequences and to combine sequence information with protein structural information, profile information or RNA secondary structures [56]. As seen in figure

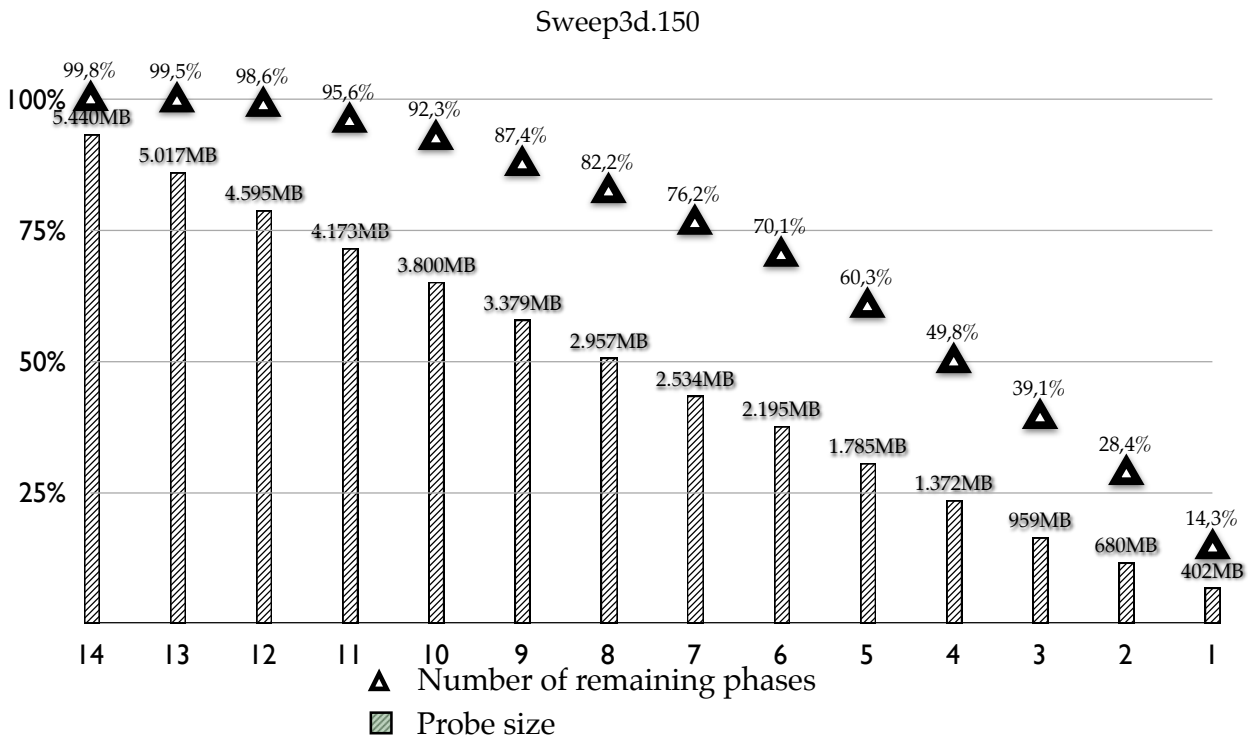

Figure 6.5: Prediction quality (without extrapolation, touched-set or compression) of the Sweep3D application with the cube sized as 150 units, versus probe size

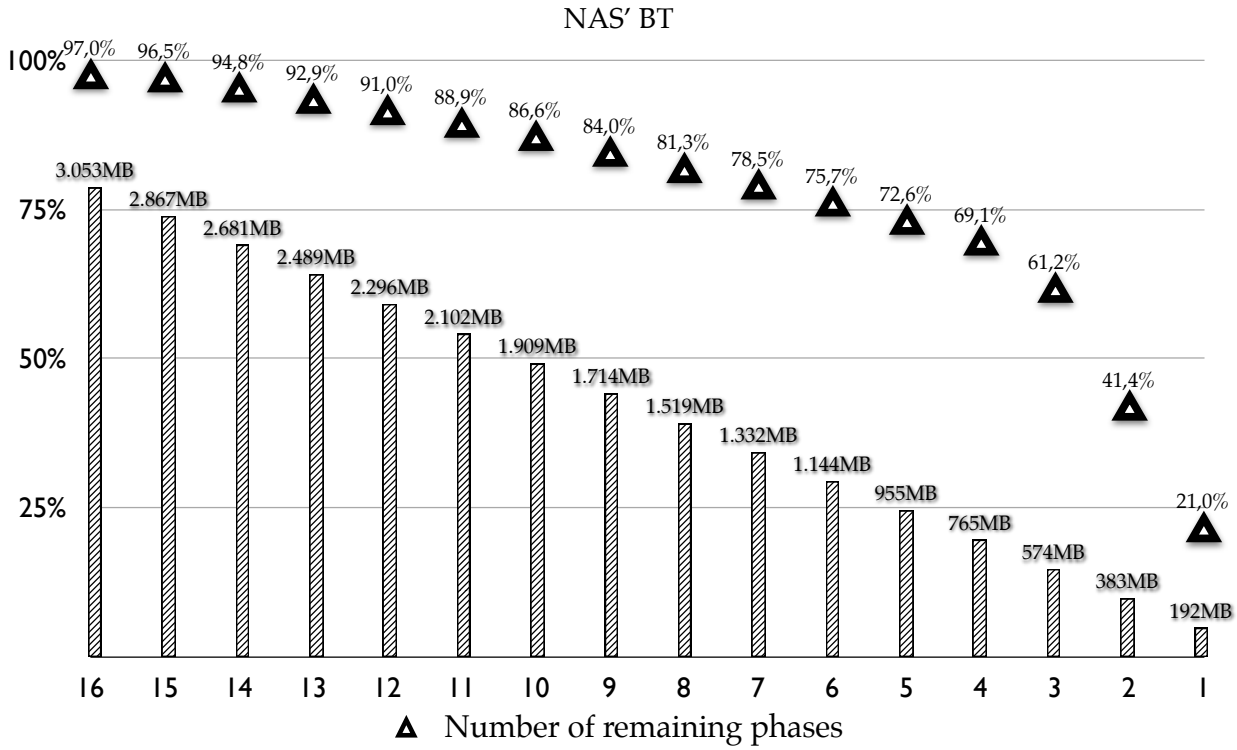

Figure 6.6: NAS' Bt.B

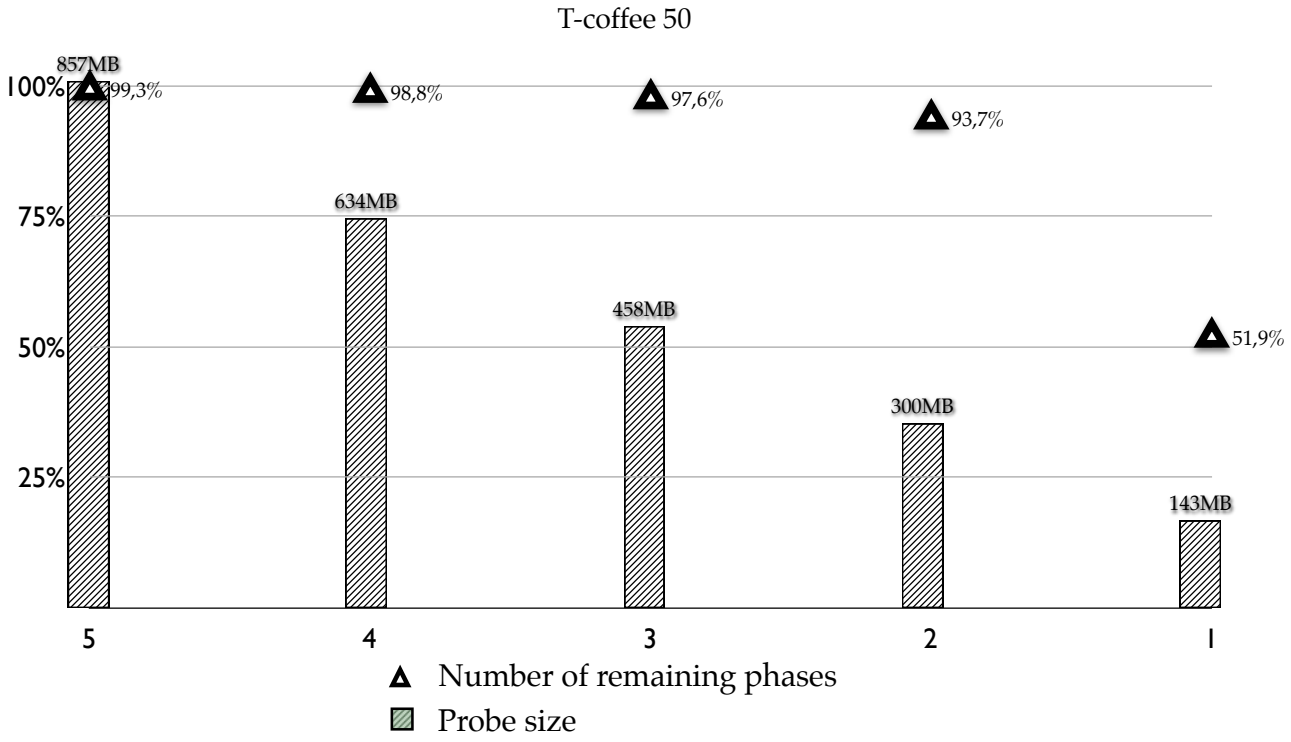

Figure 6.7: Prediction quality (without extrapolation, touched-set or compression) of the t-coffee application.

6.7, about 94% of its execution time is dominated by two phases. During these two phases, the application was working over a really small segment of data, so our reduction techniques left us with a Probe of a mere 5,6 megabytes, able to be transferred in under a minute even under our 1mbps network. This is a reduction of 98% from the original Probe, that was of 857 megabytes.

Chapter 7

Conclusion and future work

What is the use of repeating all that stuff, if you don't explain it as you go on? It's by far the most confusing thing I ever heard! — The Mock Turtle

7.1 Work contribution

This research started in order to complement previous research of [9] on the multi-cluster model, where one of the parameters of the equation for performance prediction in such model was left out as future work.

One of the ways to characterize the performance of a worker to finalize such research is what we call the software Probe.

Our Probe methodology is able to fill this gap and find the $Perf_{Avail}$ parameter necessary for equation 4.9 mentioned in chapter 4, which is itself one of the parameters of the whole model, depicted as equation 4.8 in the same chapter.

The Probe is able to predict the execution time of a worker in a matter of seconds, instead of much longer times took by running the whole worker, and much more precisely than trying to correlate the application with benchmarks.

Initial research demonstrated that the checkpoints' sizes made the Probes too big to be transported over the internet in feasible time. Further research reduced the Probes to a fraction of their original sizes, making them small enough to be transported to remote clusters in order to characterize their machines in viable time.

One characteristic of the current Probe model is that the prediction depends on the original application's input data. This is not exactly a defect, but a inherent characteristic of the application to be characterized. If the application is data-

dependent, so will the Probe be.

There are two aspects of the data-dependency "issue" that should be observed:

- Even when a data-dependent application yields wildly different results from one execution to another with different data sets of equivalent sizes, the Probes still fulfill their objective of give performance information to the scheduler, in the sense that the same Probe while run in different machines gives different results - so it is still an useful result for the matter of machine selection.
- Being the Probes as fast as they are, it is possible to construct a set of predefined probes with different input data that correlates to the spectrum of possible performances. There is ongoing research on characterizing these different input data sets started by Fritzsche, Rexachs and Luque [22], and the union of both lines of research stands as a promising future work (see next section).

7.2 Publications

This research has had as outcome the publication of the following Papers:

- *Software Probes: Towards a Quick Method for Machine Characterization and Application Performance Prediction* [80], which received the awards of best paper and best presentation on the 7th International Symposium on Parallel and Distributed Computing (ISPDC) in 2008,
- *Software Probes: A Method for Quickly Characterizing Applications' Performance on Heterogeneous Environments* [79], presented on the Workshop on Design, Optimization and Management of Heterogeneous Networked Systems of the 38th International Conference on Parallel Processing (ICPP) in 2009,
- *Improving Probe Usability* [78], presented on the Cloud Computing and Services Workshop of the 25th IEEE International Conference on Advanced Information Networking and Applications (AINA), in 2011.

7.3 Future Work

Probes are useful enough in a series of other models, not only in master/worker applications in hierarchical multi-clusters. In fact, any scheduling logic that needs information about how a given computing element will perform while running an

application can benefit from the use of Probes. Selection algorithms such as [2] and [12] can be simplified with the performance prediction given by our Probes.

Environments such as grids and clouds can use our Probes directly to replace the "benchmark" parameter of the schedulers, as it is way more related to the real application.

Different application models, such as workflows [41] may also benefit from Probes, as we may run Probes for every node of the workflow on every machine available, and run the workflow nodes where they run best.

Although the applicability of our methodology in such environments seems obvious, experiments to prove the usefulness of the Probes on them are necessary.

As already stated on the previous section, the work of Fritzsche, Rexachs and Luque [22] correlates input data sets with their performance characteristics, and it is able to predict the execution time of applications where the input data may change it radically, such as the parallel traveling salesman problem (TSP) [54], of the parallel clustering and classification of large number of documents [63]. It uses clustering techniques to correlate the position of the cities in a 2d map and the city where the travel starts with the time taken for the TSP to find the solution.

Fritzsche's solution for the specific problem of the TSP can be generalized to different kinds of applications. Their limitation was that to correlate different input data, the execution time for the thousands of experiments required was not a feasible task. Our proposal of fast Probes can make that research more general to encompass any kinds of applications, and their research can improve ours in the case of data-dependent applications, using a collection of probes of the same application with different characteristics.

Bibliography

[1] The top 500 supercomputers list - `http://www.top500.org`, June 2011.

[2] I Al-Furiah, S Aluru, S Goil, and S Ranka. Practical algorithms for selection on coarse-grained parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, Jan 1997.

[3] D Andrade, M Arenaz, B Fraguela, and J Tourino. Automated and accurate cache behavior analysis for codes with irregular access patterns. *In Proceedings of Workshop on Compilers for Parallel . . .*, Jan 2006.

[4] D Andrade, B Fraguela, and R Doallo. Analytical modeling of codes with arbitrary data-dependent conditional structures. *Journal of Systems Architecture*, Jan 2006.

[5] D Andrade, B Fraguela, and R Doallo. Precise automatable analytical modeling of the cache behavior of codes with indirections. *portal.acm.org*, Jan 2007.

[6] Manuel Diego Andrade. Automated and accurate cache behavior analysis for codes with irregular access patterns. *Concurrency and Computation: Practice and Experience*, 19(18):2407–2423, 2007.

[7] Murali Annavaram, Ryan Rakvic, Marzia Polito, Jean-Yves Bouguet, Richard A. Hankins, and Bob Davies. The fuzzy correlation between code and performance predictability. pages 93–104, 2004.

[8] E Argollo, A Gaudiani, D Rexachs, and E Luque. Tuning application in a multi-cluster environment. *Proceedings of the Euro-Par 2006*, pages 78–98, Jan 2006.

[9] Eduardo Argollo and Emilio Luque. Performance prediction and tuning in a multi-cluster environment. *Ph.D. Dissertation*, Oct 2006.

[10] D Bailey. Unfavorable strides in cache memory systems (rnr technical report rnr-92-015). *Scientific Programming*, Jan 1995.

[11] DH Bailey, E. Barszcz, JT Barton, DS Browning, RL Carter, L. Dagum, and et al. Fatoohi. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, Jan 1991.

[12] J Barbosa, J Tavares, and A.J Padilha. Linear algebra algorithms in heterogeneous cluster of personal computers. *hcw*, 00:147, 2000.

[13] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello. MPICH-V project: A multiprotocol automatic fault-tolerant MPI. *International Journal of High Performance Computing Applications*, 20(3):319, 2006.

[14] Doug Burger and Todd M Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[15] M Cierniak, M Zaki, and W Li. Compile-time scheduling algorithms for a heterogeneous network of workstations. *The Computer Journal*, 40(6):356–372, 1997.

[16] H Curnow and B Wichmann. A synthetic benchmark. *The Computer Journal*, Jan 1976.

[17] A.S Dhodapkar and J.E Smith. Comparing program phase detection techniques. *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 217– 227, 2003.

[18] Jack Dongarra. The linpack benchmark: An explanation. In E. Houstis, T. Papatheodorou, and C. Polychronopoulos, editors, *Supercomputing*, volume 297 of *Lecture Notes in Computer Science*, pages 456–474. Springer Berlin / Heidelberg, 1988.

[19] J Duell. The design and implementation of berkeley labs linux checkpoint/restart. *Tr, Lawrence Berkeley National Laboratory*, 2000.

[20] J Duell, PH Hargrove, and ES Roman. Requirements for linux checkpoint/restart. 2002.

[21] B Fraguela, R Doallo, and E Zapata. Probabilistic miss equations: evaluating memory hierarchy performance. *IEEE Transactions on Computers*, Jan 2003.

[22] P. Fritzsche. Podemos Predecir en Algoritmos Paralelos no Deterministas? *Ph.D. dissertation*, 2007.

[23] C Glasner and J Volkert. Adaps-a three-phase adaptive prediction system for the run-time of jobs based on user behaviour. *CISIS '09. International Conference on Complex, Intelligent and Software Intensive Systems, 2009.*, Jan 2010.

[24] S Golomb. Run-length encodings (corresp.). *Information Theory, IEEE Transactions on*, 12(3):399 – 401, 1966.

[25] J Goux, J Linderoth, and M Yoder.... Metacomputing and the master-worker paradigm. *Preprint MCS/ANL-P792-0200 ...*, Jan 2000.

[26] G Griem, L Oliker, J Shalf, and K Yelick. Identifying performance bottlenecks on modern microarchitectures using an adaptable probe. *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004.

[27] G Hamerly, E Perelman, and B Calder. How to use simpoint to pick simulation points. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):25–30, 2004.

[28] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, Sep 2005.

[29] P.H. Hargrove and J.C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. *Journal of Physics: Conference Series*, 46(1):494–499, 2006.

[30] J.W Haskins and K Skadron. Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation. *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on*, pages 195– 203, 2003.

[31] Kim Hazelwood, Dan Connors, David Kaeli, and Vijay Janapa Reddi. Using pin for compiler and computer architecture research and education. *ACM Sigplan 2007, PLDI Tutorial 2007*, Jun 2007.

[32] A Hoisie, O Lubeck, and H Wasserman. Scalability analysis of multidimensional wavefront algorithms on large-scale smp clusters. *Frontiers of Massively Parallel Computation*, Jan 1999.

[33] A Hoisie, O Lubeck, and H Wasserman. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. *International Journal of High Performance Computing Applications*, pages 330–346, Jan 2000.

[34] JK Hollingsworth, BP Miller, and J Cargille. Dynamic program instrumentation for scalable performance tools. *Scalable High-Performance Computing Conference, 1994. Proceedings of the*, pages 841–850, 1994.

[35] JK Hollingsworth, A Snavely, S Sbaraglia, and K Ekanadham. Emps: An environment for memory performance studies. *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 223b–223b, 2005.

[36] K Hoste and L Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, Jan 2007.

[37] L John. Spec cpu2000: Measuring cpu performance in the new millennium. *IEEE Computer*, Jan 2000.

[38] A Joshi, L Eeckhout, and L John. The return of synthetic benchmarks. *spec.org*.

[39] A Joshi, L Eeckhout, L John, and C Isen. Automated microprocessor stressmark generation. *Proceedings of International Symposium on High Performance ...*, Jan 2008.

[40] Ajay Joshi, Lieven Eeckhout, Robert Bell, Jr, and Lizy John. Distilling the essence of proprietary workloads into miniature benchmarks. *Transactions on Architecture and Code Optimization (TACO*, 5(2), Aug 2008.

[41] DS Katz, JC Jacob, E. Deelman, C. Kesselman, G. Singh, M.H. Su, GB Berriman, J. Good, AC Laity, and TA Prince. A comparison of two methods for building astronomical image mosaics on a grid. In *Parallel Processing, 2005. ICPP 2005 Workshops. International Conference Workshops on*, pages 85–94, 2005.

[42] Junghwan Kim, Jungkyu Rho, Jeong-Ook Lee, and Myeong-Cheol Ko. Cpoc: Effective static task scheduling for grid computing. *High Performance Computing and Communcations*, pages 477–486, 2005: Springer.

[43] W Korn and P Teller.... Just how accurate are performance counters? *Performance*, Jan 2001.

[44] V Kumar, A Grama, and A Gupta. Introduction to parallel computing: design and analysis of algorithms. *cs.umn.edu*, Jan 1994.

[45] Lizy Kurian and Lieven Eeckhout. Performance evaluation and benchmarking. Jul 2007.

[46] S Laha, JH Patel, and RK Iyer. Accurate low-cost methods for performance evaluation of cachememory systems. *Computers, IEEE Transactions on*, 37(11):1325–1336, 1988.

[47] J Lau, E Perelman, G Hamerly, T Sherwood, and B Calder. Motivation for variable length intervals and hierarchical phase behavior. *IEEE International Symposium on Performance Analysis of Systems and Software*, 2005.

[48] J Lau, J Sampson, E Perelman, G Hamerly, and B Calder. The strong correlation between code signatures and performance. *Performance Analysis of Systems and Software, 2005. ISPASS 2005. International Symposium on Performance Analysis of Systems and Software*, pages 236–247, 2005.

[49] J Lau, S Schoemackers, and B Calder. Structures for phase classification. *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on - ISPASS*, pages 57– 67, 2004.

[50] C Lee, C DeMatteis, and J Stepanek. . . . Cluster performance and the implications for distributed, heterogeneous grid performance. *hcw*, Jan 2000.

[51] Y Li and Z Lan. Frem: A fast restart mechanism for general checkpoint/restart. *Computers, IEEE Transactions on*, PP(99):1 – 1, 2010.

[52] Yawei Li and Zhiling Lan;. A fast restart mechanism for checkpoint/recovery protocols in networked environments. *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 217 – 226, 2008.

[53] J McCalpin and CA Oakland. An industry perspective on performance characterization: Applications vs benchmarks. *Proceedings of the Third Annual IEEE Workshop Workload Characterization, keynote address, Sept*, 2000.

[54] J Mohan. Experience with two parallel programs solving the traveling salesman problem. *osti.gov*, Jan 1983.

[55] R Murphy and P Kogge. On the memory access patterns of supercomputer applications: Benchmark selection and its implications. *Computers, IEEE Transactions on*, 56(7):937 – 945, Jul 2007.

[56] C Notredame, D G Higgins, and J Heringa. T-coffee: A novel method for fast and accurate multiple sequence alignment. *J Mol Biol*, 302(1):205–17, Sep 2000.

[57] L Pastor and J.L Bosquwe Orero. An efficiency and scalability model for heterogeneous clusters is -. *Cluster Computing, 2001. Proceedings. 2001 IEEE International Conference on AB* -, pages 427–434, 2001.

[58] H Patil, R Cohn, M Charney, R Kapoor, A Sun, and A Karunanidhi. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. *MICRO-37 2004. 37th International Symposium on Microarchitecture*, pages 81– 92, 2004.

[59] E Perelman, G Hamerly, and B Calder. Picking statistically valid and early simulation points. *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pages 244– 255, 2003.

[60] A Petitet, R Whaley, J Dongarra, and A Cleary. A portable implementation of the high-performance linpack benchmark for distributed-memory computers. *Innovative Computing Laboratory*, Jan 2004.

[61] A Phansalkar and L John. Performance prediction using program similarity. *Proceedings of the 2006 SPEC Benchmark Workshop*, Jan 2006.

[62] George A. Reis, Jonathan Chang, David I. August, Robin Cohn, and Shubhendu S. Mukherjee. Configurable transient fault detection via dynamic binary translation. In *IN: PROCEEDINGS OF THE 2ND WORKSHOP ON ARCHITECTURAL RELIABILITY*. Citeseer, 2006.

[63] A Ruocco and O Frieder. Clustering and classification of large document bases in a parallel environment. *Journal of the American Society for Information Science*, Jan 1997.

[64] I Sharapov, R Kroeger, G Delamarter, and R Cheveresan. A case study in top-down performance estimation for a large-scale parallel application. *... ACM SIGPLAN symposium on Principles and practice of parallel ...*, Jan 2006.

[65] T Sherwood, E Perelman, and B Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. *International Conference on Parallel Architectures and Compilation Techniques*, Jan 2001.

[66] T Sherwood, S Sair, and B Calder. Phase tracking and prediction. *Proceedings of the 30th Annual International Symposium on Computer Architecture, 2003*, pages 336 – 347, 2003.

[67] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *SIGPLAN: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, 37(10):45–57, 2002.

[68] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. Discovering and exploiting program phases. *IEEE Micro*, Jan 2004.

[69] H Siege, L Wang, J So, and M Maheswaran. Data parallel algorithms. *docs.lib.purdue.edu*.

[70] A Snavely, L Carrington, N Wolter, and J Labarta. A framework for performance modeling and prediction. *Supercomputing*, Jan 2002.

[71] A Snavely, X Gao, C Lee, N Wolter, J Labarta, J Gimenez, and P Jones. Performance modeling of hpc applications. *Parallel Computing (ParCo2003)*, 2003.

[72] S Sodhi and J Subhlok. Skeleton based performance prediction on shared networks. *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, pages 723–730, 2004.

[73] S Sodhi and J Subhlok. Automatic construction and evaluation of performance skeletons. *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, pages 88–98, Apr 2005.

[74] S Sodhi, J Subhlok, and Q Xu. Performance prediction with skeletons. *Cluster Computing*, Jan 2008.

[75] E Strohmaier and H Shan. Architecture independent performance characterization and benchmarking for scientific applications. *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.(MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on*, pages 467–474, 2004.

[76] E Strohmaier and H Shan. Apex-map: A synthetic scalable benchmark probe to explore data access performance on highly parallel systems. *LECTURE NOTES IN COMPUTER SCIENCE*, 3648:114, 2005.

[77] E Strohmaier and H Shan. Apex-map: a parameterized scalable memory access probe for high-performance computing systems: .... *Concurrency and Computation: Practice & Experience*, Jan 2007.

[78] Alexandre Strube, Emilio Luque, and Dolores Rexachs. Improving probe usability. *Cloud Computing and Services Workshop, The 25th IEEE International Conference on Advanced Information Networking and Applications (to appear)*, 2011.

[79] Alexandre Otto Strube, Dolores Rexachs, and Emilio Luque. Software probes: A method for quickly characterizing applications' performance on heterogeneous environments. *Parallel Processing Workshops, International Conference on*, 0:262–269, 2009.

[80] AO Strube, Dolores Rexachs, and Emilio Luque. Software probes: Towards a quick method for machine characterization and application performance prediction. *Parallel and Distributed Computing, 2008. ISPDC '08. International Symposium on Parallel and Distributed Computing*, pages 23–30, 2008.

[81] J Subhlok, P Lieu, and B Lowekamp. Automatic node selection for high performance applications on networks. *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 163–172, 1999.

[82] H Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. DobbÕs Journal*, 30(3):202–210, 2005.

[83] A Toomula and J Subhlok. Replicating memory behavior for performance prediction. *Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, pages 1–8, 2004.

[84] Haluk Topcuouglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[85] Helmut Wanek, Erich Schikuta, and Irfan Ul Haq. Grid workflow optimization regarding dynamically changing resources and conditions. In *GCC '07: Proceedings of the Sixth International Conference on Grid and Cooperative Computing*, pages 757–763, Washington, DC, USA, 2007. IEEE Computer Society.

[86] V Weaver and S McKee. Can hardware performance counters be trusted? *Workload Characterization*, Jan 2008.

[87] V.M. Weaver and S.A. McKee. Using dynamic binary instrumentation to generate multi-platform simpoints: Methodology and accuracy. *Proceedings of the 3rd international conference on High performance embedded architectures and compilers*, 4917:305–309, 2008.

[88] VM Weaver and SA McKee. Using dynamic binary instrumentation to generate multi-platform simpoints: Methodology and accuracy. *Proceedings of the 3rd international conference on High performance embedded architectures and compilers*, 4917:305–319, 2008.

[89] R Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*.

[90] J Weinberg. Quantifying locality in the memory access patterns of hpc applications. 2005.

[91] A. Wong, D. Rexachs, and E. Luque. Parallel application signature. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1 –10, aug. 2009.