Software Probes: towards a quick method for machine characterization and application performance prediction

Alexandre Strube
alexandre@caos.uab.cat

Dolores Rexachs
dolores.rexachs@uab.es

Emilio Luque
emilio.luque@uab.es
University Autonoma of Barcelona
Computer Architecture and Operating System Department (CAOS)
Barcelona, SPAIN

Abstract

Computers perform different applications in different ways. To characterize an application performance into a machine, the usual method is a throughout execution of it. This work is a step into a synthetic probe able to characterize a master-worker application's performance in a fraction of the time required to run it entirely. This is specially important for CPU-intensive scientific applications, who runs for very long, as it makes sense that it runs as efficiently (and fast) as possible. To know how, and for how long a master-worker application is going to run can guide the decision to use this machine or not. Our software probe takes into account only the performance-relevant parts of the application, discovering a program's relevant phases. Running solely these significant phases is a powerful way to quickly characterize the application's performance on a machine. It can help to select the best computing nodes in a grid or in a multi-cluster to run this application, and even quickly predict the total execution time for this application/data set in the machine analyzed. We also present ongoing work on a fully synthetic probe generated from programs' phases.[1]

1 Introduction

Our objective is to build an application's probe, to quickly characterize an application. If the probe is representative enough of the whole program, we can determine the performance of a machine just running it, what will be hundreds - or thousands - of times quicker than the application's execution.

Computers give different performance indexes according to the application it is running. The most precise way to determine the performance of a given application running on a computer is to run this application itself on the machine, as hardly a benchmark can give us a precise image of some machine's performance that can match the behavior of our applications. Instead, most of them reflects a narrow set of applications at best, and it's hard to reflect the behavior of new programs using old benchmarks [20].

In changing heterogeneous environments, such as grids or multi-clusters, where the computational resources are not necessarily known until the execution begins, it is interesting to decide if a machine is good enough to run our application in a short time. To run the application thoroughly can be too slow, and to run a benchmark is not precise enough. This work is focused on parallel master/workers applications, running on multi-clusters.

On those environments, master-worker applications can benefit from a correct node selection. In one side, by selecting the fastest nodes by examining its computation and communication characteristics, and by other side, those who are able to help with the computation being busy most of the time, that is, both for brute performance and overall cluster efficiency. Argollo [4] proposed a methodology to select the best number of nodes in a multi-cluster environment while maintaining the efficiency over a defined threshold in master/worker applications. According to his work, efficiency can be achieved through the correct nodes' selection.

An advantage of a quick probe of an application has to do with administrative issues. A large application running for hours long, interrupting a whole cluster of machines just for testing issues, can be a nuisance to say the least. A spe-

[1]This work has been supported by the MEC-Spain under contracts TIN 2004-03388 and TIN2007-64974

cial case for this administrative issue is when the computers don't belong to the users wanting to run the given application, but instead are lent for a specific amount of time, or one rented cluster, where the computing time is paid. In this case, the rented cluster may or may be not the most adequate solution, and to be able to quickly know how good (or bad) this cluster is, is an advantage that cannot be ignored. It even allows to test competing solutions in a time that could take for testing only a few with the full executions, a case probably unfeasible.

A software probe makes this quick performance determination possible. This work explains the methodology for a first approach to this concept under the master/worker programming paradigm, where the tasks itself are sequential, and do not communicate among each other. The only communications involved are the master sending the tasks to the workers, and the workers returning the results. We aim to quickly characterize the best machines to run a worker according to its performance. This work then points out future directions towards a fully synthetic general probe generated according to an application's performance characteristics.

This paper is organized as follows: Section 2 explains some basic concepts for this work. Section 3 discuss in more detail our implementation of a software probe. Section 4 describes some tools we used for implementation. Section 5 shows up the results of this work, while section 7 tells about some conclusions, open lines and our ongoing work on this field.

2 Program Characterization

Being this work focused on master/worker applications where most of the work happens on nodes running the worker tasks, it is enough to analyze a worker's performance. To build a probe, we needed to identify some basic concepts who characterize an application. According to [14], "the large scale of programs is cyclic in nature". On their paper, they measured under simulation, 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.

Periodic Behavior is defined as a repeatable pattern seen for a given architecture metric. Figure 1 shows some measurements taken in a long-running application. 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, [14] 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. They called this metric "basic block distribution analysis".

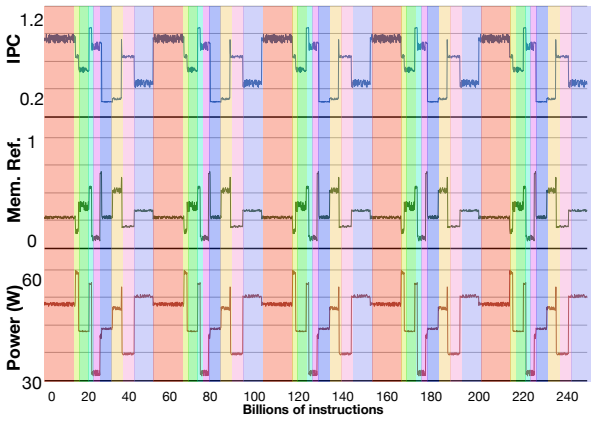

Figure 1. Some hardware metrics. The program phases are quite noticeable, no matter what metric is chosen.

Periods, also mentioned in the aforementioned work as phases, are sets of intervals within a program's execution that present similar behavior, regardless of temporal adjacency [11].

A key point is that the phase behavior seen in any program metric is directly a function of the code being executed. Because of this, a metric that is related to the code can describe phase behavior.

An application's behavior can be defined as a function of the segments of code it runs. As compilers can optimize the program, the source code does not reflect the execution directly. Therefore, a more basic definition is used to describe program behavior. Basic blocks are defined as "sections of code executed from start to finish, with one entry and one exit" [15, 16], and [9] stated that Basic Blocks has no internal control flows.

To build a frequency map of an application, we group basic blocks into arrays called basic block vectors [14]. Basic block vectors are one-dimensional arrays that represent how many times each basic block was executed during a given interval - measured in number of instructions committed. That is, a basic block vector position is incremented each time the program runs this basic block. This makes possible the creation of a fingerprint for each interval of execution, which tells where in the code the application is spending its time during each interval.

A basic block vector that stores the values of the whole program's execution is called *Target* basic block vector. Comparisons among two basic block vectors gives their

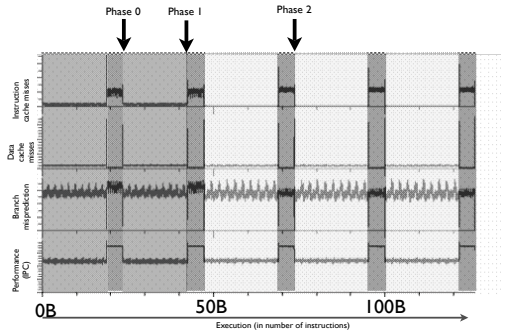

Figure 2. Phases in the gzip execution.

similarity. Therefore, all basic block vectors of an execution can be passed through a clustering algorithm to discover this execution's phases. Once we have this phases, we monitor the application to save program's phases.

2.1 Context and State Save

To be able to run only the application relevant phases, we firstly run the application thoroughly, saving the program context at each phase's beginning (such as the arrows in figure 2). The first arrow marks one phase - in fact, this phase was run previously, as can be seen from the graph. The same happens with the second one, seen as a thinner stripe occurring timely during the execution. The third phase, seen as the lightest grey, happens three time, and *Simpoint* chose the most centric one. Phases repeat, but we run only one instance of it, as what matters to us is the performance of the phases and not the program output. A similar method was described by [13], who uses checkpoints to save a program's state in a machine to then feed them into a functional simulator, to shorten simulations in future-generation hardware.

In our case, we developed a tool which monitors the application thoroughly to save the basic block vectors, feed it into *Simpoint*, collect the results, re-run the application and save checkpoints previously to each phase's beginning (due to warmup, explained below). It is necessary to run the application twice, one containing instrumentation for capturing the basic blocks, and the second one, when the phases are already known, to save checkpoints, as shown in figure 3 The same tool is used on the remote machines as the probe: it is in charge of loading the checkpoints, run each phase, measure this phase's time, stop the execution and step to the next checkpoint. More information in chapter 3.

2.2 Warmup

Resuming a program from disk has a drawback: some machine's components contents will not be in the same state that they would be if running the application from the beginning. Basically, those are the cache memory, TLBs and branch predictor. We use the strategy of *Calculated Warmup* [9], where the program state is saved somewhat earlier than the exact point where the phase begins, so the aforementioned components can warm up before the measurement begins. In figure 2, the context would be saved previously the arrows who points phases, i.e. a given number of instructions previous to the phases' beginnings themselves.

2.3 Data Dependency Issues

The phases are obviously data-dependent, that is, a probe must be used directly only when the application is not data-dependent, as the checkpoints carry actual data on them.

For data-dependent applications, e.g. the traveling salesman problem, [8] proposed a solution, where it discovers patterns on the inputs which leads to similar execution times. That is, by discovering what pattern on the input leads to those similar outputs (specific to each application), it is possible to cluster sets of input data with similar execution times. A number of sets of probes, to ranges of parameters of an application, may be used to overcome the data dependency issue.

3 Machine Characterization Method

3.1 General Scheme for the Probe Creation

The software probe creation is done previous to the measurements of the machine we want to characterize. The probe creation being such a complex operation, can take quite long. But the advantage is that it is done only once, and then the probe can be run in as many machines as we want to.

The probe generation process uses external tools as well as a program built by us, known as *ProbeGen*. The *ProbeGen* application is responsible for instrumenting the application, saving the basic block vectors, feed them into *Simpoint*, getting from *Simpoint* the phases, and then finally rerunning the application, and saving the checkpoints where required. This application, along with those saved checkpoints, are to sent to the machine to be probed, where the application loads the checkpoints, measures the phases, interrupts the execution and proceeds to load the next one.

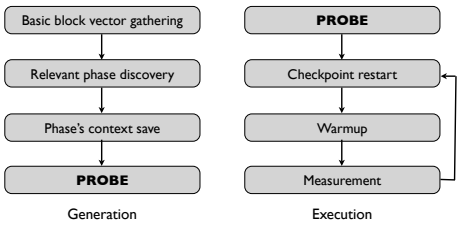

Figure 3. Probe creation general scheme. While the left side of the image is done in a machine under our control, the right side is done on the machine to be characterized.

The process consists in the following steps: *basic block vector gathering*, *phase discovery* and *context saving*. Each of them will be explained in details.

- *Basic block gathering*. In this step, *ProbeGen* runs the application thoroughly, instrumented. The instrumentation collects the basic block vectors.

- *Phase discovery. ProbeGen* proceeds then to he second step of phase discovery. Once the basic block vectors were gathered, to feed them into a clustering algorithm to find the relevant program's phases. We use the *Simpoint*[15] tool for this function.

- *Context saving.* With the phases in hand, *ProbeGen* runs the application thoroughly again to save those phases in form of checkpoints. Its worth noticing that the instrumentation is saved with the program. The reason is that when resuming from the saved contexts, we have to (1) run the program only up to the phases' end, as if the program were not instrumented, it would run normally until its completion, and (2) the instrumentation contains the code for measuring the phase's execution time.

- *Probe execution.* The same *ProbeGen* application also contains the functionality to run the probe itself and measure the phases' times. As soon as the *ProbeGen* is in possession of the checkpoints, it can be sent, along with them and with the application to be measured, to any machine one wants to measure. A single parameter makes *ProbeGen* to switch to this measurement mode, where it loads the checkpoints, wait for the machine to warmup, and then measure this checkpoint's phase time. Once the phase was run and measured, it stops the execution, and starts again with the next phase. As each phase contained on each checkpoint is very small, this happens in a magnitude of seconds.

4 Tools for Probe Creation and Use

Several concepts used on this work come from the computer architecture simulation field, as is unfeasible for them to create every piece of equipment to be tested. These simulations are usually slow, in the magnitude of months. The engineers of this field developed techniques to reduce their benchmarking time but maintaining a level of trust on their results. Their problems in determining performance of simulated machines - and their solutions - are somewhat similar to ours, determining the performance of an application running into an unknown machine, so we can apply some of their knowledge and tools into our field of study. Furthermore, to be able to interfere into a program's working without touching its source code nor having any previous knowledge of it, we made use of a set of tools, which are going to be described now.

- *Simpoint*. It was described in [15] a technique to find the relevant phases of a program, based into the basic block vectors gathered in a execution. Based on it, they developed *Simpoint*. Its an offline tool, based on the k-means clustering algorithm. It also returns the weight of each phase. Besides their own experiments, [6] found that the basic block vectors analysis and phase detection is accurate.

- *Pin*. It is a dynamic instrumentation toolkit developed by Intel [12]. With it, we are able to instrument an application to gather its basic blocks. Pin does not need to recompile code or change any application file.

- *Atom*. It is an acronym from *Analysis Tools for Object Modification* - is a static instrumentation toolkit from intel. It is necessary to statically instrument the application's phases when the context is saved to disk. This is because a dynamic instrumentation toolkit cannot instrument an application restored from it's disk saved context.

- *Blcr*. To save the program context, we use a checkpointing library, i.e. we use checkpoints only to be able to run the program from each phase's begin for the phase's duration. Checkpoints can be defined as "the process of saving the entire state of a job to disk, than later restore it" [7]. He also proposed and implemented a checkpointing library for Linux, the Berkeley Linux Checkpoint Restart.

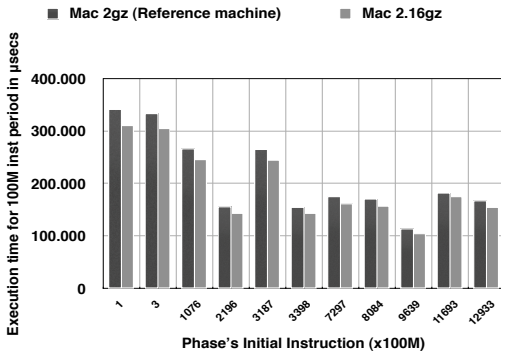

Figure 4. Phases' execution time of sweep3d on the macintoshes.

4.1 ProbeGen

As previously stated, *ProbeGen* is the tool we built to use Pin to instrument the application, feed the basic block vectors into *Simpoint*, get the phases from it, and run the application in order to save the checkpoints. That is, it generates our probe.

ProbeGen also serves as the probe itself, running on the machine to be probed, running the phases from the checkpoints and measuring them.

Our tool was built in C, using the Pin and Atom instumentation libraries, and runs under Linux. The Linux is not a requirement on the probe itself nor the concept, it was just that the Pin, Atom and Blcr libraries are only available on this operating system. Besides, Linux clusters are today the mainstream, it is more and more hard to find clusters running other OSses.

5 Experiments and Results Analysis

To prove the concept of the probe, we did some experiments. They were conducted in the following way: Two machines were probed, one we call the Reference one, and one that would be the machine to be characterized. The phases' time 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, which validates the phase method.

A more complex experiment was to create a probe of the BlueGene's *Sweep3D* [10] kernel. It is a time-independent, Cartesian-grid, "discrete ordinates" deterministic particle transport code taken from the DOE Accelerated Strategic Computing Initiative (ASCI) workload. *Sweep3D* represents "the core of a widely utilized method of solving the Boltzmann transport equation. Estimates are that deterministic particle transport accounts for 50-80% of the execution time of many realistic simulations on current DOE systems". Its relevance in real systems and the configurability were crucial to make the decision of using it to prove our probe. This experiment was conducted on the single processor worker of the *Sweep3D*'s master/worker version.

We ran it into two different architectures: a pair of Apple Macintoshes. The reference one used the Merom Core 2 Duo CPU, running at 2gz and with the i950 chipset, and the probed one with an 2.2gz Penryn Core 2 Duo CPU using the Santa Rosa chipset, and in a pc cluster, where the reference one was a 2.6gz Prescott Pentium IV CPU with 256kb of cache memory using the i865 chipset and the probed one with a 2.8gz Northwod Pentium 4 CPU, with 512k of cache and the i810 chipset. On the cluster, we are going to report the results of only two machines, as identical machines led to similar results.

On the mac reference machine, the 5-times average of the the full execution with the problem size of 50-cubed spatial grid points took 502,512 seconds (8 minutes). The probe took about 3 seconds. On the pc reference one, execution's average time with the 150-cubed problem size took 32193 seconds (almost 9 hours) and the probe, 310 seconds. In both cases, the probe took approximately 1/100th of the full execution. The bigger the problem sizes, the bigger the gain, as the number of phases tends to remain constant. Most of this time is related to load from the disk into memory the context, that is, the phases. The computing time was about 5 seconds in the worst case.

Figures 4 and 5 show each phase's execution time in the reference and probed machines. They show the difference in execution time of the reference and probed machines. By applying the weights given by *Simpoint* and comparing them, it is possible to extrapolate the execution time on the probed machine. All phases have the same number of instructions - in this case, 100 million instructions. As all the phases have the same number of instructions, the difference between the phases' time is given by the way each phase exercises the code. It may be caused by different phases' instruction sets, cache/tlb misses or memory access patterns.

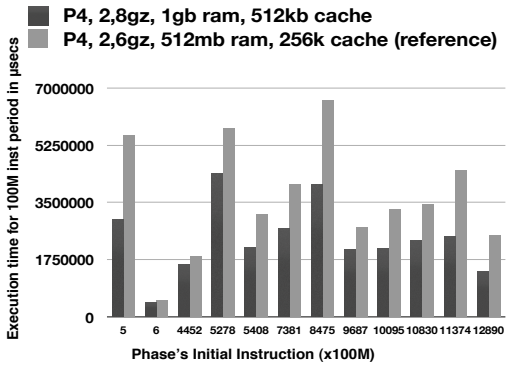

Figure 5. Phases' execution time in sweep3d in PCs.

5.1 Execution time prediction

It is possible to extrapolate the execution time for the full execution from the probe. Being $t(i)$ the time (in this case, in seconds) of each phase and $w(i)$ the phase weight, its sum (formula 1) gives us the weighed factor P which makes possible to create a proportion between execution times on the reference machine and the probed one. If we use this formula for the macintoshes' data shown at figure 4, the weighed proportion among the 2,0gz machine and the 2,16gz one is 91,9928%, and this means that the program will run at 91,9928% of the time it took to run on the reference one.

$$P_{machine} = \sum_{i=1}^{T} [t(i) \cdot w(i)] \quad (1)$$

It is possible to extrapolate the execution time without a reference machine, just by using formula 1. However, the comparison with a reference machine improves the prediction ratio by more than 10% while reduces the probing time, as phases may be smaller.

Being the execution time on the reference machine $T(ref)$ and the weighed proportion among the machines P, we can calculate the execution time on probed machine with the formula 2, being T_{pred} the predicted time on the probed machine:

$$T_{pred} = T_{ref} \cdot \frac{P}{100} \quad (2)$$

Using the formula to our data, we have the following results on the macs:

$$T_{pred} = \frac{502,519 \ \ seconds \cdot 91,9928}{100} = 462,281$$

Which means that the faster macintosh would run the application in 462,281 seconds. When we ran the application thoroughly on it, the observed time was of 465,166 seconds, a 99,37% precise prediction. On the PCs, the results were the following:

$$T_{pred} = \frac{32193 \ \ seconds \cdot 89,6673}{100} = 28866,594$$

The full execution took 24415,365 seconds on the pc, a 84,58% precise prediction. Although the probed machine had a higher clock, its bus was slower, and as memory access dominates this algorithm's execution time, it took longer than the slower but more efficient processor. This lower accuracy is related to lack of memory who caused the system to do swap, and noise on the pcs caused by other users, but we still can tell how good and how efficient a machine is for an execution.

6 Next steps

The next step is to define what we must characterize. In [13], it is stated that memory is the most relevant aspect of a machine performance, and uses this characteristic for their simulation purposes, and [17] says that memory latency usually dominates the performance behavior of scientific applications. In [5], it is proven that in some simulations, the increment in memory performance is more determinant to the overall performance of applications than any other factors analyzed. Therefore, the initial step of our studies is the analysis of memory access patterns of application's phases, and the subsequent creation of a synthetic tunable probe that could mimic this memory pattern. Both [18] and [20] are different approaches of the development of a synthetic probe that can represent memory access. We aim to create a probe who is able to represent not only the memory access precisely, but as well to mimic another factors important to represent the performance of an application.

The is ongoing work on synthetic probes in [18], [19] and [20]. Their approach is statistical, i.e. they map memory accesses in terms of miss probability when using a random access pattern.

The works of [1, 2, 3] try to map statistically the cache behavior, thus recognizing its pattern. In one of their experiments, they considered a multi-level cache hierarchy, where memory was seen as another cache level. They based their work on probabilistic miss equations.

Our approach is somewhat different. The proposal is to be able to reproduce the memory access, not probabilistically, but the access itself. The thesis is that the memory access can be represented as waves, where the frequency is the amount of accesses during a given period. Treating the memory access pattern as a analog wave permits us to decompose this wave into its components, who can generate

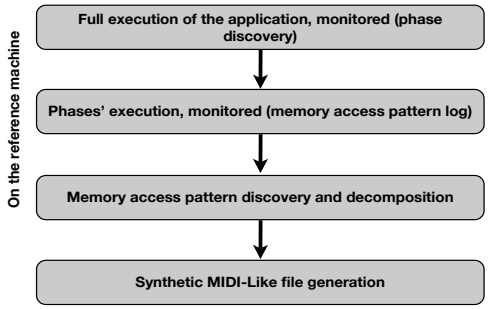

Figure 6. Synthetic Probe Generation Scheme.

equations. This equations will then be used to recompose the waves - that is, the memory access pattern of a program's phases - allowing us to "play" these patterns on a probed machine.

The idea is similar to converting a song recording file into a synthetic format, such as MIDI, and playing it back. This MIDI-like file is way smaller than an address trace could ever be, as it contains the equations responsible to generate the patterns, not the pattern themselves. This resolves the issue of the probe being the cause of noise on the pattern because of its own inner working, a problem faced by the other proposals. Figure 6 outlines the probe generation scheme.

Such a probe must achieve an interesting prediction, as each program's relevant phases will be analyzed. Current approaches use whether the entire program memory trace or just the program kernels, that is, some benchmark's main loops, and the rest of the computation is discarded. The fact that the whole relevant behavior is represented is a plus when compared to the other alternatives, turning it into a useful way to represent program behavior.

The representation of the memory access into a synthetic wave instead of access probability must result into a more precise mimic of the application's behavior.

7 Conclusion and Future Work

With this work, we proposed a first step into a scheme to characterize a parallel Master/Worker application/data set pair performance into a machine in a fraction of time required to run the application thoroughly on this machine by using a software probe. The context saving tool gave us the capability of resuming the application's execution, and the instrumentation gave us the ability to choose exactly when do this, as well as to measure the phases and stop the execution when we needed to. The formulas predicted the execution time on the probed machines with good results and quickly. However, the context-saving approach is not our final goal, as there are some issues with it that make it not suitable for practical everyday use, such as phases' context size, the need of tools installed on the machines to probe and the deterministic nature of the phases. A very promising technique is that of [8], who uses data mining techniques to recognize similar patterns to be able to predict nondeterministic behavior.

To overcome these difficulties, a radically different approach is needed. Our group is studying the creation of a generic synthetic parametric probe, able to represent the behavior of a program when fed with the correct input parameters obtained from the specific program to be characterized. The phases' approach on the subject fits well this future direction, as it narrows the amount of data to analyze in several orders of magnitude.

References

[1] 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 Computers (CPC2006)*, pages 2407-2424, Jan 2006.

[2] D. Andrade, B. Fraguela, and R. Doallo. Analytical modeling of codes with arbitrary data-dependent conditional structures. *Journal of Systems Architecture*, 52(7), Jan 2006.

[3] D. Andrade, B. Fraguela, and R. Doallo. Precise automatable analytical modeling of the cache behavior of codes with indirections. *ACM Transactions on Architecture and Code Optimization*, 4(3) Jan 2007.

[4] E. Argollo. Performance prediction and tuning in a multi-cluster environment. Ph.D. dissertation, Computer Architecture and Operating Systems Department, Universitat Autonoma de Barcelona, Barcelona, Spain, 2006.

[5] L. Carrington, A. Snavely, and N. Wolter. A performance prediction framework for scientific applications. *Future Generation Computer Systems*, 22(3):336–346, 2006.

[6] A. Dhodapkar and J. Smith. Comparing program phase detection techniques. *In Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 217– 227, 2003.

[7] J. Duell. The design and implementation of berkeley labs linux checkpoint/restart. *Lawrence Berkeley National Laboratory*, Berkeley, CA, USA, Tech. Rep. LBNL-54941, 2003.

[8] P. Fritzsche. Podemos Predecir en Algoritmos Paralelos no Deterministas? Ph.D. dissertation, Computer Architecture and Operating Systems Department, Universitat Autonoma de Barcelona, Barcelona, Spain, 2007.

[9] 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.

[10] A. Hoisie, O. Lubeck, and H. Wasserman. Scalability analysis of multidimensional wavefront algorithms on large-scale smp clusters. *In Proceedings of the 9th Symposium on the Frontiers of Massively Parallel Computation*, pages 4-16 Jan 1999.

[11] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. *in Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 236-247, 2005.

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

[13] I. Sharapov, R. Kroeger, G. Delamarter, and R. Cheveresan. A case study in top-down performance estimation for a large-scale parallel application. *in Proceedings of the 11th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 81-89, Jan 2006.

[14] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. *in Proceedings of the International Conference on Parallel Architectures and Compilation Techinques*, pages 3-14, Jan 2001.

[15] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *SIGPLAN Not.*, 37(10):45–57, 2002.

[16] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE micro*, pages 84-93, Jan 2004.

[17] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. *in Proceedings of the ACM/IEEE 2002 Conference on Supercomputing*, page 21, 2002.

[18] E. Strohmaier and H. Shan. Apex-map: a parameterized scalable memory access probe for high-performance computing systems: Research articles. *Concurrency and Computation: Practice and Experience*. John Wiley and Sons Ltd. 19(17):2185-2205, 2007.

[19] E. Strohmaier and H. Shan. Apex-map: A synthetic scalable benchmark probe to explore data access performance on highly parallel systems. *in Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 113-123, 2005.

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

[21] A. Strube. Performance determination using software probes. Master Thesis, Computer Architecture and Operating Systems Department, Universitat Autonoma de Barcelona, Barcelona, Spain, 2007.