Software Probes: A method for quickly characterizing applications' performance on heterogeneous environments

Alexandre Strube*, Dolores Rexachs†, Emilio Luque‡
Universitat Autònoma de Barcelona
Computer Architecture and Operating System Department (CAOS)
Barcelona, SPAIN
*alexandre@caos.uab.es
†dolores.rexachs@uab.es
‡emilio.luque@uab.es

***Abstract*—This work describes ongoing work for measuring the performance of an application running on a machine, where this measurement takes a fraction of the time required to run the application itself thoroughly. We call it *Performance Software Probe*. The objective is to have knowledge of this machine/application performance previous to the execution, and without the need to even install this application on the machine to characterize. Our goal is to enhance efficiency of master/worker applications on highly heterogeneous multiclusters, where the available machines - and their respective performance indexes - are not known until the time we have them available for execution.** [1]

I. Introduction

Performance evaluation and prediction is something between a science and a form of art. Being both, its possibilities depend mostly on creativity, and it is useful for a variety of fields.

However, what must we evaluate? Throughout the years, we have seen several works on computer performance evaluation. Units of measurements, such as the old (and not always well regarded) Whetstone [1], were good for comparing machines among themselves in their time. This is done even today, with the Top500's High-Performance Linpack Benchmark [2].

Although useful for having *some* idea about a computer's performance, it is surely not enough to know how a given application will perform on a given machine. In fact, it's not always possible to characterize a machine's absolute performance using benchmarks [3], as computers give different performance indexes according to the application it is running, and benchmarks usually reflects a narrow set of applications at best.

As consequence, it is hard to reflect the behavior of new programs using old benchmarks [4]. This means the best way to characterize this engagement among a computer and an application its to run the application itself, thoroughly.

However, this brings problems of its own. Running a full program may take too long for evaluation purposes. Setting up the system and the program to run together also may be a problem, especially when the machines are yet being evaluated, being considered. For instance, when a big system is going to be bought or rent specifically to run a set of applications, it is quite obvious that this system must perform as good as possible when running these specific applications.

Also in changing environments, such as grids and multiclusters outside of direct administrator's control (for instance, collaboration projects among universities), where the computational resources are not necessarily known until execution begins, it is interesting to decide if a machine is good enough to run our application in a short time.

Our goal is to enhance efficiency of master/worker applications which runs on highly heterogeneous multiclusters. On them, available machines come and go, and we don't know their performance until they are ready to be used. Considering that we choose machines on a multicluster according to their performance [5], if we don't know their performance prior to the execution, the selection may be a challenge.

Another problem of taking a whole application to a system just to evaluate it has to do with time, in different ways:

- *Installation*: An application may have enough dependencies and system requirements that the simple intent of install may not be viable for some projects' time constraints.
- *Measurement*: Some applications may take weeks to finish an execution of a single data set. This evaluation time again may be excessive.

To overcome these issues, we defined the concept of a Performance Software Probe in [6]. This work is its ongoing research.

Quoting the Oxford English Dictionary, the word probe means:

- a blunt-ended surgical instrument used for exploring a wound or part of the body.
- a small device, esp. an electrode, used for measuring, testing, or obtaining information

[1]This research has been supported by the MEC-MICINN Spain under contract TIN2007-64974

Being a probe a small device created to explore and gather information, our analogy to a software Probe is a piece of code able to explore a machine. In our case, reflecting the performance characteristics of a specific application.

This work is organized as follows: Section II talks about the performance model and where this work is located on it. Section III describes the idea of a Probe, how it is able to characterize an application for evaluating a machine's performance. Section IV demonstrates Probe usage in real environments and some results. Section V shows where the current Probe approximation is worthy. Section VI shows some conclusion and what direction our research is taking.

II. Performance Model

In the research of Argollo [5], an analytical model of performance prediction for master/worker application running in heterogeneous multiclusters was developed. This scheme consists of a *master* cluster, with its own nodes, usually the one closest to the user, and sub-clusters. Communication between a sub-cluster and the master cluster is done by means of entities called *communication managers*. This way, the master cluster sees the sub-cluster as a very fast node with a very slow network connection.

In this model, you characterize the whole environment by means of computation and communication parameters. For instance, the performance model of a sub-cluster when running a computation-bounded master/worker application during its steady time ($Perf_{Ste}$) is given by the equation (1). On it, $CPerf_{Avail}$ is given by equation (2), $Oper$ 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 know beforehand, and that sub-cluster available performance, described in equation (2. On it, $Perf_{Avail}$ is the relative computation performance of a machine while running a given application. While every other parameter of these equations is discoverable or already know, this one is not. In [5], to discover it, a whole long-running task was sent to *every* available node on the whole multicluster. That took a *long* time.

This work is about a way to accelerate the discovery of $Perf_{Avail}$.

III. The Software Probe

Given our bigger constraint in evaluating performance characteristics of a machine is its execution time, our objective with the software Probe is to reduce this evaluation time as much as we can.

A work related to ours is that of Sodhi and Subhlok [7], where their performance skeletons intend to mimic application behavior in a shorter execution time for evaluating shared resources. While they focus on shared resources and network usage, we focus this work on computation for master/worker applications by characterizing the worker, as our research group already have ongoing work on characterizing communication patterns, such as the study conducted by Wong [8]. The work of [7] claims that no monitoring is necessary, although later it's stated that the execution trace is taken. For our method, focused on the computation part of the applications, instrumentation to reach instruction-level precision is adamant. The work of [9] 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 Probe methodology consists of two steps: Acquiring knowledge of an application to generate a Probe with performance indexes related to this application it's based on, and then characterize a machine with this Probe. This general schema is in figure 1.

A. Application characterization

As previously stated, we want to evaluate the performance of a machine while running an application, although without the hassle involved in running this application to its completion. The dominant factor that drives us to create a Probe instead of running an application thoroughly is its execution time. That means our first goal is to reduce it without loosing the performance relevant aspects of a thorough execution.

In Master/Worker applications, the master is ours - that means, it is usually a machine under direct control, which will always be used, while *some* of the workers may or may not be used. The decision of which of those workers is to be used is what drives our research, as the master will always be present.

Sherwood and Calder [10] stated that "the large scale of the programs is cyclic in nature". Further, Sherwood, Sair and Calder [11] proposed a profiling technique able to understand an execution as a series of different phases that may repeat. They use the basic block distribution analysis technique described by Sherwood, Perelman and Calder in [12]. Once an application is profiled, for instance from an execution trace of an architecture simulator such as simpleScalar [13], the basic block profile is then fed into their *SimPoint* tool. A basic block vector contains the index of basic blocks executed during that instruction interval and their occurrence. Each basic block is uniquely labelled during a program's execution. Every basic block in that interval also has a counter of how many times it ran.

Our methodology to evaluate an application's performance on a machine consists of the following steps:

1) Data collection: To collect basic blocks of real application running in a real system (in opposition to the simulated architectures used by the *SimPoint* team), we instrument the application to be characterized, as we do not have cpu traces.

$$Perf_{Ste} = min\left(CPerf_{Avail}, Oper * \frac{TPut_{EC}}{S_{Comm}}, Oper * \frac{N * TPut_{InetIN}}{S_{TaskInter}}, Oper * \frac{N * TPut_{InetOUT}}{S_{ResultInter}}\right) \quad (1)$$

$$CPerf_{avail} = \sum_{workers} Perf_{Avail} \quad (2)$$

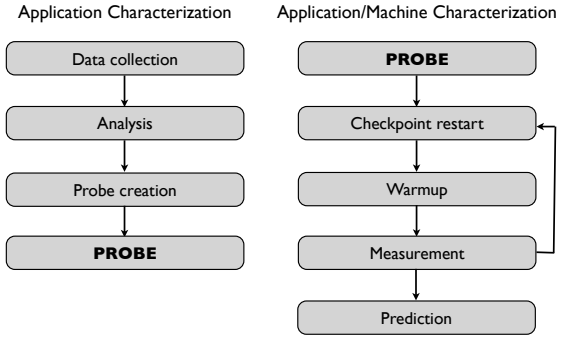


Fig. 1. Probe creation general scheme. While the left side of the image is done in a machine under our control, the right side is done on the machine to be characterized.

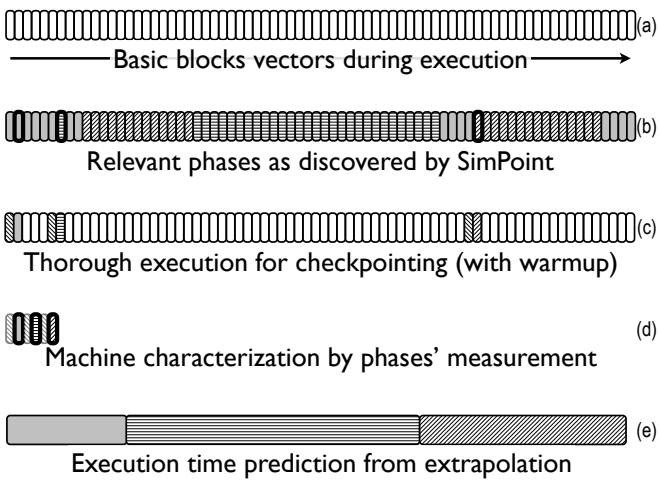


Fig. 2. Stages for Probe generation and machine/application characterization

This is done with the Pin instrumentation toolkit [14]. This toolkit is able to instrument an application at basic block level, which makes straightforward to collect them. We define a fixed instruction interval where the basic blocks are collected, and each of these intervals is a basic block *vector*. We then run the application monitored thoroughly, to collect basic block vectors, as can be seen in Figure 2, item (a).

As we instrument applications at basic block level, we save the occurrences of each of them in a basic block vector, for a fixed instruction interval. In this work's experiments in section IV, we used an interval of 1.000.000.000 instructions. For each of those intervals a basic block vector is generated.

About data dependency and problem size: A side effect of the method used for characterization has to do with problem size and data dependency. Our Probe is valid for a given problem size. However, in some domains, this not holds true. NAS' Benchmarks, for instance, present the same behavior over time, where the dominant behavior repeats itself, and all the phases hold more or less the same weight, only repeating more. Research has shown that more than the data itself, performance is dependent on some data patterns which are characteristic to the type of program being executed and its underlying algorithm, as studied by Fritzsche [15].

About the set of probes required for data-dependent applications One obvious advantage of our Probe for characterizing multiple machines against data-dependent applications is that after defining a space of data patterns relevant for characterization using the aforementioned work of Fritzsche, several Probes can be created, being each of them representative of that specific data pattern. As our Probes run in orders of magnitude less time than applications themselves, we are able to explore much wider areas of data patterns in less time than a single execution of an application may take for characterizing one single data set.

2) Analysis: The collected basic block vectors are fed into *SimPoint*, which can discover program phases based on these vectors' similarity. It uses clustering techniques to find repetitive patterns [16]. These repetitive patterns, called *phases*, are classified and the tool outputs a single occurrence of it, counted in number of instructions. *SimPoint* also outputs each phase's weight, which is the phase's percentile on the whole execution, i.e. a phase with a weight of 0.8 dominated the application's execution during 80% of its run. See Figure 2, item (b), where each color represents a single phase. The item where the color is stronger means this item is selected for representing the behavior, where the faded ones are similar to it, what means they have a similar performance behavior.

There are several applications, especially in the scientific field, where a small set of phases dominate the execution. The worker of a master/worker application typically consists of a single dominating phase, and some smaller phases, usually initialization and finalization, packing of data for send back to the master, etc. They are not necessarily relevant. In our research, we noticed that 1% of the execution is comprised of a lot of irrelevant phases. Ergo, to reduce characterization time, we discard the 1% less relevant phases. In our experiments, this was enough to maintain a good prediction level while

greatly reducing Probe size, as much less checkpoints are necessary. We can reduce Probe size even further, as after this 1% threshold we still may have a number of phases of little relevance in the overall execution. Results of these further reductions are discussed in detail in section IV-C.

3) Probe creation: With the phases now known, we must create the Probe itself. What the Probe will carry to the machine to be characterized will be: the set of phases discovered by *SimPoint* (saved in the form of checkpoints), code to run these phases, instrumentation code to stop execution after the phase size was reached, measure its execution time and send these results back.

This implies that we must run the application thoroughly a second time, with a specific monitoring tool. It counts the application's executed instructions until it reaches a point right before a phase's number, and then the application is checkpointed to disk, as in Figure 2, item (c). We checkpoint a number of instructions before the exact phase's time to ensure the machines to be characterized with this checkpoint are "warm", i.e. its components (TLB, caches, etc) are in a state consistent to that of a throughout execution [17]. This operation is inexpensive, and greatly improves precision. Its worth noting that this offline process to create a probe must be done only once.

The saved checkpoints are, as of today, an important part of our Probe. Nevertheless, given that our field of research is heterogeneous multiclusters, where network connections are a strong issue, big checkpoints may be a constraint on those environments. We have set of clusters scattered around the world that form multiclusters, and connected through internet, and connection droppings and bandwidth limitations are the norm, not the exception [18]. There are works specialized in reducing checkpoint's size, by saving only state necessary for the future, such as that of Rodriguez [19] but for it to succeed, applications must be recompiled. Our technique doesn't need any source code at all. As we know the relevance of each phase, we can discard the less important ones, knowing the degree of precision lost by this action.

Regarding Characterization Time: All these executions take time. Instrumented executions may take orders of magnitude longer than "clean" ones. Given the two executions, the first to find phases and the second for probe creation are both instrumented, the time for one characterization is way longer than the execution itself. However this is done offline, and only one time for a problem size, and once the Probe is built, there's no need to repeat this procedure again, and the Probe may be used as many times as necessary.

B. Application/Machine characterization

A machine's performance while running an application can be predicted without running it in its entirety, but alternatively by running only its relevant portions and extrapolating from them. As we know which of them they are and have them saved in form of checkpoints, running them in succession is enough to predict how it would run entirely, as essentially what we do is remove application's execution repeated behavior.

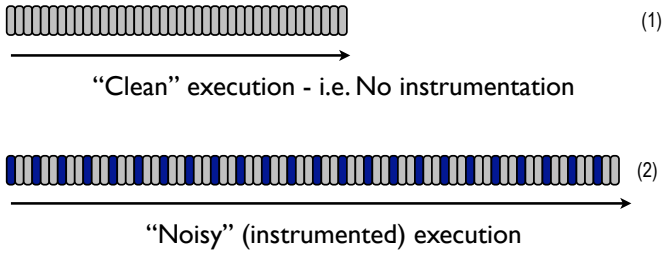


Fig. 3. Instrumentation noise in execution time.

In our case, the gross of the time is spent estimating the performance of the workers, to decide which one are going to be selected for this execution, so we focus on quickly characterizing the performance of the workers while running a given application.

1) Checkpoint restart: A shell script is responsible for running every phase in succession. When finished, it may send measurement results back. The script itself doesn't need to do much else, as the "intelligence" is embedded into the instrumentation code. For checkpointing, as of today, we are using the Berkeley Checkpoint/Restart (blcr) [20] library embedded in our instrumentation. This library checkpoints to system level, which may be in itself a disadvantage; However, this library was chosen in the first place because of its popularity among fault-tolerance solutions in parallel applications such as MPICH-V [21], fault-tolerant OpenMPI [22] and the Lam/Mpi Checkpoint/Restart framework [23] for instance which make this specific library easily available in academic clusters.

For the Probe, as it is today, to work correctly on the machines to be characterized, we need that at least the checkpointing library to be installed. Other dependencies may be carried along with the probe. Another solution is a virtual environment where all dependencies are provided.

Those issues are already known, and this paper is meant to demonstrate the quality of prediction in real hardware based on the concept of phases. They are all being handled in our ongoing research, mentioned in section VI.

2) Measurement: When checkpoints are restarted, the application is not exactly in the point that measurement must start, but instead a number of instructions before it, to ensure warmup - see Figure 2, item (c) and (d), where there's a period previous to the phases, the warmup. When the execution reaches the point where this specific phase starts, our instrumentation code sets a timer. When the phase size is reached - that is, when we run the program after the same number of instructions we used to classify our basic blocks, this time is counted, and instrumentation code writes it to an output file and interrupts the program execution. The machine's command returns to the shell script, which proceeds to run the next phase or send the results back to the user measuring the machine, when passed the last phase.

3) Execution time prediction: The measurement is then compared to the machine used to characterize the application

in the first time. Every phase given by *SimPoint* also contains its weight on the overall program, e.g. a phase with a weight of 50% means that it ran for half the time of the whole program's execution. Although only the phases' times and their weights would be enough to characterize a machine, this would not take into account the noise generated by instrumentation, which extends execution time, as illustrated in Figure 3.

To ensure a prediction where instrumentation noise is removed, we compare a "clean" execution, that is, without instrumentation, to that instrumented. The difference among them is instrumentation noise, such as in equation 3. Given instrumentation doesn't change among machines, its proportion in execution may be calculated. This equation gives is the proportion of overhead in execution. It may be counted in terms of both time (and by that $T_{Instrumented}$ and $T_{Uninstrumented}$ refer to execution time) or instructions executed. In our experiments they perform similarly.

$$Noise = \frac{T_{Instrumented}}{T_{Uninstrumented}} \tag{3}$$

The estimation of execution time is given by equation 4, which is a summation of each phase measured parameters, where $T_{FullApp}$ is the execution time during our offline characterization of the whole worker, W_{Phase} is the phase's weight, T_{Off} is the elapsed time the probe took to run during the offline characterization, and T_{New} is the time that phase took to run on the machine to be characterized.

$$\sum_{Phases} \left(\frac{T_{FullApp} * W_{Phase}}{T_{off}} * T_{New} \right) \tag{4}$$

Let's exemplify the prediction. If an hypothetical application that during its characterization ran for one hour, and we found that it had three phases: one which weight was 60%, another with 30% weight and another with 10% weight. During the offline characterization, we measured that the first phase took 2 seconds to run (representing 60% of one hour, that is, 36 minutes), the second on took 2 seconds (the one with 30% weight, representing 18 minutes) , and that the less important took 1 second (representing 6 minutes of execution).

On the machine to be characterized, the probe for the 60% phase took 1 second to run, the 30% took 1.5s and the 10% took 0.5s.

If we use our formula,

$$\left(\frac{60m * 60\%}{2} * 1 \right) + \left(\frac{60m * 30\%}{2} * 1.5 \right) + \left(\frac{60m * 10\%}{1} * 0.5 \right) \tag{5}$$

That translates as

$$(18) + (13.5) + (3) = 34.5 \tag{6}$$

Which means that this worker would take 34.5 minutes to run in this machine. This hypothetical probe would take only 3 seconds to predict an execution time of more than 2000 seconds. Our experiments shows that we are able to reduce in this order of magnitude the characterization time while keeping a good prediction ratio.

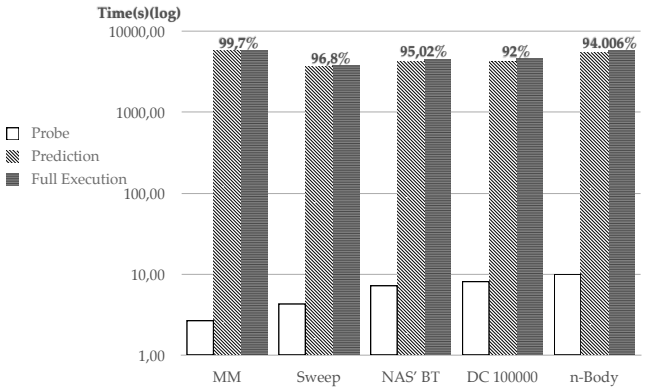


Fig. 4. Execution time, prediction and probing time in log scale.

IV. 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 [24], a Matrix Multiplication, a *n*-Body problem, the NAS Benchmarks [25] BT and DC. We show these specific experiments because they provided us 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 where the Probes were built, 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 they share the brand name, 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 pipeline 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.

A. Precision

The average of execution times for real runs and predicted, as well as the time required for probing, are shown in Figure

4, in logarithmic scale. As stated in [3], benchmarks hardly represent the overall behavior of an application, so there would be little point in comparing them with this method.

We chose problem sizes who took similar execution times for the sake of presentation. For the tested applications, the precision always stood above 92%, while the probing time was always 0.2% of the original application's execution time for this set of experiments. Longer applications in general yields even better reductions, consequence of more phases' repetitions. This probing time doesn't take into account checkpoint transmission and checkpoint loading, instead focus on its phase execution time.

B. 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 small 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. In this specific case, our Probe would not worth the effort, but this is just an example of a very short execution. However, in a local multicluster, such as we possess in our university, scattered throughout departments with heterogeneous sets of machines but a one gigabit connection for all nodes, this is not an issue.

However, when looking at the n-Body example, its state was kept constant around 4.2 megabytes, and the whole Probe had approximately 35 megabytes. This program took more than one hour 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 multicluster, with a 1gbps network throughout the university, is negligible.

When Sweep 3d used the 150-unit cube, the probe was around 2.7 gigabytes, while bt.b was around 1.8 gigabytes. Both of themwould be overkill, so now we experiment with reduced probes while trying to maintain prediction quality.

C. 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?

To answer to this question, we selectively verified our prediction quality with less and less phases. Figure 5 shows some of those results. We now proceed to discuss them individually.

1) n-Body: : as seen in figure 5, item (a), execution dominated about 40% of its time by a single phase, while the other ones are not that important.

2) Sweep3d.50: : in figure 5, item (b) is a short execution, therefore there's no dominant behavior, so every phase is important, and the curve is smooth. Although this makes easier to choose the prediction precision, it makes bigger Probe size reductions more difficult. Removing the five less relevant phases would cut Probe size by half, while keeping prediction around 78% of prediction quality. The tradeoff between probe size and prediction quality must be decided by the user.

3) BT.B: : the case shown in figure 5, item (c) tells a different story. Although the execution is dominated only by three phases, the big number of phases and their relative importance makes it difficult to trim our Probe while not affecting considerably our prediction.

4) Sweep3d.150: : however, figure 5, item (d) shows diverse behavior. Its execution is highly dominated by two phases, being others of little relevance. With this data in hand, we can selectively remove the less important phases while maintaining the desired prediction quality.

While this method can reduce substantially Probe size and yet do a good prediction, sometimes even this would be overkill. The Probe for the case c, even after trimmed down to 43% of its original size, is still a benemoth of 890 megabytes, which would take about 2h20m to be transferred in a 1mbps network link, which may not be realistic in multi-clusters scattered across the internet. However, fast networks among universities are quite common, and that issue was not found in real environments we had access so far.

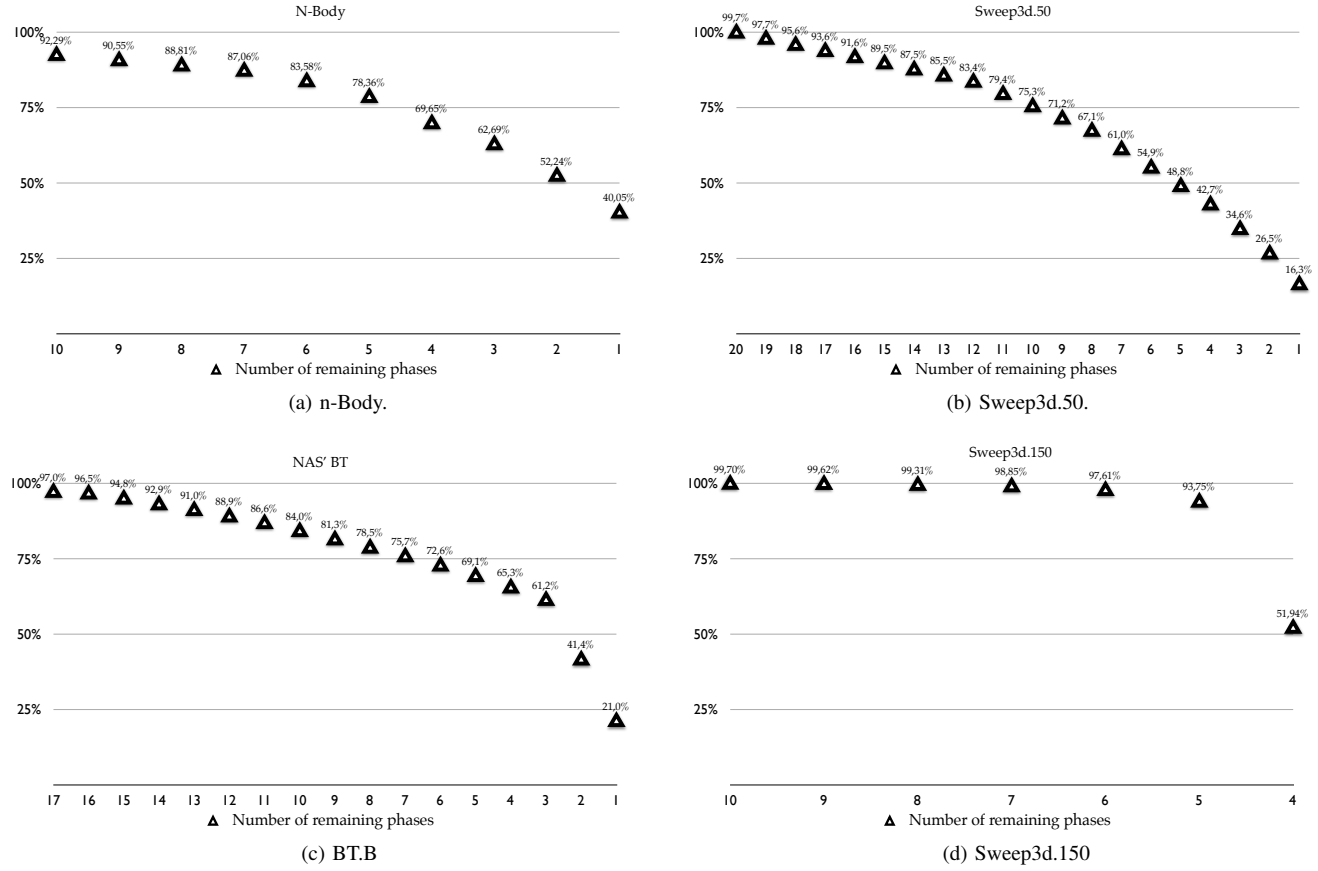
The last experiment tells a different story. As can be noted in Figure 6, execution is dominated by two phases, in more than 93% of the time. However, the remaining phases takes the most space on the probe. If we remove those less relevant phases from our probe, we can yet achieve this 93% of prediction with only a fraction of the required space for the original probe. In this case, we reduce our probe size in 75%.

With these experiments, we were able to show that our prediction method works. It can reduce the time required to characterize an application immensely, while maintaining a good degree of confidence on the results. Our worst result was still below 8%, which we consider enough, given we reduced our characterization time in three orders of magnitude.

V. Where does it fit?

Machine characterization by means of checkpoints have a variety of uses. It's a very precise way of characterizing an application, as it's the application itself being run, but only when it's relevant in terms of performance. This is very important in our field of study, with a high degree of machine heterogeneity, with lesser degree of software heterogeneity.

Given the checkpoints' size, it is a very useful way of characterizing a multitude of machines where network bandwidth is not an issue, and machines have relatively similar operating systems and libraries' versions, such across university departments, or high-bandwidth grid projects, where you can specify your requirements in terms of operating systems and libraries. On the latter, the machines able to be characterized would comprise only a subset of the available ones, but anyway they



(a) n-Body. (b) Sweep3d.50.

(c) BT.B (d) Sweep3d.150

Fig. 5. Prediction quality when removing phases.

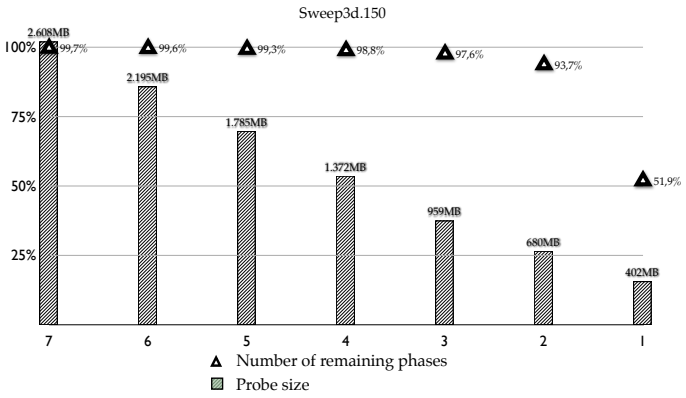


Fig. 6. Probe size versus prediction accuracy.

can be more precisely characterized and quicker than with other methods.

- *When network bandwidth is not a problem*: Companies and universities networks, where fast networks are available, so the Probes' sizes are not a problem, so the characterization will not be dominated by transmission time.
- *Where machines' softwares are relatively similar*: it's common policy to keep stable software versions as long as possible. In the scientific field, this is especially true, where researchers may need very specific versions of libraries for their experiments. This is an usual (and quite logical) requirement for running an application anyway. That is, we must have the environment set to run a given application.
- *Virtualized environments*: On those environments, the set of dependencies an application or a tool may need are all provided in a single virtual image, which is instanced throughout the environment.

In the aforementioned environments, our Probe is especially useful when we need to characterize a lot of different machines in different clusters of our multicluster systems, previous to each execution, thus discarding worthless machines.

VI. Conclusion and Future Work

This work presented a way to characterize the performance of a given application when running on a machine, in a fraction of the time required to run the same application on this machine, which we call software Probe, with the objective of selecting machines in multi-clusters for executions over an efficiency threshold. We use techniques from the simulation

world in real-life environments, which supposes the need for instrumentation and checkpointing, something that is not necessary in simulators, given the simulators themselves may give the data and abilities required to perform the "jump" to specifics points of a program.

Although the use of *SimPoint* in itself is not new for simulating new architectures, we use it in a novel way, to describe and characterize real machines way faster than it was possible before, with a precision usually found in way slower methods of characterizing application/machine pairs. This method for quickly characterizing performance of the worker greatly improves our general method for performance prediction of master/worker applications in heterogeneous multiclusters.

All our Probes ran for less than 0.2% of the time of the application they represented, and we were able to predict the full execution's time with more than 92% precision in all cases tested.

REFERENCES

[1] H. Curnow and B. Wichmann, "A synthetic benchmark," *The Computer Journal*, Jan 1976. [Online]. Available: http://comjnl.oxfordjournals.org/cgi/content/abstract/19/1/43

[2] 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. [Online]. Available: http://archimede.mat.ulaval.ca/intel/mkl/10.0.1.014/benchmarks/mp_linpack/www/

[3] J. McCalpin and C. Oakland, "An industry perspective on performance characterization: Applications vs benchmarks," *Proc. Third Ann. IEEE Workshop Workload Characterization, keynote address, Sept*, 2000.

[4] 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*, pp. 467–474, 2004.

[5] E. Argollo and E. Luque, "Performance prediction and tuning in a multicluster environment," Oct 2006.

[6] A. Strube, D. Rexachs, and E. 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*, pp. 23–30, 2008.

[7] S. Sodhi and J. Subhlok, "Automatic construction and evaluation of performance skeletons," *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pp. 88a–88a, Apr 2005.

[8] A. Wong, "Evaluación de computadores de altas prestaciones: El papel de la aplicación," *Master Thesis*, 2007.

[9] V. Weaver and S. McKee, "Using dynamic binary instrumentation to generate multi-platform simpoints: Methodology and accuracy," *Lecture Notes in Computer Science*, vol. 4917, p. 305, 2008.

[10] T. Sherwood and B. Calder, "The time varying behavior of programs," 1999. [Online]. Available: citeseer.ist.psu.edu/sherwood99time.html

[11] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction is -," *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on AB -*, pp. 336– 347, 2003.

[12] 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. [Online]. Available: http://doi.ieeecs.org/10.1109/PACT.2001.953283

[13] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, no. 3, pp. 13–25, 1997.

[14] 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*, pp. 81– 92, 2004.

[15] P. Fritzsche, "Podemos Predecir en Algoritmos Paralelos No-Deterministas?" Ph.D. dissertation, PhD Thesis, University Autonoma of Barcelona, Computer Architecture and Operating Systems Department, 2007.

[16] J. MacQueen and W. M. S. I. U. ..., "Some methods for classification and analysis of multivariate observations," *projecteuclid.org*, Jan 1966. [Online]. Available: http://www.projecteuclid.org/DPubS/Repository/1.0/Disseminate?view=body&id=pdf_1&handle=euclid.bsmsp/1200512992

[17] G. Hamerly, E. Perelman, and B. Calder, "How to use simpoint to pick simulation points," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 4, pp. 25–30, 2004.

[18] E. Argollo, A. Gaudiani, D. Rexachs, and E. Luque, "Tuning application in a multi-cluster environment," *Proceedings of the Euro-Par 2006*, Jan 2006. [Online]. Available: http://www.springerlink.com/index/ljg1qq04614153m2.pdf

[19] G. Rodriguez, M. Martin, P. Gonzalez, and J. Tourino, "Portable checkpointing of mpi applications," *12th Workshop on Compilers for Parallel Computers*, Jan 2006. [Online]. Available: http://www.des.udc.es/~gralvarez/personal/files/publications/cppc-cpc06.ps

[20] P. Hargrove and J. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, pp. 494–499, 2006.

[21] A. Bouteiller, T. Herault, G. Krawezik, and P. Lemarinier, "Mpich-v project: A multiprotocol automatic fault-tolerant mpi," *International Journal of High Performance Computing ...*, Jan 2006. [Online]. Available: http://hpc.sagepub.com/cgi/content/abstract/20/3/319

[22] J. Hursey, J. Squyres, T. Mattox, and A. Lumsdaine, "The design and implementation of checkpoint/restart process fault tolerance for open mpi," *Workshop on Dependable Parallel*, Jan 2007. [Online]. Available: http://mirror.kapook.com/open-mpi/papers/dpdns-2007/dpdns-2007.pdf

[23] S. Sankaran, J. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The lam/mpi checkpoint/restart framework: System-initiated checkpointing," *International Journal of High Performance Computing Applications*, vol. 19, no. 4, p. 479, 2005.

[24] 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*, Jan 2000. [Online]. Available: http://hpc.sagepub.com/cgi/content/abstract/14/4/330

[25] D. Bailey, E. Barszcz, J. Barton, and D. Browning, "The nas parallel benchmarks," *International Journal of High Performance Computing Applications*, Jan 1991. [Online]. Available: http://hpc.sagepub.com/cgi/content/abstract/5/3/63

[26] A. Joshi, L. Eeckhout, L. John, and C. Isen, "Automated microprocessor stressmark generation," *Proceedings of International Symposium on High Performance ...*, Jan 2008. [Online]. Available: http://www.elis.ugent.be/~leeckhou/papers/hpca08.pdf

[27] R. Murphy and P. Kogge, "On the memory access patterns of supercomputer applications: Benchmark selection and its implications," *Computers, IEEE Transactions on*, vol. 56, no. 7, pp. 937 – 945, Jul 2007. [Online]. Available: http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=4216292&isnumber=4216283&punumber=12&k2dockey=4216292@ieeejrns

[28] M. D. Andrade, "Automated and accurate cache behavior analysis for codes with irregular access patterns," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 18, pp. 2407–2423, 2007. [Online]. Available: http://dx.doi.org/10.1002/cpe.1173