Improving Probe usability

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*—During our research for characterizing individual computing nodes using Software Probes, one of the remaining issues for quickly probing them was that of the checkpoints' sizes. This article shows the different approaches we used in order to reduce the Probes' sizes. We demonstrate that in some cases it is possible to reduce the set of Probes of an application up to 95 % of its original size, thus making our approach for machine characterization useful even through slow networks such as the Internet.**

***Keywords*-Performance prediction; Software Probes; Multicluster; Scheduler**

I. Introduction

The need to quickly characterize machines is present on several different use cases. From a multicluster scheduler for master/worker applications that selects and discards computing nodes according to its performance for that application to workflow engines and cloud environments, good and quick information about execution time can improve the general state of such environments.

In [1], the method to optimize executions of master/worker applications was developed. Its shortcome is that it needs to know the execution time of the tasks on every computing node available beforehand. The execution time to perform this action is not acceptable for scheduling prior to an execution.

Cloud, grid schedulers and workflow engines, on the other hand, trust on the results of generic benchmark indexes to decide where to send processes. Although generic and readily available, benchmarks tend to lack precision or relatedness to the program to be run.

The tradeoff between thorough executions, and the lack of relatedness with real codes from benchmarks, was our motivation for creating a methodology that can predict application performance while being as fast as a micro-benchmark but as precise as running the whole application itself as possible. This resulted in what we call a software Probe. When sent to computing elements, the probe is able to precisely tell how long a worker will take to run on that given machine, in seconds.

Our probe speeds up both machine and application characterization. This means that out methodology is not only useful with the master/worker but also for every system that can benefit from precise information regarding application execution time, such as grid schedulers and workflow engines, as they can be optimized when they have this information for the next executions.

The novelty of this paper is a series of developments made in order to reduce probe size. Probes in this research consisted of the relevant parts of an application's execution. For it to be run autonomously, a series of checkpoints must be created in special moments during the execution, which means that a probe must carry several checkpoints at the same time.

The probe is sent to machines in remote locations through the internet, where bandwidth can be relatively slow and variable. Some applications can keep a considerable state in memory, so a complete probe could be in order of dozens of gigabytes of memory, which in slow and unreliable networks such as the Internet, could take a long time to be sent. The fact that the probes were so big was always a difficult point in our research, so we came up with a series of solutions that, when used together, are able to reduce the probe size in around 90%, which in turn makes the Probe usage feasible on today's networks.

This makes the act of probing faster, which helps us in two different fronts: speeding up the process of probing several different machines, thus being able to better inform schedulers faster, and improving quality, as Probes for different workloads or working set sizescan be sent to the same machine, in the time when only a single regular probe could be sent, which is already faster than running a task by itself.

This paper is organized as follows. In section II we list some work related to ours, and section III describes the issues about determining the performance of an application over a machine. Section IV discusses the process of Probe creation, and in section V, we describe our approach to solve the issues mentioned at the introduction and the current implementation and their results are shown in section VI. Section VII closes the article and suggests future work.

II. Related Work

The work of Sherwood, Sair and Calder [2] is one of the foundations of this research. They created a method to find repetitive behavior on applications to reduce runtime for benchmarking future processors on simulators. They proposed a profiling technique that identifies different phases that may repeat, using basic block distribution analysis. Once an application is profiled, for instance from an execution trace of an architecture simulator, the basic block profile is fed into their *SimPoint* tool. A basic block vector (BBV) contains the index of basic blocks executed during that instruction interval and their occurrence.

A work related to ours is that of Sodhi and Subhlok [3]. Their performance skeletons intend to mimic application behavior in a shorter execution time for evaluating shared resources. While they focus on shared resources/network usage, we focus this work on the computation 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 [4].

The plugins created by Weaver and McKee in [5] runs under the QEMU emulator or the Valgrind tool as a method to gather basic block vectors faster than when using functional simulators, but to use them in functional simulators. Our work instruments applications in real machines, for characterization on those real environments. Also, a system called Adaps [6] states that it is possible to use our probe methodology to create a predictor for their system.

III. Application Performance

In this work we consider performance under two points of view: how fast a machine runs different programs, and how fast a program runs on different machines.

The use of traditional benchmarks for determining performance of machines is only useful to have an index to compare machines between themselves, and only under some specific circumstances. Benchmarks are hardly able to represent an application's performance [7].

It is a known fact that estimating performance of applications on machines is a hard task. Functional simulators for future generations of hardware tend to be slower than real hardware in orders of magnitude, but yet they need to run real code on those simulated machines in order to estimate performancem, as stated in the previous paragraph.

On the other hand, full executions of applications for determining their performance is not desirable in many different contexts, given the time taken. One example is that from the aforementioned study of future hardware through simulation. But there are other cases where a full execution of an application is not desirable either:

A. Efficiency in multi-clusters

The work in multi-clusters already introduced at section I and further developed in [8] 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 and the estimated execution time, as in equation 1:

$$Efficiency = \frac{T_{best}}{T_{ex}} \tag{1}$$

On [1] and [8], to be able to estimate execution time, a single worker task was sent to *every available node in the whole system and ran to its completion*. This was an issue even for small worker tasks, as the number of tasks could eventually be smaller than the number of available nodes. In the case of exceptionally slow computing nodes, this also can delay the whole characterization.

B. Grid and Workflow Scheduling Policies

Another field where fast characterization with good prediction quality can help is that of scheduling policies for workflows, grid and cloud environments. Usually, scheduling policies base their decisions on benchmarks, execution of first task or historical data. The absence of good, fast data about execution time of applications on specific computing nodes makes experiments on real environments difficult, therefore, most of them are simulated. For instance, [9] simulated experiments where individual nodes' performance is random around the average node. The work of [10] states that their performance values are "declared by the service providers", but also suggests that they are probably based on historical executions.

The information obtained by our Probes can be kept by the scheduling/queuing system as historical data as well, given it is almost as precise as a real execution of a given application, thus needing no great changes in schedulers.

IV. Probe Creation

Our Probe methodology consists of two steps: application and machine characterization. This general schema is shown in figure 1.

A. Application characterization - probe construction

The main goal is to evaluate the performance of a machine while running an application quickly and precisely. Our methodology to evaluate an application's performance on a machine consists of the following steps:

1) Data collection: The most basic concept of our probe is the *basic block*. A basic block is a set of instructions with only one point of entry and exit - that is, an arbitrary number of instructions where no jumps in the process counter occurs.

During an execution, it is possible to trace the number of executions of each basic block during a fixed number of instructions, thus creating a *basic block vector* (BBV).

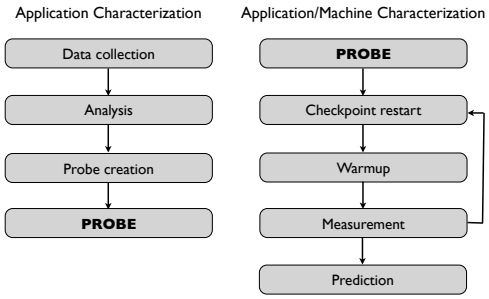

Figure 1. Probe creation general scheme. While the left side of the image is done in a machine under our control, the right side is done on the machine to be characterized.

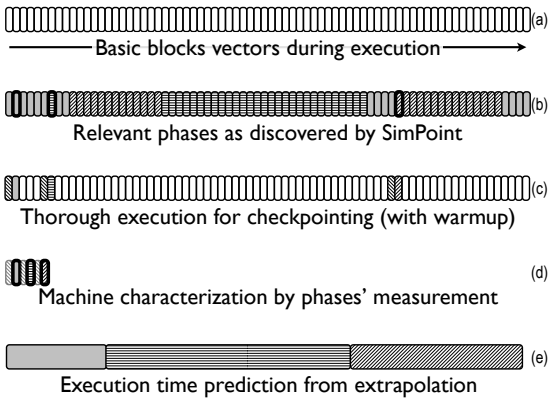

Figure 2. Stages for Probe generation and machine/application characterization

One execution will generate a number of these BBVs. The contents of these BBVs are the number of times each basic block was executed during that instruction interval. It is possible to compare the similarity between the BBVs. BBVs have millions of instructions, so similar BBVs have the tendency to be executed in similar times. This way, it is possible to characterize a program execution in phases that repeat according to this similarity. Similar phases tend to have similar execution time. By running a phase a small number of times, it is possible to extrapolate its behavior to the whole program. Therefore the execution of all the unique phases makes the extrapolation of the program's execution time possible.

To collect basic blocks of real application running in a real system (in opposition to simulation as in other projects), we instrument the application with Pin [11]. We used intervals of 10, 50 and 100 million instructions to characterize phases (see section VI). We then run the application monitored thoroughly, to collect the basic block vectors, as in Figure 2, item (a).

About data dependency and problem size: A side effect of the method used for characterization has to do with the workload, as our Probe is valid for a given one. However, in some domains, this is not 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 data patterns characteristic to the underlying algorithm, as studied by Fritzsche [12].

2) Analysis: The collected basic block vectors are fed into *SimPoint*, which discover sprogram phases based on BBVs' similarity. These phases are classified and the tool outputs a single occurrence of it, counted in number of instructions. *SimPoint* also outputs each phase's weight, see Figure 2, item (b), where each fill pattern represents a single phase. The bold BBVs 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.

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 special reduced 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 again, with instrumentation that counts the executed instructions until it reaches a point right before a phase's number, and then the application is checkpointed, 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. Its worth noting that this offline process to create a probe must be done only once.

In heterogeneous environments network connections are an issue, so big checkpoints are problematic. Our experiments have shown that connection droppings and bandwidth limitations are the norm, not the exception [1]. We created a method to transport only the required parts of the checkpoint, without previous knowlege of the source code, in opposition to [13], for instance. This method can reduce the Probe size around 90%, as can be seen in section V.

B. Probe use for fast performance prediction

The application/machine performance is determined by running the Probe phases, measuring phases' time and extrapolating them to the whole execution, according to its weights.

1) Checkpoint restart: A shell script runs every phase - that is, restarts the program from the checkpoint. Instrumentation on the Probe itself interrupts the program on the

end of the phase and measures time. This time is sent back to the master. The current implementation uses BLCR [14], because of its wide availability.

2) Measurement: We start the checkpoint before the phase, to let the architecture warm up - see Figure 2, item (c) and (d). After the warmup, a timer and an instruction counter alarm are set. After the number of instructions is reached, execution is stopped, time is measured and sent back. The script proceeds to the next phase.

3) Execution time prediction: Prediction can be achieved both by extrapolating from the phase time to the full execution and by comparison of the same phases' time in another machine, in order to remove residual noise from instrumentation, such as in equation 2. 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.

$$Noise = \frac{T_{Instrumented}}{T_{Uninstrumented}} \tag{2}$$

The estimation of execution time is given by equation 3, 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.

$$\sum_{Phases} \left(\frac{T_{FullApp} * W_{Phase}}{T_{off}} * T_{New} \right) \tag{3}$$

For instance, an 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,

$$\left(\frac{60m * 60\%}{2} * 1 \right) + \left(\frac{60m * 30\%}{2} * 1.5 \right) + \left(\frac{60m * 10\%}{1} * 0.5 \right) \tag{4}$$

That translates as (18)+(13.5)+(3) = 34.5, which means that this worker would take 34.5 minutes to run in this machine.

V. Reduction

One issue with our 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 section VI, one of the Probes is around 1.3 gigabyte. This means a transmission time of around three hours in a 1mbps connection, something not entirely unusual on the Internet. A strategy to reduce probe size was essential.

The way we achieve reduction of the probe is done by three different methods. They are

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

They will be detailed on the following sections.

A. 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 and the loss of precision.

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. 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.

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 amounts of checkpoints.

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.

The experiments of removing these phases can be seen in figure 3. On it, we show the overall prediction quality and how this quality degrades while reducing the number of phases. It can also be seen how the Probe's size is also reduced.

The removal of little-relevant phases is valid, and in fact is necessary, as it can greatly increase the total Probe size

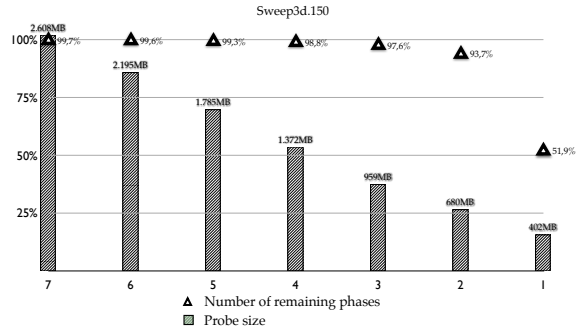

Figure 3. Prediction Quality x Probe Size.

with little gain in quality. However, how much of prediction quality must be kept is up to the user.

B. Compression

Compression itself is a widely studied subject in computer science 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.

C. 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 [15].

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. In that amount of time - also known as *tracking window*, 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 [16] is the memory contents accessed during the tracking window - in our case, during a phase's execution. All memory not accessed during the tracking window does not need to be present in the checkpoint.

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:

1) Probe generator: During the Probe generation phase, the last step was to run the application to the end in our characterization machine in order to save the instrumented checkpoints. Now, after saving a checkpoint, a trace of every memory page accessed during a phase execution is recorded. We do this by changing the instrumentation code to be performed only when it is saving a checkpoint, to avoid the creation of a trace (and its massive overhead) when in checkpoint restore mode. This is made because restore mode in our system means that this phase is being characterized, and therefore a memory trace in instrumentation will change the results in orders of magnitude.

2) Checkpointing library kernel module: 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 kernel module to also create a trace of the tuple [memory page address / file offset].

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 touched set ξ in the checkpoint file is given by the set definition 5:

$$\xi = (\theta \cap \varphi) \tag{5}$$

And the set of pages ζ to be removed from the checkpoint file is in definition 6:

$$\zeta = (\theta \cup \varphi) - \xi \tag{6}$$

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, are useless, are filled with zeros, so the file can be now considered sparse. This has the advantages of keeping binary compatibility with standard BLCR; which means we can run these Probe phases on machines running standard versions of BLCR it is easily compressed, as all sequences of zeros we introduced are multiples of a memory page size.

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, which can be sent through the network and uncompressed on the fly, with a pipe from the process receiving the byte stream to the *gunzip* utility. 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 section.

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 the 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.

VI. Experimental Results

Now we proceed to show some results regarding probe size reduction. We hereby focus solely on reduction as precision quality is not affected by the methods described in this work.

In table I, we can see the Probe's sizes unchanged, with the number of phases as given by Simpoint for a 99.5% quality - which means a preliminary reduction was already made, and a maximum number of 30 phases. The *reduced* column is the value with the following reductions applied: *touched-sed approach*, *compression* and *removal of the less-relevant phases*, keeping a minimum prediction quality of 75%. The amount of *warm-up* was of 10,000 instructions, and the basic block vector size was of 100 million instructions.

Table I
Probe size reduction with 100M instructions

Application	Original	Reduced	% Reduction
Sweep3d.150	1362456308	121029068	91.11%
Sweep3d.50	168158756	23472358	86.04%
BT.A	373487077	72353660	80.82%

VII. Conclusion

Probes are a very useful tool for predicting execution time in remote machines, for several different programming models with significant serial parts. It is able to precisely predict the execution time of applications in a matter of seconds.

The usefulness of the probes were limited by its size, given by the checkpoints. In this paper, we have shown the methods we now use for reducing probe size. The different techniques together are able to reduce the size of the Probe to be sent in one order of magnitude. This turns Probes into a viable solution for performance prediction in today's network speeds.

Future work in this research is required with the multi-threaded, multi-core machines of today. How well will the Probe be able to represent applications in processors with some duplicated functional units (such as the Intel's Hyperthreading ones) or with shared caches where different parts of the code or even different processors can be running is a question that still needs to be answered.

Acknowledgment

This research has been supported by the MICINN-Spain under contract TIN2007-64974.

REFERENCES

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

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

[3] 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)*, pp. 88–98, Apr 2005.

[4] A. Wong, D. Rexachs, and E. Luque, "Parallel application signature," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009, pp. 1–4.

[5] V. Weaver and S. 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*, vol. 4917, pp. 305–309, 2008.

[6] 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.

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

[8] E. Argollo, D. Rexachs, F. Tinetti, and E. Luque, "Efficient execution of scientific computation on geographically distributed clusters," in *Applied Parallel Computing*. Springer, 2006, pp. 691–698.

[9] J. Kim, J. Rho, J.-O. Lee, and M.-C. Ko, "Cpoc: Effective static task scheduling for grid computing," *High Performance Computing and Communcations*, pp. 477–486, 2005: Springer.

[10] H. Wanek, E. Schikuta, and I. U. Haq, "Grid workflow optimization regarding dynamically changing resources and conditions," in *GCC '07: Proceedings of the Sixth International Conference on Grid and Cooperative Computing*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 757–763.

[11] 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.

[12] P. Fritzsche, D. Rexachs, and E. Luque, "A computational approach to tsp performance prediction using data mining," in *Advanced Information Networking and Applications Workshops, 2007, AINAW'07. 21st International Conference on*, vol. 1. IEEE, 2007, pp. 252–259.

[13] G. Rodriguez, M. Martin, P. Gonzalez, and J. Tourino, "Portable checkpointing of mpi applications," *12th Workshop on Compilers for Parallel Computers*, pp. 396–410, Jan 2006.

[14] 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.

[15] S. Golomb, "Run-length encodings (corresp.)," *Information Theory, IEEE Transactions on*, vol. 12, no. 3, pp. 399 – 401, 1966.

[16] Y. Li and Z. Lan, "Frem: A fast restart mechanism for general checkpoint/restart," *Computers, IEEE Transactions on*, vol. PP, no. 99, pp. 1 – 1, 2010.