Performance Prediction of Distributed Applications Using Block Benchmarking Methods

Bogdan Florin Cornea and Julien Bourgeois

LIFC, University of Franche-Comté, 1 cours Leprince Ringuet, 25201, Montbéliard, France
{bogdan.cornea, julien.bourgeois}@univ-fcomte.fr

***Abstract*—An ongoing work is presented for accurately predicting the performance of distributed applications in heterogeneous systems. We are developing dPerf, a tool built using the Rose framework for performing static analysis and an automatic instrumentation on the input source code of programs written in C, C++ or Fortran. The accuracy in predicting program computation time resides in using hardware counters, as well as in applying two block benchmarking techniques that we propose in this paper. The current work makes use of a network simulator in order to calculate the communication time used in our approach. Afterwards, the computation and communication times are being summed up obtaining an estimation of the distributed application execution time. The approach is proven experimentally using NAS Integer Sort benchmark, the communications being simulated with SimGrid.**

***Keywords*-dPerf, performance prediction, block benchmarking, static analysis, trace-based simulation, heterogeneous systems, MPI, P2PSAP**

I. Introduction

The two main criteria for a performance analysis are efficiency and precision of the employed method or tool. Estimating the performance of a parallel program is a difficult task. As various approaches exist, the most accurate remains the simulation, but it is often very time consuming and therefore lacks efficiency. So far, the estimation on the communication and computation times were generally performed in a dynamic manner. In this paper we present two methods based on the fundamental idea [1] which uses static and semi-static MPI program analysis. We are extending that principle and we create a tool that combines static analysis with instrumented execution and with trace-based simulation for heterogeneous systems. Most prediction tools accept as input a source code written in one particular language. The described work refers to the development and results obtained with a tool which supports multiple programming languages passed as input, and is capable of applying two different block benchmarking methods. This tool outputs the computation time and information about the communications in a distributed application. For this we developed dPerf (**d**istributed **Perf**ormance Prediction) using Rose in order to analyze programs meant to run on parallel and later on decentralized systems, and obtain accurate and scalable performance predictions. The programs to be analyzed communicate using either MPI or the P2P self-adaptive protocol (P2PSAP) currently under development at the LAAS-CNRS laboratory [2, 3]. The dPerf approach is to use methods available in Rose [4] for performing static analysis on a C, C++ or Fortran input source code, these being the three most intensively used languages in the High Performance Computing (HPC) community. We create a model of an input source code as a result of static and dynamic analyses, we perform a micro benchmark on the target system and determine which is the threshold number of iterations necessary for performing very accurate and scalable block benchmarking. Our approach is valid for both homogeneous and heterogeneous systems, with one restriction. In the case of heterogeneous systems such as computing grids and P2P, the micro benchmarking part must be performed if the architecture type changes. Section II briefly describes the related work in the field of performance prediction, followed by the methodology and requirements for developing dPerf, explained in section III. Section IV shows experimental results that justify our proposed block benchmarking technique, ending with the conclusions and future work in section V.

II. Related Work

Over the years, performance prediction methods were developed with respect to that period's computing systems. The various performance tools were meant to demand as little manpower as possible, to take less time than an actual execution of the evaluated application, to provide developers with an insight of their application behavior, or to guide scientist in choosing future HPC architecture configuration that best suits their requirements. Existing performance prediction methods can be classified into *analytical* [5–7], *profile-based* (based on compilers and instrumentation tools) [1, 8], *simulation-based* [9, 10], and *hybrid* [11–14], the latter category being a combination of profile- and simulation- based.

Most aforementioned methods estimate application performance for computing systems with single processors, or they are developed for specific applications. One common drawback of performance prediction techniques up to this

day is the support for homogeneous computing systems only. Due to this, we describe a modern hybrid model that targets heterogeneous systems which aims at obtaining scalable models of application's performance. Unlike the work presented in [14], our tool addresses distributed applications written in C, C++ or Fortran, our approach handles communication made using MPI or P2PSAP, and the prediction is based on block benchmarking techniques that extend and improve the work presented in [1].

III. Methodology and requirements

Analyzing application performance is a difficult task. It becomes even more difficult to achieve when the applications are written for parallel execution. These programs are of higher complexity than sequential ones, and therefore, any approach that targets parallel programs would also work for sequential ones, and all methods for evaluating parallel application performance, will also apply for sequential applications. In order to perform source code analysis, a series of requirements must be met.

The first requirement is for our tool, dPerf, to accept three of the most intensively used languages in parallel programming, that is C, C++ and Fortran. For this, we use the Rose compiler [4], a framework that meets the first requirement and more. The work-flow is shown in Fig. 1. One of Rose features is the ability to decompose an input code into Intermediate Representations (IR). The simplest IR generated with Rose is the Abstract Syntax Tree (AST), containing the tree representation of a source code passed as input to Rose. From this, Rose can also represent the Control Dependence Graph (CDG) as well as the Data Dependence Graph (DDG), depicting the control and data dependences between a source code instructions. The supergraph of all these representations is the System Dependence Graph (SDG). An SDG contains all dependences, of control or data type, as well as paths to show variable propagation inside a program.

A second requirement is for dPerf to take various measurements of the code such that it will introduce as less noise as possible into the measurements taken. This is achieved by using the Performance Application Programming Interface (PAPI) for accurately timing processing costs in nanoseconds. Calls to the PAPI library are injected into the analyzed code by dPerf for reading hardware counter values and for computing instruction block duration, or computation time.

Our tool performs static analysis of distributed programs by using the IR obtained with Rose (see Fig. 1). To our approach, the SDG in particular is of great importance due to its complexity. When static analysis is not possible either due to the non-deterministic type of a distributed code, or to a dependence among variables that can not be solved using the SDG alone, the dPerf tool will need additional information regarding data dependences, these being obtained upon user interaction.

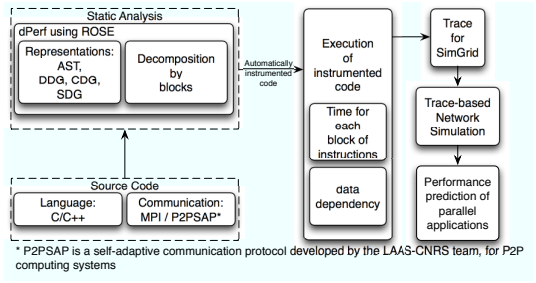

Figure 1. Work flow diagram

At the current development level, the source codes used as input are deterministic. As depicted by Fig. 1, the approach for achieving accurate static or dynamic performance prediction is:

- A parallel or distributed program is given as input to dPerf, our Rose-based translator;
- dPerf obtains the AST and the SDG for the given input code;
- Using the AST, the input code is decomposed into instruction blocks as small as possible. Each block is verified for calls for communication (MPI or P2PSAP). If such calls exist, then the block is split so that the call would not be contained by the instruction block. (see Fig. 2);

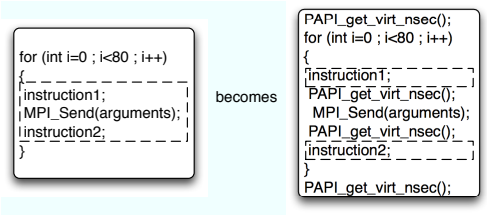

Figure 2. Decomposing a block which contains an MPI call into simplest communication-free sub-blocks. Same principle applies to P2PSAP calls.

- Once all blocks and communication primitives have been identified, calls to PAPI are inserted before and after each block;
- The altered code is unparsed using Rose and an instrumented code is obtained, this being written in the same language as the original input code;
- Upon execution of the instrumented code, trace files corresponding to each of the parallel processes are obtained. The traces contain only the instruction computation time and the relevant communication parameters;
- The traces are passed as input to SimGrid [9], this simulator being responsible for adding the communication time to the computation time supplied by the trace files.

The result, a succession of $t_{compute}$ and $t_{communication}$, represents the performance prediction of the distributed application written in C, C++ or Fortran passed as input to

dPerf, the translator that we developed. Assuming that for a distributed application we have a fixed number of machines, the two major advantages when using a decomposition in $t_{compute}$ and $t_{communication}$ are:

- $t_{compute}$ is computer architecture-dependent, that is, for any identical machines, $t_{compute}$ remains unchanged. If a change in network topology occurs, only the communication part needs to be updated.
- $t_{communication}$ is network-dependent, that is, as long as the network conditions do not change, neither will the communication time. If a machine architecture modification occurs, only the computation time needs to be updated.

Libraries and tools used

DPerf uses three essential components previously mentioned: PAPI, Rose and SimGrid.

1) Hardware Counters and PAPI: Most computer architectures nowadays use microprocessors consisting of special-purpose registers that count processor activity. By accessing these registers, accurate measurements of user and system events is achieved. Our measurements rely on the information retrieved from these hardware counters.

For GNU/Linux systems, access to the performance counters is possible only if the *Performance-Monitor Counters* module was enabled in the kernel. Two measurement infrastructures can enable the performance counters module: perfctr [15] and perfmon [16]. Based on [17], our approach uses *perfctr* and PAPI [18], [19]. Once *perfctr* has been enabled, by using PAPI developers gain access to a wide range of information given by the counters with a minimum noise level introduced into the measured system, thus improving performance analysis results. Two interfaces are available within the PAPI library. interface. *The PAPI high-level interface* is used for performing quick and simple measurements, such as retrieving the total number of available hardware counters or getting the number of nanoseconds since the previous call to PAPI, and so on. *The PAPI low-level interface* is less restrictive, and it provides an advanced interface for our performance prediction tool. Example of calls:

- PAPI_get_real_nsec accesses the counter and gets the real (user and kernel) time in nanoseconds. The user and kernel time (see Fig. 3) refers to the total time from the first CPU time unit, called time slot, assigned for the user process, until the last time slot required by the process to finish the operations, including all other processes that occur at the same time.
- PAPI_get_virt_nsec accesses the counter and gets the virtual (user) time in nanoseconds. We are interested only in the current process or user time, and therefore the virtual (user) time is captured.

Before, time measurements were taken with gettimeofday(). Gettimeofday() returns results in microseconds but its measurement depends on the time slot assigned to the process that initiated gettimeofday [20]. Moreover, the processor-cost for calling the gettimeofday function itself is quite significant.

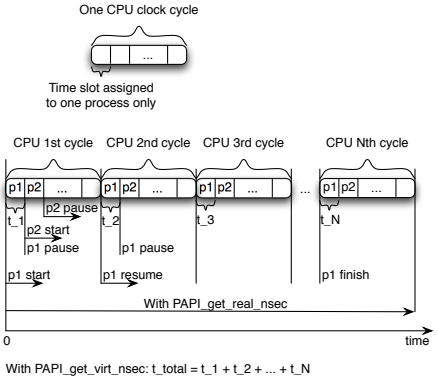

Figure 3. Measuring the real and virtual time cost with PAPI

2) The Rose framework: This is a compiler infrastructure available to developers who want to build custom tools for source-to-source program transformation and analysis. It can analyze large scale applications. Since the custom tools based on Rose accept C, C++, Fortran, OpenMP and UPC programs, it means that these tools cover the most part of applications running on parallel and distributed systems. Rose is most suited for building tools for static analysis, as well as program and performance analysis.

Here are some major features of Rose compiler:

- it builds IR based on the Sage III IR, a improved version of the Sage++ compiler preprocessor toolkit [21];

- it preserves in the Rose AST the entire information found in the original source code;

- the custom built program analysis tools can be built on top of any of Rose representation: AST, Call Graph (CG), DDG, CDG, or SDG;

- it has a great number of methods for analyzing and (or) modifying the AST, while checking for any violations of the programming language used for the input code;

- at the end of a transformation process, Rose can unparse the modified AST into a new source code in the same programming language as the input code.

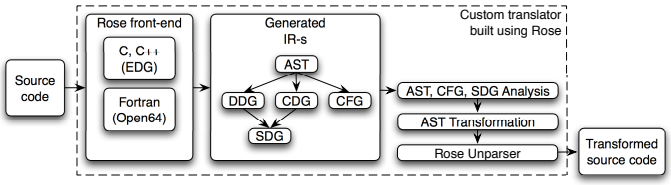

Figure 4. Diagram of a custom tool built using the Rose compiler framework

We have chosen to use Rose for building our custom performance prediction tool, dPerf, mainly due to the following

reasons:

- our target applications are developed using C, C++ or Fortran, languages supported by Rose;

- due to its front-end, Rose does not loose any information about the original code when it parses and builds an intermediate representation (IR) from an input source code;

- several program analyses are developed for Rose (see Fig. 4) making it easier to access call graphs (CG), CDG, DDG, data dependence (DD), SDG, and communication patterns. All these possible analyses being an important step forward static analysis of distributed applications;

- it is a project under development, hence it will be supported and maintained for a long time.

A series of analyses can be performed on the IR but depending on the complexity of the input source code other dependence flows could be required. For this reason, dPerf gains a second advantage compared to other program analyzers, this being the use of methods available within Rose for analyzing the DDG, CDG, and SDG of an input code. In the following, it is described the relevance of each IR to our approach.

The *AST*[1] is the fundamental syntactic representation of a single file source code. It can be easily analyzed and based on its traversal, any transformation can be performed. DPerf uses the AST to identify key elements such as statements, basic blocks and communication calls. As long as the statements present in the source code do not depend on other variables, the AST suffices for performing static analysis on the input program. The *DDG*[1] offers information about the dependences among analyzed data. The *CDG*[1] is a representation of control dependences between the vertices of an AST. The *SDG* is a super-graph [22] containing the data and control dependences combined into one representation[1]. By analyzing the SDG, dPerf can solve variable dependences. This is a crucial aspect when performing static program analysis. Making use of the Rose methods for analyzing the SDG, dPerf would achieve a higher precision when predicting application performances.

All transformations are made at AST level. The CDG, DDG, and SDG only describe dependences between the vertices of the AST. DPerf stores the memory address of the vertex that will be subjected to modifications, then this address is used to perform the desired modifications to the AST.

3) The network simulator: The instrumentation results are trace files to be passed as input for a network simulator. These traces are vital because that the main goal is to achieve performance prediction in P2P environment, where the hardware and software are different from one machine to another, as well as the network connection characteristics in between each peer. *SimGrid*, the framework for building custom grid simulators, contains a special module called MSG capable of performing trace-based network simulations. Using the output from dPerf, SimGrid's MSG module can solve the communication time aspect of our distributed application performance prediction.

[1]Representations for the AST, CDG, DDG, and SDG can be found at the following address, under the same title as the current paper: http://lifc.univ-fcomte.fr/page_personnelle/recherche/136

IV. Experiments

This section presents our experimental work regarding the benchmarking by block and the static analysis, these being done by dPerf. The accuracy of the proposed performance prediction tool was tested with the NAS Integer Sort (IS) benchmark.

A. Input Source Code Analysis

Our work is currently focused on developing dPerf, our tool for performance prediction of distributed applications. At the current development stage, dPerf successfully performs the following:

- Obtaining the AST and the SDG for the given input source code. Regarding the AST, each file has its own IR. In the case of SDG, the representation contains the entire source from the input code plus all the functions defined externally and called from within the input file, provided that all involved sources are in the same directory.
- Identifying communication primitives and inserting the code for calls to PAPI low-level interfaces at specific points in the input source code, for timing instruction blocks using hardware counters (see Fig. 2).
- Decomposing the input code into instruction blocks. Several decisions must be made for each identified block. If the block does not depend on input values, or if it belongs to a loop statement with constant and defined condition, the entire block can be instrumented. If this is not the case, the block is instrumented using the threshold method, so that the cache memory effect could be covered.
- It outputs the transformed source code. After compiling and executing this new source code, it produces traces containing only computation times ($t_{compute}$) for each instrumented block of the original source code, and the parameters of the communication primitives. The aforementioned traces will serve as input to SimGrid, so that it calculates the $t_{communication}$ needed for interprocess communication. In the end, we will obtain an overall estimated time for the input source code.

In the following, we also describe the accuracy of the presented approach with respect to the real, unaltered execution of the input source code.

Analysis of the MPI application

Researchers had shown that static and semi-static methods give promising results [1, 23, 24]. The innovation of the method presented in this paper relies on :

- an increased precision by taking into account the cache memory effect and the compiler optimization levels;
- reduced slowdown due to an analysis based on modified loop bounds (see Fig. 5). The *slowdown* is the phenomenon that occurs when predicting parallel application performance. $slowdown = \frac{t_{prediction}}{t_{normal\,execution}}$ where $t_{prediction} = t_{obtain\,trace\,files} + t_{simulation}$. $t_{obtain\,trace\,files}$ is the time taken to apply dPerf on the source code of IS and a one time execution of the code transformed by dPerf in order to obtain the trace files. $t_{simulation}$ is the time taken by SimGrid to perform a trace-based simulation. DPerf support for modifying

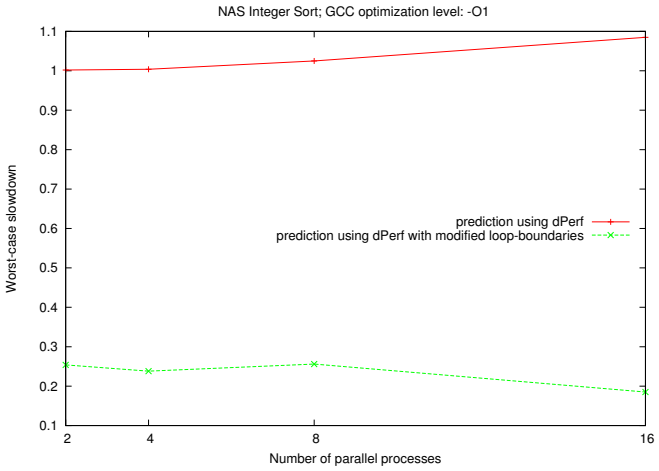

Figure 5. The slowdown for regular performance prediction compared to the method that uses loop-boundaries modification. Both methods are implemented in dPerf.

loop boundaries is undergoing, thus not fully ready yet. At this point, loop boundary modifications are based on the information available through the IR available with Rose. A dependency analysis was performed and we obtained an execution time reduced by more than 75 percent.

Prefetching effect and the Threshold iteration rule

When performing benchmark by block of instructions, the instruction prefetching effect is taken into account. The time cost for one iteration and for tens or hundreds of iterations of the same block is significant. Nevertheless, we observed that after a certain number of iterations, the time per iteration is constant within a small error interval. For this reason, we prepared a block benchmarking technique by a rule called *Threshold iteration*, that is a prediction technique based on modified loop-boundaries. Let the reference number of iterations of a block be denoted by th, or threshold, with t_{th} being the time in nanoseconds where th is reached. For a block with a single iteration, the Threshold iteration rule is expressed as follows:

$$t_{avg_{block}} = \frac{t_{th}}{th}$$

where $t_{avg_{block}}$ is the average time for one block iteration, value that takes into account the time for data pre-loading from memory. Above th, $t_{avg_{block}}$ is constant or within an ε_{th} error interval. Let x be the number of iterations of a block, and t_x its average execution time, then

$$\varepsilon_{th} = t_x - t_{th}, \forall x > th$$

In the case of blocks belonging to loop statements with n cycles, the formula becomes:

$$t_{avg_{loop}} = \frac{t_{th}}{th} \times n$$

The use of this block benchmarking approach makes it possible to scale-up the application prediction times while maintaining accuracy.

B. Experimenting with dPerf and NAS IS

The application that serves as input source code for the following experiments is the NAS Integer Sort (IS) benchmark [25],[26]. NAS IS uses non-blocking point-to-point communication (with MPI_Send and MPI_Irecv), and also collective communication such as MPI_Allreduce, MPI_Alltoall or MPI_Alltoallv. The experiments are performed on a 16-nodes heterogeneous cluster. On each computing node, only one thread will run, regardless of the number of available cores per machine.

- 8 nodes are Intel Pentium D @ 2.8 GHz, 1MB cache, 1Gbps network adapters;
- 8 nodes are Intel Core 2 Duo @ 2.33 GHz, 4MB cache, 1Gbps network adapters.

The network topology is the following:

- The 8 Pentium-s are connected to a HP Procurve 2848 switch, supporting 1Gbps on each port, and
- the 8 Core2Duo-s are connected to a Cisco Catalyst 2900XL, with Ethernet ports of 100Mbps

The maximum bandwidth between the two switches is limited by the second switch to 100Mbps.

We aim at calculating $t_{predicted}$ or $t_{simulated}$, an estimation of IS execution time. Our methods were validated by experiments using the IS code compiled with each of the four optimization levels available in the GCC compiler [27], that is 0,1,2,3 and s. The experimental work described in the following only refers to the IS code compiled with optimization level 1.

The first part of the experiment consists of the original IS application which is executed, in turns, on 2, 4, 8 and 16 nodes. The full execution is measured using PAPI; this will represent the reference time for all comparisons. *The second part* of the experiment applies dPerf on the IS source code. IS is passed as input to dPerf. A static analysis is performed and calls to MPI are prepared for instrumentation by inserting the necessary calls to PAPI library. Two block benchmarking methods are applied at this point in the transformation:

- simple block benchmarking, and
- optimized block benchmarking based on the Threshold iteration rule. This will considerably reduce the time needed for obtaining the performance prediction result.

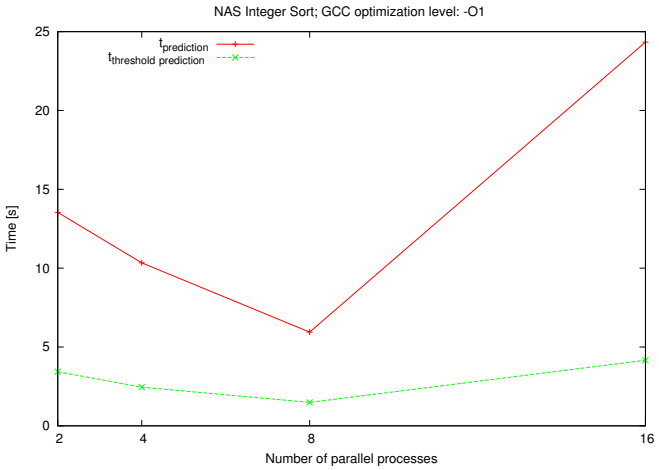

Figure 6. Total time to obtain the prediction when using dPerf with simple or with Threshold-based prediction.

$t_{prediction}$, the time for a prediction with dPerf using *simple block benchmarking*, closely follows the real execution curve, whereas $t_{threshold\,prediction}$, the performance prediction process time with dPerf *based on the Threshold iteration rule* is noticeably faster (see Fig. 6), the accuracy of the results being just as high in the first as in the second case (see Fig. 7).

After all automatic transformations are performed, dPerf uses Rose to unparse the modified code and to obtain a transformed source code. The latter is compiled and, in turns, it is executed on 2, 4, 8 and 16 nodes. The result of each execution are trace files containing $t_{compute}$ and the communication parameters instead of each MPI call. The trace files are passed to SimGrid's trace replay mechanism, and this will simulate all communications based on the parameters found in the trace files. Afterwards, SimGrid will sum up the simulated $t_{communication}$ and all the $t_{compute}$. The outcome is an estimation of execution time for a parallel application, which in this case is NAS IS. We compared a regular execution of NAS IS to two predictions obtained using dPerf (see Fig. 7).

As seen in Fig. 7 and Fig. 8, both benchmarking methods that we use for predicting performance give accurate results.

According to our performance prediction tool and by relying on the work flow in Fig. 1, we state that for one heterogeneous cluster with fixed nodes and N different network topologies, the time per performance prediction is:

- for the very first topology,
  $t_{prediction_1} = t_{obtain\,trace\,files} + t_{simulation}$
- for the 2nd to Nth topologies,
  $t_{prediction_i} = t_{simulation}\ with\, i = 2..N$

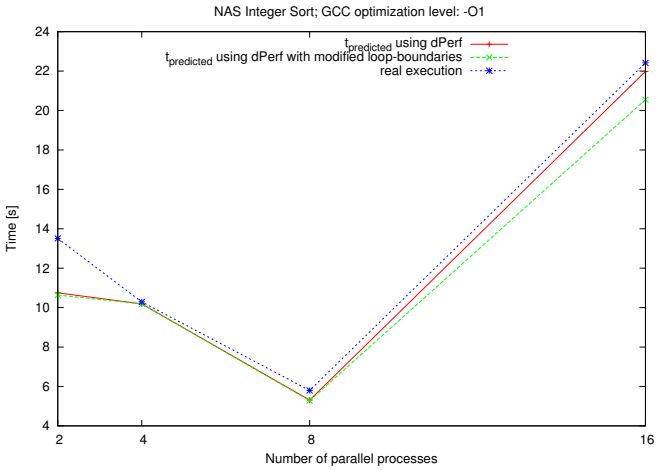

Figure 7. Real and predicted IS execution time with dPerf. First, dPerf applies a simple benchmarking method. Then, a second experiment is made when the Threshold iteration rule is enabled.

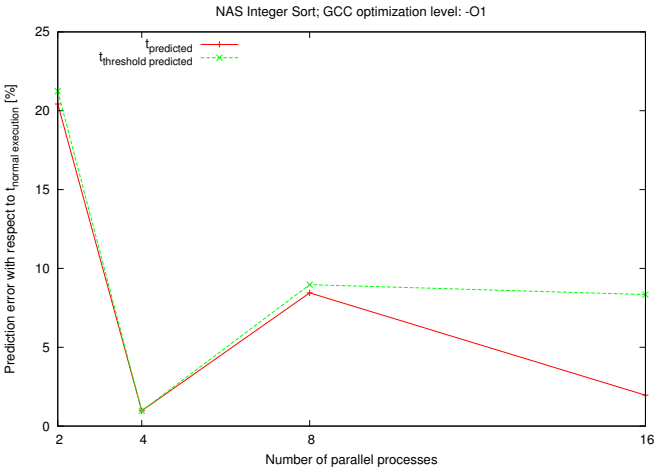

Figure 8. Error percentage of the predictions obtained with dPerf with respect to the normal execution time

since $t_{obtain\,trace\,files}$ remains unchanged and is already known.

This results in having
$t_{per\,prediction} = \frac{\sum_{i=1}^{N} t_{prediction_i}}{N}$.
Fig. 9, denotes the efficiency in predicting the execution time if we change only the network conditions, fact that gives portability to our method.

V. Conclusion and Future Work

In this paper we have presented *dPerf*, our tool for performance prediction that uses static analysis, for the $t_{compute}$, combined with trace-based network simulation, for $t_{communication}$, all this for calculating the overall execution time of a distributed application. The *computation part* is obtained through static analysis of the source code IR, which depends only on the application input parameters, followed

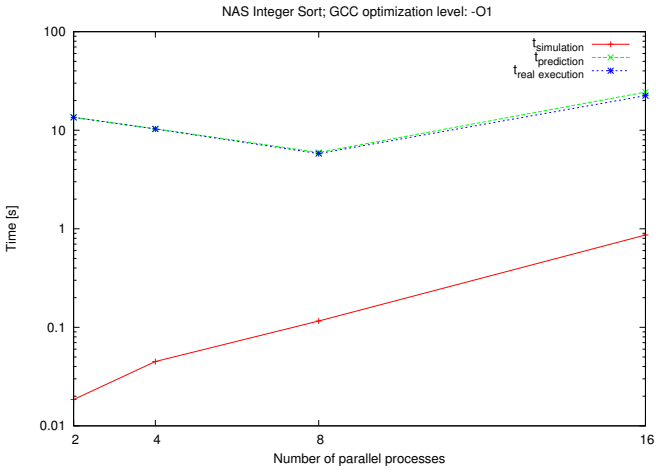

Figure 9. Logarithmic representation of IS real execution, prediction and simulation times

by the execution of the instrumented source code. Re-executing the instrumented code is necessary if we modify the architecture. $t_{compute}$ rely on the use of PAPI and the hardware counters which gives very accurate measurements with little noise. We are using SimGrid to obtain the *communication part* for the experiments presented in this paper. Our approach applies to heterogeneous, and implicitly to homogeneous computing systems. The current development stage and the accuracy of dPerf was tested using the NAS Integer Sort benchmark.

The approach presented in this paper will be developed until we will obtain a performance prediction method with scalable and architecture-independent results. We grant special attention to the use of the SDG, a representation that we intend to exploit in such a manner as to entirely solve all data-dependencies. DPerf accepts code that communicates using MPI and ongoing work exists for applications communicating using P2PSAP. As P2PSAP is destined for P2P computing systems, our approach aims at implementing the necessary support for programs that communicate using P2PSAP. In this manner, dPerf implicitly aims at offering performance prediction results for P2P distributed systems. Rose support for Fortran is undergoing, and therefore it remains an ongoing work from our part so that dPerf could fully apply to Fortran applications. In the end, three of the most widely used programming languages in parallel computing will be accepted as input. We envisage the possibility for developers to choose the performance prediction type that best suits the development state of their application. In this way, dPerf will provide prediction results throughout the entire development, thus the entire life-cycle, of a distributed application. Regarding network simulation, we are interested in adding P2P support to SimGrid and estimate performance of distributed applications in the P2P environment.

As stated in [28–30], an analysis of workload on the parallel or distributed architecture offers a more accurate performance prediction. We envisage a more advanced analysis of the workload distribution on the target architecture. By studying the instruction blocks in assembler, our model could considerably increase its precision. This would be an important feature for dPerf when predicting performance for regular and P2P systems.

We have presented in this paper dPerf, an ongoing effort for obtaining a prediction tool that addresses homogeneous, heterogeneous, and later on P2P systems, and that would be highly portable. DPerf will be applied to parallel or distributed applications written in C,C++, or Fortran which communicate using MPI or P2PSAP.

ACKNOWLEDGMENT

This work is funded by the French National Agency for Research under the ANR-07-CIS7-011-01 contract [31].

We thank Jérôme Vienne for collaborating with us on the efficient use of Rose intermediate representations.

REFERENCES

[1] J. Bourgeois and F. Spies, "Performance prediction of an NAS benchmark program with ChronosMix environment," in *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*. Springer-Verlag, 2000, pp. 208–216.

[2] T. T. Nguyen, D. El Baz, P. Spiteri, G. Jourjon, and M. Chau, "High performance peer-to-peer distributed computing with application to obstacle problem," in *IPDPSW '10: IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, 2010, pp. 1 –8.

[3] D. El Baz and T. T. Nguyen, "A self-adaptive communication protocol with application to high performance peer to peer distributed computing," in *PDP '10: Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE Computer Society, 2010, pp. 327–333.

[4] M. Schordan and D. Quinlan, "A source-to-source architecture for user-defined optimizations," in *Modular Programming Languages*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2003, vol. 2789, pp. 214–223.

[5] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: towards a realistic model of parallel computation," *Proceedings of the 4th ACM SIGPLAN symposium on Principles and practice of parallel programming*, vol. 28, no. 7, pp. 1–12, 1993.

[6] D. Sundaram-Stukel and M. K. Vernon, "Predictive analysis of a wavefront application using LogGP," *7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, vol. 34, no. 8, pp. 141–150, 1999.

[7] A. J. C. van Gemund, "Symbolic performance modeling of parallel systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 2, pp. 154–165, 2003.

[8] R. H. Saavedra and A. J. Smith, "Analysis of benchmark characteristics and benchmark performance prediction," *ACM Transactions on Computer Systems*, vol. 14, no. 4, pp. 344–384, 1996.

[9] H. Casanova, A. Legrand, and M. Quinson, "SimGrid: A generic framework for large-scale distributed experiments," in *UKSIM '08: Proceedings of the 10th International Conference on Computer Modeling and Simulation*. IEEE Computer Society, 2008, pp. 126–131.

[10] S. Prakash and R. L. Bagrodia, "MPI-SIM: using parallel simulation to evaluate mpi programs," in *WSC '98: Proceedings of the 30th conference on Winter simulation*. IEEE Computer Society Press, 1998, pp. 467–474.

[11] V. S. Adve, R. Bagrodia, J. C. Browne, E. Deelman, A. Dube, E. N. Houstis, J. R. Rice, R. Sakellariou, D. J. Sundaram-Stukel, P. J. Teller, and M. K. Vernon, "PO-EMS: End-to-end performance design of large parallel adaptive computational systems," *IEEE Transactions on Software Engineering*, vol. 26, pp. 1027–1048, 2000.

[12] A. Snavely, N. Wolter, and L. Carrington, "Modeling application performance by convolving machine signatures with application profiles," in *WWC '01: IEEE International Workshop on Workload Characterization*. IEEE Computer Society, 2001, pp. 149–156.

[13] R. M. Badia, F. Escalé, E. Gabriel, J. Gimenez, R. Keller, J. Labarta, and M. S. Müller, "Performance prediction in a grid environment," in *Grid Computing*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004, vol. 2970, pp. 257–264.

[14] J. Zhai, W. Chen, and W. Zheng, "PHANTOM: predicting performance of parallel applications on large-scale parallel machines using a single node," in *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2010, pp. 305–314.

[15] M. Pettersson. Perfctr project webpage. [Online]. Available: http://user.it.uu.se/~mikpe/linux/perfctr/

[16] Perfmon project webpage. [Online]. Available: http://perfmon2.sourceforge.net/

[17] D. Zaparanuks, M. Jovic, and M. Hauswirth, "Accuracy of performance counter measurements," in *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 23–32.

[18] PAPI project website. [Online]. Available: http://icl.cs.utk.edu/papi/

[19] PAPI sc2008 handout (papi-2008.pdf). [Online]. Available: http://icl.cs.utk.edu/graphics/posters/files/

[20] S. A. Finney, "Real-time data collection in Linux: A case study," *Behavior Research Methods, Instruments, and Computers*, vol. 33, pp. 167–173, 2001.

[21] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka, "Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools," in *OON-SKI '94: The 2nd annual object-oriented numerics conference*, 1994, pp. 122–136.

[22] P. E. Livadas and S. Croll, "System dependence graphs based on parse trees and their use in software maintenance," *Information Sciences*, vol. 76, no. 3-4, pp. 197–232, 1994.

[23] J.-B. Ernst-Desmulier, J. Bourgeois, F. Spies, and J. Verbeke, "Adding new features in a peer-to-peer distributed computing framework," in *PDP '05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*. IEEE Computer Society, 2005, pp. 34–41.

[24] J.-B. Ernst-Desmulier, J. Bourgeois, and F. Spies, "P2pperf: a framework for simulating and optimizing peer-to-peer-distributed computing applications," *Concurrency and Computation: Practice & Experience*, vol. 20, no. 6, pp. 693–712, 2008.

[25] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS parallel benchmarks—summary and preliminary results," in *SC '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. ACM, 1991, pp. 158–165.

[26] NAS parallel benchmark website. [Online]. Available: http://www.nas.nasa.gov/Resources/Software/npb.html

[27] Available GCC optimization levels. [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[28] D. Skinner and W. Kramer, "Understanding the causes of performance variability in HPC workloads," *IEEE Workload Characterization Symposium*, vol. 0, pp. 137–149, 2005.

[29] G. Marin and J. Mellor-Crummey, "Cross-architecture performance predictions for scientific applications using parameterized models," in *SIGMETRICS '04/Performance '04: Proceedings of the joint international Conference on Measurement and Modeling of Computer Systems*. ACM, 2004, pp. 2–13.

[30] G. Marin, "Application insight through performance modeling," in *IPCCC '07: Proceedings of the Performance, Computing, and Communications Conference*. IEEE Computer Society, 2007.

[31] ANR CIP project web page. [Online]. Available: http://spiderman-2.laas.fr/CIS-CIP