A RECORD&REPLAY MECHANISM USING PROGRAMMABLE NETWORK INTERFACE CARDS

Dieter Kranzlmüller
GUP (Institute of Graphics and Parallel Processing)
Joh. Kepler University Linz
Altenbergerstr. 69, A-4040 Linz, Austria
kranzlmueller@gup.jku.at

Laurent Lefèvre
INRIA /LIP (UMR CNRS, INRIA, ENS, UCB)
Ecole Normale Supérieure de Lyon
46 allée d'Italie, 69364 Lyon Cedex 07, France
laurent.lefevre@inria.fr

ABSTRACT
Nondeterministic program behavior leads to different results in successive program executions, even if the same input data is provided. For this reason, re-executions of a program (as needed during cyclic debugging) are only possible, if certain precautions are taken. The most common solution is provided by record&replay mechanisms, where an initial record phase is used to extract characteristic behavioral data, which is afterwards used to control equivalent executions during subsequent replay phases. With the novel record&replay mechanism on Myrinet network interface cards (NIC)[1], program perturbations during the record phase are avoided by performing the initial monitoring activities directly on the NIC. This approach ensures, that the CPU of the computing nodes is not affected by the monitoring activities, while subsequent re-executions can still be controlled with the data collected on the NICs.

KEY WORDS
nondeterminism, program analysis, record&replay, monitoring.

1 Introduction

Software development for parallel and distributed systems introduces a series of challenges for the programmers, who have to cope with multiple, concurrently executing and communicating tasks and the associated increased complexity of the development process and the resulting code. Among others, one of these challenges stems from the possibility of nondeterministic behavior, which is quite common in parallel and distributed code.

Nondeterministic program behavior occurs if two executions of a given program may yield different results, even if the same input data is provided. Thus, deviations in results may be observed, although the input and the program code remains the same. The reasons for this behavior are nondeterministic choices in the code, which may be determined by scheduling decisions of the processor or the operating system, cache contents, cache conflicts, and memory access patterns, network throughput and network conflicts, or nondeterminism on the interconnection network.

The consequences of nondeterminism for the program developer are described with three different effects:

- **Irreproducibility problem** [1]: A particular program execution, even if observed by the user, cannot be repeated at will, since the program may never exhibit the same behavior. This prohibits the application of cyclic debugging techniques, where a program is executed over and over again to locate the origin of an observed program failure or incorrect results.
- **Completeness problem** [2]: Some errors may occur only sporadically or may never be observed, since certain situations in nondeterministic programs may seldom or never take place. Testing all possible executions of a particular program may be impossible, even for one given set of input data, no matter how many executions are actually performed.
- **Probe effect** [3]: The observation of a program may actually result in modifications of its behavior. Due to the delays introduced by the monitoring code and the memory required by the monitor, the observed program behavior may be different compared to an execution without a monitoring tool attached.

The approach described in this paper focuses on parallel programs using a communication library compliant to the Message Passing Interface standard MPI [4]. In such

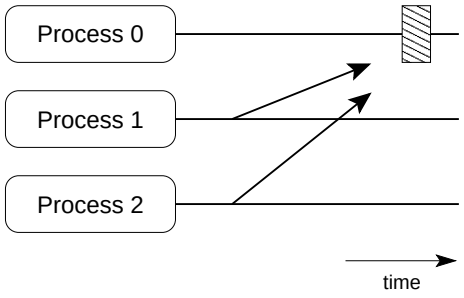

Figure 1. Racing messages in a simple parallel program

[1]This work is partially supported by the French "Programme d'Actions Integrees Amadeus" funded by the French Ministry of Foreign Affairs and the Austrian Exchange Service (ÖAD), WTZ Program Amadeus under contract no. 13/2002

programs, nondeterminism is introduced e.g. by utilizing the `MPI_ANY_SOURCE` parameter at wild card receives, which means that the calling process receives the next message from an arbitrary sender process. A simplified example of such a situation is given in Figure 1. Two processes, process 1 and process 2, are each sending a message to process 0. Process 0 specifies a wild card receive, thus allowing each of the two messages to be accepted.

Obviously, it may be possible that different messages are received during subsequent executions of the program, leading to the results and consequences described above. For example, in Figure 1, if the message from process 1 is received before the message from process 2, process 0 may compute different results than if the messages arrive in reverse order.

The wild card receive represents a so-called race condition, and the messages are usually called racing messages. While wild card receives are only one source of nondeterministic behavior, we believe that our principal approach may be applicable to other kinds of race conditions as well. (Another example is the access to shared variables in shared memory systems, where the access order may determine the contents of the shared variable.)

A main characteristic of our approach is to perform the program monitoring on the network interface cards (NICs) [5]. The optimization of the Record and Replay mechanism to fit onto the NIC required some fundamental analysis of the communication mechanism, with special attention to the overtaking mechanisms This approach should minimize the probability of the probe effect, while still guaranteeing reproducibility of a program's execution. (Solutions to the completeness problem are possible with our approach using event manipulation techniques as described in [6], but are omitted in this paper due to space constraints.)

The paper is organized as follows: Section 2 describes related solutions in this area, including their drawbacks compared to our approach, which is introduced in more detail in Section 3. Details about the actual implementation of our approach as well as some results are given in Section 4, before conclusions and an outlook on future work summarizes the paper.

2 Related work

While the problem of nondeterminism in parallel and distributed programs occurred with the first parallel machines several decades ago, there still exists no optimal solution today for every possible situation. However, all of the available solutions can be divided into the following two distinct categories:

- Controlled execution techniques
- Record&replay techniques

The former group represents all those approaches, where the execution of the target program is controlled by an external entity, which may either be the human user or some kind of algorithm or intelligent code. To enforce this method of controlled execution, code is included in the program's source to override its original behavior. Instead, whenever places of nondeterministic behavior are executed by the target program, the actual outcome of the race condition is determined by the external entity.

The approaches by [7] and [8] offer possibilities to specify the intended communication behavior of a program: Whenever a race condition would occur in the original program, the outcome of the race condition is defined by a rule set as enforced by the controlled execution mechanism.

The macro step approach as described in [9] replaces the specification with a graphical user interface, where the human user is able to control the program's execution at race conditions during run time. Whenever a process arrives at a nondeterministic event, the user has to decide which action should be taken. (This approach is well-suited for debugging purposes, but may be a bit limited for real-world applications.)

While each of the controlled execution techniques represents a possible solution to both, the irreproducibility effect and the completeness problem, their high amount of overhead represents a substantial drawback with regards to the probe effect. This characteristic is improved with record&replay techniques, which actually represent the majority of solutions to the nondeterministic problems described above.

Within record&replay approaches, the execution of the target program is distinguished in two phases. During the initial record phase, the execution of a program is (passively) observed, obtaining only minimal information about the nondeterministic choices taken. The resulting trace data is then used during multiple, subsequent replay phases to enforce the same program behavior as previously observed. By tracing only a minimal amount of information during the initial record phase, the perturbation of the program's execution and consequently the probe effect is expected to be less critical [10]. The replayed execution may then be delayed as much as desired to collect additional data about a program's execution, since the logical execution order of the program is controlled with the initial trace data. Implementations of this approach are described in [11] and [12].

One of the first implementations of record&replay has been proposed with Instant Replay [13], where the relative order of events in a parallel system is traced and subsequently enforced during replayed executions. Instant Replay has been the starting point for a series of other tools, such as IVD [14], PDT [15], and DDB [16]. However, most of these solutions focus only on the irreproducibility effect and the probe effect. With event manipulation as described in [6], record&replay can also be applied to address the completeness problem.

The record&replay technique proposed in this paper is comparable to other record&replay techniques and even

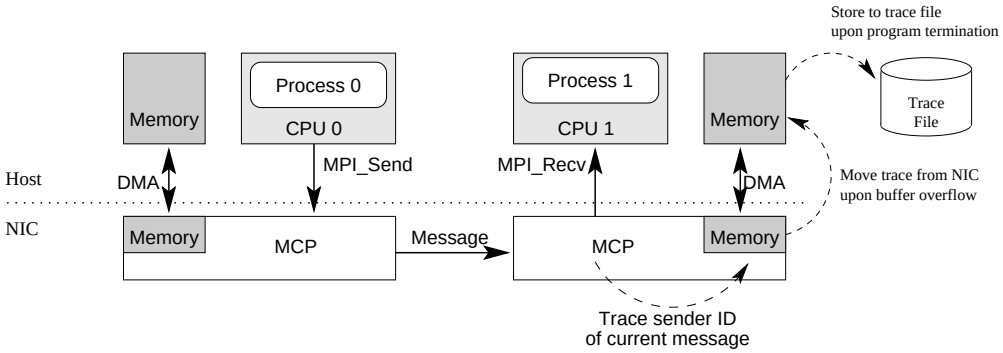

Figure 2. Operational scheme of the record phase using the Myrinet NICs

permits event manipulation, but introduces even less monitoring overhead during the initial record phase. The idea is to use a hybrid approach through minimal invasive instrumentation and exploitation of additional hardware. By offloading the monitoring code onto additional hardware units, the CPU is not affected by the monitoring operations compared to the original program.

Well-known hardware-based approaches for program monitoring include the exploitation of hardware counters, such as the Performance Counter Library PCL [17] and the Performance Application Programming Interface PAPI [18] or dedicated hardware monitors such as ZM4 [19] and the SMiLE monitor [20].

The advantages of each of these solutions are that they induce only relatively little intrusion on the observed program. The biggest drawback of these solutions is that they are usually highly system dependent, including the hardware environment (e.g. CPU type) and the operating system. In addition, it is much more difficult or even impossible to extract high-level information about the program's execution, since the sensors attached to the code are operating on a rather low-level. The latter is not very critical to our approach, since only event ordering information needs to be provided during the initial record phase.

3 Overview of approach

The record&replay technique described in this paper focuses on programmable network interface cards (NIC). Instead of performing the monitoring functionality on the host CPU of the running processes, the initial record phase is offloaded to the NICs. Of course, this requires that modification to the code executed on the NIC are permitted. Among the vendors providing programmable NIC, we have chosen Myrinet NICs [21] as utilized in many high performance clusters around the world, including our own clusters at ENS Lyon [22] and GUP Linz[2].

For this project, we used Myricom's M3F-PCIXD-2 cards, which use fiber optics for interconnecting the nodes with the Myrinet switch. The cards are equipped with a 200 MHz LANai 9.2 RISC processor, which supports a limited set of efficiently implemented instructions. Additionally, 2 MBytes of local memory and a PCI DMA bridge for communication with the host CPU are available on the NIC.

Exchanging data between host CPU and Myrinet card is achieved with one of two possibilities: Programmed Input/Output (PIO) or Direct Memory Access (DMA). PIO offers dedicated commands to access memory locations and to extract the status of the NIC. DMA allows to perform the transfer between the host CPU and the NIC CPU independently from the host CPUs operation. Whenever a DMA operation is completed, the host and the NIC are informed via a dedicated interrupt.

The software on the Myrinet NICs is called GM. It consists of a dedicated software library, a kernel module, and the Myricom Control Program MCP. More advanced communication libraries, such as implementations of the Message Passing Interface standard MPI [4], are implemented on top of the communication library.

With this system architecture available, our record&replay mechanism is deployed as follows:

During the initialization step, the Myrinet NIC is prepared to perform the initial record phase. This is achieved by modifying the MCP code executed on the NIC. With the NIC code modifications in place, the program is executed by the user (without any additional user intervention). While the program executes, the NIC generates the event ordering data and stores it to temporary memory on the NIC. Whenever this memory is filled up or immediately before the program terminates, the data produced by the NIC is moved from the NIC to the host memory and afterwards stored to a trace file. With this data available, arbitrary numbers of re-executions can be initiated, which will then use the information stored in the trace files to control the program's execution.

Please note, only the initial execution requires modifications to the NIC code. The replayed executions are performed solely by the CPU, and do not require any service (apart from the regular communication services) from the NIC.

[2] `http://www.gup.uni-linz.ac.at/cluster`

4 Actual implementation and results

The initialization required by our record&replay technique requires the preparation of the Myrinet NIC to collect the necessary event order information. This is achieved by performing the following steps:

(1) A modified MCP is loaded onto the NIC.

(2) The MPI Program is instrumented by including a modified MPI header file.

(3) The program is compiled and linked with the modified MPI library.

The most important part is the execution of the modified MCP to perform the monitoring activities on the NIC. While this may seem adventurous for the inexperienced user, it was relatively straight forward using the programming environment provided by Myricom. On contrary, it is much more difficult to design the MCP's operation such that the amount of overhead on the NIC is as minimal invasive as possible. In our record&replay approach, the MPC performs the following steps:

(4) During the initialization of the MPI program, the buffer memory for the monitoring data is reserved. The amount of buffer memory required by the monitoring operations should be as small as possible to fit into the 2 MB limitation of the NIC and to avoid influencing the NICs regular operation.

(5) During the program's execution, the buffer is used to store the order of incoming messages. This operation does not affect the operation of the host CPU.

(6) Upon buffer overflow or program termination, the data is transferred from the NIC to the main memory.

(7) Upon program termination, the monitoring data is stored to a trace file.

The last step (7) can be rather time-consuming, depending on the amount of data generated by the MCP. However, since the regular computational tasks of the target program are already finished, it is no longer critical.

The most critical part of this scheme is step (5), which has to be carried out for every communication operation. In order to minimize the amount of work performed in this step, we applied the following optimization:

Based on the MPI Standard 1.1, Section 3.5, "Semantics of point-to-point communication", messages in MPI are non-overtaking. This means, that the delivery of two successive messages from one process to the other, if both messages match the same receive, will always arrive at the receiver in the same order, in which they have been submitted by the sender. Thus, it is sufficient to store the order of incoming messages at the receiver node, which must afterwards be guaranteed during the replay by the receiving process. If the same order of the racing messages at wild card receives can be enforced, equivalent program behavior will be observed [6].

Figure 2 contains a graphical sketch of the operation of our technique. Two MPI processes, process 0 and process 1, being executed on two different CPUs, CPU0 and CPU1, are shown. Process 0 sends a message using MPI_Send to process 1. The message transfer is initiated on the Myrinet NIC attached to CPU 0, which transfers the message data (using DMA) from the host memory to the destination NIC. At the destination, the Myrinet NIC traces the sender ID of the incoming message - which is afterwards used during the replay phase - and hands the message over to process 1 running on CPU 1, whenever process 1 calls MPI_Recv. The trace data is stored locally on the Myrinet NIC of CPU1. Whenever the trace buffer is filled or the program terminates, the trace data is moved to the CPUs memory using DMA transfer.

After program termination, the trace data of the observed program execution is moved to a trace file and thus available for subsequent replay phases. In contrast to the initial record phase, where the amount of overhead should be as small as possible, the overhead during the record phase is no longer critical, since the replayed execution will be managed based on the trace data. For this reason, the replay is implemented in the user code as executed by the target process instead of modifying the Myrinet MCP. In addition, this allows to implement much more monitoring functionality due to less critical time and space constraints.

With the trace data available, the program's execution is controlled during replay at every wild card receive operation. This is achieved by replacing every `MPI_ANY_SOURCE` parameter with the actual source process as observed during the initial record phase. By permitting only the first of the incoming messages from each process, the same racing messages will be taken by the receive during replay as during the record phase.

As with traditional record&replay techniques, the subsequent replay phases are used to extract more data and provide this data for program analysis to the user. Examples of the application of our approach are given in Figure 3 with three screenshots of different analysis activities within the DeWiz program analysis tool [23]. Each of these graphical displays shows data obtained during the replayed execution.

The top-most screenshot of Figure 3 shows a typical event graph, also known as space-time diagram, of the program's execution. Time is on the horizontal axis, while the processes are arranged vertically. The graph contains the relations between communicating processes. A similar display is shown on the bottom- left, where more performance relevant information, such as the states of the executing processes, are displayed. The display on the bottom right offers statistical data about characteristics of the program, which have also been sampled during the replay phase.

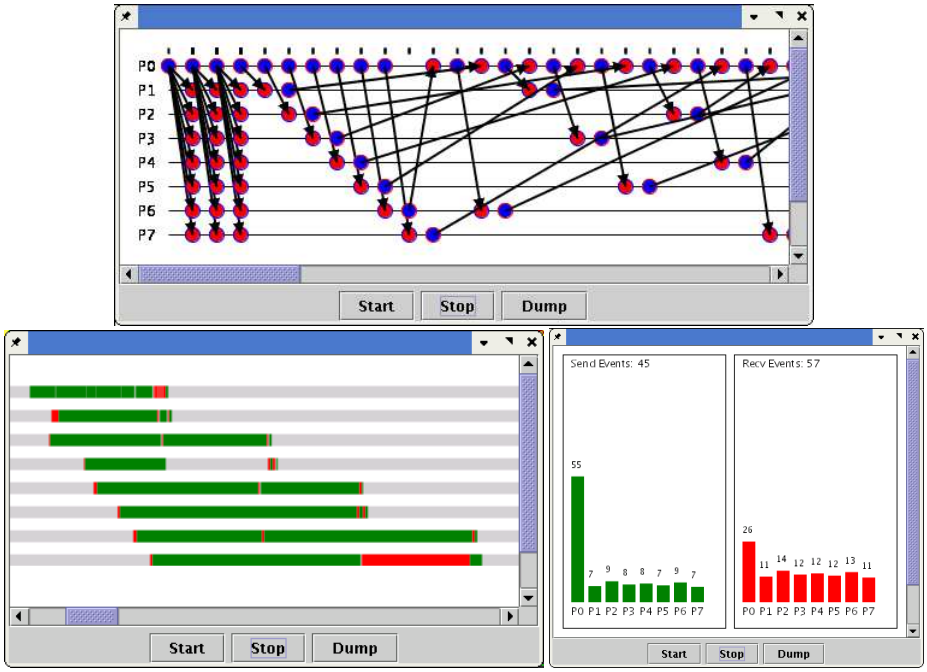

Figure 3. Event graph, time graph, and counter statistics of a program execution as obtained during subsequent record phases

5 Conclusions and future work

Record&replay techniques represent the usual approach to dealing with nondeterministic program behavior. Unfortunately, these approaches suffer from the sometimes large amount of monitoring overhead and the associated probe effect. To reduce the impact of the monitoring activity on the running program, we implemented a dedicated record&replay approach executed on Myrinet programmable network interface cards.

Major parts of the initial record phase are carried out independently by the code on the NIC without interfering with the target CPU. For this reason, the amount of perturbation (and thus the probe effect) induced onto the target program should be neglectable, and the traced data should represent observation data from a progam run without attaching a standard monitoring tool. The resulting trace data is used during subsequent execution of the same program to enforce an equivalent execution, thus eliminating the reproducibility effect.

Since the subsequent replay phases do not suffer from perturbations by the monitoring code, we are able to attach even highly invasive debugging tools, such as gdb. The delay introduced by gdb will only affect the execution time of the replayed execution, but does not perturb the order of event occurrences. Thus, the logical control flow of the program will be the same as during the initial execution.

The next step in this project is the evaluation of monitoring functionality in programmable (Myrinet) network switches (complementary to the NIC) and the integration of event manipulation features, where the order of events can artificially be modified by the debugging user. Through enforcing a different event ordering than previously observed, investigations of the consequences at nondeterministic choices can be investigated.

In addition, we plan to enable a transparent integration of the record&replay mechanism into the execution environment of the parallel system. Since the record phase does not interfere with the program's behavior, it may be turned on all the time, storing the trace data in some hidden memory on the users hard disk. Upon request, e.g. by providing a corresponding command line parameter, the user may thus be able to replay the most recent program execution, and thus investigate the recently observed behavior.

Acknowledgments We would like to express our gratitude to the French Ministry of Foreign Affairs and the Austrian Exchange Service (ÖAD) for supporting this project under the joint "Programme d'Actions Integrees Amadeus", contract no. 13/2002.

Several of our colleagues at GUP Linz and ENS Lyon, especially Martin Maurer, Eric Lemoine, and Reinhard Brandstädter, contributed to this work. We are most thankful for their input.

References

[1] D.F. Snelling and G.-R. Hoffmann, *A Comparative Study of Libraries for Parallel Processing*, Proc. Intl. Conf. on Vector and Parallel Processors, Computational Science III, Parallel Computing, Vol. 8 (1–3), pp. 255–266 (1988).

[2] H. Krawczyk and B. Wiszniewski, *Analysis and Testing of Distributed Software Applications*, Research Studies Press Ltd., Baldock, England (1998).

[3] J. Gait, *The Probe Effect in Concurrent Programs*, IEEE Software - Practise and Experience, Vol. 16, No. 3, pp. 225–233 (March 1986).

[4] Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard - Version 1.1*, `http://www.mcs.anl.gov/mpi/` (1995).

[5] M. Maurer. Fehlersuche in nichtdeterministischen parallelen Programmen mit Unterstützung von programmierbaren Netzwerkkarten. Diploma thesis, GUP Linz, Joh. Kepler University Linz, Austria, (June 2004) [in German].

[6] D. Kranzlmüller, *Event Graph Analysis for Debugging Massively Parallel Programs*, PhD thesis, GUP Linz, Joh. Kepler University Linz, Austria, `http://www.gup.uni-linz.ac.at/~dk/thesis` (September 2000).

[7] K.C. Tai, R.H. Carver, R.H., *Testing Distributed Programs*, in: Zomaya, A.Y., (Ed.), "Parallel and Distributed Computing Handbook", McGraw-Hill, New York, Chapter 33 (1996).

[8] M. Oberhuber, *Elimination of Nondeterminacy for Testing and Debugging Parallel Programs*, Proc. AADEBUG '95, 2nd International Workshop on Automated and Algorithmic Debugging, Saint Malo, France, pp. 315–316 (May 1995).

[9] P. Kacsuk, *Systematic Testing and Debugging of Parallel Programs by a Macrostep Debugger*, Proc. DAPSYS '98, 1998 Workshop on Distr. and Parallel Systems, Budapest, Hungary, pp. 105-112 (1998).

[10] A. Fagot and J. Chassin de Kergommeaux, *Systematic Assessment of the Overhead of Tracing Parallel Programs*, Proceedings EUROMICRO PDP '96, 4th EUROMICRO Workshop on Parallel and Distributed Processing, IEEE Computer Society Press, Braga, Portugal, pp. 179–186 (January 1996).

[11] E. Leu and A. Schiper, *Execution Replay: A Mechanism for Integrating a Visualization Tool with a Symbolic Debugger*, In: Y. Roberts, L. Bouge, M. Cosnard, D. Trystram, (Eds.), Proc. CONPAR 92 - VAPP V, Springer, LNCS, Vol. 634 (1992).

[12] F. Teodorescu and J. Chassin de Kergommeaux, *On Correcting the Intrusion of Tracing Non-deterministic Programs by Software*, Proc. EUROPAR'97, 3rd Intl. Euro-Par Conference, Springer, LNCS, Vol. 1300, Passau, Germany, pp. 94–101 (1997).

[13] T.J. LeBlanc, J.M. Mellor-Crummey, *Debugging Parallel Programs with Instant Replay*, IEEE Transactions on Computers, Vol. C-36, No. 4, pp. 471-481 (April 1987).

[14] M. Mackey, *Program Replay in PVM*, Technical Report, Concurrent Computing Department, Hewlett-Packard Laboratories (May 1993).

[15] C. Clemencon, J. Fritscher, R. Rhl, *Visualization, Execution Control and Replay of Massively Parallel Programs within Annai's Debugging Tool*, Proc. High Performance Computing Symposium, HPCS '95, Montreal, Canada, pp. 393-404 (July 1995).

[16] J. Sienkiewicz, T. Radhakrishnan, *DDB: A Distributed Debugger Based on Replay*, Journal of High Performance Computing, National University of Singapore, Vol. 4, No. 1, pp. 37–45 (December 1997).

[17] R. Berrendorf and H. Ziegler, *PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors*, Technical Report FZJ-ZAM-IB-9816, Research Center Jülich (October 1998).

[18] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, *A Portable Programming Interface for Performance Evaluation on Modern Processors*, The International Journal of High Performance Computing Applications, Vol. 14, No. 3, pp. 189–204 (Fall 2000).

[19] R. Klar, P. Dauphin, F. Hartleb, R. Hofmann, B. Mohr, A. Quick, and M. Siegle, *Messung und Modellierung paralleler und verteilter Rechensysteme*, B.G. Teubner, Stuttgart, Germany (1995) [in German].

[20] W. Karl, M. Schulz, and J. Trinitis *Multilayer Online-Monitoring for Hybrid DSM systems on top of PC clusters with a SMiLE*, In: Proceedings of the 11th Intl. Conference on Modeling Techniques and Tools for Computer Performance Evaluation, Springer, LNCS, Vol. 1786, Chicago, IL, USA (March 2000).

[21] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic and W. Su *Myrinet: a gigabit per second local area network* IEEE-Micro, 15(1) (February 1995).

[22] E. Lemoine, C. Pham and L. Lefèvre *Packet Classification in the NIC for Improved SMP-based Internet Servers"* IEEE Proceedings of the International Conference on Networking (ICN 2004), Guadeloupe, French Caribbean (Feb. 2004).

[23] D. Kranzlmüller, Michael Scarpa, Jens Volkert, *DeWiz - A Modular Tool Architecture for Parallel Program Analysis*, Proc. Euro-Par 2003, 9th International Euro-Par Conference, Springer Verlag, Lecture Notes in Computer Science, Vol. 2790, Klagenfurt, Austria, pp. 74-80 (August 2003).