Incremental Monitoring on Programmable Network Interface Cards

Extended Abstract

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`

Dieter Kranzlmüller, Martin Maurer
GUP, Joh. Kepler University Linz
Altenbergerstr. 69, A-4040 Linz, Austria
`kranzlmueller@gup.jku.at`

Abstract
Monitoring of program behavior is a basic necessity for performance tuning and debugging. A major problem is the overhead associated with the monitoring tasks, which influences the program's behavior and affects the observation results. At the same time, a certain amount of data is needed for the program analysis activities. For this reason, monitoring approaches have to balance the amount of overhead with the amount of extracted information. The proposed incremental monitoring approach provides a solution implemented to run on programmable network interface cards (Myrinet), which perturbs the program's execution as little as possible[1]. *The resulting data is then used to control the program during subsequent executions, while at the same time incrementally increasing the amount of monitoring. With this multistep approach, any amount of trace data can be extracted without substantial intrusions onto the program's execution.*

Keywords: monitoring, program analysis, perturbation, programmable network interface cards

1 Introduction

Program analysis is an important task to achieve and guarantee a certain level of software quality, which is determined by characteristics such as efficiency and reliability. The corresponding tasks are denoted as performance tuning and error debugging, respectively.

The input for these kinds of program analysis activities is usually state data about the program's behavior during runtime. This data is acquired at program execution with so-called monitoring activities, which are performed additionally to the program's original behavior. Obviously, this additional code introduces perturbations to the observed program, which, in the worst case, may even invalidate the observation results. For this reason, software tool developers implementing monitoring tools try to minimize the overhead generated during program observation in order to avoid the so-called *probe effect* [4], which defines the intrusion of observation onto the target.

A solution for reducing the effects of the monitor overhead while providing a flexible enough system to do arbitrary tracing is presented in this paper. The general idea is to perform monitoring in several consecutive executions, where each step increases the amount of extracted observation data. In fact, the first execution step delivers only the minimal information required to perform an equivalent re-execution during follow-up steps.

The monitoring of only minimal information generates the smallest possible overhead, while at the same time guarantees that re-executions of the program will exhibit the same program behavior. Therefore, it is possible to subsequently increase the amount of monitoring while still obtaining the same behavior as during the initial step.

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

This paper is organized as follows: Section 2 provides an overview of monitoring strategies and how related approaches address the monitor overhead problem. Afterwards, the idea of our approach is briefly introduced with some details about its implementation on Myrinet network interface cards. Conclusions and an outlook on future work in this project summarize the paper.

2 Monitoring Approaches

The monitoring overhead and the related probe effect manifest themselves in two possible dimensions:

- *Time*: The functionality introduced through instrumentation delays the occurrence time of events during the execution of the program. Compared to the original program, the monitored code performs additional activities, which lead to increased runtime.

- *Space*: The monitor itself requires a certain amount of memory for storing intermediate results and trace data. Depending on the amount of information required for the analysis activities and the chosen interval for event buffering, monitoring may require a substantial amount of space which may otherwise be used by the target application.

As a *second order perturbation*, the delay introduced through monitoring may also modify the actual behavior of the program. Due to the differences in occurrence times, relations between events and thus the event ordering may be substantially different from the un-monitored program run [7]. This is especially important for nondeterministic programs, where the actual behavior of the program depends on the relative order of events. In parallel and distributed programs, this situation occurs whenever race conditions are included in the code. Different timings of events at race conditions may yield different results, even if the same input data is provided.

A typical example for race conditions in parallel systems is the access to shared variables by two (or more) processes. With the message passing paradigm, race conditions occur whenever two (or more) messages race towards a receive event, which permits either of these messages to be accepted at the receiving process. Depending on the process that accesses the shared memory first or on the (racing) message that arrives at the receiving process first, different program behavior may be observed [7].

In order to overcome these probe effect problems, most of todays tools address the monitor overhead with either or both of the following two characteristic solutions:

- *Minimization of monitor overhead through minimal invasive instrumentation*:
  The idea of these approaches is to introduce only as little code as possible into the observed program, both on the number of instrumented code points as well as on the amount of instrumentation per code point.

- *Minimization of monitor overhead through exploitation of additional hardware*:
  The primary idea is to offload the monitoring activities onto additional hardware components, such that the CPU is not affected from additional operations compared to the original program.

The idea of tracing only the minimum amount of information required to allow re-execution of programs is described in [8] and [12]. By perturbing the initial execution only slightly, it is considered to be a sufficient approximation of an unperturbed execution [3]. Afterwards replayed executions are used to apply performance tracing and generate additional timing information.

Examples for the hardware oriented approaches are the exploitation of hardware counters, such as the Performance Counter Library PCL [1] and the Performance Application Programming Interface PAPI [2] or dedicated hardware monitors such as ZM4 [6] or the SMiLE monitor [5].

While each of these solutions are able to induce only a relatively low intrusion on the observed program, there are some limitations. Both approaches are highly dependent on the target system, including CPU architecture, operating system and similar things. In addition, the user has to balance the amount of monitoring concerning intrusion and amount of extracted data. While hardware monitoring is generally only little intrusive, extraction of high-level information is usually not possible.

3 Need of programmable network cards

Based on these observations above, we developed the monitoring approach described in this paper. A basic necessity of our approach is the availability of programmable network interface cards (NICs), which allow to modify the communication behavior by changing the code executed on the NIC.

Today, only few vendors provide possibilities to program their NIC products. In our case, we have chosen the Myrinet NICs [11] utilized in clusters at both, ENS Lyon [13, 9] and GUP Linz [2]. Myrinet is a well-known high-performance, packet-communication and switching technology that is widely used to interconnect clusters of workstations for achieving high performance through distributed computing.

The Myrinet cards used for this implementation are Myricom M3F-PCIXD-2 cards, which use fibre optics for interconnecting the nodes with the Myrinet switch. The processor on these cards is a 200 MHz LANai 9.2 RISC CPU, which supports only a limited set of efficiently implemented commands. In addition, the NIC contains among other components 2 MB of local memory and a PCI DMA bridge for communication with the host CPU.

In order to exchange data between the hosts CPU and the Myrinet card, two possibilities are available:

- Programmed Input/Output (PIO)
- Direct Memory Access (DMA)

With PIO, dedicated commands are used to read and write memory address spaces, and to extract the status of the network cards. With DMA, transfer of data between host memory and NIC memory is performed independently from the host CPU and the NIC CPU. After the transfer is completed, host and NIC are informed via interrupt.

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

The software used to run on Myrinet cards is called GM which consists of three parts:

- Software library
- Kernel module
- Myricom Control Program (MCP)

On top of this Myrinet software layer, more advanced communication libraries, such as the Message Passing Interface standard MPI, are implemented. In our case, the MPICH Implementation provided by Myricom has been applied.

4 Monitoring on Myrinet NICs

Based on the system architecture described above, our monitoring system is implemented as follows:

1. Preparation and instrumentation
2. Initial record phase
3. Repeated replay and analysis phases

The first step is required to prepare the Myrinet NIC for collecting monitoring information. In concrete, the following tasks have to be performed:

- Loading a modified MCP onto the network card.
- Instrumenting the MPI Program by including modified MPI header file.
- Compiling the program with the modified MPICH library.

The modified MCP includes the code to perform the monitoring on the NIC. This is the most important and delicate part, since debugging on the NIC is rather difficult. However, with the programming environment provided by Myricom, including the monitoring functionality on the NIC is not too difficult.

An important part of the monitoring tool design is to minimize the amount of overhead on the card by optimizing the tasks performed by the MCP. In concrete, the following steps are performed:

- Upon initialization of the MPI program, a certain amount of buffer memory is reserved to hold the monitoring data. (This has to be done with care since only 2 MB of memory are available on the card).

- During execution, the order of incoming messages is stored in the above reserved memory. Please note, that the tracing is performed concurrently to the MPI process, which received the message.

- If the buffer is full, the data from the NIC has to be transferred to the main memory. This transfer is also performed, when the computation on the host finishes.

- Upon finalization of the program, the data as provided by the NIC is stored to a tracefile which serves as the input for the follow-up replay phases. This time consuming task does not influence the program's behavior, since the working tasks of the host program are already concluded at this point in time.

The optimization of the MCP code is based on the fact that overtaking of messages in MPI is not permitted [3] Based on this rule it is sufficient to store the order of incoming messages at the receiver nodes. If the same order is enforced during subsequent replay phases, the same behavior will be observed.

Thus, with the trace data from the initial record phase, arbitrary numbers of equivalent executions of the program can be initiated. Please note, that the replay will be controlled by code added to the MPI process and not to the Myrinet MCP. The reason to perform this on the host process is twofold: On the one hand, enforcing the correct message order on the MCP would require substantial changes to the MCP code (compared to the changes of the MPI process). On the other hand, the overhead occurring on the MPI process is no longer an issue, since the order is controlled by the minimal perturbed initial record phase.

In order to perform equivalent execution, the receive operations of MPI have been modified. Instead of using the specified `MPI_ANY_SOURCE` as a filter for the incoming message, the actual source process as observed during the initial record phase is provided to the receive operation.

[3]See MPI Standard 1.1, Section 3.5 "Semantics of point-to-point communication".

In addition to enforcing the correct message order, the replay phases are also used to extract additional information. Depending on the users requirements, arbitrary data can be extracted without substantially affecting the program's behavior in terms of event ordering. In practice, the first subsequent replay phase is usually applied to extract a complete event graph of the program as shown in Figure 1.

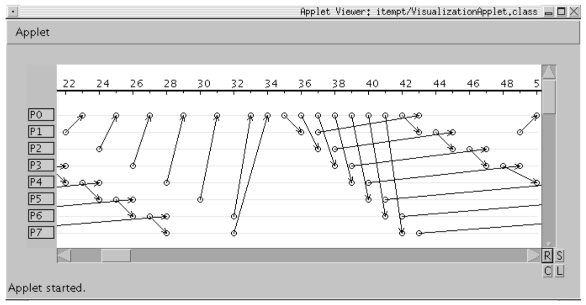

Figure 1: Event graph of program execution as obtained during follow-up replay phase

The display shown in Figure 1 represents a screenshot of the debugging tool DeWiz, with processes arranged vertically and time on the horizontal axis. Events (e.g. MPI operations) occurring during program execution are displayed as nodes in this space-time diagram, with arcs connecting corresponding pairs of communication events (e.g. send and receive operations).

While this display provides a good overview of the program's execution, more data is usually needed to detect the causes of incorrect behavior or performance bottlenecks. Such data is then extracted during additional replay phases. It is also possible to attach standard debuggers to the program and perform traditional cyclic debugging, as long as the execution of the program is controlled by the data of the initial execution.

5 Conclusions and Future Work

Observing a program's execution during runtime is a difficult task since the observation itself perturbs the observation target. For this reasons, monitoring of software is a delicate task which requires a good balance between the amount of extracted data and the intrusion on the observed program.

This paper introduced a monitoring approach where the initial execution of the program is traced on a programmable network interface cards. The advantage of this approach is its minimal intrusion on the observed program. In fact, with this approach it is even possible to perform monitoring without the users' knowledge, and thus permit re-execution of arbitrary programs e.g. by adding a flag to the `mpirun` command.

The work described above is only the first results of this project. We are also investigating possibilities to determine the usage of the networking infrastructures (in order to define the costs of networking in a cluster), or to monitor other additional networking characteristics. It may also be interesting to add Quality-of-Service functionality directly on the NIC with an approach similar to adding the monitoring code.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] 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 Modelling Techniques and Tools for Computer Performance Evaluation, Springer, LNCS, Vol. 1786, Chicago, IL, USA, March 2000.

[6] 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].

[7] 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.

[8] 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.

[9] 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

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

[11] 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), Feb. 1995

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

[13] L. Lefèvre and R. Westrelin *High performance communications libraries for windows 2000 : from a developer standpoint* In International conference on parallel and distributed processing techniques and aplications (PDPTA 2002), volume 4, pages 1665-1671, Las Vegas, Nevada, USA, june 2002.