Correction of Monitor Intrusion for Testing Nondeterministic MPI-Programs

D. Kranzlmüller[1], J. Chassin de Kergommeaux[2], and Ch. Schaubschläger[1]

[1] GUP Linz, Joh. Kepler University Linz, Austria
[kranzlmueller|schaubschlaeger]@gup.uni-linz.ac.at
[2] ID-IMAG, Grenoble,France
Jacques.Chassin-de-Kergommeaux@imag.fr

Abstract. Software analysis tools apply monitors to retrieve state information about program executions. Unfortunately such observation introduces the probe effect, which means that analysis data are influenced by the monitor's requirements to computing time and space. Additionally, nondeterministic parallel programs may yield different execution patterns due to alterations of event ordering at race conditions. In order to correct these intrusions, two activities are carried out. Firstly, the monitor overhead is removed by recalculating event occurrence times based on measurements of the occurred delay. Secondly, event manipulation and program replay is applied at places, where reordering of events has occurred. The resulting data describes the program's execution without monitoring.

1 Introduction

Performance analysis and error detection rely on observations of program state information that are generated by monitoring tools during execution. Since analysis facilities assume that valid data have been obtained, correctness of the monitor's results is crucial. A main problem is the probe effect [1] which means, that a program's execution is perturbed by observation. The critical issue is the size of the monitor overhead, which consists of the time spent in the instrumented code and the amount of memory needed by the monitor [5].

As a consequence, a primary goal of software monitoring tools must be to generate very small monitor overhead. Yet, even the smallest overhead will introduce an execution delay of the target program, which influences the occurrence time of events. Thus, the data describes events that occurred later in time compared to an execution without monitoring. An additional problem to the displacement of observed events is introduced with nondeterministic behavior. In that case, different executions may be observed in successive program runs, even if the same input data are provided. Furthermore, analysis data may not reflect the same execution as if monitoring were turned off, and errors may be hidden in program branches not taken during the monitored execution.

This paper describes an overhead correction algorithm for the topics mentioned above. It targets on MPI-programs [6], where nondeterminism is introduced at so-called wild card receives. Such receives do not specify the origin of a

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 154–158, 1999.
© Springer-Verlag Berlin Heidelberg 1999

message, but instead provide a special parameter to accept a message from any available process, and different messages may be accepted at a corresponding receive event [8]. In fact, if two or more messages are racing towards the receive, their order may be determined by different processor speeds, scheduling decisions, cache contents, or conflicts on the network. Therefore, it is unpredictable which message may be accepted and different executions of the same program may occur, even if the same input data are provided.

2 Correcting the Monitor Overhead

Perturbation of programs as described above occurs in two areas, time and space. We are concentrating on perturbation of monitor data in time, which can be classified into delay of event occurrence, and alteration of event ordering [5]. An example for both perturbations is displayed in figure 1. Two executions are visualized as space-time diagrams, with the time on the horizontal axis and the processes arranged vertically. The arcs between the process axes are communication events, with the tip of the arc pointing to the receiver.

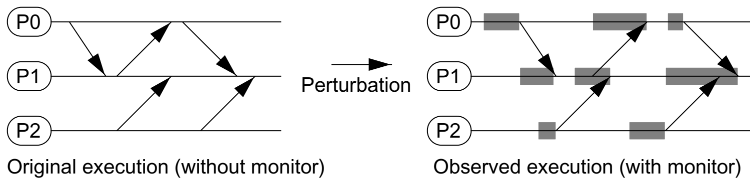

Fig. 1. Perturbation of program execution due to monitor overhead

A problem during correction of monitor traces is clock synchronization, which is required because most parallel and distributed systems do not have a global clock. In that case, times on different processors are not comparable, and artifacts, like messages being received before sent, may occur. For synchronization, we are experimenting with three approaches: an existing global clock (e.g. on the IBM SP/2), simple clock synchronization with Lamport's algorithm [3], and extensive clock synchronization with Maillet's method [4].

For the following sections we assume, that either a global clock exists or clock synchronization has been performed. Then the correction algorithm starts by computing the timings of each event. Other approaches ([5], [10]) have already discussed this issue, and our approach for this first step of intrusion correction is similar to [9], which was described for parallel systems based on threads.

In message passing systems the monitor overhead can be identified for non-blocking sends and blocking receives as follows: for the send operation the overhead is generated by monitor functionality before (t_{init}) and after (t_{exit}) the original communication functions (t_{comm}). In addition, a receive operation may be delayed by a certain amount of blocking ($t_{blocking}$). Therefore, the times for a generic send and receive function call are:

$$t_{send} = t_{init} + t_{comm} + t_{exit} \qquad t_{receive} = t_{init} + t_{blocking} + t_{comm} + t_{exit}$$

Hence, the goal of the correction algorithm is to remove the times t_{init} and t_{exit} from t_{send} and $t_{receive}$ for all events in the traces. (For simplification, we assume t_{init} and t_{exit} to be equal for both send and receive events.) A critical point is the blocking time, which can computed based on the assumption that communication time and monitor overhead is fixed. Please note, that communication time means the amount of time spent in the communication function without blocking, which is fixed, because the message is already available. Additionally the monitor overhead is fixed because the activities of the monitor are the same for similar events. These pre-defined portions of the formulae can be evaluated independently from the program, and we apply standard measurements of the SKaMPI benchmarks [7].

After collecting these measurements about the communication functions the correction algorithm is applied to each event in the traces. If an event is a send event, the new event time is computed as

$$t_{event} = t_{prec_event} - t_{prec_exit} + t_{send} - t_{init}$$

where t_{prec_event} and t_{prec_exit} denote the times of the preceding event and its exit time, respectively. For receive events the corrections also have to follow Lamport's rule, that messages can only be received after they have been transferred. Thus it is important to determine the transfer time (t_{trans}) for communication between processes, which can be evaluated before the correction algorithm with benchmarks, or based on the times stored in the traces. The latter is possible, since blocking is explicit and receives are waiting until messages are completely available. In that case, the transfer time can be computed based on the times of connected sends and receives as follows:

$$t_{trans} = (t_{recv} - t_{exit}) - (t_{send} - t_{exit})$$

The correction algorithm uses this transfer time to compute the new time for the receive events with the following formula:

$$t_{event} = \max(t_{prec_event} - t_{prec_exit} + t_{receive} - t_{init}, t_{send} + t_{transfer})$$

If these formulae are applied to all events in the tracefiles, the resulting trace will contain corrected times as if monitoring would have been turned off. The approach described so far is comparable to conservative approaches [9], which treat nondeterministic events as if they would occur as specified in the traces. However, this need not be the case for obvious reasons and the following section tries to give a solution for that problem.

3 Correcting Nondeterministic Events

The problem of conservative approaches is, that reordering of events is not considered. Yet, if several messages are racing towards a wild card receive, it is unpredictable, which one will succeed. The accepted message is always the message, that arrived first at the receive. Thus, it is important to analyze the reasons

that determine the message's arrival date. Besides network transfer time, contention, and scheduling decisions the most obvious prerequisite for the arrival time is the injection time at the sender. From a set of messages being sent to a common receiver, usually the first injected message will be the first to arrive.

However, since the event times of send and receive events are affected by the monitor overhead, it is also possible that the overhead influences the injection time of the messages. It could be, that the global order of send events is scrambled, leading to different arrival times of racing messages. In order to address this problem, the previous approach is extended: For each wild card receive, we determine the racing messages, which are all messages that are possibly accepted at that particular receive [8]. For each of these racing messages we compute the corrected time of the corresponding send event. This produces the following set of send times for n racing messages:

$$T_{send} = \{t_{send(1)}, t_{send(1)}, \ldots, t_{send(n)}\}$$

Based on these send times, we can determine the new order of message arrival by selecting the smallest value. This provides the new value for each wild card receive as:

$$t_{event} = \max(t_{prec_event} - t_{prec_exit} + t_{receive} - t_{init}, \min(T_{send} + t_{trans}))$$

At present, we also assume, that the transfer time is equal between all processes, and thus can be determined as above. For varying transfer times calculation has to be performed under consideration of the amount of data and the distance between communicating processes.

After detecting an exchange of event ordering we apply two activities, event manipulation and artificial replay. Both activities have been developed for the MAD environment and serve the purpose of race condition analysis [2]. With event manipulation we can define a different order of message arrival at wild card receives. This is done by specifying the first message from the set of racing messages that arrives at a particular point of exchange (POE). During replay the program is re-executed under the following constraints:

- before the POE: all messages arrive in the order defined by the traces of a previous program run.
- at the POE: the manipulated process waits until the user selected message arrives. Since this is one of the racing messages, its arrival is guaranteed.
- after the POE: the process is executed again without control because the following message transfer is unknown. It is unclear, how the exchanged message order affects the future of the process.

It is important to notice, that only one event manipulation can be evaluated per artificial replay, and only the first message to arrive can be specified. Due to the nondeterministic behavior it is not possible to make any assumptions about the execution after the POE. As a consequence, it may also be necessary to apply event manipulation iteratively, if more than one wild card receive has to be investigated.

4 Conclusion

Monitor intrusion correction is an important issue for tool developers, that want to provide correct and valid data for users. While existing correction algorithms apply conservative approaches, which do not investigate the probability of event reordering, our approach integrates event manipulation and artificial replay. Therefore it is possible to correct the times of each event and additionally the order of racing messages at wild card receives. Furthermore, by specifiying an interval for the arrival time of racing messages, our approach can easily be extended to produce more than one single execution, which is believed to be the program run without monitor, but at set of executions which certainly contains the actual program run.

Acknowledgments The work described in this paper was partially sponsored by the Austrian-French joint-cooperation project Amadee, Austrian contract number 24, French contract number 97005, 1997.

References

[1] J. Gait: A Probe Effect in Concurrent Programs. Software - Practise and Experience, Vol. 16(23), pp. 225–233 (March 1986).
[2] D. Kranzlmüller, S. Grabner, J. Volkert: Debugging with the MAD Environment. Parallel Computing, Vol. 23, No. 1-2, pp. 199–217 (Apr. 1997).
[3] L. Lamport: Time, Clocks, and the Ordering of Events in a Distributed System. Comm. ACM, pp. 558–565 (July 1978).
[4] E. Maillet, C. Tron: On Efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems. Journal of Parallel and Distributed Computing, Vol. 28, pp. 84–93 (July 1995).
[5] A. Malony, D. Reed: Models for performance perturbation analysis. Proc. Workshop Parallel Distributed Debugging, ACM SIGPLAN/SIGOPS and Office of Naval Research, (May 1991).
[6] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard - Version 1.1. http://www.mcs.anl.gov/mpi/ (June 1995).
[7] R. Reussner, P. Sanders, L. Prechelt, M. Müller: SKaMPI: A Detailed, Accurate MPI Benchmark. Proc. 5th European PVM/MPI Users' Group Meeting, Springer, Lecture Notes in Computer Science, Vol. 1497, Liverpool, UK, pp. 52–59 (Sept. 1998).
[8] M. Ronsse, D. Kranzlmüller: RoltMP - Replay of Lamport Timestamps for Message Passing Systems. Proc. 6th EUROMICRO Workshop on Parallel and Distributed Processing, University of Madrid, Spain, pp. 87–93, (Jan. 1998).
[9] F. Teodorescu, J. Chassin de Kergommeaux: On Correcting the Intrusion of Tracing Non-deterministic Programs by Software. Proc. EUROPAR'97 Parallel Processing, 3rd Intl. Euro-Par Conference, Springer, Lecture Notes in Computer Science, Vol. 1300, Passau, Germany, pp. 94–101 (Aug. 1997).
[10] W. Wu, R. Gupta, M. Spezialetti: Experimental Evaluation of On-line Techniques for Removing Monitoring Intrusion. Proc. of SPDT'98, SIGMETRICS Symposium on Parallel and Distributed Tools, Welches, Oregon, pp. 30–39, (Aug. 1998).