PARADIS: Analysis of Transaction-Based Applications in Distributed Environments

Christian Glasner[1], Edith Spiegl[1], and Jens Volkert[1,2]

[1] Research Studios Austria, Studio AdVISION,
Leopoldskronstr. 30, 5020 Salzburg, Austria
{christian.glasner, edith.spiegl}@researchstudio.at
http://www.researchstudio.at/advision.php

[2] GUP - Institute of Graphics and Parallel Processing,
Joh. Kepler University Linz, Altenbergerstr. 69, 4040 Linz, Austria
volkert@gup.uni-linz.ac.at
http://www.gup.uni-linz.ac.at

Abstract. The analysis of long running and distributed applications poses a great challenge to software developers. PARADIS is a novel tool that helps the programmer with accomplishing this task. It reconstructs the corresponding event graph from events collected during a program run and provides techniques to address the problems arising from large traces. It offers several modules for specific examinations like the analysis of applications which process transactions and due to its modular architecture it allows an easy extension of the functionality. We show the usefulness on the basis of a real-life application and discuss future enhancements.

1 Introduction

Program analysis and debugging are complex tasks in the area of software engineering. Using profilers is a common way to understand the dynamic behavior of an application, since they allow measuring the time being spent in particular functions and consequently help to identify possible performance bottlenecks. Unfortunately there are lots of reasons for bad runtime behavior that cannot be tracked by this technique because simple time measurement only shows which functions were responsible for the execution time but not the underlying causes.

This reason led to event-based debugging. The programmer or the tool automatically instruments an application at arbitrary points and at these points information about state changes happening during a program run are recorded. Each state record is associated with an event, that triggered the logging activity and a series of recorded events is called a trace. These traces can be analyzed either after termination of the inspected application (post-mortem) or simultaneously (on-line). In practice this event-based approach is limited by the number of events which have to be gathered. If a high number of events are recorded it not only slows down the execution of the program because of the logging activities (if not avoided by additional hardware), but it also complicates the later analysis. Possible problems arising are the time spent for trace processing and analysis, the consumption of disk storage and working memory and the complexity of the graphical representation.

To keep the number of events manageable one can instrument the program very economically. However, trends show that for utilizing the capacity of all available resources more and more applications get distributed across multiple threads, processes or even nodes. This leads to higher complexity of the applications and to new sources of error. Taking into consideration multiprocessor machines, clusters or grids combining hundreds or thousands of processors, the need for detailed information about the program execution to find out the reasons for an application's unintentional behavior seems obvious. Even if enough program information is gathered and sufficient computing power is provided one still has to face the task of filtering the data to get valuable results during the analysis and visualization.

In this paper we present PARADIS, a tool for the event-based analysis of distributed programs. In Section 2 we discuss related work, while Section 3 focuses on the modular architecture of the tool. Section 4 describes a real world example and finally an outlook on future work concludes the paper.

2 Related Work

There are several tools that address performance analysis of parallel programs. According to the programming paradigm of the underlying program, they log communication events like Send and Receive or resource assignment in shared memory systems. They deal with large trace files but do not offer support for the analysis of transaction-based applications, where events happening in the context of a single transaction belong semantically together.

Paradyn [1] for instance, which was developed at the University of Wisconsin, uses dynamic instrumentation to gather performance data such as CPU times, wallclock times, and relevant quantities for I/O, communication, and synchronization operations. As it allows dynamically setting and removing predefined probes during a program's execution the amount of generated trace data can be kept relatively small, even when monitoring over a long period.

Vampir [2] and Vampir NG [3] analyze trace files produced by the Vampirtrace library which has an API for defining events. Instrumentation is realized by linking the application with the library, after adding the calls to Vampirtrace to the source code. Similar to PARADIS they offer a hierarchical visualization, but as the application is targeted at clustered SMP nodes, the display provides three dedicated layers that represent cluster, nodes and processes [4].

Alike PARADIS DeWiz [5] utilizes the event graph model to represent a program run. By connecting a set of specialized DeWiz modules (analysis, visualization etc.), the user can build an event graph processing pipeline. The different modules communicate using TCP/IP which makes it possible to distribute a DeWiz system across several computers. This loose coupling of the modules contributes to the flexibility of DeWiz, but causes an administrative overhead and performance problems during on-line monitoring and analysis activities.

3 Our Approach

We consider PARADIS a tool for the breakdown of distributed programs, like on-line database systems with plenty of users, eBusiness, eCommerce or eGovernment systems, to name only a few. Nevertheless, the techniques might also prove very useful in the field of high performance computing, where message passing and shared memory computing is common. Our intent is to allow users to define events (eg. send and receive events when using MPI [6]), which are logged during a program run. These events form an event graph which provides the basis for our investigations. An event graph [7] is a directed graph, where the nodes are events and the edges stand for relations between the events.

To obtain a description of the application flow it is necessary to order the events in a causal manner. For this purpose we apply Lamport's [8] "happened-before relation". While the order for events occurring on one given object (node, process, thread,...) is given implicitly by the real-time timestamps of the events on the calling object with a given local time, the creation of relations between two dependent events on different objects can be more complicated if considering distributed systems without any global clock and where the local clocks are not synchronized and drifting. To get these events ordered we use a logical time-stamping mechanism (totally ordered logical clocks [9]).

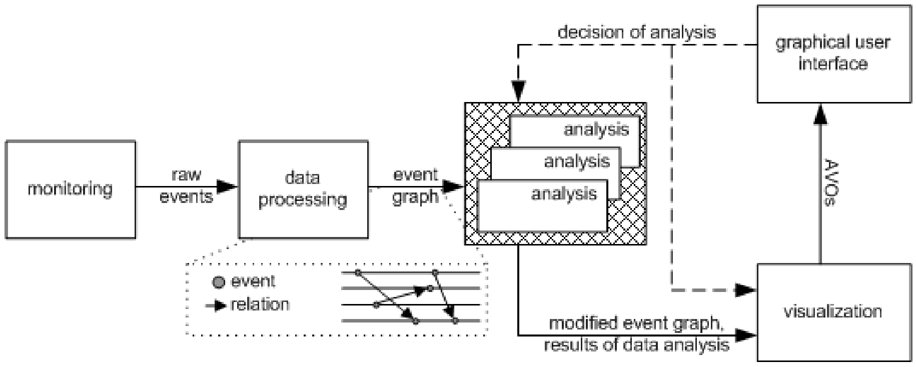

Fig. 1. Block diagram of the PARADIS system. An event graph is constructed from the recorded trace data and represents the basis for all further analysis tasks

Based on these conditions we create the event graph and offer various modules for textual and graphical representations and examinations. Figure 1 shows the logical units of PARADIS and their communication and is explained in the following sections.

3.1 Monitoring

To enable event tracing, first the program has to be instrumented. At the moment this is done statically by inserting calls to dedicated monitoring functions at points of interest in the source code of the inspected program. Being aware of the limitations due to the need of recompilation we are working on a dynamic instrumentation module using dyninstAPI [10]. It will allow the examination of already running programs without having to change the source code.

After the instrumentation each participating node in the computing environment executes a modified program which logs program state information. To comply with our system, for each event the values described in Table 1 (*Tr*) have to be recorded. Each event belongs to a particular category denoted by *type*. Possible categories are "communication", "critical section", "function call", "inspection of variable", and also user defined ones. Each *type* can have several subtypes. *Identification* is usually the id of the thread and the name or address of the node where the event has occurred. *Timestamp* is used to calculate durations (eg. blocking times) and statistics. For the creation of happened-before relations we utilize the logical Lamport time which is stored in the field *logicalclock*.

Table 1. Information stored for each event in the PARADIS monitoring environment. *Kind* indicates the traced (*Tr*) and the deduced (*De*) information

Information	Description	Kind
type	Category to which the event belongs	Tr
identification	Id of the thread and machine that produced the event	Tr
timestamp	Local physical time measured at the occurence of the event	Tr
logicalclock	Used to reconstruct the logical sequence in the event graph	Tr
data	Any other information for the later analysis	Tr/De
isHeadOf	Events that relate as source events to this event	De
isTailOf	Events to which this event does relate as a source event	De

The field *data* serves as storage for any further event-specific information. For instance to find all events referring to the same critical section one has to store the id of the critical section for which the events occurred. Since we use PARADIS for the analysis of distributed transaction-based applications we also store the id of the triggering transaction for each event. This information offers the possibility to trace the processing of a complete transaction in a distributed environment and to locate dependencies between single transactions.

3.2 Data Processing

The data processing unit commutes the raw events recorded during a monitored program run into the internal data model where it is negligible whether the analysis takes place post-mortem or on-line. For holding the relations, PARADIS offers two data structures (see Table 1) for each event. These two structures, *isHeadOf* and *isTailOf*, store the connected events according to the event graph definition and the application specific relations.

Figure 2 shows a more detailed image of the data processing unit. Each component of the program which takes part in the monitoring process is recording program information according to its instrumentation. These "raw events" are gathered by Trace Collectors (TC). It is necessary to offer different techniques for the collection to accommodate different node types. For instance nodes in a cluster with a shared file system may store their traces in dedicated directories from where the TC fetches them, while single clients which do not grant access to a TC will most probably prefer sending their traces in uncertain time intervals. PARADIS allows new nodes to start, and already participating nodes to discontinue partaking in the monitoring process at any time.

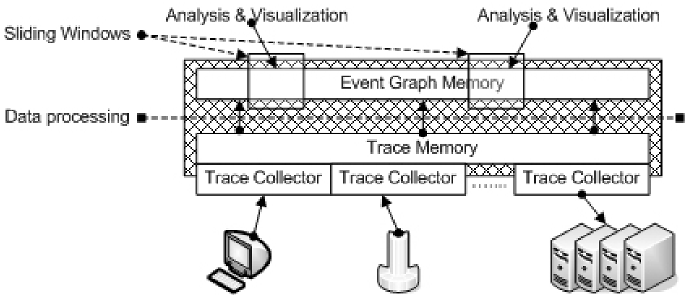

Fig. 2. Trace collecting and processing mechanism of PARADIS

Each raw event gets stored to the Event Graph Memory which offers a fast and efficient indexing and caching mechanism for the ongoing creation of relations, graph transformations and examinations. If more raw events are produced than the data processing unit can handle at a certain time, we use a selection mechanism, named "wave-front analysis", where those events are worked up first which occurred in neighbouring time segments. This leads to more significant results, as the event graph gets more detailed in the respective time frames and it is more likely to find related events. Additionally it is possible to filter specific events, for instance only those which cause blocking, to get intrinsic information even during high workloads.

3.3 Analysis

This unit works directly on the event graph which leads to a low main memory usage as there exists at most one single instance of any event at any time. The event graph memory module caches clusters of the most likely to be requested events ("sliding window"). By design an unlimited number of analysis modules may be active simultaneously while the data processing unit is still generating the graph. At the moment the analysis unit implements several filtering techniques, like for example by time of appearance or by blocking type to name only a few, but it is possible to extend the set with graph transformation, pattern matching or export functionality for transferring PARADIS data to other graph analysis or visualization tools like VisWiz [11] or GraphML [12].

As PARADIS additionally provides support for events which occurred in the context of transactions, it governs the identifications of all transactions, to accelerate access to the associated events. This is necessary because multiple nodes and processes may work on the same transaction in parallel which can lead to a spreading of events which belong semantically together in the event graph memory.

3.4 Visualization

For simply visualizing communication between a handful of processes a space-time diagram like the one adumbrated in Figure 1 may be sufficient, but for large distributed programs with a vast number of processes, events and relations it may not. As our target programs are transaction-based applications we propose an abstraction on transaction level. Figure 3a shows an overview of some transactions of an example application, where a single bar represents one single transaction. Dedicated events like the blocking time before obtaining a critical section or establishing communication are emphasized on the lower and upper half of the bar and with different colors. Thereby one gets an impression on how long each transaction took place and how much time was spent unproductive due to blocking. As a bar contains all events of one transaction regardless on which object they occurred it is necessary to provide a more detailed view for the analysis of the communication patterns between the different nodes and processes in the distributed environment (see Figure 3b).

To ease the detection of reasons of an unintended runtime behavior, PARADIS supports the user by visually emphasizing different events and blocking times. For figuring out the reasons for race conditions or communication patterns, transactions that relate directly (communication partners, previous owner of critical sections,...) to the transaction under analysis are displayed too. The abstraction from transaction layer to process layer and the detailed information about each single event contribute to the systematic breakdown of distributed applications. To maintain independence from the user interface and the output device the visualization unit creates AVOs [13] (Abstract Visualization Object). The graphical representation of these objects is chosen by the viewing component. At the moment our graphical user interface is running under Microsoft Windows and the AVOs are rendered with OpenGL.

4 Results

We have tested our tool with traces of a real-world document-flow and eBusiness system. Users submit tasks using a web interface and the system executes the tasks by distributing subtasks to different layers (client layer, frontend server, backend server). In order to fulfil one task at least one transaction is created. The

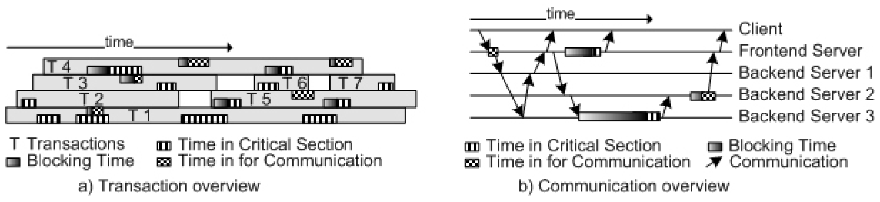

Fig. 3. The Transaction overview (a) shows all transactions in a given period, while the Communication overview (b) represents the inter-node communication of a single transaction

application was instrumented to produce events which comprehend the necessary information for the ensuing analysis. It was then started on a test environment, which is able to simulate real-world conditions. As this application is based on transactions, each recorded event can be associated with a specific transaction. One program run generates in one minute about 3000 transactions, 5.6 million critical section operations (request, enter, leave) and 76000 delegation operations (http and rpc requests, execution requests,...) which leads to a total of almost 6 million events per minute.

We have implemented several analysis and visualization modules like the one described in the previous section and especially the transaction overview proved very valuable for getting a first impression which transactions were unproductive over longer periods. With this information a goal-oriented search for the causes was possible, though we must admit that for really large numbers of transactions merging and filtering techniques become necessary to hide complexity.

5 Future Work

According to our tests the close coupling of the data processing and analysis units contributes causally to the celerity of PARADIS. Future work focuses on two aspects: the first is tuning our tool. We are optimizing the most time consuming tasks and want to reduce the amount of events, without losing important program information. One approach is the fractional outsourcing of the generation of the event graph, where relations between events which do not represent any type of inter-node communication are created on the nodes where they occurred. Another technique is the introduction of "meta-events" which encapsulate more than one event. This reduces the number of events and lessens the access to the event graph memory, but needs more effort in administration.

The second aspect is how to widen the field of application. We are developing new visualizations like highlighting positions with surpassing blocking times. Furthermore we are designing a mechanism for setting checkpoints in order to use record and replay techniques to repeat applications with non-deterministic behavior for debugging purpose.

Acknowledgements

This work was funded by ARC Seibersdorf research GmbH Austria and was originally initiated by an Austrian software company, which develops large distributed applications. The Institute of Graphics and Parallel Processing, at the Joh. Kepler University Linz/Austria was taking part as a consulting member.

References

1. B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, T. Newhall. The Paradyn Parallel Performance Measurement Tools. IEEE Computer 28, pp. 37-46, November 1995.
2. H. Brunst, H.-Ch. Hoppe, W. E. Nagel, M. Winkler. Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach. In V. N. Alexandrov, J. Dongarra, B. A. Juliano, R. S. Renner, C. J. K. Tan (Eds.): International Conference on Computational Science (ICCS), San Francisco, CA, USA, May 28-30, 2001, Proceedings, Part II. , Springer, LNCS 2074, pp. 751-760, 2001.
3. H. Brunst, W. E. Nagel, Allen D. Malony. A Distributed Performance Analysis Architecture for Clusters. IEEE International Conference on Cluster Computing, Cluster 2003, IEEE Computer Society, Hong Kong, China, pp. 73-81, December 2003.
4. S. Moore, D. Cronk, K. London, J. Dongarra. Review of Performance Analysis Tools for MPI Parallel Programs. In Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer, pp. 241-248, 2001.
5. C. Schaubschläger, D. Kranzlmüller, J. Volkert. Event-based Program Analysis with DeWiz. In M. Ronsse, K. De Bosschere (Eds.): Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003), Ghent, September 2003.
6. Message Passing Interface Forum, MPI: A Message-Passing Interface Standard - Version 1.1, `http://www.mcs.anl.gov/mpi/`, June 1995.
7. D. Kranzlmüller. Event graph analysis for debugging massively parallel programs. PhD Thesis, Institute for Graphics and Parallel Processing, Joh. Kepler University Linz, Austria, `http://www.gup.uni-linz.ac.at/~dk/thesis/`, September 2000.
8. L. Lamport. Time, clocks, and the ordering of events in a distributed system. In Communications of the ACM, Vol. 21, No. 7, pp. 558-565, July 1978.
9. C. Fidge. Fundamentals of Distributed System Observation. In IEEE Software, Volume 13, pp. 77-83, 1996.
10. B. Buck, J. Hollingsworth. An API for Runtime Code Patching. In Journal of High Performance Computing Applications, 14(4), pp. 317-329, 2000.
11. R. Kobler, Ch. Schaubschläger, B. Aichinger, D. Kranzlmüller, J. Volkert. Examples of Monitoring and Program Analysis Activities with DeWiz. In Proc. DAPSYS 2004, pp. 73-81, Budapest, Hungary, September 2004.
12. U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, M. S. Marshall. GraphML Progress Report: Structural Layer, Proposal.Proc. 9th Intl. Symp. Graph Drawing (GD '01), LNCS 2265, pp. 501-512, 2001.
13. R. B. Haber, D. A. McNabb. Visualization idioms: A conceptual model for scientific visualization systems. In G. Nielson, B. Shriver, L. J. Rosenblum: Visualization in Scientific Computing, pp. 74-93, IEEE Comp. Society Press, 1990.