A Concept of Portable Monitoring of Multithreaded Programs

Bartosz Baliś[1], Marian Bubak[1,2],
Włodzimierz Funika[1], and Roland Wismüller[3]

[1] Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland
[2] Academic Computer Centre – CYFRONET, Nawojki 11, 30-950 Kraków, Poland
[3] LRR-TUM – Technische Universität München, D-80290 München, Germany
{bubak, funika, balis}@uci.agh.edu.pl,
wismuell@in.tum.de
phone: (+48 12) 617 39 64, fax: (+48 12) 633 80 54 phone: (+49 89) 289-28243

Abstract. Multithreading is potentially a powerful way to achieve high performance in parallel programming. However, there are few tools that support multithreaded programs development. This papers presents an analysis which has led to a concept of building an autonomous monitoring system for multithreaded programs on top of which various tools can be based. Many problems specific to monitoring multithreaded programs are presented, as well as the ideas to solve them, specifically we focus on efficiency, scalability and portability of the monitoring system.
Keywords: Multithreading, monitoring, parallel tools, shared memory

1 Introduction

Multithreaded programming is potentially the most efficient way of parallel programming as threads are faster than processes both in creation, context switching and possibly also communication. Moreover, on many modern architectures which support shared memory, multithreading is a natural way of parallel programming, as threads naturally share address space and usually communicate via shared resources.

On the other hand, multithreading is inherently more difficult and error-prone than multiprocessing. This increases the need for specialized on-line tools supporting development of multithreaded programs. Unfortunately, multithreading support in existing tools is rather poor.

The goal of this paper is to present a concept for a portable, autonomous monitoring system for multithreaded programs focusing on SMP architectures. We present concepts to deal with many problems related to monitoring threads, specifically the monitoring system is designed to be efficient, scalable, and portable. The communication protocol between tools and the monitoring system is based on the OMIS specification [11]. OMIS is a standardized interface for communication between tools and monitoring systems, designed to support a variety of tool classes and extendible to provide new functionality when needed. We mostly focus on two types of tools: debuggers and performance analyzers, therefore we describe the functionality of the monitoring system to support these kinds of tools. However, the design is going to be generic, so that other types of tools can also be supported.

2 Analysis of Problems in Thread Monitoring

Multithreaded programming poses many problems not present in developing “ordinary” multiprocess parallel applications. Most of the problems derive from the fact that multiple threads share single address space which is not the case with processes. Access to shared data needs synchronization which can be tricky and error-prone. A thread can unexpectedly change a value of some variables which can lead to data corruption. Below we analyze a number of problems connected with monitoring threads.

2.1 Data races

An important class of problems is a *data race* [12]. A race is a situation in which two or more threads access the same piece of data concurrently and at least one access is a write one. Races lead to indeterministic behaviour since data being raced can be accessed in different sequence from execution to execution sometimes (but possibly not every time), probably causing an error. This makes it particularly hard to determine the cause of a program's incorrect behaviour. In addition, due to the *probe effect*, monitoring overhead can easily mask data races which occur in a program when it runs normally. Thus, data races can remain undetected for a long time.

2.2 Performance measurement

Threads communicate between themselves via shared resources, thus this communication is not explicit as in message passing. This makes it harder to measure performance of multithreaded applications (in comparison with a message passing application), since it is hard to measure the amount of communication passed between threads or time needed for this communication, which are essential pieces of information for certain performance measurements.

2.3 Lack of the OS support

Other problems with threads are related to operating system limitations. Some operating systems do not support directly thread monitoring. First of all, standard system interfaces like `ptrace` or the `/proc` file system [16] are supported by the kernel, and - if at all - allow only for dealing with kernel threads. Monitoring user-level threads usually requires a detailed knowledge of the system-specific thread library. There is no standardized interface for monitoring user-level threads. It is up to a particular vendor to provide e.g. a library with interface for monitoring user-level threads (e.g. pthreads). In addition, on some systems the mentioned interfaces are suited only to sequential debugging, since they provide only synchronous communication between debugger and application (i.e., a debugger pauses when the application is running and vice versa - when the application hits a breakpoint, it pauses and transfers the control to the debugger). A debugger for multithreaded programs should be able to independently control each target thread which requires asynchronous communication (neither the whole debugger nor the application process should be stopped, but only the specific thread).

2.4 Shared code image

Another problem stems from the fact that multiple threads often share a single code image. In such a case, if we instrument (e.g. insert a breakpoint) one of these threads, whenever any of the other threads reaches the instrumentation point (e.g. the breakpoint) we need a kind of filtering so as to ensure the proper actions are executed only for the really instrumented thread (e.g. only the proper thread is stopped by the breakpoint). The cost of such a filtering may be very high therefore it must be handled very efficiently.

2.5 Efficiency and scalability

Other issues which must be taken into consideration when implementing a monitoring system for multithreaded applications are efficiency and scalability of the monitoring system. A multithreaded application can potentially create hundreds of threads. For this reason, we must take care that monitoring system's design and implementation are scalable. Specifically, we should ensure that no part of the monitoring system must handle an excessive number of requests at a time. Moreover, as mentioned earlier, threads are more 'lightweight' than processes - their creation time and context switch overhead are much lower than for processes. Therefore, the overhead of monitoring threads should also be proportionally lower than for processes! Otherwise, the monitoring system's influence on the monitored application will be relatively very high.

3 State of the Art

Until today, there were numerous attempts to create tools for multithreaded applications. The bulk of them are debuggers, e.g., gdb, TotalView, kdb, Node-Prism and LPdbx. Wildebeest [1] is an example of a debugger based on gdb, which supports both kernel and user threads. However, it is strictly limited to HPUX platforms and implements only synchronous thread control. TotalView [7] is a commercial debugger which supports a variety of platforms and offers a rich set of debugging capabilities. It is well suited for multithreaded applications and provides even support for applications developed in OpenMP [6]. However, it does not allow for asynchronous thread control unless this feature is supported by the operating system. kdb [4] was designed to address the limitations of other debuggers, specifically it was designed to handle user-level threads and be able to control each target thread independently.

One of the rare efforts to address the performance analysis of multithreaded applications is the `Tmon` tool [9]. `Tmon` is a monitoring system combined with visualization module used to present *waiting graphs* for multithreaded applications.

To sum up, none of the mentioned efforts features a separate facility for monitoring multithreaded applications, designed to potentially support multiple classes of tools.

4 Monitoring Concepts

This section provides a more detailed description of our design concepts as well as a deeper insight into the general problems arising in debugging and performance analysis of multithreaded programs. Solutions to most of these issues are proposed.

4.1 Support for debugging

Some of the problems related to debugging multithreaded applications are outlined in Section 2. The two primary problems described there are the issue of the independent control of each target thread and the overhead caused by instrumenting a thread which shares its code image with several other threads. In case of debugging the second problem applies to setting and hitting breakpoints. These two problems can be addressed by placing part of monitoring system directly in the application. With this approach, this *application monitor* can handle events or perform actions on behalf of the *local monitor*, which in turn is the part of the monitoring system residing on each node of the target system. The local monitor thread, instead of establishing a synchronous connection via `ptrace` or the `/proc` filesystem, can delegate this task to the application monitor by means of an asynchronous communication (e.g. message passing). Thus, blocking the whole monitor process can be avoided. In a similar way the problem of instrumentation (breakpoints) can also be addressed. The applicability of hitting a breakpoint can be checked locally in the application monitor while the local monitor will only be notified if the breakpoint was hit by a proper thread. The approach described here was successfully used for breakpoints as described in [4] and [10].

It should be noted here that only some tasks can be delegated to the application monitor. Some of them are simply impossible to be handled (e.g. a process cannot stop itself) while some others may be intentionally left to be handled by a separate (local monitor) process (thread). The latter may be dictated by the safety reasons. The monitor code and data inside a user thread may be corrupted by bugs in the application. For this reason, it might be safer to leave some important debugging functions in the local monitor. Only these functions which most affect efficiency should be placed in the user application space.

4.2 Support for performance analysis

In most performance measurements applicable for message-passing applications the key role is played by the time spent on or the volume of communication between processes. In multithreaded programs, it is particularly hard to measure these characteristics, since there is no explicit communication between threads - all communication is done via shared resources. Moreover, this method of communication needs a frequent synchronization to ensure exclusive write access to shared data. This factor also affects performance as due to synchronization some threads may be waiting for others. This *waiting time* can be used as a basis of performance measurements related to multithreaded programs. In [9], an approach is presented to measure waiting time and create *waiting graphs*.

The performance analysis of multithreaded programs has to address the following issues:

- definition of performance measurements for multithreaded applications
- mechanisms for instrumentation of thread libraries and application code to gather performance data
- efficiency in gathering performance data

Definition of performance measurements

First we have to answer the question what we really need to measure, i.e. define the performance measurements for threads. In case of threads we are interested in at least three types of measurements - CPU load related to computations, amount of data passed between threads, and delays related to communication and synchronization, parallelizing operations or remote memory access. The measurements will be defined with regard to constraints, e.g. within code region like loop, function or location, e.g. within particular threads.

Instrumentation

The instrumentation is the main way to gather performance data. It allows for collecting information about interesting events. In case of multithreaded applications the user is interested in the information on accesses to shared objects and operations related to synchronization, i.e., locking/unlocking mutexes or condition waits. For the instrumentation, two primary techniques are considered: *binary wrapping* and *dynamic instrumentation*. The binary wrapping is a technique in which the functions to be instrumented are wrapped with additional code that performs event detection and execution of associated actions [5]. This method has a drawback that it requires relinking the application. Dynamic instrumentation is an advanced technique to patch the code of a running process, directly in the memory. This approach is supported by the DyninstAPI [3]. However, while the DyninstAPI successfully allows to instrument function calls, it does not solve the problem of efficient instrumentation of generic objects, e.g. data shared by multiple threads to detect accesses to this data. While software approaches exist to handle this kind of instrumentation for the purpose of debugging (*watchpoints* of *data breakpoints* [16, 14]) they are not suited for performance analysis, since either their overhead is too high, or they are limited to the observation of a very limited number of memory cells. Performance measurement on shared data can, however, be supported with minimal hardware extensions. A design of a hardware monitor suitable for this task is outlined in [8].

Efficient data gathering

The efficiency problems in gathering performance data derive from the fact that on one hand the data can be generated rapidly, on the other hand, it may be rarely accessed. If we decide to immediately report each event of interest to the monitoring system, the resulting overhead will probably be unacceptable due to frequent context switches and excessive communication. To reduce the number of context switches, we can store the data *locally* and send it to the monitoring system e.g. on an explicit demand. We can further reduce the communication by using *efficient data structures* to store the performance data. Instead of saving a raw trace of events, we can initially process the data and save the summarized information (e.g. only the number of events if this information is sufficient to perform the measurements).

We have a long experience in monitoring message-passing applications with an OMIS-compliant monitoring system OCM and OCM-based tools, among others a performance analyzer PATOP and a debugger DETOP. During the previous work with the OCM and PATOP [2] we used both local storing and specialized data structures to hold performance data, namely *counters* and *integrators*. The support of these features was enabled by placing parts of the monitoring system directly in the context of application processes. The described mechanisms proved to be very efficient. Full performance monitoring overhead was only about 4%.

4.3 Detecting data races

Data races are particularly hard to detect as mentioned in section 2. On the other hand, it is particularly important to detect this kind of bugs, as they can easily cause many other bugs. In [15] a two-phase approach for data race detection is presented. The program being analyzed is executed twice. During the first execution, information about synchronization operations is collected. The second execution is a *replay* of the first one. In this execution, all read and write accesses to global data are instrumented, causing a rather high, but unavoidable intrusion. The information collected during the first execution is used to force the second one to still be *equivalent* to the first one. Thus, the replay will possibly fail (i.e. the internal program flow will differ across the two executions) only if a race condition occurs. This solution avoids the probe-effect otherwise caused by the intrusion of collecting information about memory operations. This approach can be realized using an OMIS-compliant monitoring system. The first execution is already supported by OMIS as it basically consists in instrumenting synchronization operations. The replay operation can also be implemented as an additional service in OMIS.

4.4 Monitoring interface

An autonomous monitoring system must offer a suitable interface that is both flexible and powerful enough to handle very different kinds of tools. In our design, we use OMIS [11] for this purpose, because besides fulfilling these requirements, the existing specification already allows for threads and includes some basic services to inspect and control threads, although those services are currently not implemented in the OMIS-compliant monitoring system OCM. In addition, OMIS specifies a well-defined way to extend it with new objects and services, allowing to smoothly integrate shared data objects or synchronization objects, such as mutexes or condition variables, as well as the associated services. The new services will mainly monitor accesses to these objects, using the instrumentation techniques discussed in section 4.2.

Apart from OMIS, there are few other approaches defining standardized interfaces for monitoring systems. A notable project is DPCL [13], which is entirely based on dynamic instrumentation. DPCL is, however, not well suited for our purpose, because it does not take threads into consideration and defines a more low-level interface as compared to OMIS.

4.5 Design alternatives for a monitoring system

The monitoring system is planned to support multithreaded applications on shared memory multiprocessor architectures. The first question to be answered is what should be the architecture of the monitoring system. In case of the OCM, a monitoring system for clusters of workstations, this architecture is a collection of *local monitors*, one per node in the target system, with one central component, a so called NDU (*Node Distribution Unit*) responsible for accepting a request from a tool, splitting it into sub-requests for local monitors which execute the request and send the (partial) replies back to the NDU which in turn assembles them into a single reply for the tool.

For the SMPs, several architectures of the monitoring system are taken into consideration:[1]

1. Pool of local monitors, each assigned to a specified subset of nodes, and one central component on top of the local monitor to play the role of the NDU, called *service manager*.
2. One local monitor per SMP node and one service manager.
3. One local monitor per SMP node and one service manager per each monitored application.

[1] The term 'node' or 'SMP node' refers to a building block of an SMP system, possibly consisting of a couple of processors with local, shared memory, like in cc-NUMA systems.

4. Hybrid of some of the above.

The first two architectures resemble the centralized architecture of the OCM, while the 3rd approach introduces decentralization which should address possible scalability problems. On the other hand, the 2nd and 3rd approaches address the *locality* issue, i.e., each SMP node would have its own dedicated local monitor, while in the 1st approach a single local monitor may handle multiple SMP nodes which in turn reduces the total number of local monitors. Which of these architectures is the most appropriate one depends on the relative importance of the factors scalability, locality, and number of local monitors and cannot be decided a priori.

The second important question is whether the monitoring system should itself be multithreaded or not. At the moment, it seems that it is a natural solution to implement local monitors as threads of a single process (e.g. for the efficiency reasons - fast communication between monitor threads). It may still be necessary to have some parts of the monitoring system as separate processes e.g. to better handle asynchronous control of threads.

Another decision to take is whether there will be a single monitoring system in OS, or there will be one instance of it per application. The OCM follows the latter approach, which is easier to implement as the monitoring system has to deal only with a single application at a time and it runs as a user process instead of root so no concern about access control and permissions is necessary. This approach is also potentially better scalable (only one application to be handled). On the other hand, it introduces more monitoring processes overall in the system so its total overhead could be higher.

In the OCM, some parts of the monitoring system reside directly in the target application process space. These parts are responsible for handling certain events *locally* to reduce the overhead of monitoring. This particularly helps in gathering data for performance measurements as described in Section 4.2. In a monitoring system for SMPs, this solution should also be applied, especially because it can also help in debugging multithreaded applications (Section 4.1).

A generic architecture of the monitoring environment is shown in Fig. 1. In this figure, local monitors and the service manager are threads of a single Global Monitor process. Application consists of a collection of threads with an additional application monitor thread.

5 Conclusion

Monitoring systems for multithreaded programs are faced with a number of severe problems, concerning their efficiency, scalability and portability. In this paper, we have outlined the most important problems that need to be solved and have presented proper solution concepts, as well as design alternatives for the resulting monitoring system. In an ongoing project at Institute of Computer Science, AGH, these concepts will be evaluated in detail by fully implementing this monitoring system. In particular, the project addresses several issues which until now are not adequately solved by existing approaches:

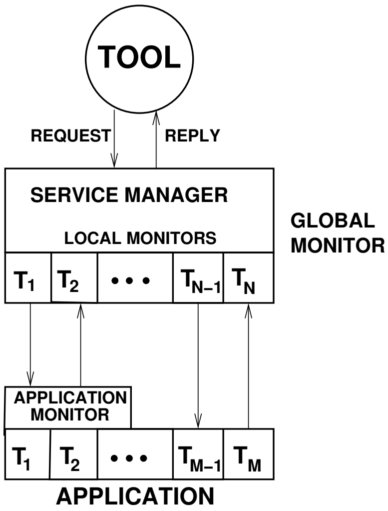

Fig. 1. Possible architecture of the monitoring environment

1. A universal autonomous monitoring facility for multithreaded applications will be implemented. None of current tools for multithreaded programming support features a separate monitoring system.
2. Both kernel and user threads will be supported. Few current products support user-level threads, even fewer do it in a portable way.
3. The problem of asynchronous thread control regardless of platform support will be addressed. Current approaches to solve this problem usually only work when the operating system supports independent thread control.
4. The efficiency and scalability in monitoring threads will be one of the primary concerns. This includes the concept of application monitors which perform certain tasks on behalf of the global monitor. Current projects using this approach do it to a limited extent (e.g. only to handle breakpoints).

References

1. S. S. Adayapalam. In Search of Yeti: Footprint Analysis with Wildebeest. In Mireille Ducassé, editor, *Proceedings of AADEBUG 2000, Fourth International Workshop on Automated Debugging*, Munich, Germany, August 2000. `http://www.irisa.fr/lande/ducasse/aadebug2000/proceedings/04srikanth.ps.gz`.
2. M. Bubak, W. Funika, B. Baliś, and R. Wismüller. On-line OCM-based Tool Support for Parallel Applications. In Y. Ch. Kwong, editor, *Annual Review of Scalable Computing*, volume 3, chapter 2, pages 32–62. World Scientific Publishing Co. and Singapore University Press, Singapore, 2001.
3. Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14(4):317–

329, Winter 2000.
citeseer.nj.nec.com/buck00api.html.
4. P. A. Buhr, M. Karsten, and J. Shih. KDB: A Multi-threaded Debugger for Multi-threaded Applications. In *Proc. of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 80–89, Philadelphia, Pennsylvania, USA, May 1996. ACM Press.
5. J. Cargille and B. P. Miller. Binary Wrapping: A Technique for Instrumenting Object Code. *ACM SIGPLAN Notices*, 27(6):17–18, June 1992.
6. J. Cownie and S. Moore. Portable OpenMP debugging with TotalView. In *Proceedings of EWOMP'2000, Second European Workshop on OpenMP*, 2000.
7. TotalView Multiprocess Debugger. WWW-Site of Etnus Inc., Framingham, MA, USA, 1999.
http://www.etnus.com/products/totalview/index.html.
8. Robert Hockauf, Jrgen Jeitner, Wolfgang Karl, Robert Lindhof, Martin Schulz, Vicente Gonzales, Enrique Sanquis, and Gloria Torralba. Design and Implementation Aspects for the SMiLE Hardware Monitor. In *Proceedings of SCI-Europe 2000*, pages 47–56, Munich, August 2000. SINTEF Electronics and Cybernetics.
http://wwwbode.cs.tum.edu/~schulzm/papers/2000-08-gt-scieuro.pdf.
9. M. Ji, E. W. Felten, and K. Li. Performance Measurements for Multithreaded Programs. In *Measurement and Modeling of Computer Systems*, pages 161–170, 1998.
citeseer.nj.nec.com/article/ji98performance.html.
10. P. B. Kessler. Fast breakpoints. Design and implementation. *ACM SIGPLAN Notices*, 25(6):78–84, June 1990.
http://www.acm.org:80/pubs/citations/proceedings/pldi/93542/p78-kessler/.
11. T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. *OMIS — On-line Monitoring Interface Specification (Version 2.0)*, volume 9 of *LRR-TUM Research Report Series*. Shaker-Verlag, Aachen, Germany, 1997. ISBN 3-8265-3035-7.
http://wwwbode.in.tum.de/~omis/OMIS/Version-2.0/version-2.0.ps.gz.
12. R. H. B. Netzer. What are Race Conditions? Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems*, 1:74–88, March 1992.
13. D. Pase. Dynamic Probe Class Library (DPCL): Tutorial and Reference Guide, Version 0.1. Technical report, IBM Corp., Poughkeepsie, NY, July 1998.
http://www.ptools.org/projects/dpcl/tutref.ps.
14. Paul E. Roberts. Implementation and Evaluation of Data Breakpoint Schemes in an Interactive Debugger. Master's thesis, Department of Computer Science, University of Utah, December 1996.
www.cs.utah.edu/flux/papers/perobert_thesis.ps.gz.
15. M. Ronnse and K. De Bosschere. Non-intrusive on-the-fly data race detection using execution replay. In M. Ducasse, editor, *Proceedings of AADEBUG 2000 Fourth International Workshop on Automated Debugging*, Munich, Germany, August 2000.
http://www.irisa.fr/lande/ducasse/aadebug2000/proceedings/20ronsse.ps.gz.
16. Jonathan B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. John Wiley & Sons, 1996.