Monitoring of Multithreaded Applications on SGI Computers

Bartosz Baliś[1,2], Marian Bubak[1,2], Włodzimierz Funika[1,2], Roland Wismüller[3,4], and Grzegorz Kaplita[1]

[1] Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland
[2] Academic Computer Center – CYFRONET, Nawojki 11, 30-950 Kraków, Poland
[3] LRR-TUM – Technische Universität München, D-80290 München, Germany
[4] Institute for Software Sciences, University of Vienna, A-1090 Wien, Austria
{balis,bubak,funika}@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
This paper presents a design of a monitoring infrastructure called TOM (Thread-enabled OMIS Monitor) on which various tools can be based. TOM's design is focused on asynchronous thread control, portability and support for a variety of kinds of tools. The platform for the first prototype implementation is cc-NUMA SGI Origin.
Keywords: monitoring, multithreading, OMIS, tools, SGI

1 Introduction and Motivation

The capability of multithreading is a standard for shared memory architectures. Threads are more efficient than processes and they have a lower overhead of communication, context switching and creation cost. Consequently, they are a natural choice for parallel programming, wherever appropriate. Although this approach has many advantages, there is a serious difficulty: multithreaded programming is difficult and poorly supported by tools. Available tools are limited and they usually do not provide all the required functionality. The development of multithreaded applications is difficult. We have to decide which language, library, platform and operating system we should choose. There are many different solutions, out of which one of the most popular is the POSIX threads standard, called pthreads. Pthreads define a standardized interface and behaviour for thread creation and management subroutines as well as related data types. Almost every operating system provides a user-level pthreads library, although the underlying implementation is system-dependent. The latter feature makes pthreads hard to debug – there is no standard interface for accessing pthreads internals, and as they are supported in a user-level library, system mechanisms such as ptrace or procfs are often not suitable. Besides debugging tools, application developers are interested in monitoring the inside working of applications in order to learn whether the application performs well, how large are the delays due to synchronization, etc. These features are offered by tools such as performance analyzers or visualizers.

Both debuggers and performance analyzers need monitoring services, which allow to manipulate or to get information about an application. For the reasons mentioned above, these services are usually integrated with tools and depend on many system-specific factors. For example, on the SGI Origin/IRIX platform most available tools supporting pthreaded applications are commercial and expensive. IRIX provides some standard mechanisms for debugging, but they usually support only processes, or IRIX-specific threads (LWPs) created by the sproc() system call. There exists a library for debugging pthreads, but it is specific to IRIX and proprietary – used only by tools delivered by SGI.

Currently, most of thread-enabled tools are debuggers, however they are usually limited in various senses. For example, Wildebeest [1] supports both kernel and user threads, but it is strictly limited to HPUX platforms and implements only synchronous thread control. TotalView [9] provides powerful capabilities; it is well suited for multithreaded applications, supports multiple platforms, and provides even support for applications developed in OpenMP [6]. However, it is commercial, and relies only on mechanisms provided by the operating systems. For example, it only allows for asynchronous thread control if this feature is supported by the operating system. One interesting debugger, which seems to overcome the above limitations, is kdb [4]. It seems to handle user-level threads and be able to deal with threads asynchronously. The `Tmon` tool [7] is a monitoring system combined with visualization module and is designed for performance analysis of multithreaded applications. None of the above mentioned efforts features a separate facility for monitoring multithreaded applications, designed to potentially support multiple classes of tools.

The monitoring infrastructure TOM (Thread-enabled OMIS Monitor), which is the subject of this paper, is designed to overcome at least some of the mentioned difficulties. TOM is supposed to be a separate monitoring infrastructure, started independently of the tools. Monitoring services of TOM are accessible via a standardized interface, OMIS (On-line Monitoring Interface Specification) [8]. TOM introduces some ideas which allow to monitor threads in a portable manner.

The rest of this paper is organized as follows. First we briefly overview some major problems in monitoring threads. Next, we outline the monitoring interface OMIS. Then we describe the goals for TOM and a proposed architecture for a thread-enabled monitoring system for a cc-NUMA architecture. Finally we present the design and functionality of the first prototype of TOM.

2 Thread Monitoring Problems

We intend to propose a robust architecture for a thread-enabled application monitoring system. Let us first have a brief look at some problems specific to monitoring multithreaded applications. We have presented a more detailed analysis of these problems in [2].

2.1 Asynchronous Control

We are going to be able to deal with each thread asynchronously, for example, execute one thread step-by-step while others are still running. Unfortunately, standard interfaces provided by operating system such as ptrace or procfs usually do not support threads. This means that, for example, manipulating one thread with ptrace would require stopping the whole process.

2.2 Single Code Image

One of the primary features that distinguishes multithreaded applications from multiprocess ones is the shared address space. This includes both code and data space. The fact that multiple threads may execute the same code image may cause transparency problems. For example, a breakpoint set in one thread is immediately visible in all other ones! However, in fact we want only one of the threads to be stopped.

2.3 Communication Measurement Difficulties

Communication metrics (e.g., delay or amount of data sent between threads) are very important for performance analysis. In multiprocess application using message passing, communication was rather easy to measure, since it was done explicitly, by calling appropriate functions. It is different with threads; here, communication is done via reading and writing in a shared address space, which makes it hard to measure it.

2.4 Probe Effect

Threads are much more lightweight than processes. This means that the overhead of monitoring should also be proportionally lower than in case of processes. Otherwise, the probe effect may be so high that it significantly distorts the measurements themselves.

2.5 Support for different kinds of threads

A serious difficulty is the fact that there are different kinds of threads and many thread standards. Some applications use user-level threads, which are difficult to monitor, since they are entirely managed in the user space, thus they are invisible to the kernel. Even in case of the most popular thread standard – pthreads, the particular implementations differ from each other in significant ways. These differences do not occur only between different platforms but even between different versions of the same operating system. For example, the IRIX6.5 pthreads implementation is totally different from the one in IRIX6.4. As a result, tools handling pthreads are poorly portable. No standard interface for monitoring pthreads exists. In addition, platform-specific interfaces for debugging pthreads, even if exist, may be not publicly available, as is the case with IRIX 6.x.

2.6 Data Locality

The first implementation of TOM is supposed to work on SGI machines with a cc-NUMA architecture. In this kind of architecture, we have a collection of SMP nodes connected through some kind of routers (sometimes called xbows). On each SMP node there is a number of processors sharing a local memory. Upon this collection, a single system image is built including a virtual shared memory. However, the memory is in fact distributed, which means that accesses to a remote node's memory involves higher latencies then accesses to a local memory. Thus, data locality is important, i.e., we desire that threads reference their local memory as often as possible.

3 Monitoring Interface

Monitoring services provided by TOM will be requested via a standard interface OMIS (Online Monitoring Interface Specification) [8]. OMIS defines a standard format for monitoring requests as well as an API via which tool can interact with the monitoring system. The OMIS 2.0 specification defines a large number of services of different types, e.g. for debugging and performance analysis. OMIS defines three types of services: information (get information about objects), manipulation (manipulate object), and event services (detect events), OMIS also offers a possibility to extend the interface with new objects and services, which allows to provide a new type of functionality if needed. For further details on syntax and semantics of OMIS services and the OMIS API functions refer to [OMIS].

4 Architecture of TOM

4.1 Previous Experience – the OCM

In the design phase we have taken advantage of the experience gained during development of an OMIS-compliant monitoring environment for clusters of workstations – the OCM (*OMIS-Compliant Monitoring system*) [3]. The OCM is composed of a collection of Local Monitors, one on each node in the target system, and one central component, the so called Node Distribution Unit (NDU). The NDU is the part of the OCM that communicates with a tool using the monitoring system. The tool sends requests to the NDU, which splits them into sub-requests for the Local Monitors. Each Local Monitor is responsible for executing the request locally, and sending back the reply to the NDU. The NDU collects and assembles the replies from Local Monitors and sends the global reply to the tool. The architecture of the OCM is show in Fig. 1.

4.2 Components of TOM

There are significant differences between TOM and the OCM. First of all, the former works in a shared memory environment, while the latter in a distributed

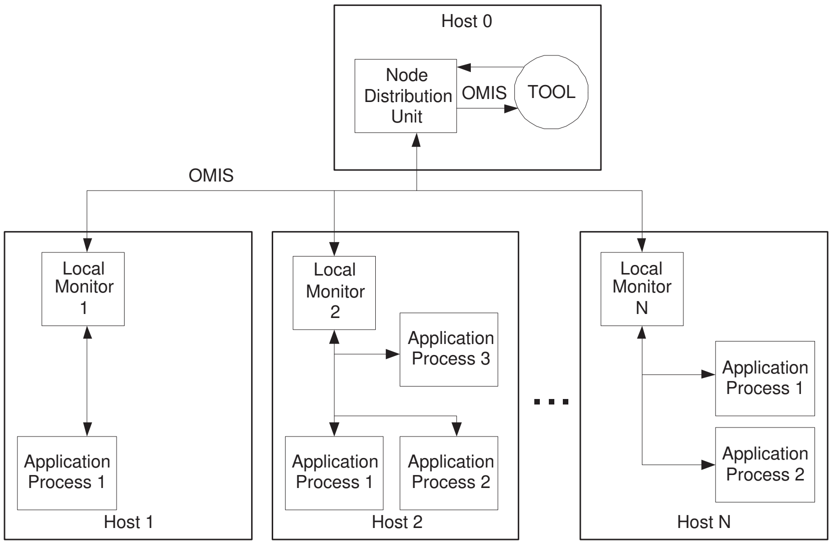

Fig. 1: The OCM – OMIS-Compliant Monitoring System

one. Second, the OCM is focused on processes, while TOM is designed for thread monitoring. These differences enforce a different architecture for TOM. During our survey, we have finally decided that TOM should be composed of three types of components: A Service Manager, Local Monitors, and Applications Monitors.

4.2.1 Service Manager

The Service Manager is very similar to the Node Distribution Unit in the OCM. Like the NDU, it is responsible for communication with tools which are using the monitoring system. It accepts requests from these tools, analyzes them and distributes them across the appropriate Local Monitors. It holds information about the localization of every component of the monitored system. Each thread is registered in the Service Manager.

4.2.2 Local Monitor

Local Monitors in TOM are also similar to those in the OCM. They are supposed to exist on each SMP node of the target machine and execute OMIS requests concerning local objects (i.e., threads located on the same SMP node). This model ensures the locality: once the request is distributed by the SM across LMs, all further interactions between LMs and application threads do not require references to a remote memory.

4.2.3 Application Monitors

Application monitors are a new concept in TOM. They serve two purposes: portability and efficiency in monitoring. They are components of TOM embedded in the monitored application. In fact, they are additional threads which are able to accept requests from Local Monitors. Since AMs have access to the application's address space, they can, e.g., read the value of some variables and pass it to the LM. Thus, no OS mechanisms are needed. In this way, portability is addressed. Application monitors are also used to handle some actions locally and buffer monitoring data before it is passed to the LM, SM and tool. Application monitors can handle asynchronous communication with the monitored system. All application threads can be controlled independently.

AMs can also solve the problem of the single code image. This can be done in the following way. If some instrumentation point is hit, the application monitor can check whether it was hit in the appropriate thread, and only if this is the case, the event will be reported to the LM.

However, application monitors have a disadvantage. They are embedded in the monitored application, so they cannot be protected from being damaged by incorrectly working threads. This is a serious safety issue. A corrupted thread in the application can destroy the application monitor precluding further debugging which is exactly the thing we want to do in such a case.

Some solutions may be used to fix this. First, memory protection can be applied to prevent the memory pages used by AMs from being written by other threads. Second, the LMs may be able to perform all services the AMs are normally responsible for, but using a different (synchronous) mechanism. The latter would be more expensive but only used in an emergency situations, i.e, when the AM does not respond.

4.2.4 Instrumentation

Apart from knowledge about the building blocks of TOM, we should get acquainted with methods to gather monitoring data. To obtain information about the state of a target process, we need to instrument the application. Currently, we use binary wrapping for this [5]. The monitored application needs to be relinked with a special version of the programming library, which contains additional monitoring code which informs the monitoring system about interesting events. For example, TOM may be informed when a thread will block on semaphore and when it will start again. One of the tasks during the implementation phase will consist in the preparation of the instrumented libraries. The instrumentation we use is *selective*, which means that it is activated or deactivated on demand, i.e., only those events are detected which are explicitly requested by a tool, which minimizes the monitoring overhead.

4.3 Difficulties

Some aspects of the design described above turned out to be very hard to be put into practice. In the initial phase of the implementation, we have encountered many difficulties that resulted in some changes in the originally designed architecture. In particular, these problems are connected with the placement of the Local Monitors. We have planned to create one Local Monitor one each SMP node. However, IRIX does not seems to provide a mechanism suitable for such a solution. A cooperation between TOM, POSIX thread library, and the kernel is needed. After each decision made by the kernel or the library scheduler, the placement of Local Monitors should be verified. We have not solved this problem so far. For that reason, this idea will not be implemented in the first prototype. Instead, TOM will contain only one Local Monitor. This solution will decrease the overall performance of our system, but we hope it will be acceptable. To reduce the influence of this simplification we will concentrate on the amount of data exchanged between Application Monitors and Local Monitors. We will try to reduce the communication between these components to a necessary minimum. After the first prototype will be finished, we will verify this solution in a series of tests.

4.4 First Prototype of TOM

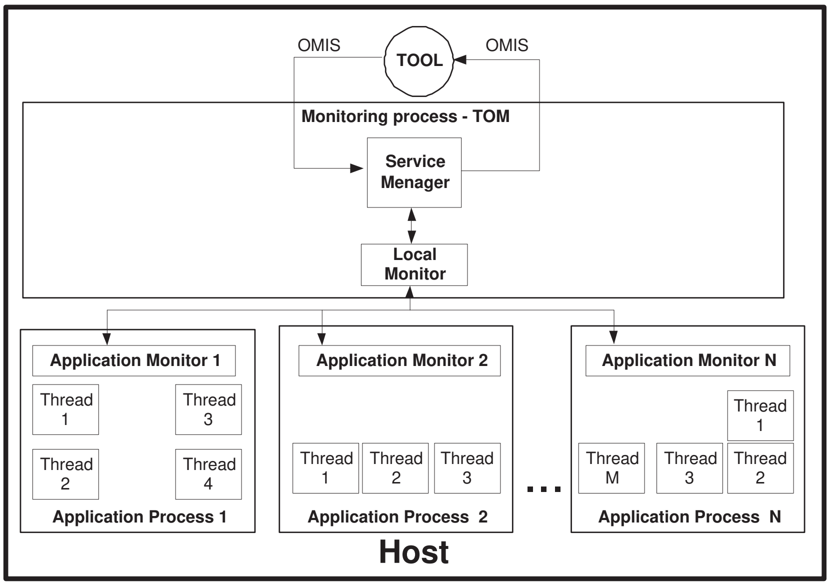

Fig. 2: Thread enabled OMIS Monitor

The architecture for TOM's first prototype is shown in Fig. 2. During our survey, we have finally decided that TOM will be composed of the following subsystems: one Service Manager, one Local Monitor and many Applications Monitors – one for each monitored process. Each of these entities will be a separate thread, except the Local Monitor which will be composed of two threads: one of them will be responsible for messages sent to Applications Monitors, while the second one will handle signals and shared memory segments, thus it will be responsible for communication with the Applications Monitors. The Service Manager and the Local Monitor will be parts of TOM's process, while Applications Monitors will be embedded in the target application as additional threads. The communication between AM and TOM will be via shared memory segment and signals, whereas the TOM process will be communicating with a tool via a TCP socket connection. One instance of TOM will be created per each target application. This approach increases security, because it does not need root privileges. For the same reason it is also easier to implement.

5 Summary and Status

We believe that the presented approach can solve many problems related to thread monitoring. Selective instrumentation allows us to efficiently gather monitored data. The concept of application monitors enables portable asynchronous thread control and solves other problems such as instrumentation of a shared code image. The universal interface OMIS via which TOM provides monitoring services enables various types of tools to be based on TOM. Currently, the implementation of the first prototype has started. This prototype will be used to perform tests to verify our decisions regarding the system's architecture.

Acknowledgements. This work has been carried out within the Polish-German collaboration and is supported, in part, by KBN under grant 4 T11C 026 22.

References

1. Adayapalam, S.S.: 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. www.irisa.fr/lande/ducasse/aadebug2000/proceedings/04srikanth.ps.gz.
2. Baliś, B., Bubak, M., Funika, W., and Wismueller, R.: A Concept of Portable Monitoring of Multithreaded Programs. In: P.M.A. Sloot, C.J.K. Tan, J.J. Dongarra and A.G. Hoekstra (eds): Computational Science - ICCS 2002, Proceedings Part II, LNCS 2330, pp. 884-893, Springer, April 2002, ISBN 3-54043591-3.
3. Bubak, M., Funika, W., Baliś, B., and Wismüller, R.: On-line OCM-based Tool Support for Parallel Applications. In: Yuen Chung Kwong (ed.): Annual Review of Scalable Computing, 3, Chapter 3, 2001, Singapore
4. Buhr, P. A., Karsten, M., and Shih, J.: 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. Cargille, J., Miller, B.P.: Binary Wrapping: A Technique for Instrumenting Object Code. In: ACM SIGPLAN Notices, vol. 27, no. 6, June 1992.
6. Cownie, J., and Moore, S.: Portable OpenMP debugging with TotalView. In *Proceedings of EWOMP'2000, Second European Workshop on OpenMP*, 2000.

7. Ji, M., Felten, E. W., and Li, K.: Performance Measurements for Multithreaded Programs. In *Measurement and Modeling of Computer Systems*, pages 161–170, 1998. citeseer.nj.nec.com/article/ji98performance.html.
8. Ludwig, T., Wismüller, R., Sunderam, V., and Bode, A.: OMIS — On-line Monitoring Interface Specification (Version 2.0). In: Technical Report TUM-I9733, SFB-Bericht Nr. 342/22/97 A, Technische Universität München, Munich, Germany, July 1997.
9. TotalView Multiprocess Debugger. WWW-Site of Etnus Inc., Framingham, MA, USA, 1999. http://www.etnus.com/products/totalview/index.html.