TOM – Efficient Monitoring Infrastructure for Multithreaded Programs*

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

[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
[4] Institute for Software Sciences, University of Vienna, A-1090, Wien Austria
{balis, bubak, funika}@agh.edu.pl, wismuell@in.tum.de, orkid@student.uci.agh.edu.pl
*p*hone: (+48 12) 617 39 64, *f*ax: (+48 12) 633 80 54 *p*hone: (+49 89) 289-28243

Abstract. Multithreading is an efficient and powerful solution for parallel programming. However, multithreaded programming is difficult and there are few tools that support the development of multithreaded applications. Even fewer or no tools introduce portable concepts to deal with threads on many platforms. In this paper, we describe the TOM monitoring infrastructure for multithreaded applications. The key concept of TOM are Application Monitors which are additional monitoring threads in the monitored application. The concept of Application Monitors allows efficient and portable solutions to the most important problems in thread monitoring. We describe the current implementation of TOM with a focus on Application Monitors. In addition, we provide a case study implementation of fast breakpoints based on these Application Monitors.

Keywords: multithreading, monitoring, debugging, performance analysis, parallel tools

1 Introduction

The most popular thread standard today is POSIX threads, called pthreads [9]. Pthreads define a standardized interface and behaviour for thread creation and management subroutines, as well as related data types. Almost every operating system (OS) provides a pthreads library, although the underlying implementation is strongly system-dependent. This makes it hard to develop tools that support the debugging of pthreads applications – there is no standard interface for accessing pthreads internals, and standard system mechanisms such as ptrace or procfs [11] are often not pthreads-enabled. Besides debugging, application developers are also interested in the monitoring of running applications in order to analyse the application performance, to find how large are the delays due to

* This work has been carried out within the Polish-German collaboration and is supported, in part, by KBN under grant 4 T11C 026 22, and, in part concerning SGI, under grant 6 T11 0052

synchronization, etc. These features are offered by tools such as performance analyzers or visualizers. Instrumentation of an application is necessary to collect information for performance measurements. Efficient instrumentation is not trivial and multithreading introduces even more problems such as transparency due to the single code image [5].

In this paper, we describe the TOM monitoring infrastructure for multithreaded applications, which supports the development of portable tools, such as debuggers or performance analyzers. The key concept in TOM are Application Monitors which are additional monitoring threads in the monitored application. The Application Monitors concept allows for efficient and portable solutions to the most important problems of thread monitoring. We describe the current implementation of TOM with special focus on Application Monitors and provide a case study of fast breakpoints based on this concept.

2 Troubles with Thread Monitoring

Below we shortly summarize the most important problems related to the monitoring of multithreaded programs.

Asynchronous control. We wish to deal with threads asynchronously like with processes, for example, execute step-by-step only one thread of a process while others are still running. Unfortunately, OS interfaces used for this purpose, such as ptrace or procfs, usually do not support threads. This means that manipulating one of the threads with ptrace would require stopping the whole process.

Single code image. Threads share not only data but also code address space. This introduces *transparency* problems when the monitoring system instruments a thread's code. A breakpoint set in one thread is immediately visible in all other ones while we are going to apply the breakpoint to one thread only.

Portability. Though the common pthreads interface is widely agreed, the underlying implementations of libraries are very different from system to system or even between two versions of the same system (such as in IRIX 6.4 and IRIX 6.5). In addition, there is no standard debug-interface for pthreads. Some vendors do provide a pthreads debug library, but it is not portable and sometimes not even available to the public. As a result, in some cases only the vendor tools are thread-aware, as it is the case for IRIX.

3 TOM – Thread-enabled OMIS Monitor

The architecture of TOM is shown in Fig. 1. TOM is an autonomous infrastructure composed of three types of components: Service Managers (SM), Local Monitors (LM), and Application Monitors (AM). Tools connect to TOM via SMs, send monitoring requests and receive replies. The communication interface is based on the OMIS specification (On-line Monitoring Interface Specification) [10, 2]. On each local unit of the system, which tightly shares a common memory, one LM is supposed to reside. For example, on a cluster there is a single SM

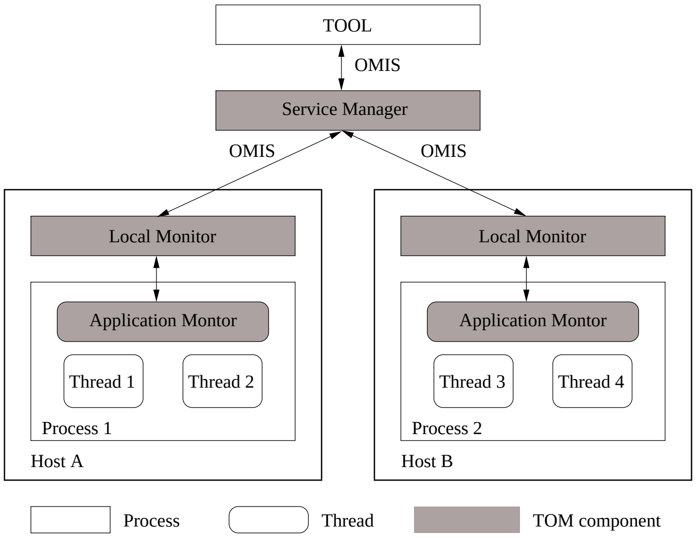

Fig. 1. Thread enabled OMIS Monitor

and one LM per host, while on a multiprocessor shared-memory machine, such as SGI Origin, there may be one SM, and one LM per each SMP node. Finally, there is one AM thread per each process of the application. The monitoring requests are executed by either LMs or AMs, while the SM's task is to distribute the requests to appropriate LMs and collect the corresponding replies.

In the rest of the paper, we focus on the Application Monitors, which are the key concept in TOM enabling efficient and portable monitoring of threads.

4 Application Monitors

4.1 General concept

The Application Monitors (AMs) are designed for two main objectives: portability and efficiency of monitoring. An AM is an additional thread in each process of the application. The main benefit is that, as an AM shares address space with the application, it can perform some actions directly, without OS mechanisms involved.

An LM sends asynchronous requests to an AM and waits for responses. In this way, we can read the address space (e.g., variable values), set breakpoints or even insert a general instrumentation code dynamically. The asynchronous control of threads is addressed owing to the communication pattern: asynchronous queries/replies as opposed to synchronous mechanisms such as ptrace. At the same time we also benefit from a higher portability. With AMs we can also easily solve the single code image problem. This can be done in such a way that when an instrumentation point is hit, the AM checks if it was hit in the appropriate thread, and only in this case the instrumentation code will be executed.

4.2 Avoiding of AM corruption and transparency

Since AMs share the address space with the other application threads, they may be subject to corruption by the code of the user's application as a result of bugs. This may render debugging impossible while this is exactly the situation when we especially wish to do that. A solution could be to `mprotect` the appropriate pages. In this case, write attempts would simply cause page faults which should be enough to fix the problem. However, the disadvantage of this solution is a loss of efficiency. For a full protection, we would need to protect both text and data pages of the AM. This would mean that whenever we need to write the AM's memory, we would have to temporarily unprotect the appropriate pages. This is a high-cost operation since it involves a switch to kernel mode. An alternative solution, efficient though not perfect, is to ensure that the AM's data is allocated in memory segments located far away from the pages of other threads. This considerably reduces the probability that a thread accidentally writes the AM's data space. We can achieve that by providing our own memory allocation subroutines or by using a shared memory segment to store the AM data. In our case, the latter option is natural, since in the current implementation the AMs and the Local Monitor already use a shared memory segment for communication.

A second problem is transparency. The additional thread must not be seen by other threads and must not affect the execution of the application. In general, this can be achieved by proper instrumentation of all thread library functions which are affected by this problem. These are usually the operations that involve 'all' threads (e.g., return number of all threads) or 'any' thread. However, in case of pthreads there are no such operations, thus the problem of transparency is not a serious one.

5 Case Study: Breakpoints

In this section, we present a detailed design and implementation of fast breakpoints based on the concept of Application Monitors. Though the concept of fast breakpoints itself is not new, and similar approaches have been proposed [8, 4], we believe that it can illustrate the advantage of the AMs concept. Furthermore, multithreading poses new problems related to fast breakpoints which will be described below.

5.1 Description of the fast breakpoints concept

The general idea is as follows. As a prerequisite, we need a way to stop an arbitrary thread. The AM achieves this by sending the thread a signal. The signal handler for this signal will then wait for a condition variable, which effectively stops the thread. When the thread should be continued again, the AM signals this condition variable to wake-up the thread.

At start-up, the AM creates a configurable number of condition variables and handler functions. This number determines the maximum number of breakpoints allowed. Each handler function contains two parts: a call to the pthreads routine that waits for a condition variable, and ‘empty space’ (i.e. a couple of NOP-instructions) to save the original instruction when the breakpoint is set (see below). Thus, the process of setting a breakpoint is done in the following stages:

1. The debugger sends the AM a request ‘set breakpoint at address AAA’.
2. The AM thread receives the request and saves the original instruction at address AAA in the space reserved in the handler function.
3. The AM replaces the instruction at address AAA with a jump to the handler function.

This solution seems quite obvious and relatively easy to implement, however, there are serious problems related to the third operation. The following subsection discusses them.

5.2 Troubles with instrumentation of fast breakpoints

The question is: how can we safely replace the original instruction with another one, e.g., a jump to the breakpoint handler? In general, we must distinguish two cases:

– the jump instruction is not longer than the shortest possible instruction for the architecture (usually one word),
– the jump is longer than the shortest possible instruction.

In the first case, the original instruction can be replaced using an atomic memory-write operation. This means that we can safely do the replacement on-the-fly, since any thread either will see the original state or the new state; there are no intermediate, inconsistent states possible.

In the second case, we may run into serious problems, since we cannot do the replacement atomically. Even if we stop all threads before the replacement, and continue them again afterwards, some of the threads might accidentally have been stopped in the middle of the code region we wish to change. After the replacement they may be executing an incorrect instruction (e.g., the ‘second half’ of our jump). It is still possible to handle this case by careful examination of the threads’ stacks. If any thread happens to execute the ‘forbidden’ code, we may continue it for a while and stop it again until we get the desired result. However, if the thread happens to be waiting in this region of code, the retry strategy will not work. The most serious problem though, is when the application’s code contains a jump into the code region to be replaced. After changing the code region, this jump will branch into ‘the middle’ of an instruction, which usually will crash the application. Since at the level of the machine code there is no easy way to check whether a given address is a branch target, it is extremely difficult to properly handle this case.

The only reliable solution is to always try to use a short jump instruction. However, while on most architectures such jump instructions are available, they are usually ‘short’ jumps, i.e., the range of the jump is limited to a nearby memory region. Since it is not easy to guarantee that the handler function is always within a nearby memory region, using short jumps is not always possible.

5.3 Optimal fast breakpoints instrumentation

Fortunately, on each processor, there is an instruction which is not longer than the shortest possible instruction for the architecture, and which can be used instead of a jump. This is the trap instruction which is used by 'normal' debuggers to set breakpoints. Although this solution is guaranteed to work, its disadvantage is that it is not as efficient as a normal jump instruction, since the trap instruction involves a switch into kernel mode and a signal delivery. Still, in our approach there is an advantage over traditional debuggers – the signal is caught by the same process, which avoids an additional process switch and is therefore considerably faster.

The optimal solution is a mixture of the options described above. If the address of the handler function is within the short-jump range, we use a jump instruction. Otherwise, we use the trap instruction.

5.4 Evaluation

We have performed a test to compare the efficiency of breakpoints implemented in three different ways: 1) trap instruction with signal handled by an external process, 2) trap instruction with signal handled inside the AM, 3) with a jump instruction. The test procedure was very simple – a thread was hitting a breakpoint, and the handler immediately continued execution; this was done a number of times in a loop. The total time to execute the loop was measured. The test was performed on an 128-processor SGI Origin machine with R14k processors installed, and IRIX 6.5 OS. Table 5.4 shows the results. The leftmost column indicates the number of iterations, the following columns provide the real time in system tics.

No. iterations	TRAP / signal ext.	TRAP / signal AM	JUMP
10,000	3213 ± 74	15.7 ± 0.48	0 ± 0
100,000	32112 ± 544	159 ± 2.1	0.8 ± 0.42
1,000,000	323870 ± 3532	1585 ± 14.5	6.8 ± 0.42

Table 1. Evaluation of different implementations of breakpoints

The table shows clearly that with application monitors and trap instruction we obtain a result two orders of magnitude better than with the traditional method. Additionally using the jump instruction improves the result by still another two orders of magnitude!

6 Related Work

Until now, thread-enabled tools for parallel programming are still not as well supported as those for multiprocess applications. Most of existing tools are debuggers, for example, Wildebeest, TotalView, kdb, NodePrism 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 HP-UX platforms and implements only a synchronous thread control. TotalView [6] 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 even provides support for applications developed in OpenMP. However, it does not allow for asynchronous thread control unless this feature is supported by the operating system. Kdb [4] was designed to overcome the limitations of other debuggers, specifically for handling user-level threads and for controlling each target thread independently. It does not support pthreads, though.

There are some efforts to address performance analysis of multithreaded applications. One example is the `Tmon` tool [7], which is a monitoring system combined with a visualization module used to present *waiting graphs* for multithreaded applications. An interesting approach is the thread-enabled Paradyn tool [12]. Paradyn is able to associate performance data with individual threads and relies on dynamic instrumentation to lower the instrumentation cost and overall monitoring intrusiveness.

7 Comparison of AMs with Other Concepts

Our approach is different from those described above not only because it unites the main benefits of the mentioned projects, but – what is more important – it introduces the concept of Application Monitors, which are *active* monitoring components and thus more powerful. The benefits of using AMs are manifold: **AMs encapsulate their functionality in a standardized protocol**. External users (a debugger, a monitoring system, etc.) just send standardized request and receive replies. This increases portability, since all platform-specific details are hidden in the implementation, and interoperability, since multiple users can potentially concurrently use the AM's services. **AMs can access the application's address space directly**, without the need to deal with interfaces such as ptrace or procfs. This greatly eases and speeds up many operations, for example reading memory, which is the basis for sophisticated data visualization capabilities of debuggers. **AMs enable dynamic insertion of arbitrary instrumentation** into the application's code. The only project which so far offers such a possibility, is DynInst [3]. DynInst is a very powerful approach, which allows to insert (nearly) arbitrarily complex code at run-time. AMs are more restricted, since only the jump to the handler function is inserted dynamically; all the other required code is included into the application by linking it with the AM library. This restriction, however, makes our approach much more lightweight and portable; for example, we do not need a complex code generator like DynInst. The simpler structure also makes the AM approach easier to maintain and less error-prone. **AMs enable asynchronous control of threads**. For example, the concept of fast breakpoints described in this paper allows independent debugging of a single thread while keeping others running, even if the particular implementation is based entirely on user-level threads, which are not supported by the OS. Finally, **AMs enable the portable monitoring of threads**, independent of the concrete pthreads implementation (e.g. user level vs. kernel threads) and independent of OS support.

8 Conclusion and Future Work

We have described our experience in design and implementation of a portable and efficient monitoring system which supports multithreaded applications. The key concept of our solution are Application Monitors which are additional threads for each process of the application. The presented concept is generic – it is designed to support a variety of tools, for example, debuggers and performance analyzers. We have shown a case study of fast breakpoints based on the concept of Application Monitors.

Currently, we have implemented the first prototype of TOM which contains the infrastructure for Application Monitors with basic monitoring services, and a simple external monitoring infrastructure. The tasks for the future are as follows. First, an existing OMIS-compliant monitoring system for clusters – the OCM [2] will be extended with threads support so that we can benefit from the full monitoring functionality offered by OMIS. Second, the Application Monitors will be extended by additional monitoring functionality. Finally, the thread-enabled OCM will be integrated with Application Monitors to obtain a fully functional OMIS-based thread-enabled monitoring system – TOM.

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.
2. M. Bubak, W. Funika, B. Baliś, R. Wismüller. On-line OCM-based Tool Support for Parallel Applications. In *Annual Review of Scalable Computing*, vol. 3, pp. 32–62. World Scientific Publishing and Singapore University Press, Singapore, 2001.
3. B. Buck and J. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.
4. P. A. Buhr, M. Karsten, and J. Shih. KDB: A Multi-threaded Debugger for Multi-threaded Applications. In *Proc. SPDT'96: SIGMETRICS Symposium on Par. and Distrib. Tools*, pp. 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. TotalView Multiprocess Debugger. WWW-Site of Etnus Inc., Framingham, MA, USA, 1999. `http://www.etnus.com/products/totalview/index.html`.
7. M. Ji, E. W. Felten, and K. Li. Performance Measurements for Multithreaded Programs. In *Measurement and Modeling of Computer Systems*, pp. 161–170, 1998.
8. P. B. Kessler. Fast Breakpoints. Design and Implementation. *ACM SIGPLAN Notices*, 25(6):78–84, June 1990.
9. Portable Operating System Interface: The Pthreads standard (POSIX 1003.1c).
10. Ludwig, T., Wismüller, R., Sunderam, V., and Bode, A. OMIS – On-line Monitoring Interface Specification (Version 2.0). Shaker Verlag, Aachen, vol. 9, LRR-TUM Research Report Series, 1997. `http://wwwbode.in.tum.de/~omis`
11. J. B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. John Wiley & Sons, 1996.
12. Z. Xu, B. Miller, O. Naim. Dynamic Instrumentation of Threaded Applications. In: Proc. *7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Atlanta, Georgia, May 4-6, 1999.