An Efficient Multi-level Trace Toolkit for Multi-threaded Applications

Vincent Danjean, Pierre-André Wacrenier, Raymond Namyst

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

An Efficient Multi-level Trace Toolkit for Multi-threaded Applications

Vincent Danjean — Pierre-André Wacrenier — Raymond Namyst

N° 5513

Mars 2005

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--5513--FR+ENG

An Efficient Multi-level Trace Toolkit for Multi-threaded Applications

Vincent Danjean , Pierre-André Wacrenier , Raymond Namyst

Thème NUM — Systèmes numériques
Projet Runtime

Rapport de recherche n° 5513 — Mars 2005 — 12 pages

Abstract: Nowadays, observing and understanding the behavior and performance of a multithreaded application is nontrivial, especially within a complex multithreaded environment such as a multilevel thread scheduler. In this report, we present a trace toolkit that allows a programmer to precisely analyze the behavior of a multithreaded application. A application's run generates several traces that are merged and analyzed offline. The resulting super-trace contains not only classical information such as the number of elapsed cpu cycles per functions but also details about thread scheduling at multiple levels.

Key-words: Profiling, multithreaded application, two-level scheduling, SMP

Unité de recherche INRIA Futurs
Parc Club Orsay Université, ZAC des Vignes,
4, rue Jacques Monod, 91893 ORSAY Cedex (France)
Téléphone : +33 1 72 92 59 00 —Télécopie : +33 1 72 92 59 ??

Mécanismes de traces efficaces pour programmes multithreadés

Résumé : Aujourd'hui, il est très difficile d'observer et de comprendre finement les performances des applications reposant sur des supports d'exécution multithreadés, en particulier lorsque la plateforme de threads utilisée est complexe (ordonnancement multiniveaux). Dans ce rapport, nous présentons un environnement permettant d'observer précisément le comportement des applications multithreadées. Lors de son exécution, l'application génère plusieures traces qui sont fusionnées et analysées après coup. La supertrace résultante donne non seulement des informations classiques comme la consommation de cycle CPU par fonction, mais aussi le détail de l'ordonnancement multiniveau des processus légers de l'application.

Mots-clés : Profilage, application multithreadée, ordonnancement à deux niveaux, SMP

1 Introduction

Analyzing bottlenecks, debugging deadlocks, understanding performances need a fine-grain analyze of the behavior of a parallel application. This problem becomes even more tricky when dealing with multilevel multithreading. Let us recall that there are three main families of threads:

User-level threads are managed by the application, can be tailored to the particular requirements of the application and have very efficient basic operation (creation, destruction, switching and synchronization); the disadvantages are that because the operating system knows nothing about them, they cannot use all available system resources, especially multiprocessors.

Lightweight processes (also called LWPs or kernel-level threads) are managed by the kernel and they have access to all kernel resources, particularly multiple processors. For instance, several LWPs belonging to the same process can be simultaneously active. The disadvantages are that they consume kernel resources (the number of LWPs is usually limited) and they tend to incur much more overhead since all LWP scheduling and switching require kernel intervention.

Hybrid threads (multilevel threads, mixed threads) were defined in order to take advantage of the two previous techniques, the key idea is to map user-level threads onto a pool of LWPs. This leads to a two-level scheduling: the kernel manages LWPs which manage user-level threads in a distributed way. SUN'S SOLARIS operating system provides such hybrid-threads.

Analyzing the scheduling of a multithreaded application executed by an hybrid thread system, observing the behavior of a such application in its global context are difficult tasks that require some kind of support from the kernel, from the (hybrid-)thread library and from the application. For this purpose, the code must be instrumented in order to record selected events in one or several *trace buffers*. This leads to multilevel instrumentation. In this framework, we may notice the work of Shende [She01] who has defined a strategy for utilizing multilevel instrumentation in order to improve the coverage of performance measurement in layered software. His approach, based on the *node/process/thread* model, was successfully implemented in the TAU portable profiling and tracing toolkit. For instance, to deal with Java's multithreaded environment [MS00], each thread creation is recorded into TAU's performance database (requiring mutual exclusion with other threads) in order to create a per-thread performance data structure.

In [XMN99], Xu *et al* use the dynamic environment PARADYN [MCC+95] to profile multithreaded applications in a statistical way. In their approach, each thread has its own private copy of performance counters or timers; locks are used to access to only a few global bookkeeping data structures.

However, in the framework of the parallel environment PM^2 [NM95] based on an hybrid thread library, our goal is to debug and to optimize low level middlewares, for instance reactive communication library [ABMN02, DN03], tricky mechanisms like scheduler activations [ABLL91, DNR00]; in this aim we have to study lock mechanisms and interruption handler

routines, for instance. When dealing with such low level middlewares and parallel processes, there is no secret: the instrumentation must be as less intrusive as possible. Especially, we do not want to introduce new synchronization points within the kernel or within the thread scheduler in order to limit interdependent intrusion effects[1]. That for, we have defined a lightweight multilevel instrumentation toolkit in order to be able to precisely trace the behavior of a multithreaded program. For this purpose we have to meet the following requirements:

To be the lesser intrusive as possible. The tracing overhead must be very small not only to allow an accurate performance analysis but also to minimize the intrusive effect on the global scheduling of the application: excessive increasing of the execution time of a critical section, new synchronization points and new context switch points. Therefore, system calls and high level synchronization mechanisms must be avoided.

To deal with multilevel instrumentation. Since our goal is to study multilevel schedulers or high performance communication libraries, we need to record both kernel and user level events. For instance, we need to record all the kernel's scheduler decisions and all the thread library's scheduler decisions to get a complete knowledge of the scheduling of a multithreaded application.

To deal with a huge amount of data. We may record a lot of events like scheduler decision, entry and exit of some functions executed by a thread or by the kernel. This may generate several mega bytes per second.

In this paper we propose a solution based on two independent buffer traces: the first one contains kernel-level events, the second one contains user-level events. First, we present the FAST KERNEL TRACE toolkit, our starting point. Then we justify our approach and give some technical details. Finally, we analyze the introduced overhead on several multithreaded applications.

2 From Kernel Tracing to Multi-level Tracing

2.1 The Fast Kernel Traces Toolkit

Kernel instrumentation may be done during compilation [YD00, RC01] or dynamically at runtime like KERNINST [TM99] does. It is worth noting that operating systems like LINUX 2.6.10 and SOLARIS 10 (DTRACE) are already provided with a dynamic instrumentation toolkit which allows to instrument the running operating system kernel. For our purpose we chose to use the `FKT` toolkit [RC01] which is a simple and efficient SMP LINUX kernel dedicated trace toolkit. It is based on a source level instrumentation, which is achieved thanks to a set of macro-functions. Therefore the modification of a trace point needs to recompile the source code and to restart the kernel. Nevertheless, operation like `tracing`

[1] Note that, Malony *et al* [MS04] have shown that while it is possible to compensate overhead due to the intrusion a single process, parallel overhead compensation is a more complex problem because of interdependent intrusion effects.

`start`, `tracing stop` or `tracing store` can be done from the user-level space. It is worth knowing that FKT uses a well optimized storage mechanism [Thi03] since the pages of trace file's buffer-cache are directly used and that these pages are organized in a circular buffer way. This allows to use the TLB mechanism to directly write buffer's pages on the disk, avoiding useless memory copy and limiting memory consumption.

Figure 1 details an `FKT` macro. The `KEYMASK` argument and the kernel variable `fkt_active` allows to enable/disable the tracing. A new system call is defined to set the variable `fkt_active` from the user-space. The `CODE` argument denotes the recorded event. Here `P1` and `P2` are two integer arguments left to the programmer (one may record from zero up to five integer arguments).

2.2 Meeting Hybrid Scheduling's Requirements

In order to precisely rebuild the behavior of multithreaded programs, it is necessary to be able to determine at any time the current running user-level threads on the SMP. Note that Kernel's view is insufficient: indeed, the kernel has no knowledge about user-level threads which are scheduled by the LWP pool. On the other side the user space's view is insufficient too: LWPs usually are unaware of kernel's context switches, so it is difficult, from the user-space point of view, to get the identity of the running LWPs at a given date and to get the processor number on which user code is running.

To solve these problems, new system calls might be created in order to request the identity of processor that is recording an event, for instance, or in order to notify the kernel scheduler about the user-level scheduler's context switches. However, this kind of solution is too intrusive: system calls are expensive (cf. micro benchmarks given in Section 3.3) and, moreover, this kind of solution introduce much more context-switch points than the uninstrumented execution would encounter. Another solution would be to define a mechanism based on an *up-call*: in order to transmit the kernel point of view to the user-space, the kernel force the application to call a given function, like the POSIX signal's mechanism does. However this solution is also expensive since the thread state must be saved at every up-call.

Our proposition is to generate a trace from both point of views. The kernel's trace will be generated by FKT and user level's one will be generated by Fast User Trace (FUT), a tool similar to FKT. The key-points of this solution are: (1) Dealing with hybrid scheduling,

```
#define FKT_PROBE2(KEYMASK,CODE,P1,P2)                \
  do {                                                \
      if( KEYMASK & fkt_active )                      \
         fkt_header( ((unsigned int)(CODE)),          \
            (unsigned int)(P1), (unsigned int)(P2) );\
     } while(0)
```

Figure 1: A definition of an `FKT` macro for an event with two parameters.

Kernel and user-level traces are both necessary to get a full description of a multithreaded application run. Both traces use the cycle counter register to stamp the events since this clock is very accurate. (2) Dealing with SMP, the cycle counter register of each processor is perfectly synchronized with each others by the hardware. (3) All context switches (user and kernel) are recorded, so that we will be able to deduce what happens from a scheduling point of view within the system.

After the run, the both traces are merged into a so-called *super-trace* which contains the following event data: the event code, time-stamp, size and parameters; the identifiers of the user-level thread, LWP and processor that executed the recording. By reconciling the kernel and the user points of view, this toolkit allows to trace multithreaded applications and, moreover, it allows to put the application run back in its execution context, since any kernel event can be recorded too. Hence it is possible to get an accurate analysis of low level middlewares like a multithreaded communication library.

2.3 Description of the Tracing Toolkit

This multilevel tracing toolkit has been implemented on top of MARCEL/LINUX/INTEL x86 system; MARCEL is the hybrid thread library of the portable parallel environment PM^2 [NM95].

In order to instrument the kernel, the user needs to apply a given patch against the LINUX kernel. This patch introduces instrumented points in the kernel code allowing to record events like context switches, entries/exits of hardware interruption (IRQ) and software interruption (system calls). The thread library MARCEL is instrumented in order to record user-thread scheduling decisions; for instance, events like user-level context switches, creation and termination of LWPs are recorded. Moreover this instrumentation allows to trace any function of the library MARCEL. In this way, one may accurately trace the performances of the library and determine the cause of the preemption of a user-level thread (elapsed time-slice, unacquired lock,...).
The API of FUT is similar to the FKT's one. Event recording is done by `FUT_PROBE`x`()` macros and some event types are already defined (like user-level thread creation or destruction). A basic code instrumentation tool is also implemented in order to automatically attribute entry/exit codes for each function. The `PROF_IN()` and `PROF_OUT()` macros may be used to trace the entry and the exit of a function. The code instrumentation may either directly be done by programmers or inserted automatically by compilers, like GCC does.

Once the two traces have been recorded, they are merged in a super-trace where events are ordered with respect to the time-stamps. During the merge the relation between user-level events, user-level threads , LWPs and processors is established. However, some kernel events, like those that are recorded during interruption routines, are not to be associated with any user-level thread or event.

We have developed a tool (called SIGMUND) which allows to apply filters to the super-trace in order to extract a sub-trace from it. One may specifies events that match some criteria (a given kind of events, a given user-thread, a time-slice). Some basic measures

```
$> sigmund --trace-file supertrace.log --thread 15 \
   --event CONTEXT_SWITCH --list-events
type  date_tick  pid cpu thr   code name                 param(s)
[...]
USER  97615576  7137   1   7  23014 USER_CONTEXT_SWITCH  15
USER  97757052  7137   1  15  23014 USER_CONTEXT_SWITCH  8
USER  98006248  7136   0   6  23014 USER_CONTEXT_SWITCH  15
KERN  98139183  7136   0  15  23014 KERN_CONTEXT_SWITCH  6152
KERN  98638163  2352   2   ?  23014 KERN_CONTEXT_SWITCH  7136
USER  99060185  7136   2  15  23014 USER_CONTEXT_SWITCH  7
[...]
$> sigmund --trace-file supertrace.log --thread 15 --active-time
130193845 cycles
```

`type`: event level – `date_tick`: event date – `pid`: LWP identity
`cpu`: processor identity – `thr`: user-thread level identity
`code`: event code – `name`: event name – `param(s)`: associated parameter values

In this example, we can see that user-level thread 15 was firstly scheduled by LWP 7137 on CPU 1; then it was scheduled by LWP 7136 on CPU 0. Then after LWP 7176 was preempted by the kernel (in order to schedule another application), it was scheduled on CPU 2. Then the user-level scheduler preempted thread 15 in order to run thread 7. Here we can see that this 2-level scheduler does not take enough care of the affinity.

Figure 2: Super-trace analysis with sigmund

may also be computed like, for instance, the (active) execution time of a given thread or the reactivity of a communication library to a given communication event (the elapsed time between the detection of a given event by the kernel and its treatment by the application). A filter was developed in order to translate the super-trace into the file-format of `Pajé` [dKdOS00], a generic graphic trace viewer.

Figure 2 shows two requests about the user-level thread 15 on a super-trace. The instrumented program is executed on a SMT bi-processor machine (thus 4 logical processors, numbered from 0). For this execution, the 2-level thread library defined 4 LWPs in order to execute user-level threads. Figure 3 shows how to observe thread's reactivity.

3 Implementation Details and Performance Analysis

We addressed some technical issues in order to limit the intrusion of tracing. In the following paragraphs, we detail the time-stamping, the trace format and the concurrent recording mechanism. Then we discuss about the overhead introducted by the instrumentation.

```
type  date_tick  pid cpu    thr     event                param(s)
USER  5150163706  2732 2  8(work/6) USER_CONTEXT_SWITCH  1(daemon)
KERN  5150169922  2732 2  1(daemon) SYSTEM_CALL          142(select)
USER  5150182646  2732 2  1(daemon) USER_CONTEXT_SWITCH  11(work/9)
USER  5152816866  2733 3  9(work/7) USER_CONTEXT_SWITCH  12
KERN  5170071750  1630 0  ?         IRQ                  24(eth0)
USER  5176768370  2731 1 10(work/8) USER_CONTEXT_SWITCH  5(work/3)
USER  5179394810  2732 2 11(work/9) USER_CONTEXT_SWITCH  13(work/11)
USER  5182046038  2733 3 12(work/10)USER_CONTEXT_SWITCH  14(work/12)
USER  5205964954  2731 1  5(work/3) USER_CONTEXT_SWITCH  15(work/13)
USER  5208624942  2732 2 13(work/11)USER_CONTEXT_SWITCH  17(work/15)
USER  5211315302  2733 3 14(work/12)USER_CONTEXT_SWITCH  18(work/16)
USER  5235191514  2731 1 15(work/13)USER_CONTEXT_SWITCH  19(work/17)
USER  5237854634  2732 2 17(work/15)USER_CONTEXT_SWITCH  20(work/18)
USER  5240544282  2733 3 18(work/16)USER_CONTEXT_SWITCH  21(work/19)
USER  5264421734  2731 1 19(work/17)USER_CONTEXT_SWITCH  4(work/2)
USER  5267084698  2732 2 20(work/18)USER_CONTEXT_SWITCH  2(work/0)
USER  5269736086  2733 3 21(work/19)USER_CONTEXT_SWITCH  3(work/1)
USER  5293652362  2731 1  4(work/2) USER_CONTEXT_SWITCH  6(work/4)
USER  5296353186  2732 2  2(work/0) USER_CONTEXT_SWITCH  7(work/5)
USER  5298968062  2733 3  3(work/1) USER_CONTEXT_SWITCH  8(work/6)
USER  5322881710  2731 1  6(work/4) USER_CONTEXT_SWITCH  1(daemon)
KERN  5322893274  2731 1  1(daemon) SYSTEM_CALL          142(select)
USER  5322907374  2731 1  1(daemon) USER_EVENT           received_msg
```

Our tracing toolkit allows to emphasis the reactivity of multithreaded applications. We can compute the elapsed time between the network message arrival (a hardware interrupt is raised by the network card) and the treatment of this message by the right user-thread in the application.
This figure shows the relevant parts of a trace of a run where 20 threads are devoted to some computation (denoted `work/0` *to* `work/19`*) and one special thread (denoted* `daemon`*) is listening to the network in order to manage the incoming messages as soon as possible. In this program, the network thread executes a non-blocking system call*[a] *to* `select()` *and call* `pthread_yield()` *to give hand to the other thread in the system if no message is available. Here the considered algorithm leads to very bad latencies, as the* `daemon` *thread has to wait that all the other threads use their quantum before getting hand back whereas the message has already been received by the OS. However, most of thread libraries do not provide anything else to deal with this kind of problems. A description of an adequate support in thread libraries to improve thread reactivity to external asynchronous events is described in [BDN02].*

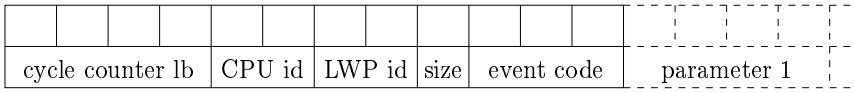
[a] blocking system call must be avoided with user-level thread library

Figure 3: Using our mecanisms to observe thread's reactivity

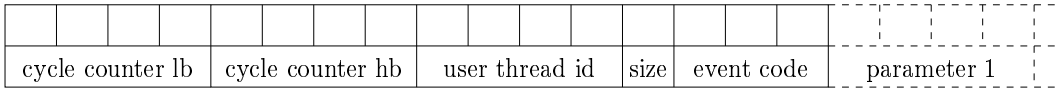
3.1 About the Time-stamping and the Trace Format

FKT and FUT use the cycle counter register as time reference; this register stores the number of elapsed cycles since the last machine switch on. It is directly readable from the user-space. It is as accurate as possible and it is 64 bit wide that leads to a 136 years period (2^{32} s) on 4 GHz (2^{32} Hz) machine, moreover cycle counters of a SMP are synchronized.

Note that only 32 bits are required to stamp the kernel events. Indeed, as soon as the cycle register was once saved, there are enough kernel events (like kernel scheduling decisions or clock interruptions) that are recorded during a period (2^{32} cycles) to infer the 32 higher bits. However, this argument does not hold for user-level threads which may not produce any event during several seconds.



Kernel trace layout ($\geq$ 12 byte wide)



User-level trace layout ($\geq$ 16 byte wide).

Figure 4: Kernel and user-level trace entry layout

In order to limit the intrusiveness, event buffers are created and initialized when before the real launching of the application. An initial section containing context information (function names, running LWPs) is also recorded in both buffers. The initial section size is about several hundred of kilobytes.

3.2 Mutual Exclusion Mechanism

Dealing with threads and SMP machines, we have to take care of concurrent accesses to the trace buffers. Actually this problem of concurrency appears as soon as we want to record asynchronous events such as hardware interrupts or signals, even on single processor machine. Indeed, asynchronous events may be raised at any time and we do not want to try to block then in order not to perturb the scheduling. Therefore the instrumentation code must be fully reentrant.

The basic idea of our approach is to atomically increment the buffer length variable. However, high level mutual exclusion mechanisms are forbidden. We have this problem using the *atomic* CPU instruction `cmpxchgl`.The principle consists in storing the buffer's length value in a register, then storing the new buffer's length in a second register and finally calling `cmpxchgl` in order to set the new buffer's length. This subroutine is retried until the `cmpxchgl` call is successful. As a result, the event trace may be not time-stamp ordered, thus the merging tool may have to reorder the trace.

Table 1: Micro benchmarks (Linux 2.6.4 bi-Xeon SMT 2.8 GHz)

Function/Macro		cycle
Macro	`PROF_IN`	260
System call	`getpid()`	1900
buffered io	`printf("test")`	672

Table 2: Overhead measures (Linux 2.6.4 bi-Xeon SMT 2.8 GHz)

	execution time	# recorded	events (size)	flow (Mo/s)
`Sumtime` program				
without any profiling	230 ms	-		-
profiled (context switches)	288 ms (+23%)	161 484	(3.72 Mo)	13
profiled (all events)	430 ms (+80%)	821 844	(13.4 Mo)	31
`SuperLU_MT` program				
without any profiling	7.17 s	-		-
profiled (context switches)	7.30 s (+1.8%)	374	(0.007 Mo)	0.001
profiled (all events)	7.50 s (+4.6%)	836054	(8.39 Mo)	1.1

3.3 Analysis of the Tracing Overhead

In table 1, we compare the cost of recording a single trace sample with the cost of a few other basic operation. Let us note that, from [MS04], the TAU measurement overhead per (flat) event is about 1400 cycles on a INTEL XEON processor.

We also measured the overhead and the size of generated traces. These two values depends on the instrumentation level and on the application. Here we have considered three instrumentation levels: no instrumentation, scheduling instrumentation and complete instrumentation (where system calls and every function of the application and of the hybrid-thread library MARCEL are traced). It is worth noting that there is only one binary instrumented code: the degree of instrumentation is selected through to the global variable `fkt_active`.

The `Sumtime` program is a kind of torture test for the hybrid-thread library: it recursively builds a complete binary tree of threads for a given height. As a matter of fact, this program spends most of its execution time in creation, synchronization and destruction of user-level threads. Hence highly frequent scheduling events have to be recorded. This leads to 23% overhead for scheduler degree instrumentation and 80% for a complete instrumentation. This is the worst case, clearly this is not the best way to perform a performance analyze of this kind of program, however the information harvest may prove useful for debugging purpose. The second program is a multithreaded direct solver for sparse systems of linear equations based on the library SUPERLU. [DGL99]. As there is a lot of computation within threads, the overhead of the instrumentation becomes quite reasonable.

4 Conclusion

Hybrid-thread scheduling's approach allows to efficiently exploit SMP architecture, because basic operations on threads are efficient and several user-level threads of a given application can run in a true parallel way. However, analyzing the performance of such programs is delicate, mainly because some events occur within the kernel and others occur in user space. Thus, instrumentation of these programs has to be carried out at both levels. Our environment allows to instrument a multithreaded program in order to conduct a precise analysis of a run. It avoids the introduction of synchronization points or system calls during the execution, including basic thread operations like creation, destruction and synchronization.

Our toolkit is available on SMP INTEL x86 / ITANIUM architectures, LINUX and the hybrid-thread library MARCEL. The required modifications of the LINUX kernel and of library sources are localized. Therefore thread libraries like NGPT, NPTL or LINUXTHREAD can be easily adapted. The toolkit port onto other CPU architectures relies on the availability of an instruction similar to `cmpxchgl` (which usually exists on modern processors) and an accurate and CPU synchronized clock (such as cycle counter registers).

We are currently porting our toolkit on NUMA machines where cycle counter registers are *nearly* synchronized, only. To deal with this we have to introduce calibration steps. Other interesting improvements include to record performance counter registers and to translate our traces in other trace format like VAMPIRE's one.

References

[ABLL91] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler Activations: Efficient kernel support for the user-level managment of parallelism. In *Proc. 13th ACM Symp. on Operating Systems Principles (SOSP 91)*, pages 95–105, October 1991.

[ABMN02] O. Aumage, L. Bougé, J.-F. Méhaut, and Raymond Namyst. Madeleine II: A portable and efficient communication library for high-performance cluster computing. *Parallel Computing*, 28(4):607–626, April 2002.

[BDN02] Luc Bougé, Vincent Danjean, and Raymond Namyst. Improving Reactivity to I/O Events in Multithreaded Environments Using a Uniform, Scheduler-Centric API. In *Euro-Par 2002*, volume 2400 of *LNCS*, pages 605–614, Paderborn, Germany, August 2002.

[DGL99] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Anal. Appl.*, 20(4):915–952, 1999.

[dKdOS00] J. Chassin de Kergommeaux and B. de Oliveira Stein. Pajé: an extensible environment for visualizing multi-threaded programs executions. In *Proceedings of the 6th International EuroPar Conference*. EuroPar2000, 2000.

[DN03] Vincent Danjean and Raymond Namyst. Controling Kernel Scheduling from User Space: an Approach to Enhancing Applications' Reactivity to I/O Events. In *HiPC '03*, volume

2913 of *LNCS*, pages 490–499, Hyderabad, India, December 2003. Held in conjunction with IEEE Computer Society and ACM, Springer-Verlag.

[DNR00] Vincent Danjean, Raymond Namyst, and Robert Russell. Integrating Kernel Activations in a Multithreaded Runtime System on Linux. In *(RTSPP '00*, Lect. Notes in Comp. Science, Cancun, Mexico, May 2000. Springer-Verlag.

[MCC+95] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, 1995.

[MS00] Allen D. Malony and Sameer Shende. Performance Technology for Complex Parallel and Distributed Systems. In *Distributed and parallel systems: from instruction parallelism to cluster computing*, pages 37–46. Kluwer Academic Publishers, 2000.

[MS04] Allen D. Malony and Sameer S. Shende. Overhead Compensation in Performance Profiling. In *Proc. Europar 2004 Conference*. LNCS, 2004.

[NM95] Raymond Namyst and Jean-François Méhaut. PM2: Parallel Multithreaded Machine. A computing environment for distributed architectures. In *Parallel Computing (ParCo '95)*, pages 279–285. Elsevier Science Publishers, September 1995.

[RC01] Robert D. Russell and Mrinalini Chavan. Fast Kernel Tracing: a Performance Evaluation Tool for Linux. In *Proc. 19th IASTED International Conference on Applied Informatics (AI 2001)*. IASTED, February 2001.

[She01] S.S. Shende. *The Role of Instrumentation and Mapping in Performance Measurement.* PhD thesis, University of Oregon, August 2001.

[Thi03] Samuel Thibault. Developping a software tool for precise kernel measurements. Master's thesis, University of New Hampshire, 2003.

[TM99] Ariel Tamches and Barton P. Miller. Using dynamic kernel instrumentation for kernel and application tuning. *The International Journal of High Performance Computing Applications*, 13(3):263–276, Fall 1999.

[XMN99] Zhichen Xu, Barton P. Miller, and Oscar Naim. Dynamic instrumentation of threaded applications. In *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 49–59. ACM Press, 1999.

[YD00] Karim Yaghmour and Michel R. Dagenais. Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. In *Proceeding of the 2000 USENIX Annual Technical Conference*, June 2000.

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399