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

Vincent Danjean, Raymond Namyst, and Pierre-André Wacrenier

LaBRI / INRIA-Futurs, Université Bordeaux 1
351, cours de la Libération
33405 Talence Cedex, France

Abstract. Nowadays, observing and understanding the behavior and performance of a multi-threaded application is a nontrivial task, especially within a complex multi-threaded environment such as a multi-level thread scheduler. In this paper, we present a trace toolkit that allows programmers to precisely analyze the behavior of a multi-threaded application. Running an application through this toolkit generates several traces which are merged and analyzed offline. The resulting *super-trace* contains not only classical information but also detailed informations about thread scheduling at multiple levels.

1 Introduction

Bottleneck analysis, deadlock debugging, and performance understanding are tasks which require a fine-grain analysis of the behavior of a parallel application. The problem becomes even more tricky when dealing with multi-level multi-threading applications. Let us recall that there are three main families of threads: *User-level threads* are managed by the application, offer efficient basic operations and, most importantly, can be tailored to the particular requirements of the application; however as the operating system knows nothing about these threads, they have the disadvantage of not being able to use all available system resources, especially multi-processors resources. *Lightweight processes* (also called LWPs or kernel-level threads) are managed by the kernel and have access to kernel resources. 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 tend to incur a bigger overhead since all LWP scheduling and switching tasks require a kernel intervention. *Hybrid threads* (multi-level threads) were introduced 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 themselves manage user-level threads in a distributed fashion. Although the implementation of this scheme within an operating system is very complex [1], hybrid threads offer significant performance benefits with high performance parallel applications involving only few I/O operations [2].

* This work has been supported by the *ACI Masse de données & ACI Grid*

Analyzing the scheduling of a multi-threaded application executed by an hybrid-thread system, and observing the behavior of a such application in its global context are difficult tasks which require 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 multi-level instrumentation. In this framework, we may notice the work of Shende [3] who has defined a strategy using multi-level 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 multi-threaded environment [4], each thread creation is recorded into a TAU's performance database (this requiring mutual exclusion with other threads) in order to create a per-thread performance data structure.

In [5], Xu *et al* use the dynamic environment PARADYN [6] to profile multi-threaded applications through statistics. In their approach, each thread has its own private copy of some performance counters or timers; locks are used to access the minimal set of global book-keeping data structures.

However, in the framework of the parallel environment PM^2 [2] which is based on an hybrid-thread library, our goal is to debug and to optimize low-level middlewares, such as a reactive communication library [7, 8], and tricky mechanisms like scheduler activations [9, 10]. To that effect, it is important to consider aspects such as lock mechanisms and interruption handler routines. 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 minimize interdependent intrusion effects[1]. In that respect, we have defined a lightweight multi-level instrumentation toolkit which aims to precisely trace the behavior of a multi-threaded program. In order to be efficient, this toolkit has to meet the following requirements:

To be the less 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 (which would be the result e.g of an excessive increasing of the execution time of a critical section, some new synchronization points or some new context switch points). Therefore, system calls and high-level synchronization mechanisms must be avoided.

To deal with multi-level instrumentation. Since our goal is to study multi-level schedulers and/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 in order to get a complete knowledge of the scheduling of a multi-threaded application.

To deal with a huge amount of data. The toolkit may need to record a lot of events such as scheduler decisions, starting and termination points of functions

[1] Note that, Malony *et al* [11] have shown that while it is possible to compensate overhead due to the intrusion in a single process application, parallel overhead compensation is a more complex problem because of interdependent intrusion effects.

executed by a thread or by the kernel. This may generate several mega bytes of data per second.

In this paper, we propose a solution based on two independent buffer traces: the first one containing kernel-level events, the second one containing user-level events. We will first present the FAST KERNEL TRACE toolkit which is the basis of our work. Then we will justify our approach and give some technical details. Finally, we will analyze the introduced overhead on two applications.

2 From Kernel Tracing to Multi-level Tracing

2.1 The Fast Kernel Traces (FKT) Toolkit

Kernel instrumentation may be done at compilation time [12, 13] or dynamically at run-time like in KERNINST [14]. It is worth noting that operating systems such as LINUX 2.6.10 and SOLARIS 10 (DTRACE) already provide a dynamic instrumentation toolkit which allows to instrument the running operating system kernel. For our purpose, we chose to use the `FKT` toolkit [13] 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 tracing call requires to recompile the source code and to restart the kernel. Nevertheless, basic operations such as **`tracing start`**, **`tracing stop`** or **`tracing store`** can be executed from the user-level space. It is worth knowing that FKT uses a well-optimized storage mechanism [15] which allows to use TLB mechanisms to directly write buffer's pages on the disk, avoiding useless memory copy and limiting memory consumption.

```
#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)
```

Fig. 1. A definition of a `FKT` macro for an event with two parameters.

Figure 1 shows the details of 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. `P1` and `P2` are two integer arguments left to the programmer (it is possible to record up to five integer arguments).

2.2 Meeting Hybrid Scheduling's Requirements

In order to precisely rebuild the behavior of multi-threaded programs, it is necessary to be able to determine at any time the current running user-level threads on the SMPs. Note that a kernel view is insufficient: indeed, the kernel has no knowledge about user-level threads which are scheduled by the LWP pool. On the other side, an user-space's view is also insufficient: LWPs are usually unaware of kernel's context switches, so it is difficult, from the user-space point of view, to get the identifiers of the running LWPs at a given date and to get the processor identifier on which the user code is running. To solve these problems, new system calls might be created to request the identifier of the processor which is recording an event, for instance, or to notify the kernel scheduler about the user-level scheduler's context switches. However, such a solution is too intrusive: system calls are expensive (see micro benchmarks given in Section 3.3) and, moreover, this solution would introduce a higher number of context-switch points than the uninstrumented execution would encounter. Another solution would be to define a mechanism based on *up-calls*: in order to transmit the kernel view to the user-space level, the kernel forces 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 each up-call.

Our proposition is to generate a trace from both point of views. The kernel's trace will be generated by FKT and the user-level trace 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, Kernel- and user-level traces are both necessary to get a full description of a multi-threaded 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 other registers at the hardware level. (3) All context switches (user's and kernel's) are recorded, so that we will be able to deduce what happens from a scheduling point of view within the system.

After the execution of the application, 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, the LWP and the processor which executed the recording. By reconciling the kernel- and the user-level sides, this toolkit allows to trace multi-threaded applications and, moreover, it allows to put the application run back into its execution context, as any kernel event may be recorded. Hence it is possible to get an accurate analysis of low-level middlewares such as a multi-threaded communication library.

2.3 Description of the Tracing Toolkit

The multi-level tracing toolkit FUT has been implemented on top of the Marcel/Linux/x86 system which is the hybrid-thread library of the portable parallel environment PM^2 [2].

In order to instrument the kernel, users need to apply a given patch against the Linux kernel. This patch introduces instrumented points in the kernel code allowing to record events such as context switches, starting/termination points of hardware interruption (IRQ) and software interruptions (system calls). The thread library Marcel is instrumented in order to record user-thread scheduling decisions; for instance, events such as user-level context switches, creation and termination of LWPs are recorded. Moreover, this instrumentation allows to trace any function call of the library MARCEL. 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. A basic code instrumentation tool is also implemented in order to automatically add attributes to the starting/termination points of each function. The `PROF_IN()` and `PROF_OUT()` macros may be used to trace the call and the termination of a function. The code instrumentation may either be called directly by programmers or be inserted automatically by compilers, like GCC does.

Once both traces have been recorded, they are merged in a super-trace in which events are ordered with respect to the time-stamps. During the merge, the relationship between user-level events, user-level threads, LWPs and processors is established. However, some kernel events, such as those which are recorded during interruption routines, are not to be associated with any user-level thread.

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 filter events matching some criteria (a given kind of event, a given user-thread, a time-slice). Some basic measures may also be computed like, for instance, the (active) execution time of a given thread or the reactivity of the 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). Moreover, a specific filter has been developed to translate the super-trace format into the file format of `Pajé` [16], a generic graphic trace viewer.

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

3 Implementation Details and Performance Analysis

We are addressing in this section some technical issues we encountered in order to limit the intrusion of the tracing mechanisms. We will first detail the time-stamping, the trace format and the concurrent recording mechanism. Then we will discuss about the overhead introduced by the instrumentation.

3.1 About the Time-Stamping and the Trace Format

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

```
$> 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 identifier
`cpu`: processor identifier – `thr`: user-thread level identifier
`code`: event code – `name`: event name – `param(s)`: associated parameter values

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

Fig. 2. Super-trace analysis using sigmund.

Note that only 32 bits are required to stamp the kernel events. Indeed, from the first recording of the cycle register, there is enough kernel events (such as kernel scheduling decisions or clock interruptions) that are recorded during a defined 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 for several seconds.

In order to limit the intrusiveness, event buffers are created and initialized before the real launching of the application. An initial section containing context information (function names, running LWPs) is also recorded in both buffers. The size of the initial section 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 a single processor machine. Indeed, asynchronous events may be raised at any time and we do not want to try to block them in order to avoid interferences with the scheduler. 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 solved this problem using the *atomic* CPU instruction

```
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 multi-threaded applications. One can compute the elapsed time between the network message arrival (a hardware interrupt is raised by the network card) and the processing of this message by the appropriate user-level 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 process the incoming messages as soon as possible. In this program, the* `daemon` *thread executes a non-blocking system call*[a] `select()` *and calls* `pthread_yield()` *to yield its execution in favor of another thread when no message is available.*
Here the considered algorithm leads to very bad latencies, as the `daemon` *thread has to wait for all the other threads to use their quantum before getting active even though messages could have already been received by the OS. However, most of thread libraries do not provide any mechanism to deal efficiently with this kind of problem. A description of an adequate support within thread libraries to improve thread reactivity to external asynchronous events can be found in [17].*

[a] blocking system call must be avoided when using user-level thread library.

Fig. 3. Using our mechanisms to observe thread's reactivity.

`cmpxchgl`. The idea is to store the buffer's length value in a register, then to store the new buffer's length in a second register and finally to call `cmpxchgl`

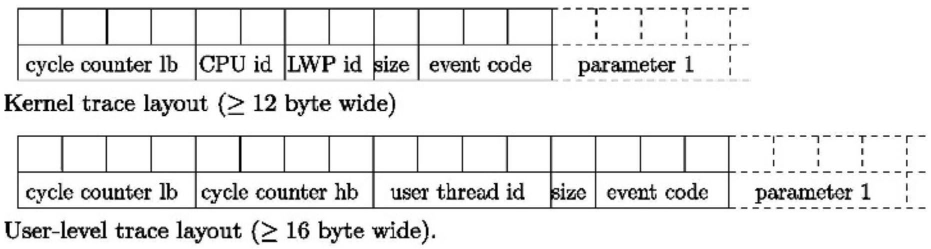

Fig. 4. Kernel and user-level trace entry layout.

in order to set the new buffer's length. This subroutine is repetitively called 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.

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 operations. Let us note that according to [11], the TAU measurement overhead per (flat) event is about 1400 cycles on a Xeon processor.

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

Function/Macro		cycles
Macro	`PROF_IN`	260
System call	`getpid()`	1900
buffered io	`printf(‘‘test’’)`	672

We also measured the overhead and the size of generated traces. These two values depend 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 all the functions of the application and of the hybrid-thread library Marcel are traced). It is worth noting that there is no need to recompile the source code: the degree of instrumentation is defined through the use of the global variable `fkt_active`.

The `Sumtime` program can be seen as a 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 creating, synchronizing and destructing user-level threads. Hence highly frequent scheduling events have to be recorded. This leads to a 23% overhead for the scheduler-level instrumentation and a 80% overhead for a complete instrumentation. This is the worst case, clearly this is not the best way to analyze the performances of our toolkit, however the gathered information may prove to be useful for debugging purposes. The second program is a multi-threaded direct solver for sparse

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

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

systems of linear equations based on the library SuperLU [18]. 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, as 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 some others in user space. Thus, instrumentation of these programs has to be carried out at both levels. Our toolkit allows to instrument a multi-threaded program in order to conduct a precise analysis of executions of this program. It avoids the introduction of synchronization points or system calls during the execution, including basic thread operations such as creation, destruction and synchronization.

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

We are currently implementing our toolkit on NUMA machines where cycle counter registers are *nearly* synchronized. To deal with this problem, we have to introduce calibration steps. Some other interesting improvements include the recording of the performance of the counter registers and the translation of our trace format into other trace format such as Vampir.

References

1. Sun microsystems: Multithreading in the solaris operating environment. `http://www.sun.com/software/whitepapers/solaris9/multithread.pdf` (2002)
2. Namyst, R., Méhaut, J.F.: PM2: Parallel Multithreaded Machine. A computing environment for distributed architectures. In: Parallel Computing (ParCo '95), Elsevier Science Publishers (1995) 279–285
3. Shende, S.: The Role of Instrumentation and Mapping in Performance Measurement. PhD thesis, University of Oregon (2001)
4. Malony, A.D., Shende, S.: Performance Technology for Complex Parallel and Distributed Systems. In: Distributed and parallel systems: from instruction parallelism to cluster computing, Kluwer Academic Publishers (2000) 37–46
5. Xu, Z., Miller, B.P., Naim, O.: Dynamic instrumentation of threaded applications. In: PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming, ACM Press (1999) 49–59
6. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., Newhall, T.: The paradyn parallel performance measurement tool. Computer **28** (1995) 37–46
7. Aumage, O., Bougé, L., Méhaut, J.F., Namyst, R.: Madeleine II: A portable and efficient communication library for high-performance cluster computing. Parallel Computing **28** (2002) 607–626
8. Danjean, V., Namyst, R.: Controling Kernel Scheduling from User Space: an Approach to Enhancing Applications' Reactivity to I/O Events. In: HiPC '03. Volume 2913 of LNCS., Hyderabad, India, Springer-Verlag (2003) 490–499
9. Anderson, T., Bershad, B., Lazowska, E., Levy, H.: Scheduler Activations: Efficient kernel support for the user-level managment of parallelism. In: Proc. 13th ACM Symp. on Operating Systems Principles (SOSP 91). (1991) 95–105
10. Danjean, V., Namyst, R., Russell, R.: Integrating Kernel Activations in a Multithreaded Runtime System on Linux. In: (RTSPP '00. Lect. Notes in Comp. Science, Cancun, Mexico, Springer-Verlag (2000)
11. Malony, A.D., Shende, S.S.: Overhead Compensation in Performance Profiling. In: Proc. Europar 2004 Conference, LNCS (2004)
12. Yaghmour, K., Dagenais, M.R.: Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. In: Proceeding of the 2000 USENIX Annual Technical Conference. (2000)
13. Russell, R.D., Chavan, M.: Fast Kernel Tracing: a Performance Evaluation Tool for Linux. In: Proc. 19th IASTED International Conference on Applied Informatics (AI 2001), IASTED (2001)
14. Tamches, A., Miller, B.P.: Using dynamic kernel instrumentation for kernel and application tuning. The International Journal of High Performance Computing Applications **13** (1999) 263–276
15. Thibault, S.: Developping a software tool for precise kernel measurements. Master's thesis, University of New Hampshire (2003)
16. de Kergommeaux, J.C., de Oliveira Stein, B.: Pajé: an extensible environment for visualizing multi-threaded programs executions, EuroPar2000 (2000)
17. Bougé, L., Danjean, V., Namyst, R.: Improving Reactivity to I/O Events in Multithreaded Environments Using a Uniform, Scheduler-Centric API. In: Euro-Par 2002. Volume 2400 of LNCS., Paderborn, Germany (2002) 605–614
18. Demmel, J.W., Gilbert, J.R., Li, X.S.: An asynchronous parallel supernodal algorithm for sparse gaussian elimination. SIAM J. Matrix Anal. Appl. **20** (1999) 915–952