Last results using the Java Instrumentation Suite

Jordi Guitart, Jordi Torres, Eduard Ayguadé, José Oliver and Jesús Labarta

European Center for Parallelism of Barcelona (CEPBA)
Computer Architecture Department, Technical University of Catalonia
C/ Jordi Girona 1-3, Campus Nord UPC, Mòdul C6, E-08034 Barcelona (Spain)

{jguitart, torres, eduard, joseo, jesus}@ac.upc.es

Abstract. This report presents the Java Instrumentation Suite (JIS), a set of tools designed to instrument Java threaded applications using dynamic code interposition (avoiding the instrumentation and recompilation of the source code and/or the Java Virtual Machine JVM). The report describes the design of JIS and highlights the results obtained using JIS for improving threaded applications performance.

1. Introduction

The rapid maturing process of the Java technology is encouraging users the development of portable applications using the Java language. As an important part of the definition of the Java language, the use of threads is becoming commonplace when programming this kind of applications. Understanding and tuning threaded applications requires the use of effective tools for detecting possible performance bottlenecks.

The support for this multithreaded environment requires, for each platform, an efficient mapping of Java threads to the underlying system threads. Thread support traverses many layers: from the application, through various Java classes (e.g., Thread and ThreadGroup), then through the platform-independent layer, the platform-specific layer, and finally to the operating system itself. While this layering provides a clean separation of responsibilities and allows many platforms to "plug in" their support, it also contributes to extra overhead.

The Java Instrumentation Suite (JIS) enables a detailed analysis of the application behavior by recording the state of each thread along the execution of the application. The instrumentation is done using dynamic code interposition at the JVM level (avoiding modifications in the source code and its recompilation).

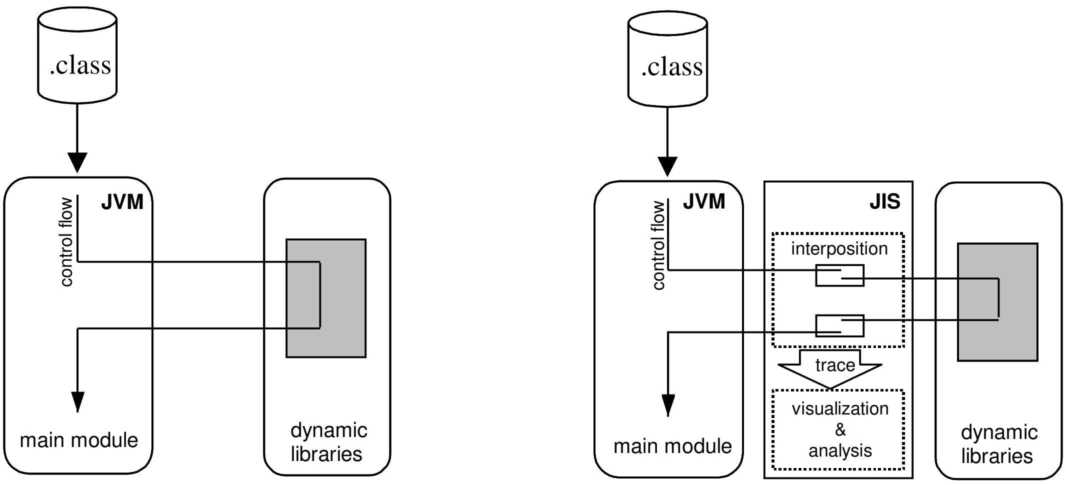
Current implementations of the JVM allow Java threads to be scheduled by the virtual machine itself (the so-called *green threads model*) or by the operating system (the so-called *native threads model*). When using green threads, the operating system does not know anything about threads that are handled by the virtual machine. From the point of view of the operating system, there is a single process and a single thread; it is up to the virtual machine to handle all the details of the threading API. In the native threads model, threads are scheduled by the operating system that is hosting the virtual machine. JIS allows the analysis of threaded applications running on these two execution models.

To our knowledge, it is the first tool that handles the Java thread scheduling representation of an application [GTAO00]. JIS is a part of Barcelona Java Suite (BJS) currently under development at CEPBA. The main purpose of BJS is to serve as platform for doing research on scheduling mechanisms and policies oriented towards the efficient execution of multithreaded Java applications on parallel servers.

The remaining of this report is as follows. Section 2 describes the basic components of JIS. Experimentation with JIS is reported in Section 3. The report is concluded in Section 4.

2. JIS implementation

JIS makes use of two components previously developed at the CEPBA: DITools [SeNC00] and Paraver [LGPC96]. DITools is a set of tools bringing an environment in which dynamically linked executables can be extended at runtime with unforeseen functionality. Code interposition is used in JIS to instrument the code of the JVM. References to specific functions of the dynamically linked libraries used by the JVM are captured and probed. This instrumentation does not require any special compiler support and makes unnecessary to rebuild neither the bytecode of the application nor the executable of the JVM. The probes executed with the intercepted calls generate a trace file with all the information that is required to analyze and visualize the execution of the threaded application. Paraver serves as the visualization tool. It allows the user to navigate through the trace and to do an extensive quantitative analysis. Figure 1 shows the interaction of the above mentioned components in JIS.



(a) (b)

Figure 1. Placement of the instrumentation

2.1. Trace contents

We use traces in order to analyze the behavior of threaded applications. These traces reflect the activity of each Java thread in the application (through a set of predefined states that are representative of the parallel execution) and collect the occurrence of some predefined events along the whole application lifetime. Table 1 summarizes the different states that we consider for a thread.

The trace can also contain events that provide additional information about the behavior of the threaded application. The following events could be recorded in the trace:

- Information relative to the monitor in which a thread is blocked (either in the queue of threads waiting to enter into the monitor or in the queue of threads waiting in this monitor).

Some of these monitors are registered by the JVM, in which case the event reports the name; otherwise, it reports a numerical identifier.

- Information relative to actions performed by a thread, like for instance processor yield, entry (exit) to (from) code protected by a mutual exclusion, notify on a monitor (to awake a previously blocked thread on that monitor for a wait), among others.
- And in general, the invocation of any method in the Java Thread API, like for instance sleep and interrupt.

STATE	**DESCRIPTION**
INIT	Thread is being created and initialized
READY	Thread is ready for running, but there is no processor available for it
RUN	Thread is running
BLOCKED IN CONDVAR	Thread is blocked waiting on a monitor
BLOCKED IN MUTEX	Thread is blocked waiting to enter in a monitor
STOPPED	Thread has finalized

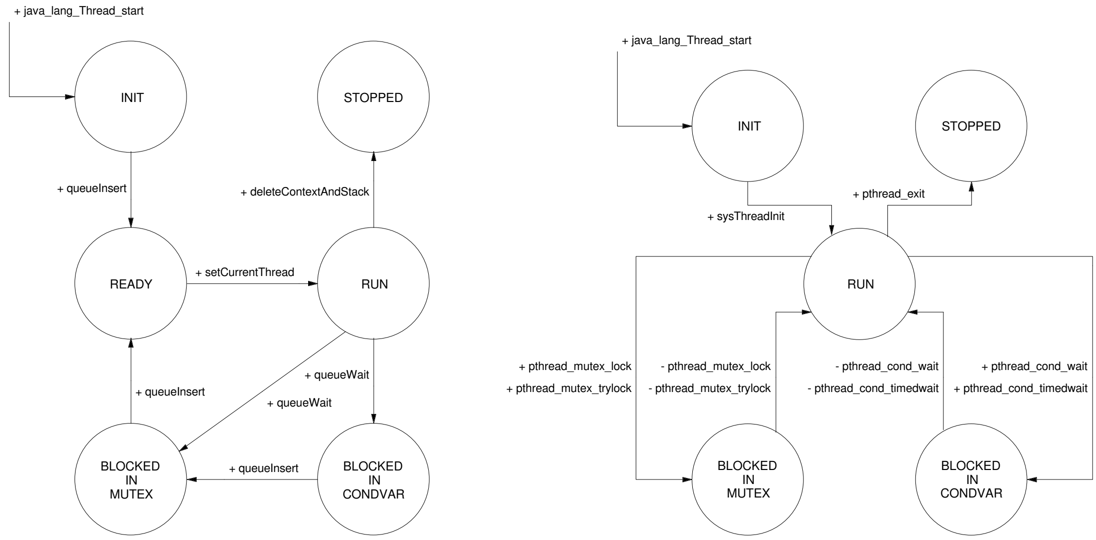
Table 1. Thread states

2.2. Code interposition

Dynamic linking is a feature available in many modern operating systems. Program generation tools (compilers and linkers) support dynamic linking via the generation of linkage tables. Linkage tables are redirection tables that allow delaying symbol resolution to run time. At program loading time, a system component fixes each pointer to the right location using some predefined resolution policies. Usually, the format of the object file and these data structures are defined by the system Application Binary Interface (ABI). The standardization of the ABI makes possible to take generic approaches to dynamic interposition.

The monitoring methodology is based on the fact that the JVM invokes a set of run-time services at key places in order to use threads or to synchronize them. The required knowledge about the execution environment can be expressed using a state transition graph, in which each transition is triggered by a procedure call and/or a procedure return. Figure 2 presents the state transition graph for application threads for both execution models (green and native threads) considered in this work, in which nodes represent states, and edges correspond to procedure calls (indicated by a + sing) or procedure returns (indicated by a - sign) causing a state transition. We can also monitor internal threads of the JVM (like the finalizer, garbage collector, ...). In this case, the invocation of other procedures (like `InitializeFinalizerThread`, `InitializeGCThread`, ...) are used to trigger the transition to the `INIT` state instead of the invocation of procedure `java_lang_Thread_start`.

This transition graph is then used to derive the interposition routines used to keep track of the state in the performance monitoring backend. These routines are simple wrappers of functions that change thread state, emit an event and/or save thread information in the internal structures of JIS. These wrappers can perform instrumentation actions before and/or after the call being interposed. Some representative procedure wrappers for green and native models are shown in Appendix B.



(a) ***(b)***

Figure 2. State transition graph for (a) green threads and (b) native threads

The interposition of other functions leads to the generation of an event in the trace and/or to an update of the internal structures of the instrumentation system (e.g. `pthread_mutex_unlock`, `pthread_cond_signal`, `pthread_cond_broadcast`, `sched_yield`). These functions do not provoke a direct change in the state of any other thread.

2.3. Instrumentation library

For each action being traced, the tracing environment internally finds which Java thread made the call and the time at which it was done. Timestamps associated to transitions and events are obtained using platform-specific mechanisms (for instance the high-resolution memory-mapped clock) offered by the system.

This data is written (optionally with additional parameters) to an internal buffer for each thread (i.e. there is no need for synchronization locks or mutual exclusion inside the parallel tracing library). The data structures used by the tracing environment are also arranged at initialization time in order to prevent interference among threads (basically, to prevent false sharing). The user can specify the amount of memory used for each thread buffer. When the buffer is full, the runtime system automatically dumps it to disk.

The instrumentation library provides the following services:

- `initlib` - Initialize the library internal data structures to start a parallel trace receiving as parameters: 1) the maximum number of threads participating in the execution, 2) the maximum amount of memory that the library has to reserve for each thread buffer, and 3) the mechanism used to obtain timestamps.
- `changestate` - Change the state of a thread.
- `pushstate` - Store the current state of a thread in a private stack and change to a new one.
- `popstate` - Change the state of a thread to the one obtained from the private stack.
- `userevent` - Emit an event (identifier and associated value) for a thread.

- `closelib` - Stop the tracing; this call makes the library dump to disk all buffered data not yet dumped and write resulting sorted trace to a file.

The library also offers combined services to change the state and emit an event: `changeandevent`, `pushandevent` and `popandevent`.

Some representative functions performed in procedure wrappers, which use these library services in order to generate trace information, are shown in Appendix C for green and native models.

3. Application analysis with JIS

In this section, we highlight some conclusions drawn from our initial experimentation with JIS. The main idea is to show the usefulness of the tool in both analyzing the behavior of a threaded application, understanding the behavior of the JVM itself and analyze the impact of some improvements done at any level of the Java execution environment. The current implementation targets the SGI Origin 2000 architecture with MIPS R10000 processors at 250 MHz running the JVM version 3.1.1 (Sun Java 1.1.6). One program has been used for the purposes of this report: LUAppl (a LU reduction kernel over a two-dimensional matrix of double-precision elements taken from [OlAN00]). A most complete explanation of JIS possibilities analyzing threaded applications can be found at [GTAO00].

JIS offers different views of the application. On one side, JIS allows a global view of the application, which allow us to recognize, for example, the execution paradigm (master/slave, work queuing, ...), the application concurrency and/or parallelism degree, ... On another side, JIS allows a more detailed analysis (timing) of the behavior and to detect, among other things, load unbalancing at the thread level or critical points that may create some possible bottlenecks.

Figure 3 shows a Paraver window in which the behavior of the LUAppl application using native threads is shown. Notice that the MAIN thread acts as a master thread that creates four slave threads (USER4 – USER7) along the application lifetime. At this specific point of the trace, each one of the slave threads is executing a subset of the total number of iterations for one of the loops that compose the LU reduction.

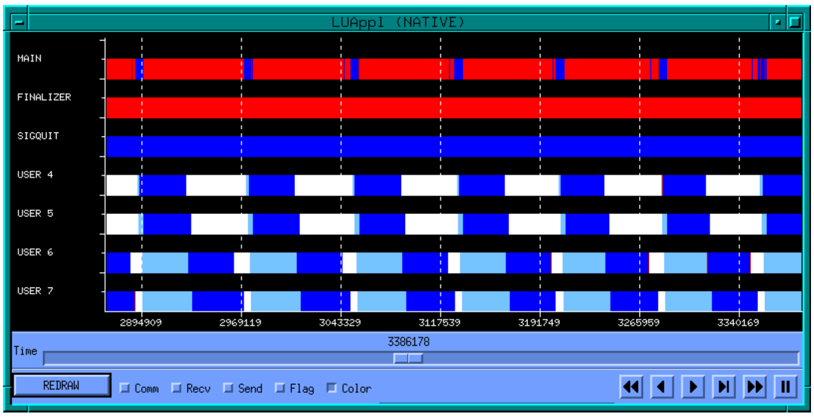


Figure 3. Visualization of the LUAppl running with native threads

3.1. Concurrency level

The analysis at this level reveals a possible inefficiency in the threaded execution of this part of the application: the system is unable to exploit the parallelism that exists among the four chunks generated for each loop. This observation was used to conduct further investigation in the implementation of the JVM. The conclusion from this investigation was to detect that the JVM does not inform the underlying threads implementation about the desired concurrency level of the application. By default, the threads library adjusts the level of concurrency itself as the application run. In the previous execution, only two kernel threads where used to execute Java threads USER4 – USER7. In order to give the library a hint about the concurrency level of the application, we automatically introduce an invocation to the `pthread_setconcurrency(level)` service of the threads library. Argument `level` is used to compute the ideal number of kernel threads for scheduling the available Java threads. Figure 4 shows the execution trace after setting `level` to the maximum concurrency degree of the application. Notice that the performance of the application noticeably improves after setting this value.

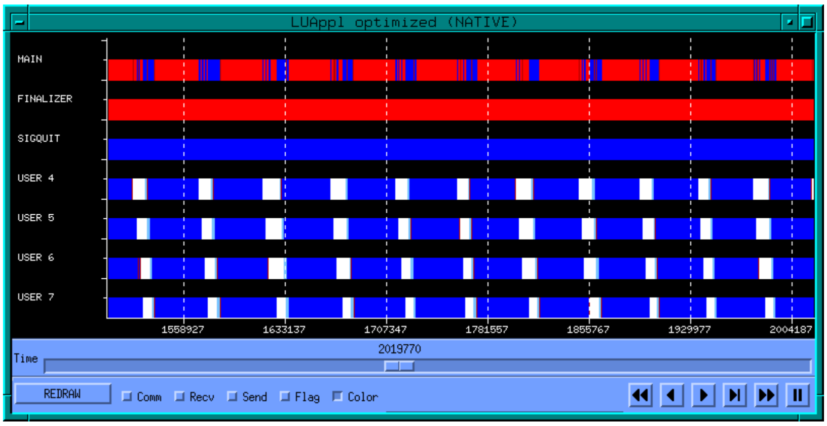


Figure 4. LUAppl running with native threads after setting the concurrency level

Table 2a shows the performance improvement of the original LUAppl application running with native threads after setting the concurrency level. We can see that improvements range from 23% to 49%. Table 2b shows the performance improvement of the instrumented LUAppl application running with native threads after setting the concurrency level. We can see that improvements range from 19% to 49%.

	original	**pthread_setconcurrency**	**% improvement**
64x64	900	689	23,44 %
128x128	4430	2795	36,91 %
256x256	42230	17542	58,46 %
512x512	214656	109976	48,77 %

(a) pthread_setconcurrency vs original

	original + instrumented	pthread setconcurrency + instrumented	% improvement
64x64	1015	824	18,82 %
128x128	4713	2996	36,43 %
256x256	32844	17975	45,27 %
512x512	215790	110857	48,63 %

(b) pthread_setconcurrency + instrumented vs original + instrumented

Table 2. Performance improvement using pthread_setconcurrency call (execution times in ms)

The invocation of this function is done with no modifications in the original source code; instead it is automatically performed as part of the code executed when the instrumentation library is dynamically loaded. This is an example that shows the use of the dynamic interposition mechanism to add (or replace) functionalities to (from) an existing implementation of the JVM.

3.2. Instrumentation overhead

The instrumentation process of JIS introduces some overhead during the execution of the application. Once the application is finished, the instrumentation library joins the per-thread buffers into a single trace (ordered in time) suitable for visualized with Paraver. This adds an extra overhead to the whole execution time of the job that does not have any impact in the trace.

For the LUAppl application, Table 3 shows the overhead introduced when using green threads when using 4 threads. In this case, we can see that, in the worst case, the overhead is smaller than 3%.

	original	instrumented	% overhead
64x64	911	934	2,52 %
128x128	7214	7290	1,05 %
256x256	60680	61095	0,68 %
512x512	414913	417265	0,57 %

Table 3. Instrumentation overhead for LUAppl running with green threads (execution times in ms)

Table 4 shows the overhead introduced when using native threads when using 4 threads. We have measured this overhead in two ways: the first (see Table 4a), comparing the execution time of original LUAppl application with the execution time of this benchmark instrumented. In this case, we can see that, in the worst case, the overhead is smaller than 13%. Notice than in one case (256x256), the instrumented version runs much faster than the original one (because the concurrency level adjusted by pthreads is better). The second way (see Table 4b), comparing the execution time of LUAppl application optimized with pthread_setconcurrency call with the execution time of the same program instrumented. In this case, we can see that, in the worst case, the overhead is smaller than 20%.

	original	instrumented	% overhead
64x64	900	1015	12,78 %
128x128	4430	4713	6,39 %
256x256	42230	32844	-22,23 %
512x512	214656	215790	0,53 %

(a) instrumented vs original

	pthread_setconcurrency	pthread setconcurrency + instrumented	% overhead
64x64	689	824	19,59 %
128x128	2795	2996	7,19 %
256x256	17542	17975	2,47 %
512x512	109976	110857	0,80 %

(b) pthread_setconcurrency + instrumented vs pthread_setconcurrency

Table 4. Instrumentation overhead for LUAppl running with native threads (execution times in ms)

4. Conclusions

In this report we have presented the design of JIS (Java Instrumentation Suite), a set of tools designed to analyze the behavior of threaded applications running on the JVM. Our main objective with this report has been to make an exhaustive collection of all the measurement results obtained while evaluating JIS, and give an idea of its internal implementation.

Acknowledgments

We acknowledge the European Center for Parallelism of Barcelona (CEPBA) for supplying the computing resources for our experiments. Special thanks are to Albert Serra who is the heart in the development of the DITools. Ministry of Education of Spain has supported this work under contracts TIC98-511 and TIC97-1445CE.

References

[GTAO00] J. Guitart, J.Torres, E. Ayguadé, J. Oliver and J. Labarta. Java Instrumentation Suite: Accurate Analysis of Java Threaded Applications. Proc.of the Second Annual Workshop on Java for High-Performance Computing,ICS00. St. Fe, New Mexico, May 7, 2000.

[LGPC96] J. Labarta, S. Girona, V. Pillet, t. Cortés and L. Gregoris. DiP: A Parallel Program Development Environtment. In proceedings of the 2th. Int. Euro-Par Conference. Lyon (France). August 1996.

[OlAN00] J. Oliver, E. Ayguadé and N. Navarro. Towards an efficient exploitation of loop-level parallelism in Java. In Proceedings of the ACM Java Grande Conference, St. Francisco (CA), June 2000.

[SeNC00] A. Serra, N. Navarro and T. Cortés. DITools: Application-level Support for Dynamic Extension and Flexible Composition. In Proceedings of the USENIX Annual Technical Conference, June 2000.

Appendix A. Complete evaluation results

A1. Green threads

- **64 x 64 (4 threads)**

ORIGINAL	*INSTR*
912	934
913	936
914	936
911	936
912	935
912	935
915	935
917	935
915	937
915	936
913,6	935,5
911	934
	% INSTR vs ORIGINAL
	2,40
	2,52

- **128 x 128 (4 threads)**

ORIGINAL	*INSTR*
7214	7301
7219	7296
7215	7296
7214	7297
7216	7295
7231	7309
7223	7291
7229	7290
7226	7297
7224	7294
7221,1	7296,6
7214	7290
	% INSTR vs ORIGINAL
	1,05
	1,05

* *execution times in ms*

- **256 x 256 (4 threads)**

ORIGINAL	*INSTR*
60800	61268
60809	61209
60803	61149
60796	61160
60772	61140
60833	61146
60769	61125
60807	61141
60790	61111
60680	61095
60785,9	61154,4
60680	61095
	% INSTR vs ORIGINAL
	0,61
	0,68

- **512 x 512 (4 threads)**

ORIGINAL	*INSTR*
414913	417265
416299	417623
416126	417587
415727	417450
415320	418182
415057	418190
416171	418097
416312	418492
415626	417926
415349	417671
415690	417848,3
414913	417265
	% INSTR vs ORIGINAL
	0,52
	0,57

* *execution times in ms*

A2. Native threads

- **64 x 64 (4 threads)**

ORIGINAL	***INSTR***	***INSTR + PTHSC***	***PTHSC***
927	1016	851	715
931	1027	836	689
928	1022	830	730
923	1025	852	726
912	1019	851	729
915	1029	824	706
902	1015	841	696
900	1025	850	715
907	1017	854	729
923	1024	851	720
916,8	1021,9	844	715,5
900	1015	824	689
% INSTR vs ORIGINAL	***% INSTR + PTHSC vs INSTR***	***% PTHSC vs ORIGINAL***	***% INSTR + PTHSC vs PTHSC***
11,46	-17,41	-21,96	17,96
12,78	-18,82	-23,44	19,59

* *execution times in ms*

- **128 x 128 (4 threads)**

ORIGINAL	***INSTR***	***INSTR + PTHSC***	***PTHSC***
4465	4737	3027	2822
4452	4721	2996	2806
4499	4734	3037	2797
4520	4739	3021	2795
4430	4740	3025	2822
4466	4724	3035	2818
4475	4720	3015	2810
4431	4738	3024	2823
4485	4713	3000	2805
4511	4743	3021	2832
4473,4	4730,9	3020,1	2813
4430	4713	2996	2795
% INSTR vs ORIGINAL	***% INSTR + PTHSC vs INSTR***	***% PTHSC vs ORIGINAL***	***% INSTR + PTHSC vs PTHSC***
5,76	-36,16	-37,12	7,36
6,39	-36,43	-36,91	7,19

* *execution times in ms*

- **256 x 256 (4 threads)**

ORIGINAL	***INSTR***	***INSTR + PTHSC***	***PTHSC***
48442	33133	17985	17586
58838	32844	17994	17577
58428	33158	17975	17644
58466	33022	18086	17542
58439	32892	17988	17750
46248	33291	18017	17724
42230	33152	18061	17584
60767	33235	18096	17655
56555	33051	18046	17752
44785	32881	17998	17710
53319,8	33065,9	18024,6	17652,4
42230	32844	17975	17542
% INSTR vs ORIGINAL	***% INSTR + PTHSC vs INSTR***	***% PTHSC vs ORIGINAL***	***% INSTR + PTHSC vs PTHSC***
-37,99	-45,49	-66,89	2,11
-22,23	-45,27	-58,46	2,47

* *execution times in ms*

- **512 x 512 (4 threads)**

ORIGINAL	***INSTR***	***INSTR + PTHSC***	***PTHSC***
215714	216598	111791	110269
215244	216525	111518	110205
215116	217307	111581	110171
216187	216648	110857	110142
215827	216537	111349	110115
214656	216420	111693	110001
215053	216783	111725	109976
215755	215790	111637	110212
216222	217066	111902	110065
215478	216437	111780	110124
215525,2	216611,1	111583,3	110128
214656	215790	110857	109976
% INSTR vs ORIGINAL	***% INSTR + PTHSC vs INSTR***	***% PTHSC vs ORIGINAL***	***% INSTR + PTHSC vs PTHSC***
0,50	-48,49	-48,90	1,32
0,53	-48,63	-48,77	0,80

* *execution times in ms*

Appendix B

B1. Representative procedure wrappers for green threads

```
/*
 * Copyright (c) 2000
 * CEPBA - UPC.    All Rights Reserved.
 *   DAC - UPC.    All Rights Reserved.
 *
 * Programmer: Jordi Guitart Fernandez (jguitart@cepba.upc.es)
 * DITools Code Interposition (GREEN MODEL)
 *
 * Java Instrumentation Suite (JIS)
 * Barcelona Java Suite (BJS)
 */

/* wrapper for InitializeClock routine */
void InitializeClock_wrapper(void)
{
    set_current_init_thread(CLOCK);
    InitializeClock();
}

/* wrapper for InitializeGCThread routine */
void InitializeGCThread_wrapper(void)
{
    set_current_init_thread(GC);
    InitializeGCThread();
}

/* wrapper for java_lang_Thread_start routine */
void java_lang_Thread_start_wrapper(struct Hjava_lang_Thread * th)
{
    start_thread((long)th);
    java_lang_Thread_start(th);
}

/* wrapper for monitorRegister routine */
void monitorRegister_wrapper(sys_mon_t * mon, char * c)
{
    monitor_register((long)mon,c);
    monitorRegister(mon,c);
}
```

```
/* wrapper for queueInsert routine */
void queueInsert_wrapper(queue_t * q, sys_thread_t * th)
{
    queue_insert((long)q,(long)th);
    queueInsert(q,th);
}
/* wrapper for queueWait routine */
void queueWait_wrapper(sys_mon_t * mon, queue_t * q)
{
    queue_wait((long)mon,(long)q);
    queueWait(mon,q);
}

/* wrapper for setCurrentThread routine */
void setCurrentThread_wrapper(sys_thread_t * th)
{
    set_current_thread((long)th);
    setCurrentThread(th);
}

/* wrapper for sysThreadCreate routine */
int sysThreadCreate_wrapper(long l, uint_t flags, void *(*f)(void *),
sys_thread_t ** t, void * v)
{
    int ret;
    ret = sysThreadCreate(l,flags,f,t,v);
    start_sys_thread((long)v,(long)*t);
    return ret;
}
```

B2. Representative procedure wrappers for native threads

```
/*
 * Copyright (c) 2000
 * CEPBA - UPC.    All Rights Reserved.
 *   DAC - UPC.    All Rights Reserved.
 *
 * Programmer: Jordi Guitart Fernandez (jguitart@cepba.upc.es)
 * DITools Code Interposition (NATIVE MODEL)
 *
 * Java Instrumentation Suite (JIS)
 * Barcelona Java Suite (BJS)
 */
```

```
/* wrapper for InitializeFinalizerThread routine */
void InitializeFinalizerThread_wrapper(void)
{
    set_current_init_thread(FINALIZER);
    InitializeFinalizerThread();
}

/* wrapper for InitializeSigquitHandler routine */
void InitializeSigquitHandler_wrapper(void)
{
    set_current_init_thread(SIGQUIT_HANDLER);
    InitializeSigquitHandler();
}

/* wrapper for java_lang_Thread_start routine */
void java_lang_Thread_start_wrapper(struct Hjava_lang_Thread * th)
{
    start_thread((long)th);
    java_lang_Thread_start(th);
}

/* wrapper for pthread_cond_wait routine */
int pthread_cond_wait_wrapper(pthread_cond_t *cond, pthread_mutex_t *mutex)
{
    int ret;
    pthread_wait((long)cond,(long)mutex);
    ret = pthread_cond_wait(cond,mutex);
    pthread_cond_run();
    return ret;
}

/* wrapper for pthread_create routine */
int pthread_create_wrapper(pthread_t *thread, pthread_attr_t *attr, void
*(*start)(void *), void *arg)
{
    int ret;
    ret = pthread_create(thread,attr,start,arg);
    start_pthread((long)*thread,(long)arg);
    return ret;
}

/* wrapper for pthread_exit routine */
void pthread_exit_wrapper(void *retval)
{
```

```
    exit_thread();
    pthread_exit(retval);
}

int pthread_mutex_init_wrapper(pthread_mutex_t *mutex, const pthread_mutexattr_t
*attr)
{
    mutex_init((long)mutex);
    return(pthread_mutex_init(mutex,attr));
}

/* wrapper for pthread_mutex_lock routine */
int pthread_mutex_lock_wrapper(pthread_mutex_t *mutex)
{
    int ret;
    pthread_lock((long)mutex);
    ret = pthread_mutex_lock(mutex);
    pthread_mutex_run((long)mutex,TRUE);
    return ret;
}

/* wrapper for pthread_mutex_trylock routine */
int pthread_mutex_trylock_wrapper(pthread_mutex_t *mutex)
{
    int ret;
    pthread_lock((long)mutex);
    ret = pthread_mutex_trylock(mutex);
    if(ret != EBUSY)
        pthread_mutex_run((long)mutex,TRUE);
    else
        pthread_mutex_run((long)mutex,FALSE);
    return ret;
}

/* wrapper for sysThreadInit routine */
void sysThreadInit_wrapper(sys_thread_t * th, stackp_t st)
{
    init_thread((long)th);
    sysThreadInit(th,st);
}
```

Appendix C

C1. Representative Pre/Post functions for green threads

```
/*
 * Copyright (c) 2000
 * CEPBA - UPC.    All Rights Reserved.
 *   DAC - UPC.    All Rights Reserved.
 *
 * Programmer: Jordi Guitart Fernandez (jguitart@cepba.upc.es)
 * Java Tracing Routines Implementation (GREEN MODEL)
 *
 * Java Instrumentation Suite (JIS)
 * Barcelona Java Suite (BJS)
 */

/* set current initiating thread */
void set_current_init_thread(int rol)
{
    current_init_thread = rol;
    if(rol != APPLICATION)
    {
    thread_table[rol].ever_busy = TRUE;
    thread_table[rol].wait_monitor = 0;
    thread_table[rol].wait_queue = 0;
    if(rol != MAIN)
    {
        thread_table[rol].state = INIT;
        paraver_thread = rol;
        paraverChangeState(CPU_INIT);
    }
    else
        thread_table[rol].state = RUN;
    }
    else
    {
    paraver_thread = current_thread;
    paraverSetUserEvent(EVENT_APPLICATION,BEGIN);
    }
}

/* update JIS structures when an application Java thread starts */
void start_thread(long thread_id)
```

```
{
    long i;
    if(consult_AVL(java_tid_index,thread_id,&i))
    {
        fprintf(stderr,"JIS Error starting application Java thread: Thread %ld
        already exists\n",thread_id);
        exit(1);
    }
    if(!head_prio_queue(&free_threads,&i))
    {
        fprintf(stderr,"JIS Error starting application Java thread: Not enought
        space at JIS structures\n");
        exit(1);
    }
    if(!delete_head_prio_queue(&free_threads))
    {
        fprintf(stderr,"JIS Error starting application Java thread:
        delete_head_prio_queue(free_threads)\n");
        exit(1);
    }
    thread_table[i].ever_busy = TRUE;
    thread_table[i].thread_java_id = thread_id;
    thread_table[i].wait_monitor = 0;
    thread_table[i].wait_queue = 0;
    thread_table[i].state = INIT;
    paraver_thread = i;
    paraverChangeState(CPU_INIT);
    if(!insert_AVL(&java_tid_index,thread_id,i))
    {
        fprintf(stderr,"JIS Error starting application Java thread:
        insert_AVL(java_tid_index,%ld)\n",thread_id);
        exit(1);
    }
}

/* update JIS structures when a sys thread starts */
void start_sys_thread(long thread_java_id, long thread_sys_id)
{
    long i;
    if(!consult_AVL(java_tid_index,thread_java_id,&i))
    {
        /* system thread */
        thread_table[current_init_thread].thread_java_id = thread_java_id;
        thread_table[current_init_thread].thread_sys_id = thread_sys_id;
    if(!insert_AVL(&java_tid_index,thread_java_id,current_init_thread))
```

```
        {
            fprintf(stderr,"JIS Error starting sys thread:
            insert_AVL(java_tid_index,%ld)\n",thread_java_id);
            exit(1);
        }
        if(!insert_AVL(&sys_tid_index,thread_sys_id,current_init_thread))
        {
            fprintf(stderr,"JIS Error starting sys thread:
            insert_AVL(sys_tid_index,%ld)\n",thread_sys_id);
            exit(1);
        }
    }
    else
    {
        thread_table[i].thread_sys_id = thread_sys_id;
    if(!insert_AVL(&sys_tid_index,thread_sys_id,i))
        {
             fprintf(stderr,"JIS Error starting sys thread:
             insert_AVL(sys_tid_index,%ld)\n",thread_sys_id);
                exit(1);
        }
    }
}

/* update JIS structures when a new thread becomes the current thread */
void set_current_thread(long thread_id)
{
    long i,j;
    if(primer_set_current)
    {
    current_thread = MAIN;
    thread_table[MAIN].state = RUN;
    paraver_thread = MAIN;
    paraverChangeState(CPU_RUNNING);
    /* NOT necessary for MAIN thread (it is CPU_RUNNING by default) */
    primer_set_current = FALSE;
    }
    else
    {
    if(!consult_AVL(sys_tid_index,thread_id,&i))
    {
             fprintf(stderr,"JIS Error setting new current thread: Thread %ld
             doesn't exist\n",thread_id);
            exit(1);
    }
```

```
/* update state of current thread */
    paraver_thread = current_thread;
switch(thread_table[current_thread].state)
{
    case STOPPED:
         paraverChangeState(CPU_STOPPED);
         if(!delete_AVL(&java_tid_index,thread_table[current_thread].thread
         _java_id))
        {
             fprintf(stderr,"JIS Error setting new current thread:
             delete_AVL(java_tid_index,%ld)\n",thread_table[current_thread].
             thread_java_id);
           exit(1);
        }
         if(!delete_AVL(&sys_tid_index,thread_table[current_thread].thread_
         sys_id))
         {
                fprintf(stderr,"JIS Error setting new current thread:
                delete_AVL(sys_tid_index,%ld)\n",thread_table[current_thread
                ].thread_sys_id);
                exit(1);
         }
    if(!insert_prio_queue(&free_threads,current_thread))
        {
                fprintf(stderr,"JIS Error setting new current thread:
                insert_prio_queue(free_threads,%ld)\n",current_thread);
                    exit(1);
        }
    break;
        case BLOCKED_WAIT:
         if(!consult_AVL(mid_index,thread_table[current_thread].wait_monito
         r,&j))
        {
                fprintf(stderr,"JIS Error blocking thread in monitor:
                Monitor %ld doesn't exist\n",
                thread_table[current_thread].wait_monitor);
                    exit(1);
        }
           if(monitor_table[j].type != USER_MONITOR)

                paraverChangeStateAndUserEvent(CPU_BLOCKED_WAIT,FALSE,1,EVEN
                T_BLOCKED_IN_MONITOR,monitor_table[j].type);
           else

                paraverChangeStateAndUserEvent(CPU_BLOCKED_WAIT,FALSE,1,EVEN
                T_BLOCKED_IN_MONITOR,monitor_table[j].monitor_id);
    break;
        case BLOCKED_MUTEX:
```

```
            if(!consult_AVL(mid_index,thread_table[current_thread].wait_monito
            r,&j))
               {
                fprintf(stderr,"JIS Error blocking thread in monitor: Monitor
                %ld doesn't exist\n",
                thread_table[current_thread].wait_monitor);
                   exit(1);
               }
              if(monitor_table[j].type != USER_MONITOR)

                   paraverChangeStateAndUserEvent(CPU_BLOCKED_MUTEX,FALSE,1,EVE
                   NT_BLOCKED_IN_MONITOR,monitor_table[j].type);
            else

                   paraverChangeStateAndUserEvent(CPU_BLOCKED_MUTEX,FALSE,1,EVE
                   NT_BLOCKED_IN_MONITOR,monitor_table[j].monitor_id);
        break;
        }

    /* update state of new current thread */
    paraver_thread = i;
    paraverChangeState(CPU_RUNNING);
    thread_table[i].state = RUN;
    current_thread = i;
    }
}

/* update JIS structures when a registered monitor is created */
void monitor_register(long monitor_id, char * name)
{
    long i;
    if(consult_AVL(mid_index,monitor_id,&i))
    {
       fprintf(stderr,"JIS Error registering monitor: Monitor %ld already
       exists\n",monitor_id);
         exit(1);
    }
    if(!head_prio_queue(&free_monitors,&i))
    {
       fprintf(stderr,"JIS Error registering monitor: Not enought space at JIS
       structures\n");
         exit(1);
    }
    if(!delete_head_prio_queue(&free_monitors))
    {
       fprintf(stderr,"JIS Error registering monitor:
       delete_head_prio_queue(free_monitors)\n");
```

```
    exit(1);
}
monitor_table[i].monitor_id = monitor_id;
if(!strcmp(name,"Monitor registry"))
monitor_table[i].type = MONITOR_REGISTRY;
if(!strcmp(name,"Sbrk lock"))
    monitor_table[i].type = SBRK_LOCK;
if(!strcmp(name,"Alarm monitor"))
    monitor_table[i].type = ALARM_MONITOR;
if(!strcmp(name,"I/O monitor"))
    monitor_table[i].type = IO_MONITOR;
if(!strcmp(name,"Event monitor"))
    monitor_table[i].type = EVENT_MONITOR;
if(!strcmp(name,"Child death monitor"))
    monitor_table[i].type = CHILD_DEATH_MONITOR;
if(!strcmp(name,"Monitor IO lock"))
    monitor_table[i].type = MONITOR_IO_LOCK;
if(!strcmp(name,"rld lock"))
    monitor_table[i].type = RLD_LOCK;
if(!strcmp(name,"Malloc lock"))
    monitor_table[i].type = MALLOC_LOCK;
if(!strcmp(name,"Async IO Poll lock"))
    monitor_table[i].type = ASYNC_IO_POLL_LOCK;
if(!strcmp(name,"hostent lock"))
    monitor_table[i].type = HOSTENT_LOCK;
if(!strcmp(name,"Finalize me queue lock"))
    monitor_table[i].type = FINALIZE_ME_QUEUE_LOCK;
if(!strcmp(name,"Has finalization queue lock"))
    monitor_table[i].type = HAS_FINALIZATION_QUEUE_LOCK;
if(!strcmp(name,"Heap lock"))
    monitor_table[i].type = HEAP_LOCK;
if(!strcmp(name,"Code rewrite lock"))
    monitor_table[i].type = CODE_REWRITE_LOCK;
if(!strcmp(name,"Java stack lock"))
    monitor_table[i].type = JAVA_STACK_LOCK;
if(!strcmp(name,"Class loading lock"))
    monitor_table[i].type = CLASS_LOADING_LOCK;
if(!strcmp(name,"BinClass lock"))
    monitor_table[i].type = BINCLASS_LOCK;
if(!strcmp(name,"JNI global reference lock"))
    monitor_table[i].type = JNI_GLOBAL_REFERENCE_LOCK;
if(!strcmp(name,"JNI pinning lock"))
    monitor_table[i].type = JNI_PINNING_LOCK;
```

```
    if(!strcmp(name,"String intern lock"))
        monitor_table[i].type = STRING_INTERN_LOCK;
    if(!strcmp(name,"Name and type hash table lock"))
        monitor_table[i].type = NAME_AND_TYPE_HASH_TABLE_LOCK;
    if(!strcmp(name,"Thread queue lock"))
        monitor_table[i].type = THREAD_QUEUE_LOCK;
    if(!insert_AVL(&mid_index,monitor_id,i))
    {
       fprintf(stderr,"JIS Error registering monitor:
       insert_AVL(mid_index,%ld)\n",monitor_id);
        exit(1);
    }
}

/* update thread state to BLOCKED when it blocks in monitor */
void queue_wait(long monitor_id, long queue_id)
{
    long i;
    if(!consult_AVL(mid_index,monitor_id,&i))
    {
       fprintf(stderr,"JIS Error blocking thread in monitor: Monitor %ld doesn't
       exist\n",monitor_id);
        exit(1);
    }
    thread_table[current_thread].wait_monitor = monitor_id;
    thread_table[current_thread].wait_queue = queue_id;
    if(queue_id - monitor_id == 24)
    thread_table[current_thread].state = BLOCKED_WAIT;
    else
    {
    if(queue_id - monitor_id == 16)
        thread_table[current_thread].state = BLOCKED_MUTEX;
    }
}

/* update state of a thread inserted in a queue */
void queue_insert(long queue_id, long thread_id)
{
    long i;
    if(!consult_AVL(sys_tid_index,thread_id,&i))
    {
       fprintf(stderr,"JIS Error inserting thread in queue: Thread %ld doesn't
       exist\n",thread_id);
    exit(1);
```

```
    }
    if(queue_id == (long)&runnable_queue)
    {
    if(thread_table[i].wait_monitor != 0)
    {
        thread_table[i].wait_monitor = 0;
        thread_table[i].wait_queue = 0;
    }
    thread_table[i].state = READY;
    paraver_thread = i;
        paraverChangeState(CPU_READY);
    }
    else
    {
      /* thread blocked in wait queue, blockes in mutex queue in the same
      monitor */
      if((thread_table[i].state == BLOCKED_WAIT) && (queue_id ==
      thread_table[i].wait_monitor + 16))
    {
        thread_table[i].wait_queue = queue_id;
        paraver_thread = i;
        thread_table[i].state = BLOCKED_MUTEX;
            paraverChangeState(CPU_BLOCKED_MUTEX);
    }
    }
}
```

C2. Representative Pre/Post functions for native threads

```
/*
 * Copyright (c) 2000
 * CEPBA - UPC.    All Rights Reserved.
 *   DAC - UPC.    All Rights Reserved.
 *
 * Programmer: Jordi Guitart Fernandez (jguitart@cepba.upc.es)
 * Java Tracing Routines Implementation (NATIVE MODEL)
 *
 * Java Instrumentation Suite (JIS)
 * Barcelona Java Suite (BJS)
 */

/* update JIS structures when an application Java thread starts */
void start_thread(long thread_id)
{
    long i;
```

```
    pthread_mutex_lock(&java_tid_index_mutex);
    if(consult_AVL(java_tid_index,thread_id,&i))
    {
       fprintf(stderr,"JIS Error starting application Java thread: Thread %ld
       already exists\n",thread_id);
         exit(1);
    }
    pthread_mutex_unlock(&java_tid_index_mutex);
    pthread_mutex_lock(&free_threads_mutex);
    if(!head_prio_queue(&free_threads,&i))
    {
       fprintf(stderr,"JIS Error starting application Java thread: Not enought
       space at JIS structures\n");
         exit(1);
    }
    if(!delete_head_prio_queue(&free_threads))
    {
       fprintf(stderr,"JIS Error starting application Java thread:
       delete_head_prio_queue(free_threads)\n");
         exit(1);
    }
    pthread_mutex_unlock(&free_threads_mutex);
    thread_table[i].ever_busy = TRUE;
    thread_table[i].wait_monitor = 0;
    thread_table[i].state = INIT;
    thread_table[i].thread_java_id = thread_id;
    pthread_setspecific(paraver_thread,(const void *)i);
    paraverChangeState(CPU_INIT);
    pthread_mutex_lock(&java_tid_index_mutex);
    if(!insert_AVL(&java_tid_index,thread_id,i))
    {
       fprintf(stderr,"JIS Error starting application Java thread:
       insert_AVL(java_tid_index,%ld)\n",thread_id);
         exit(1);
    }
    pthread_mutex_unlock(&java_tid_index_mutex);
}

/* update JIS structures when a pthread starts */
void start_pthread(long thread_pthread_id, long thread_sys_id)
{
    long i, thread_java_id;
    int ret;
    thread_java_id = (long)pthread_getspecific(last_java_thread_created);
    pthread_mutex_lock(&java_tid_index_mutex);
```

```
ret = consult_AVL(java_tid_index,thread_java_id,&i);
pthread_mutex_unlock(&java_tid_index_mutex);
if(!ret)
{
/* system thread */
thread_table[current_init_thread].thread_java_id = thread_java_id;
    thread_table[current_init_thread].thread_sys_id = thread_sys_id;
thread_table[current_init_thread].thread_pthread_id = thread_pthread_id;
pthread_mutex_lock(&java_tid_index_mutex);
if(!insert_AVL(&java_tid_index,thread_java_id,current_init_thread))
    {
         fprintf(stderr,"JIS Error starting pthread:
         insert_AVL(java_tid_index,%ld)\n",thread_java_id);
            exit(1);
    }
pthread_mutex_unlock(&java_tid_index_mutex);
pthread_mutex_lock(&sys_tid_index_mutex);
    if(!insert_AVL(&sys_tid_index,thread_sys_id,current_init_thread))
    {
         fprintf(stderr,"JIS Error starting pthread:
         insert_AVL(sys_tid_index,%ld)\n",thread_sys_id);
            exit(1);
    }
pthread_mutex_unlock(&sys_tid_index_mutex);
pthread_mutex_lock(&pthread_tid_index_mutex);

if(!insert_AVL(&pthread_tid_index,thread_pthread_id,current_init_thread))
    {
         fprintf(stderr,"JIS Error starting pthread:
         insert_AVL(pthread_tid_index,%ld)\n",thread_pthread_id);
            exit(1);
    }
pthread_mutex_unlock(&pthread_tid_index_mutex);
}
else
{
thread_table[i].thread_sys_id = thread_sys_id;
    thread_table[i].thread_pthread_id = thread_pthread_id;
pthread_mutex_lock(&sys_tid_index_mutex);
if(!insert_AVL(&sys_tid_index,thread_sys_id,i))
    {
         fprintf(stderr,"JIS Error starting pthread:
         insert_AVL(sys_tid_index,%ld)\n",thread_sys_id);
            exit(1);
    }
```

```
    pthread_mutex_unlock(&sys_tid_index_mutex);
    pthread_mutex_lock(&pthread_tid_index_mutex);
        if(!insert_AVL(&pthread_tid_index,thread_pthread_id,i))
        {
             fprintf(stderr,"JIS Error starting pthread:
             insert_AVL(pthread_tid_index,%ld)\n",thread_pthread_id);
            exit(1);
        }
    pthread_mutex_unlock(&pthread_tid_index_mutex);
    }
}

/* update JIS structures when a thread initiates */
void init_thread(long thread_id)
{
    long i;
    pthread_mutex_lock(&sys_tid_index_mutex);
    if(!consult_AVL(sys_tid_index,thread_id,&i))
    {
      fprintf(stderr,"JIS Error initiating thread: Thread %ld doesn't
      exist\n",thread_id);
    exit(1);
    }
    pthread_mutex_unlock(&sys_tid_index_mutex);
    thread_table[i].state = RUN;
    pthread_setspecific(paraver_thread,(const void *)i);
    /*paraverChangeState(CPU_RUNNING);*/

paraverChangeStateAndUserEvent(CPU_RUNNING,FALSE,1,EVENT_KERNEL_THREAD,(long)di_
whoami());
}

/* update JIS structures when a non registered monitor is created */
void mutex_init(long monitor_id)
{
    long i;
    int ret;
    pthread_mutex_lock(&mid_index_mutex);
    ret = consult_AVL(mid_index,monitor_id,&i);
    pthread_mutex_unlock(&mid_index_mutex);
    if(!ret)
    {
        /* Non registered monitor */
    pthread_mutex_lock(&free_monitors_mutex);
    if(!head_prio_queue(&free_monitors,&i))
```

```
        {
              fprintf(stderr,"JIS Error creating non registered monitor: Not
              enought space at JIS structures\n");
                 exit(1);
        }
        if(!delete_head_prio_queue(&free_monitors))
        {
              fprintf(stderr,"JIS Error creating non registered monitor:
              delete_head_prio_queue(free_monitors)\n");
                 exit(1);
        }
    pthread_mutex_unlock(&free_monitors_mutex);
    monitor_table[i].type = USER_MONITOR;
        monitor_table[i].monitor_id = monitor_id;
    pthread_mutex_lock(&mid_index_mutex);
    if(!insert_AVL(&mid_index,monitor_id,i))
        {
              fprintf(stderr,"JIS Error creating non registered monitor:
              insert_AVL(mid_index,%ld)\n",monitor_id);
                 exit(1);
        }
    pthread_mutex_unlock(&mid_index_mutex);
    }
}


/* update thread state to BLOCKED_WAIT when it blocks in a condition variable
inside a monitor */
void pthread_wait(long condvar_id, long mutex_id)
{
    long i,j;
    pthread_mutex_lock(&pthread_tid_index_mutex);
    if(!consult_AVL(pthread_tid_index,(long)pthread_self(),&i))
    {
       fprintf(stderr,"JIS Error blocking thread in condvar: Thread %ld doesn't
       exist\n",(long)pthread_self());
         exit(1);
    }
    pthread_mutex_unlock(&pthread_tid_index_mutex);
    thread_table[i].state = BLOCKED_WAIT;
    pthread_mutex_lock(&mid_index_mutex);
    if(!consult_AVL(mid_index,mutex_id,&j))
    {
       fprintf(stderr,"JIS Error blocking thread in condvar: Monitor %ld doesn't
       exist\n",mutex_id);
         exit(1);
    }
```

```
    pthread_mutex_unlock(&mid_index_mutex);
    pthread_setspecific(paraver_thread,(const void *)i);
    if(monitor_table[j].type != USER_MONITOR)
    paraverChangeStateAndUserEvent(CPU_BLOCKED_WAIT,TRUE,1,EVENT_BLOCKED_IN_MONI
TOR,monitor_table[j].type);
    else
    paraverChangeStateAndUserEvent(CPU_BLOCKED_WAIT,TRUE,1,EVENT_BLOCKED_IN_MONI
TOR,monitor_table[j].monitor_id);
}

/* update thread state to RUN when it unblocks of a condition variable inside a
monitor */
void pthread_cond_run()
{
    long i;
    pthread_mutex_lock(&pthread_tid_index_mutex);
    if(!consult_AVL(pthread_tid_index,(long)pthread_self(),&i))
    {
       fprintf(stderr,"JIS Error unblocking thread of condvar: Thread %ld
       doesn't exist\n",(long)pthread_self());
         exit(1);
    }
    pthread_mutex_unlock(&pthread_tid_index_mutex);
    thread_table[i].state = RUN;
    pthread_setspecific(paraver_thread,(const void *)i);
    /*paraverPopState();*/
    paraverPopStateAndUserEvent(1,EVENT_KERNEL_THREAD,(long)di_whoami());
}

/* update thread state to RUN when it acquires the lock inside a monitor */
void pthread_mutex_run(long mutex_id, int in_lock)
{
    long i,j;
    pthread_mutex_lock(&pthread_tid_index_mutex);
    if(!consult_AVL(pthread_tid_index,(long)pthread_self(),&i))
    {
       fprintf(stderr,"JIS Error unblocking thread of monitor: Thread %ld
       doesn't exist\n",(long)pthread_self());
         exit(1);
    }
    pthread_mutex_unlock(&pthread_tid_index_mutex);
    /*****/ thread_table[i].state = RUN;
    if(in_lock)
    {
    pthread_mutex_lock(&mid_index_mutex);
```

```
    if(!consult_AVL(mid_index,mutex_id,&j))
        {
             fprintf(stderr,"JIS Error unblocking thread in monitor: Monitor
             %ld doesn't exist\n",mutex_id);
            exit(1);
        }
    pthread_mutex_unlock(&mid_index_mutex);
    pthread_setspecific(paraver_thread,(const void *)i);
    /*if(monitor_table[j].type != USER_MONITOR)
            paraverPopStateAndUserEvent(1,EVENT_LOCK,monitor_table[j].type);
        else
paraverPopStateAndUserEvent(1,EVENT_LOCK,monitor_table[j].monitor_id);*/
    if(monitor_table[j].type != USER_MONITOR)
paraverPopStateAndUserEvent(2,EVENT_LOCK,monitor_table[j].type,EVENT_KERNEL_THRE
AD,(long)di_whoami());
        else
paraverPopStateAndUserEvent(2,EVENT_LOCK,monitor_table[j].monitor_id,EVENT_KERNE
L_THREAD,(long)di_whoami());
    }
    else
    {
    pthread_setspecific(paraver_thread,(const void *)i);
        /*paraverPopState();*/
    paraverPopStateAndUserEvent(1,EVENT_KERNEL_THREAD,(long)di_whoami());
    }
}

/* update thread state to BLOCKED_MUTEX when it blocks trying to enter a monitor
*/
void pthread_lock(long mutex_id)
{
    long i,j;
    /* active waiting for pthread completely created */
    pthread_mutex_lock(&pthread_tid_index_mutex);
    while(!consult_AVL(pthread_tid_index,(long)pthread_self(),&i))
    {
    pthread_mutex_unlock(&pthread_tid_index_mutex);
    pthread_mutex_lock(&pthread_tid_index_mutex);
    }
    pthread_mutex_unlock(&pthread_tid_index_mutex);
    thread_table[i].state = BLOCKED_MUTEX;
    pthread_mutex_lock(&mid_index_mutex);
    if(!consult_AVL(mid_index,mutex_id,&j))
```

```
    {
       fprintf(stderr,"JIS Warning blocking thread in monitor: Monitor %ld
       doesn't exist -> Monitor added\n",mutex_id);
    /* add monitor */
        pthread_mutex_lock(&free_monitors_mutex);
        if(!head_prio_queue(&free_monitors,&j))
        {
             fprintf(stderr,"JIS Error creating non registered monitor: Not
             enought space at JIS structures\n");
               exit(1);
        }
        if(!delete_head_prio_queue(&free_monitors))
        {
             fprintf(stderr,"JIS Error creating non registered monitor:
             delete_head_prio_queue(free_monitors)\n");
               exit(1);
        }
        pthread_mutex_unlock(&free_monitors_mutex);
        monitor_table[j].type = USER_MONITOR;
        monitor_table[j].monitor_id = mutex_id;
        if(!insert_AVL(&mid_index,mutex_id,j))
        {
             fprintf(stderr,"JIS Error creating non registered monitor:
             insert_AVL(mid_index,%ld)\n",mutex_id);
               exit(1);
        }
    }
    pthread_mutex_unlock(&mid_index_mutex);
    pthread_setspecific(paraver_thread,(const void *)i);
    if(monitor_table[j].type != USER_MONITOR)
    paraverChangeStateAndUserEvent(CPU_BLOCKED_MUTEX,TRUE,1,EVENT_BLOCKED_IN_MON
ITOR,monitor_table[j].type);
    else
    paraverChangeStateAndUserEvent(CPU_BLOCKED_MUTEX,TRUE,1,EVENT_BLOCKED_IN_MON
ITOR,monitor_table[j].monitor_id);
}


/* update JIS structures when a thread exits */
void exit_thread()
{
    long i;
    pthread_mutex_lock(&pthread_tid_index_mutex);
    if(!consult_AVL(pthread_tid_index,(long)pthread_self(),&i))
    {
       fprintf(stderr,"JIS Error exiting thread: Thread %ld doesn't
       exist\n",(long)pthread_self());
```

```
        exit(1);
    }
    pthread_mutex_unlock(&pthread_tid_index_mutex);
    thread_table[i].state = STOPPED;
    pthread_setspecific(paraver_thread,(const void *)i);
    paraverChangeState(CPU_STOPPED);

    pthread_mutex_lock(&java_tid_index_mutex);
    if(!delete_AVL(&java_tid_index,thread_table[i].thread_java_id))
    {
       fprintf(stderr,"JIS Error exiting thread:
       delete_AVL(java_tid_index,%ld)\n",thread_table[i].thread_java_id);
        exit(1);
    }
    pthread_mutex_unlock(&java_tid_index_mutex);

    pthread_mutex_lock(&sys_tid_index_mutex);
    if(!delete_AVL(&sys_tid_index,thread_table[i].thread_sys_id))
    {
       fprintf(stderr,"JIS Error exiting thread:
       delete_AVL(sys_tid_index,%ld)\n",thread_table[i].thread_sys_id);
        exit(1);
    }
    pthread_mutex_unlock(&sys_tid_index_mutex);

    pthread_mutex_lock(&pthread_tid_index_mutex);
    if(!delete_AVL(&pthread_tid_index,thread_table[i].thread_pthread_id))
    {
       fprintf(stderr,"JIS Error exiting thread:
       delete_AVL(pthread_tid_index,%ld)\n",thread_table[i].thread_pthread_id);
        exit(1);
    }
    pthread_mutex_unlock(&pthread_tid_index_mutex);

    pthread_mutex_lock(&free_threads_mutex);
    if(!insert_prio_queue(&free_threads,i))
    {
       fprintf(stderr,"JIS Error exiting thread:
       insert_prio_queue(free_threads,%ld)\n",i);
        exit(1);
    }
    pthread_mutex_unlock(&free_threads_mutex);
}
```