Understanding the overhead of the spin-lock loop in CMT architectures

Vladimir Cakarevic[1], Petar Radojkovic[1], Javier Verdú[2], Alejandro Pajuelo[2],
Roberto Gioiosa[1], Francisco J. Cazorla[1], Mario Nemirovsky[1,3], Mateo Valero[1,2]
[1] Barcelona Supercomputing Center (BSC)
[2] Universitat Politecnica de Catalunya (UPC)
[3]ICREA Research Professor

***Abstract*—Spin locks are a synchronization mechanisms used to provide mutual exclusion to shared software resources. Spin locks are used over other synchronization mechanisms in several situations, like when the average waiting time to obtain the lock is short, in which case the probability of getting the lock is high, or when it is no possible to use other synchronization mechanisms.**

In this paper, we study the effect that the execution of the Linux spin-lock loop in the Sun UltraSPARC T1 and T2 processors introduces on other running tasks, especially in the worst case scenario where the workload shows high contention on a lock. For this purpose, we create a task that continuously executes the spin-lock loop and execute several instances of this task together with another active tasks.

Our results show that, when the spin-lock tasks run with other applications in the same core of a T1 or a T2 processor, they introduce a significant overhead on other applications: 31% in T1 and 42% in T2, on average, respectively. For the T1 and T2 processors, we identify the fetch bandwidth as the main source of interaction between active threads and the spin-lock threads. We, propose 4 different variants of the Linux spin-lock loop that require less fetch bandwidth. Our proposal reduces the overhead of the spin-lock tasks over the other applications down to 3.5% and 1.5% on average, in T1 and T2 respectively. This is a reduction of 28 percentage points with respect to the Linux spin-lock loop for T1. For T2 the reduction is about 40 percentage points.

I. Introduction

Spin-locks are synchronization mechanisms used to provide mutual exclusion to shared software resources. In most of the cases they rely on some hardware support to work properly. This support usually consists of a processor instruction that atomically reads and writes to a memory location.

Even if more advanced implementations of the spin-lock are available, this mechanism should be used carefully since it raises a trade-off between *overhead* and *responsiveness*. Spinning on a lock can reduce the speed of other tasks doing useful work (active taks) by consuming shared resources (overhead). On the other hand, the time from the release of a lock until the lock is re-acquired is a critical parameter in parallel applications (responsiveness). There is a clear trade-off between how often a thread tries to obtain a lock and the slowdown caused on the execution of other threads. Nevertheless, there are several situations where spin-lock mechanisms have shown good results with respect to other synchronization mechanisms. First, if the time a thread has to wait is smaller than the time it takes to do a context switch, then it is useful to spin on a lock. Second, if the probability of having a conflict obtaining the lock is on average low, spinning on the lock is worthy. And third, when there is no other synchronization mechanism because there is not a full-fledged Operating System (OS). This is a common case in networking. As a represent of a networking environment in this paper we will use a runtime environment called Netra Data Plane Software Suite (NetraDPS) [4][5].

In [8][9][10] authors propose effective exponential backoff, MCS lock and self-tuning algorithms in order to reduce busy-wait memory contention in multi-threaded applications. Our approach is, in a way, orthogonal to theirs, instead of memory, we identify the fetch bandwidth and intra-core resources the as main sources of slowdown in CMT processors. In our experiments effect of memory and interconnection network contention shows to be orders of magnitude smaller than effect of intra-core resource sharing. This difference appears because of different nature of architectures on which experiments are run. While [8][9][10] analyze single-threaded/single-core processor systems where the only shared resources are memory and interconnection network, we focus on a massive multithreading architecture.

In [1] the authors propose to add "slight" delay into the spin-lock loop for out of order Intel processors. The point is that in Out of Order processors the read of the spin-lock loop maybe executed out of order, as a consequence, the processor may suffer a penalty when check if no violation of the memory order happens. The difference of our approach with respect to this is that we precisely measure impact of the spin-lock loop to co-scheduled threads. We show that there is still high impact on performance when Out of Order execution is taken out of equation. Moreover, we propose and analyze the software solution to the problem.

In this paper, we show that, as the processor industry moves toward multithreaded processors, with more hardware shared resources, the effect of the spin-lock on other active threads becomes more critical. For this reason, we focus on the delay code itself in order to reduce the overhead on the other active threads. To accomplish this, we modify the delay code, inserting long-latency instructions to reduce the amount of resources allocated by the spinning thread. We show that if the underlying processor has multiple cores (where each core is single-threaded), the instructions in the delay code are not critical as they do not affect much the threads running on the other cores. However, in processors supporting the execution of several threads per core, a bad choice in the selection of the body of the delay code can drastically affect the performance of the rest of threads running in the same core.

In particular, we focus our study in the Sun UltraSPARC T1 [2] and T2 [6] processors. These processors provide a fine-grain hardware resource sharing between running tasks. In T1, threads in different cores share the L2 cache, the Floating Point

Unit (FPU), the I/O resources and the interconnection network between these three components. We call this level *inter-core*. Tasks in the same core, in addition to previous resources, share the vast majority of the core resources: fetch bandwidth, execution units, etc. We call this level *intra-core*. The first level data and instruction cache are also shared between threads in the same core. The main difference in the T2 is that in each core there are two levels of resource sharing. Each T2 core has two *execution pipelines* that are composed basically by the execution units. Threads in the same execution pipe share the execution units, while threads in different execution pipes do not. We call these levels *intra-pipe/intra-core* and *inter-pipe/intra-core*.

The main contributions of this paper are the following:

1) We show that the a thread executing the default spin-lock loop of Linux can seriously degrade the performance of other active threads depending on the amount of resources shared among them. In fact, the larger the amount of shared resources, the higher the degradation on the performance of active threads. For example, the execution of three instances of the default Linux spin-lock loop causes a slowdown on the applications we use in this paper up to 43% (31.5% on average) for T1 and up to 61% (42% on average) for T2.
2) In addition, we examine what are the shared resources that cause most of the conflicts between the spin-lock thread and the active threads. In T1 and T2 we show that the fetch bandwidth is the most critical shared resource and it is the responsible of the performance slowdown (overhead).
3) Derived from this study, we create several versions of the delay code by adding different type of long-latency instructions, so that the fetch bandwidth needs of the spin-lock threads decreases. Our results show that, even if our improved spin-lock loops introduce overhead in other shared resources, they effectively reduce the contention in the fetch stage of the processor. Our best spin-lock loop reduces the overhead on the active threads down to 4.1% in the worst case, 3.5% on average for T1. In T2 the worst case slowdown is 2%, 1.5% on average. This is, on average, a reduction of 28 and 40 percentage points with respect to the default implementation of the spin-lock loop in Linux in T1 and T2 respectively.

The rest of this paper is structured as follows. Section II provides some background on synchronization mechanisms. In Section III we describe our experimental environment. Section IV studies the overhead of the spin-lock loop implemented in Linux for T1 and T2 processors. Section V presents our proposals of spin-lock loops. Section VI is devoted to the conclusions.

II. Background on spin locks

Synchronization mechanisms are critical for the performance of parallel or multi-thread applications. Choosing the right synchronization mechanism and tuning it properly most likely results in scalability improvement which, in turn, translates in performance improvement. However, choosing the right synchronization mechanism is not straightforward. Several variables play a role in the scalability and performance of a multi-thread applications, the most important being *correctness*, *responsiveness* and *overhead*. Unfortunately, those variables are not completely independent and improving one of them may worsen others. For example, assuming that the mechanism is correct (as we do in the rest of this paper), improving the responsiveness of a locking system may increase the overhead, hence, the overall performance may decrease.

A. To spin or not to spin

The choice of the synchronization mechanisms depends on the application and the system. For example, responsiveness and overhead depends on the *contention rate* (the expected number of processes that try to acquire the lock concurrently), *context switch overhead* (the time required to suspend and resume a process) and the *locking time* (the average time a process holds a lock). If the locking time is high, a process P will spend a lot of time waiting for a lock to become free. In this case, it could be convenient to release the CPU to the OS and sleep until the lock is available. The same situation happens when the contention rate is high, though, in this last case, the analysis is performed on the average waiting time. In both the previous examples, the overhead of the waiting processes on the running process (i.e. the one that holds the lock) is reduced. However, the responsiveness of the waiting processes may decrease.

On the other hand, if the responsiveness is the most important variable, a waiting process should not release the CPU but should spin until it is able to acquire the lock. In this case the process does not pay the overhead of the context switch and it is able to move on with its job as soon as the lock is released. If the contention rate and/or the locking time is low, this solution may provide benefits. An interesting situation may arise when a user mode process spins on a lock: the process tries to acquire the lock continuously consuming CPU power, eventually increasing the overhead on other processes that share hardware resources with it. If the process spins on the lock for a long time (high contention rate or high locking time) then, sooner or later, the OS will schedule this process out of the CPU, assigning the processor to another process (time sharing scheduling). In this case the spinning process would still suffer the context switch overhead.

Another possibility half-way between continuously spinning and going to sleep is to release the CPU without going to sleep, by means of the `yield()` system call. In this case the OS put the process to the end of its run-queue, assigning the CPU to other runnable processes. Of course, if the spinning process is the only one in the run-queue, the CPU will be assigned to it again (without any context switch) and the situation degenerates in a spin-lock mechanism with a large delay code (notice that this should be the correct way of writing a program that spins on locks).

In order to help reducing the contention of the spin-lock, processor designers provide hardware support to delay or even

pause the execution of the spinning loop. Intel [7] has extended their ISA to include the assembler instructions `pause` and `monitor/mwait`. The former introduces a pre-defined delay in the loop to alleviate the fetch bandwidth contention of the processor. Unfortunately, a precise description of the implementation of this instruction is not available at literature what makes it work as a black box and, thus, not suitable for our study. The SSE3 `MONITOR` and `MWAIT` instructions in the x86 processor are also used to alleviate the spin lock problem. Using the SSE3 instructions in the spin-lock eliminates the overhead of sending and receiving an interrupt to wake up a halted processor. MONITOR is used to specify a memory range to "monitor". MWAIT halts the processor until the address previously specified with MONITOR is accessed. With the new idle loop a processor only has to write to memory to wake up a halted processor. These instructions have, as a major drawback, that long periods of spin-lock cannot be used to executed useful work (processor preemption) by other process different from the one that is spinning.

1) Case Example 1: The key point of this discussion is that the application can choose which synchronization mechanism better suits its needs. However, there are systems or situation when the OS is not able to properly suspend and resume a process, thus releasing a CPU is not an option and spinning is necessary. For example, NetraDPS is a run-time system with no process scheduler: each process is assigned to a CPU at compile time and runs on that CPU until the end. If a process cannot proceed with its job because another process has already acquired a lock, the only option for the waiting process is spinning on the lock. The responsiveness is very high, though the overhead on the other processes becomes crucial with high contention locks, especially for multi-thread/multi-core processors.

2) Case Example 2: The OS core is another case when sleeping might not be possible. Interrupts are asynchronous and cannot be predicted in the time they come, thus, they might not be related (likely they are not) to the current process running on the CPU that received the interrupt. The interrupt handler cannot release the CPU and go to sleep because there is simply no place where to save its status (like task descriptors for processes). Interrupt handlers are designed to minimize this critical path: the *top half* of the interrupt handler performs all critical operations for handling the interrupt and restarting the device; once the device has been restarted, the *bottom half* of the interrupt handler performs the non-critical operation. Bottoms halves can be executed by a kernel daemon, thus, it is possible to go to sleep in this case. ¿From this example, it is clear that the top half of the interrupt handler and all the functions that try to acquire the same lock, must spin on the lock. Other situations arise when the OS has to perform global system operation that cannot be interrupted, such as accessing different per-CPU data structures at the same time (for example, load balancing).

There are more examples, but the common point is there are a lot of cases where spinning is the only option.

Line	Source code
01	static inline void __raw_spin_lock
02	(raw_spinlock_t *lock)
03	{
04	unsigned long tmp;
05	
06	__asm__ __volatile__(
07	"1: ldstub [%1], %0\n"
08	" membar #StoreLoad — #StoreStore\n"
09	" brnz,pn %0, 2f\n"
10	" nop\n"
11	" .subsection 2\n"
12	"2: ldub [%1], %0\n"
13	" membar #LoadLoad\n"
14	" brnz,pt %0, 2b\n"
15	" nop\n"
16	" ba,a,pt %%xcc, 1b\n"
17	" .previous"
18	: "=&r" (tmp)
19	: "r" (lock)
20	: "memory");
21	}

Fig. 1. Linux's spin-lock loop for Sun UltraSPARC T1 and T2 processors

B. Linux Spin-lock

In this section we explain the spin-lock loop implemented in Linux for T1 and T2 processors. This will be the reference point we use in our paper. Figure 1 shows the exact code of Linux implementation of spin-lock for UltraSPARC 64-bit architectures (T1, T2).

Linux uses a spin-on read (Test and Test and Set). In a call to this lock function a process first tries to obtain a lock using load-store instruction on the lock. If the lock is already acquired (it is not zero) the process will loop checking in each iteration if the lock value has changed (lines from 12 to 15). When the lock is released the process will try again to obtain it by using read-store instruction (out most loop). After the first loop iteration, the lock is in the nearest data cache of the processor, the first level data cache in T1 and T2. Only when the lock is released this address is invalidated by the thread which releases the lock. As long as the lock is busy, the spinning thread continuously executes the instruction between lines 12 and 15[1]. We refer to this inner most loop as the *Linux default spin-lock loop*.

III. Experimental Environment

This section describes the environment, the metrics and the benchmarks we use in this paper.

A. Environment

In order to run out experiments, we used a Sun UltraSPARC T1 and T2 processor.

We performed our tests on a T1 processor machine running at a clock frequency of 1GHz with 16GBytes of DDR-II SDRAM. T1 is a Chip-Multithreading (CMT) CPU with eight cores. Each core is an in-order, fine-grain multithreading (meaning that it switches between available threads each cycle) core able to run four threads concurrently.

[1]The nop instruction is in the branch delay slot and hence it is executed as a part the loop

The Sun UltraSPARC T2 processor we used has a clock frequency of 1.4GHz and 64GBytes of DDR-II FB-DIMMs. Like T1, the T2 processor has 8 in-oder cores but the in-core organization is different. The processor has two *execution pipes* (or simply pipe) per core. Each pipe has its own integer execution unit. A pipe consists of four hardware strands (hardware threads or contexts) making a total of eight strands per core. The pipes fetch and execute instructions in parallel and share per core FP unit, LSU and cryptography unit.

In our experiments we use a dedicated virtual machine (logical domain) managed by Sun Logical Domains (LDoms) software [3]; our experiments show no measurable overhead when using the LDoms software. In both T1 and T2 we used a logical domain running on two cores and using 4GBytes of dedicated memory. Using two cores means eight strands in the T1 processor and 16 strands in the T2 processor available for the LDoms manager.

To run our experiments we used as the framework NetraDPS [4][5]. NetraDPS is a low-overhead environment that provides lower functionalities than other full-fledged Operating Systems like Linux and Solaris, but it introduces almost no overhead, making it ideal for our purposes. The peculiarity of this framework is that, instead of executable files, it executes bootable images that may be run on Sun CMT processors. In NetraDPS binding function to virtual processor (strand) is done in a mapping file before compiling the application. Applications cannot migrate to other strands at run time. For this reason NetraDPS does not provide a run time scheduler, erasing the overhead it introduces.

When executing applications on NetraDPS logical domain, NetraDPS images are booted on bare virtual machine over virtual network. NetraDPS allows writing applications in high level language (ANSI C).

B. Methodology

Our goal in this paper is to measure the impact of a spinning lock over an independent active application(s) on a real multithreaded architecture. As a reference point we use the execution time of a given active application when run in isolation. Next, we measure its execution time when run together with one or several threads executing a spin-lock loop and compute the slowdown the application suffers.

The threads spinning on the lock and the active threads are mutually independent. We do not use real multithreaded applications, since in that case the spin-lock overhead would be mixed with the overhead of other parts of the application. The testing setup we propose mimics a parallel application during the period in which all its threads but one are blocked on a lock of a critical section. In this way, we simulate the worst case scenario for the spin-lock loop where the workload shows high contention on a lock. As a future work we plan to implement our spin-locks on a real OS and see their effect on real multithreaded applications.

Our objective in this framework is to ensure that the spin-lock threads are constantly running throughout the execution of active threads. We show a high-level view of our experimental environment in Figure 2. We differentiate 3 phases:

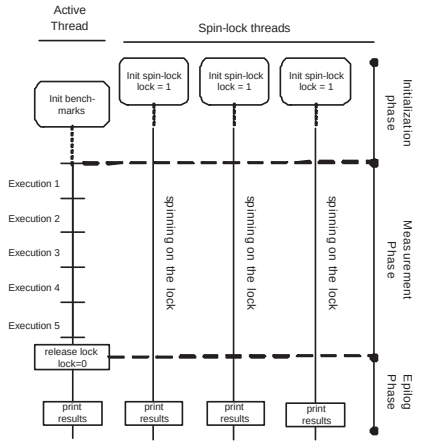


Fig. 2. Example of execution of one active thread and three spin-lock threads

1) Initialization phase: The active thread and the spin-lock thread(s) are initialized. The initialization phase of the the spin-lock threads consists of setting the lock to 1 and giving the lock to the active threads, which require a different, task specific, initialization. For example, the Matrix by Vector Multiplication benchmark (described in the next section) requires initializing the vector and the matrix. We first, start the spin-lock threads and then the active threads. In this way, we ensure that during the measurement phase the spin-lock threads cause a constant overhead on the active thread.

2) Measurement phase: To obtain reliable measurements of the spin-lock threads, we use the FAME (FAirly MEasuring Multithreaded Architectures) methodology [11][12]. In [11][12] the authors state that the average accumulated IPC of a program is representative if it is similar to the IPC of that program when the workload reaches a steady state. The problem is that, as shown in [11][12], the workload has to run for a long time to reach this steady state. FAME determines how many times each benchmark in a multithreaded workload has to be executed so that the difference between the obtained average IPC and the steady state IPC is below a particular threshold. This threshold is called MAIV (Maximum Allowable IPC Variation). The execution of the entire workload stops when all benchmarks have executed as many times as needed to accomplish a given MAIV value. For the experimental setup and benchmarks used in this paper, in order to accomplish a MAIV of 1%, each benchmark must be repeated at least 5 times.

3) Epilog Phase: After the active thread executes 5 times, it releases the lock (lock=0) which allows all spin-lock threads to finish almost instantly. Finally, all threads print some statistics and end.

C. Metrics

We use two main metrics to measure the performance of each spin-lock proposal: slowdown and responsiveness.

First, the slowdown (or overhead) that each spin-lock variant causes on the active thread. For this, we compare the execution time of the active thread when it runs in isolation (alone in the logical domain of the processor), and its execution time when it runs together with one or more spin-lock threads.

	Line	Source code
	001	.inline intmul_il, 0
	002	.label1:
B	003	mulx %o0, %o1, %o3
O		...
D	514	mulx %o0, %o1, %o3
Y	515	subcc %o2,1,%o2
	516	bnz .label1
	517	mulx %o0, %o1, %o3

Fig. 3. Main structure of the single-behavior benchmarks. In the example it is shown the MULX benchmark.

We use an indirect metric of the responsiveness of each spin-lock proposal. Each spin-lock thread measures the time interval between two spin-lock reads. It is clear that the longer this time, the lower the responsiveness. The real effect of the reduced responsiveness is something we plan to do in our future work.

D. Benchmarks

We use two sets of benchmarks as active tasks: single-behavior benchmarks and multiple behavior benchmarks. All benchmarks are compiled using Sun C compiler included in Sun Studio 12 (Sun C version 5.9). We compile NetraDPS images with the same Sun C compiler.

1) Single-behavior benchmarks: We use five single-behavior benchmarks: integer addition (ADD), 64-bit integer multiplication (MULX), branch always (BAA), a benchmark that always hits in L1 data cache ($L1\_hit$) and one benchmark that always miss in level 1 cache, but hits in level 2 cache ($L2\_hit$). They are written in assembler for UltraSPARC 2005 architecture. All single-behavior benchmarks are designed using the same principle, see Figure 3. The assembly code is a sequence of 512 instructions of the targeted type (lines from 3 to 514) ended with the decrement of integer register (line 515) and non-zero branch to the beginning (line 516). After the loop branch (line 516) we add another instruction of the targeted type (line 517) because in UltraSPARC ISA the instruction after a branch instruction is always executed (so called branch delay slot). The assembly functions are inlined in C code that defines the number of iterations for the assembly loop. The overhead of the loop and the calling code is less than 1% (more than 99% of time processor executes only the desired instruction).

The $L1\_hit$ and $L2\_hit$ benchmarks use half of available data L1 (4KB in both T1 and T2) and L2 cache (1.5MB in T1 and 2MB in T2) size respectively.

It should be noticed that our branch (BAA) benchmark is designed for the T1 processor. In this processor it is not important whether the branch is taken or not, in both cases the latency of the instruction is 3-4 cycles, depending on the exact branch instruction. In the T2 processor there is a large discrepancy between taken and not taken branch: taken branch latency is 6 cycles, while a not taken branch takes only 1 cycle to complete. All branches in our test are taken, this should be taken into account when looking at the test results for T2.

By using these single-behavior benchmarks we can capture the overhead due to the influence of spin-lock threads running in the system and identify the characteristic of the active single-behavior benchmarks that are more affected by the spin-lock threads.

2) Multiple-behavior benchmarks: In this case, we use benchmarks that emulate real algorithms with different phases in its execution. In particular, we built:

- Matrix by Vector Multiplication (integer and floating point): this benchmarks use large data structures and perform significant number of non-sequential accesses to memory. In this way, we try to cause noticeable number of data L2 cache misses that cause slowdown in the benchmark execution. The integer variant uses 800MB of memory for its matrices and access memory in highly non-sequential manner. The FP variant uses 400MB of memory space, but the access pattern is more sequential as the number of matrix elements is actually four times less. Both of them use a lot of multiplication (integer and FP respectively) which are medium latency instruction.
- QuickSort algorithm: This is a cpu-bounded task. It uses a standard recursive quicksort sorting algorithm. As input we use an array of 80KB. This program performs a lot of short latency operations (compares, level 1 cache loads and stores and recursive branches).
- Hash function: The benchmark inserts into a large 2D data structure (100 000 entries - around 2MB of data) a list of random integer numbers according to a 32bit hash function. The resulting hash value for a given number indicates the entry of the first dimension of the data structure. The function provides an even distribution. Then, the insertion function crosses the second dimension in order to insert the number. In order to check a match with other number in the second dimension of the structure, we use the less significant bits of the number. The hash function gathers the stressing issues of the both previously mentioned benchmarks. The hash function is cpu–bounded due to the arithmetic operations, while the insertion function is memory bounded due to the non–sequential accesses to a large data structure.

We choose the active thread from one of these two groups. First, the single-behavior group comprised by the ADD, MULX, BAA, $L1\_hit$, $L2\_hit$ benchmarks. And the multiple-behavior group that is composed by the Hash, QuickSort, MbV FP and MbV INT benchmarks.

IV. Results

The overhead that a thread executing a spin-lock loop puts on other threads running at the same time depends on two main factors. First, the type of operations executed in the loop, and second, the degree of hardware resources shared between the spin-lock thread and other threads. In the following, we analyze in detail the different resource sharing levels in the T1 and T2 processors.

A. Resource sharing in T1 and T2

Both T1 and T2 are massively-threaded processor. T1 and T2 combines two forms of multithreading: multicore and fine-grain multithreading. Threads share different resources depending on the resource level in which they are.

T1 and T2 present different levels of resource sharing. In the case of T1, threads can be in the same core (*intra-core* resource sharing) or in different cores (*inter-core* resource sharing).

- Inter-core: T1 has eight cores each running up to 4 threads, so in total it can run up to 32 threads. Threads in different cores share mainly the L2 cache, the interconnection network, the Floating Point (FP) unit and the I/O resources.
- Intra-core: In each core T1 can run up to 4 threads at the same time. Threads in each core, in addition to all previous resources, share most of the execution resources like the Instruction Fetch Unit (IFU), the Execution Unit (EXU), the Load/Store unit (LSU), the FP Frontend Unit (FFU) and the instruction and data Translation Lookaside Buffers (TLBs). Each core has its own private data and instruction cache.

For the T2 processor we differentiate three levels of resource sharing. The main difference from T1 resides in the fact that each T2 core provides two *execution pipelines*.

- Inter-Core: T2 has 8 cores each running up to 8 threads, so in total it can run up to 64 threads at the same time. As in T1, the main resources shared between threads in different cores are the L2 cache, the interconnection network and the I/O resources. Opposite to T1, the Floating Point - Graphical Unit (FPGU) is private to each core.
- Inter pipe - Intra core: In each core T2 provides two execution pipelines. Threads in different pipelines do not share the execution units that are duplicated. However, in addition to sharing the same resources as threads in different cores, they share the Load/Store Unit (LSU), the FPGU, the instruction and the data TLBs and cryptography unit. As in T1, each core has its own private data and instruction cache.
- Intra-pipe - Intra core: In T2, Functional Units are duplicated, one for each execution pipeline. Hence, all four threads that can run in an execution pipeline share the execution unit and the IFU (in addition to sharing the same resources as threads in different pipes).

Clearly, if two threads run in a resource level in which they share more resources, the slowdown that one causes on the other is higher.

B. Overhead introduced by the Linux spin-lock loop

Linux implements a spin-on-read (Test and Test and Set). Hence, as long as a lock is taken, the memory location on which a thread spins is in the data cache level nearest to the processor, the L1 data cache in T1 and T2. When the lock is released, this address is invalidated by the thread holding the lock when it released it. Hence, while the threads are spinning on a lock the average latency of each instruction in the loop is reported in Table I (lines from 12 to 15 in Figure 1).

As a consequence of this low CPI, threads executing the spin-lock loop are ready to fetch most of the time. In T1, the Instruction Fetch Unit (IFU) and the load/store unit (LSU), that is used as a bridge to access the instruction cache, are under constant pressure of the thread that is executing the spin-lock. Because of that, we say that they have high fetch bandwidth requirements. In T1, the thread selection policy is *Least Recently Fetched*. The same thread selection policy is used in T2 in each execution pipe, hence T2 fetches two instructions (if available) each cycle, with the restriction that they must belong to threads executing in different pipes. Whenever a thread is blocked for any reason (resource conflict, long-latency operation, etc.) it is not considered as a candidate to fetch from. Given that during most of the time the thread executing the spin-lock loop is available for fetch, it consumes cycles that could be used more effectively by another thread.

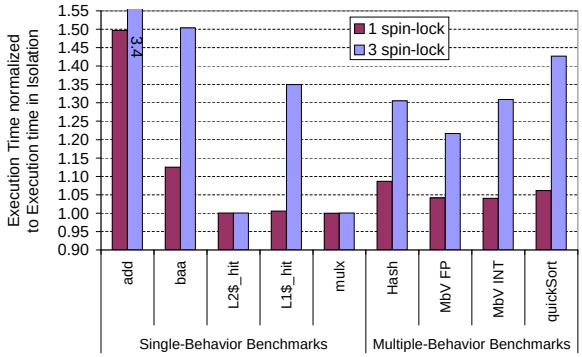


Fig. 4. Effect of 1 and 3 instances the linux spin-lock loop on different active threads when run on the same core on T1

TABLE I
LATENCY OF INSTRUCTIONS USED IN LINUX SPIN-LOCK LOOP IN T1/T2

Instruction	Latency in T1 (cycles)	Latency in T2 (cycles)
ldub	3	3
membar	variable (1)	variable (1)
branch	4	6
nop	1	1
Average CPI	2.25	2.75

Figure 4 shows the effect that 1 and 3 threads running the Linux spin-lock have on the different Single-Behavior and Multiple-Behavior benchmarks running on the same core of T1. Table II shows the CPI of each benchmark when running in isolation in T1 and T2 respectively. We observe that the active threads that suffer a high slowdown are those with low CPI because they also require a lot of fetch bandwidth.

The average slowdown caused by 1 instance of a thread executing the Linux spin-lock is 9.5%; 3 instances of the spin-lock loop introduce an average overhead is 50%. The same experiments on the T2 processor show a 8.6% and 54% overhead, respectively.

V. IMPROVED SPIN LOCK

In the previous section we have seen that the effect of the Linux spin-lock is significant mainly in those threads with low CPI. The main reason for this is that the default spin-lock is quite 'fetch hungry' as its CPI is very low. In this section our objective is to provide several modified implementations of the spin-lock loop. These modified versions aim to require less fetch bandwidth and less resources and to make them available for other active threads.

To achieve these goals, we insert a long-latency operation inside the loop. The role of this instruction is to increase the CPI of the spin-lock loop reducing its activity and the resource

TABLE II
CPI OF THE ACTIVES THREADS WHEN RUN IN ISOLATION IN T1 AND T2

	add	baa	$L2h$	$L1h$	mulx	Hash	MbVf	MbVi	Sort
T1	1	4	22	3	11	4.2	15.1	5.7	3.2
T2	1	7(6+1)	22	3	9	3.9	15.5	4.2	2.8

needs of the spinning thread. We add this instruction between lines 13 and 14 in the code shown in Figure 1. To strength the study, we show the effect of inserting instructions with different latencies and latency requirements inside the loop. The description of these instructions follows.

64-bit signed integer division (sdivx): its latency is 72 cycles in T1 and 12-41 cycles in T2. Since the division functional unit is not pipelined, it cannot overlap the execution of several division instructions. This makes threads stall whenever they have to execute a sdivx instruction and the div functional unit is being used.

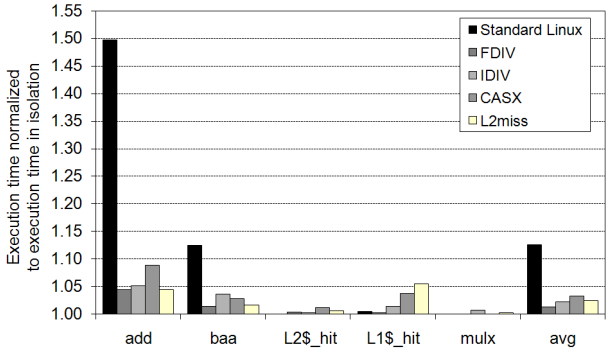
Compare and swap (casx): This instruction swaps the contents of one memory position allocated in the L2 data cache with the value of a register. This means that this instruction always accesses a memory location in L2 cache. Its latency is 39 cycles in T1 and between 20 and 30 cycles in T2 (in our experiments it takes almost always about 30 cycles). This instruction does not excessively stress the processor structures that could be used by the active thread. In fact, casx only uses one entry of the shared LSU structure that connects the core to the interconnection network. Moreover, the memory space requirements of using this instruction are very low since all the spin-locks can access the same memory position.

Double precision floating-point division (fdivd): its latency is 83 cycles in T1 and 33 in T2. In T1, there is only one available FP unit for the whole processor. This unit sequentially processes the FP instructions coming from one core and can overlap up to 3 different FP instructions coming from three different cores. Hence each issued FP instruction has to wait for all other previous FP instructions issued by the same core to finish before it is executed. This can lead to large slowdowns. In T2 there is one FP unit per core (a total of 8 FP units in the whole processor). The latency of the fdivd instruction is smaller and very similar to the latency of the casx instruction.

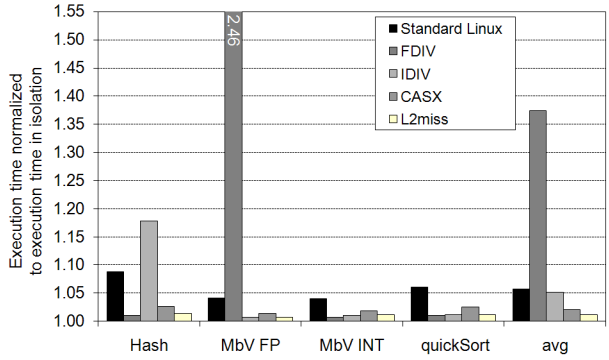
L2 cache data miss (L2miss): We also modify the spin-lock loop inserting a load instruction that always accesses the main memory: We make that load to access some data that we ensure it is not in the lower levels (L1 and L2 data cache) of the memory hierarchy. We used pointer chasing approach: during the initialization phase the array is initialized for pointer chasing in such a way that every load misses in L2 cache. We make this benchmark so that it always misses in the same set and bank of the L2 cache, hence, reducing the overhead it could introduce to other active threads. This approach consumes the same hardware resources of the casx modification, but it presents the largest latency of all instructions mentioned above - 100 cycles in T1 and over 200 in T2.

A. Results on T1

1) Results on the same core: Figure 5 shows the overhead that one instance of each spin-lock loop proposal introduces



(a) Single-Behavior Benchmarks



(b) Multiple-Behavior Benchmarks

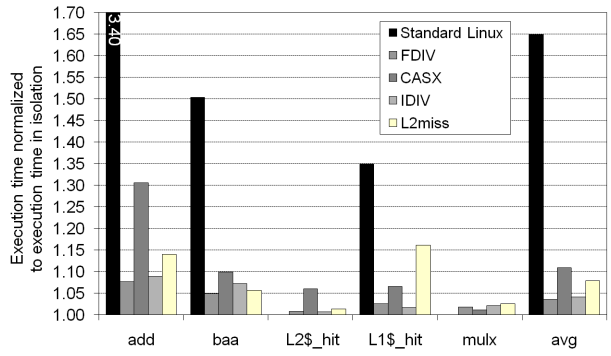
Fig. 5. Effect of 1 instance the same spin-lock loop on different active threads when run on the same core on T1

on each single-behavior benchmark (Figure 5(a)) and each multiple-behavior benchmark (Figure 5(b)). The Y-axis shows the slowdown relative to the execution of the active thread when it is run in isolation.

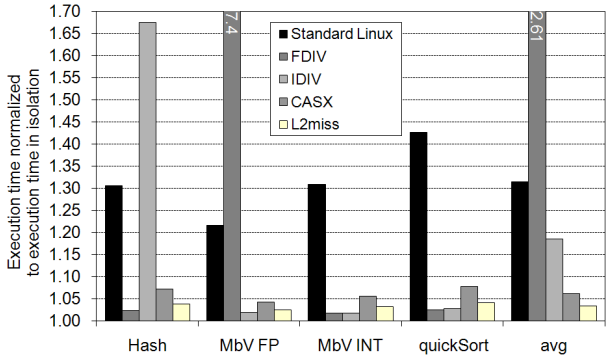
Figure 5(a) shows that, on average, the Standard Linux spin-lock loop is the implementation that affects more the execution of the single behavior benchmarks, nearly 14% on average, and up to 50% for the *ADD* benchmark. This large slowdown is due to the fact that the *ADD* benchmark presents a very low CPI, meaning that it needs a lot of resources, especially fetch bandwidth, to execute. Hence, since in the T1 processor the fetch stage is shared among all the strands in the same core, the spin-lock loop reduces the opportunities of fetching instructions for the add benchmark. Figure 5(a) also shows that none of the other implementations (FDIV, IDIV, CASX and L2miss) affects, on average, less than 5%, making them preferable to the standard implementation.

The slowdown introduced by the spin-lock threads on the multiple-behavior applications is shown in figure Figure 5(b). In this case, the Standard spin-lock only degrades the performance of benchmarks about 6% whereas the other implementations, except FDIV, present less than 5% of slowdown. This slowdown reduction respect to Figure 5(a) is due to the fact that multiple-behavior benchmarks present a higher CPI compared to single behavior ones. Hence, since their resource requirements are lower the spin-lock loops hardly affect their performance.

The only special cases appears when the MbVFP is executed in the same core as the FDIV and when the Hash benchmark is executed in the same core as IDIV spin-loop benchmark. In



(a) Single-Behavior Benchmarks



(b) Multiple-Behavior Benchmarks

Fig. 6. Effect of 3 instances the same spin-lock loop on different active threads when run on the same core on T1

the former case, MbVFP suffers the slowdown of 50% since both programs depend on the shared FP unit (there is only one unit for the whole chip in the T1 processor). In the latter case, the same happens with the integer divisor.

In general, the higher the CPI of an active thread, the less the spin-lock loops affect its execution. However, if the spin-lock loop uses a resource needed by the active thread, the latter suffers a significant slowdown.

Figure 6(a) and (b) shows the slowdown when three instances of the spin-lock loops are executed in the same core with each single- and multiple-behavior benchmark respectively. Remember that one core in the T1 processor, as previously described, can execute simultaneously up to 4 threads, meaning that, in this case, one core is fully used.

Figure 6(a) draws a more critical situation than in Figure 5(a) in which the Standard spin-lock loops degrades, on average, the performance of applications, nearly 65%. On the other hand, the modified implementations of this loop only increase the execution time about 10%. Also, Figure 6(b) shows the same critical scenario for multiple-behavior applications. The execution of 3 instances of the spin-lock increases the pressure in the shared resources reducing the opportunities of using those resources by the benchmarks. Since the fetch bandwidth and the functional units are critical in the T1 processor to execute applications, the insertion of long latency instructions makes the spinning loops reduce their resource requirements since most of the time they are stalled waiting for the completion of those instructions. This prevents the application from stalling due to the lack of resources.

To sum up, when two applications (one active application and one executing spin-lock loop) execute on the same core the main sources of interaction are the following. First, the Fetch bandwidth is the predominant source of slowdown caused on active threads. Second, the FP Frontend Unit (FFU): if FDIV spin-lock is paired with a task that use FP instructions, it suffer a significant slow down. Third, the same happens with the integer execution unit. If both the active task and spin-lock task use the same integer instruction type (Addition/Substraction/Comparation, Multiplication, Division) they will slowdown each other. Notice that if the spin-lock loop performs, for example, add operations and the active thread performs divisions, there is not interaction as each of these operations are done in a different unit. Fourth, L1 instruction and data cache: cache bound active tasks paired with CASX or L2miss spin-lock show small percentage of overhead. The impact of the instruction and data TLBs is not measured in our experiments as NetraDPS does not use TLBs. However, we think they are not going to be a bottleneck as the CASX and L2miss spin-lock loops only access 1 memory location, thus, only needing 1 entry in the Data TLB.

2) Results on different cores: If the spin-lock threads and the active threads run in different cores on the chip they will interact on the following resources. First, the L2 cache: we measured up to 2% of overhead when 4 CASX or L2miss spin-lock threadss run on a different core than the active task core. Second, the FP Unit: we measured no overhead caused by central FP unit. This overhead may exist if the active thread uses the same type of instruction as our spin-lock (FDIV). And third, the interconnection network: we did not measure any overhead related to this resource.

B. Results on T2

As previously described, the T2 processor presents three levels of resource sharing: intra-pipe, intra- core and inter-core. Each pipe has four strands from which the processor independently fetches instructions using a Least Recently Fetched algorithm among non-stalled strands.

1) Intra-pipe spin-lock overhead: Figure 7 shows the slowdown measured for our benchmarks when executing 3 instances of the loops in the same core and pipe.

An important result in this chart is the low overhead of the L2miss spin-lock loop which, in the T2 processor, hardly penalizes the execution through the whole set of benchmarks (under 1% of slowdown on average). This improvement is associated with, firstly, the long latency of a L2miss iteration (more than 220 cycles) and, secondly, with the low pressure that this loop makes to the LSU and the processor caches.

The existence of one pipelined FP unit per core greatly reduces the impact FDIV implementation of the spin-lock on FP workloads, as may be observed in Figure 7. However, the number of instructions that FPGU uses is much larger, from 23 instructions in T1 up to 129 in T2. For example, integer division and multiplication execution is assigned to FPGU unit. This may be seen in results for Hash benchmark which uses lots of integer division. The results when collision exist are not as disastrous as in T1, but still FDIV implementation may degrade performance instead of improving them.

To sum up, the main sources of overhead in intra-pipe executions are the fetch bandwidth and execution units in the pipe (integer execution units). The former is the main source of the overhead while the latter may introduce significant overhead if the active thread is executing the same instruction type as the spin-lock thread(s).

Even if the pressure in many of the shared resources has been alleviated, it keeps on being some execution overhead coming, mainly, from the fetch bandwidth congestion. This overload in the fetch unit can be easily alleviated by moving the spin-lock threads to the other pipe in the same core.

2) Inter-pipe and Inter-core spin-lock overhead: Figure 8 shows the overhead when running a single-behavior (charts labeled (a)) and multiple-behavior (charts labeled (b)) benchmarks scheduled with 4 instances of the spin-lock loops running in the second pipe in the same core.

In T2 the instruction fetch unit (IFU) is 'split' among pipes, being able to fetch instructions from one thread in each execution pipe at the same time. Hence, the fetch bandwidth congestion is effectively removed if threads are executing in different pipes. The average overhead is less than 3% for all the implementations. In this situation, the overhead mainly comes from the additional resources shared among the pipes (FP and LSU units).

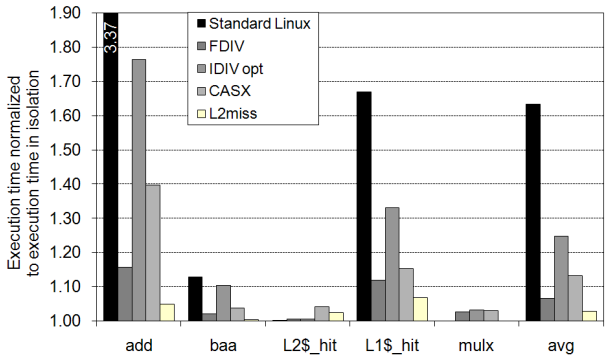
The main resources shared at core level are the Core FPGU and the L1 instruction and data cache. The Core FPGU is pipelined, thus, it can serve more requests with relatively little overhead. However, the unit is a source of a measurable overhead, both in the same pipe and different pipe experiments. For what concerns the L1 instruction and data cache, when cache intensive active tasks run together with CASX and L2miss spin-lock loop(s) the impact is low. We obtained similar results in the same-pipe and in the different-pipe experiments.

As for T1, we did not measure the impact of the instruction and data TLBs as NetraDPS does not use TLBs. However, we think they are not going to be a bottleneck as the CASX and l2miss spin-lock loops only access 1 memory location, thus, only needing 1 entry in the Data TLB. Moreover, the loop only has few instructions, thus, we do not expect Instruction TLB misses either. Finally, the results on separate cores are not shown here as the overhead is negligible.

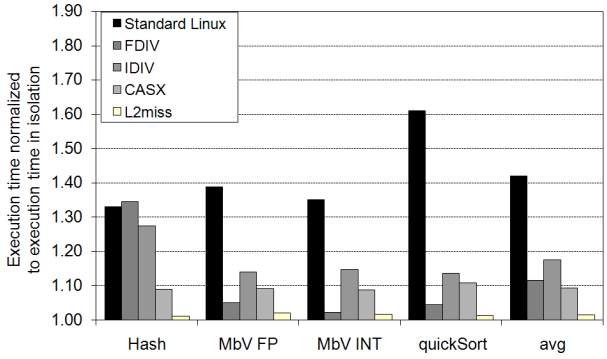
C. Comparison between T1 and T2

In both T1 and T2, the largest overhead is introduced when an active thread shares the instruction fetch bandwidth with the spin-lock threads. In case of T1 it means running the active thread and spin-lock threads on the same core, while in T2 it implies running them in the same pipe of a core.

1) Intra-core overhead: The use of Fully Buffered RAM, which has high bandwidth but slow response time, and large discrepancy between the processor and the memory frequency in T2 makes L2miss spin-lock more efficient in the newer processor. The T2 processor performs better by large margin in most of the cases. As a result of introducing a FP unit in each core of the T2 processor, FP performance is significantly improved - mainly because the contention at FFU is reduced. As these FP units are pipelined the slowdown when issuing

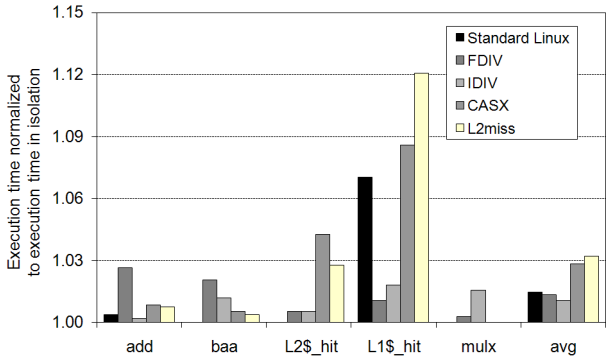


(a) Single-Behavior Benchmarks

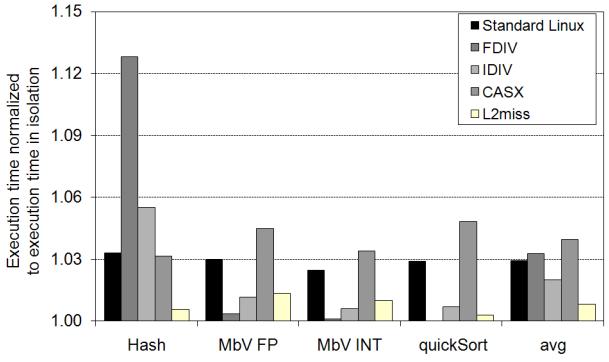


(b) Multiple-Behavior Benchmarks

Fig. 7. Effect of 3 instances the same spin-lock loop on different active threads when run on the same core and execution pipe on T2



(a) Single-Behavior Benchmarks



(b) Multiple-Behavior Benchmarks

Fig. 8. Effect of 4 instances the same spin-lock loop on different active threads when run on the same core and in different execution pipe on T2

consequent FP instructions is drastically reduced and, thus, the effect of spin-lock loop on the active thread is reduced as well. However, a large number of instructions that use FPGU

increases the chances of resource collision, making CASX more suitable for a spin-lock implementation. CASX and IDIV have similar effects in T2 as in T1. CASX remains the second best option regarding overhead on average.

2) Inter-core overhead: In both T1 and T2 the overhead when running an active thread alone on the core and the maximum number of spin-lock threads on a different core (4 in T1, 8 in T2) is very low. We did not notice any overhead when running FP spin-locks and FP intensive active benchmarks on different cores in T1. This is due to pipelining of central FP unit and the use of separate units for division, multiplication and addition in T1's FPU. Hence, only a slight overhead is observed through L2 cache interaction in both T1 and T2.

3) Influence of two execution pipes in T2: The second level of resource sharing in T2, same core - different pipe has a level of interaction between threads similar to the one observed in inter-core schedules in T1. The difference is overhead caused by shared core FPGU which may be worse than default Linux implementation. Also, there is some more overhead caused by level 1 cache trashing in CASX and L2miss spin-locks. Fortunately, these overheads are small, between 2 and 3 percent in average. We also noticed that the IDIV spin-lock performs better in all cases then standard Linux spin-lock.

4) Summarizing up the solutions: Considering all experimental results we present here, it may be concluded that, regarding the overhead introduced to an active thread, L2miss loads are the best delay instructions for improved spin-lock. It is either the best or very close to the best in all experiments we performed. More importantly, it causes the lowest overhead on realistic benchmarks in all our experiments. The other option could be CASX instruction, as it behaves very well in all experiments, while both IDIV and FDIV have their weak spots, when running with some benchmarks that use the same resources. FDIV is especially weak in T1 when the workload uses FP instructions because of the described conflicts in the core FFU. FDIV is better in T2, but it may introduce relatively high overhead when the active thread uses more complex arithmetic operations, and the overhead is not removed by moving the thread to the other pipe.

The possible argument against use of L2miss approach is that it may constantly trash critical data of the active task if they happen to be allocated in the exact cache set and bank that L2miss benchmark uses. This situation would cause an high slowdown of an active task. Fortunately, the probability of this situation is fairly low, there are 4096 sets in L2 cache and we are trashing only one.

D. Responsiveness

The effect on the workload performance introduced by additional latency of delay instruction is not directly measured in our experiments. The maximum latency to read the state of the lock exists when we add the L2 miss instruction in the spin-lock loop. It is, 110 cycles in T1 and 230 cycles in T2. This translates to 110ns and 170ns respectively in our processors. For most usages we think that this latency is acceptable. It is for approximately 2 orders of magnitude faster than switching the software context of a strand. If the impact of the L2miss latency is too high, CASX may be used instead. Its moderate latency - around 30 cycles (20-25ns) should pose only a slight response delay in time critical applications.

VI. Conclusions

In this paper, we show that a thread executing the default Linux spin-lock loop can seriously degrade the performance of other active threads when running in the T1 and T2 processors. As a rule of thumb, the larger the amount of shared resources, the higher the performance degradation over the active threads. For example, executing in the same core/pipe three instances of the default Linux spin-lock loop for T1 and T2 processors causes a slowdown on the realistic applications we used in this paper of up to 43% (31% on average) for T1 and up to 61% (42% on average) for T2. We identified the fetch bandwidth as the most critical hardware shared resource, being responsible of the performance slowdown suffered by the active threads. We create several versions of the spin-lock loop by adding different type of long-latency instructions, so that the fetch bandwidth needs of the spin-lock threads decrease. Our results show that, even if our improved spin-lock loops introduce overhead in other shared resources, they effectively reduces the contention in the fetch stage of the processor. As a consequence, our best spin-lock loop reduces the overhead on the active threads down to 3.5% on average for T1 and 1.5% on average for T2.

Although responsiveness of our proposals is larger than with the standard Linux spin-lock loop, it is better than the responsiveness of a context switch. If the responsiveness is critical we propose use medium latency delay instructions - they just slightly reduce responsiveness of the spin-lock while significantly improve performance of the active threads.

Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain under contracts TIN- 2004-07739-C02-01, TIN-2007-60625, the HiPEAC European Network of Excellence and a Collaboration Agreement between Sun Microsystems and BSC. The authors wish to thank the reviewers for their comments, Jochen Behrens, Gunawan Ali-Santosa and Ariel Hendel from SUN for their technical support.

References

[1] *Spinlock in Intel Pentium 4 and Xeon processors*, 2001.
[2] *OpenSPARC^{TM} T1 Microarchitecture Specification*, 2006.
[3] *Logical Domains (LDoms) 1.0.1 Administration Guide*, 2007.
[4] *Netra Data Plane Software Suite 2.0 Reference Manual*, 2007.
[5] *Netra Data Plane Software Suite 2.0 User's Guide*, 2007.
[6] *OpenSPARC^{TM} T2 Microarchitecture Specification*, 2007.
[7] *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3A: System Programming Guide*, 2008.
[8] Thomas E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. In *IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 1*, 1990.
[9] Phuong Hoai Ha, Marina Papatriantafilou, and Philippas Tsigas. Efficient self-tunning spin-locks using competitive analysis. *The Journal of Systems and Software 80*, 2007.
[10] John M. Mellor-Crummey and Michael L. Scott. Synchronization without contention. In *ISCA, IV*, 1991.
[11] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. Analysis of system overhead on parallel computers. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 305–316, 2007.
[12] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. Measuring the Performance of Multithreaded Processors. In *SPEC Benchmark Workshop*, 2007.