Overhead of the spin-lock loop in UltraSPARC T2

A. Introduction

Spin locks are task synchronization mechanism used to provide mutual exclusion to shared software resources. Spin locks have a good performance in several situations over other synchronization mechanisms, i.e., when on average tasks wait short time to obtain the lock, the probability of getting the lock is high, or when there is no other synchronization mechanism.

In this paper we study the effect that the execution of spin-locks create in multithreaded processors. Besides going to multi-core architectures, recent industry trends show a big move toward hardware multithreaded processors. Intel P4, IBM POWER5 and POWER6, Sun's UltraSPARC T1 and T2 all this processors implement multithreading in various degrees. By sharing more processor resources we can increase system's performance, but at the same time, it increases the impact that processes executing simultaneously introduce to each other.

We focus on the effect that spin-lock loop has on active tasks that when they execute in parallel. As standard spin-lock implementation we used Linux spin-lock loop for the Sun UltraSparc 64-bit processors and we run our experiments on Sun UltraSPARC T2. T2 processor has 8 in-order cores. It has two *execution pipes* (or simply pipe) per core. Each pipe has its own integer execution unit. A pipe consists of four hardware strands 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. These three levels of sharing resource makes T2 a very good choice to test various levels of performance interaction among threads depending on how much resources they share.

B. Linux Spin lock

Linux uses a spin-on read (Test and Test and Set) approach, see Figure 1 for assembly code implementation. In a call to this lock code, 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 11 to 14). When the lock is released the process will again try to obtain it by using read-store instruction (outmost loop). After the first loop iteration, the lock is in the nearest data cache of the processor, the first level data cache in T2. Only when the lock is released this address is invalidated by the thread which releases the lock. If the lock is already locked it is expected that a task which invokes this function will spend the most time executing lines 11 to 14[1]. We refer to this innermost loop as the *Linux default spin-lock loop*.

The main goal in spin-on-read locks is that, 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. Only when the lock is released, this address is invalidated by the thread holding the lock when it releases it. As a consequence, while the threads are spinning on a lock the average latency of each instruction in the loop is the following (lines from 11 to 14 in Figure 1):

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

As a consequence of this low CPI, threads executing the spin-lock loop are ready to fetch most of the time. In each pipe of the T2,

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

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

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

the thread selection policy is *Least Recently Fetched*. Hence, when running a thread executing the spin-lock loop it will consume most of the fetch cycles of the processor introducing overhead on the other thread.

C. Experimental

In our experiments we use a dedicated virtual machine (logical domain) managed by Sun Logical Domains (LDoms) software [1] as it introduces no measurable overhead. We used a logical domain with two cores (16 strands) and 4GBytes of dedicated memory available for our experiments.

To run our experiments we use as the framework NetraDPS [2][3]. 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. 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.

Our goal in this paper is to measure the impact of a spinning lock over an independent active application 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 it suffers. The threads spinning on the lock and the active threads are mutually independent.

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 that 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. As a future work we plan to implement some real multithreaded applications with spin-locks we propose here, and run them in a real OS.

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:

1) Initialization phase: In this phase the active thread and the spin-lock thread(s) are initialized. In the case of the spin-lock threads initialization consists of setting the lock to 1 and giving the lock to the active thread. Active threads require a different, task specific,

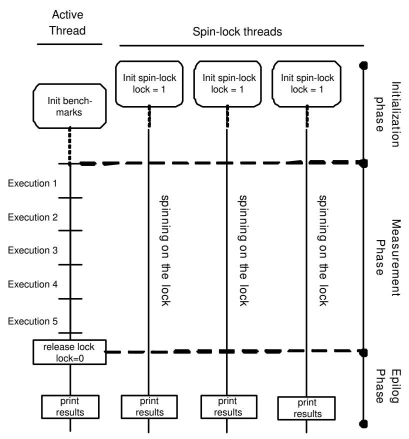

Fig. 2. Example of execution of one active thread and three spin-lock threads

initialization. First, we start the spin-lock threads and then the active thread. This way, we ensure that during the measurement phase the spin-lock threads cause a constant overhead over the active thread.

2) Measurement phase: To obtain reliable measurements of the spin-lock threads, we use the FAME (FAirly MEasuring Multi-threaded Architectures) methodology [4][5]. 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.

As an active threads we use 4 benchmarks that simulate behavior of real applications. These are:

- **Matrix by Vector Multiplication (integer and floating point):** These two benchmarks are mostly memory bound and access the memory with more or less constant stride.
- **QuickSort algorithm:** This is a cpu-bounded task.
- **Hash function:** This is the mix of a CPU intensive application and non-sequential, pseudo-random memory access.

D. Results

As may be seen in Figure 3 the default spin-lock loop in Linux causes vary high slowdown to low CPI tasks that execute on the same core than spin-locks. Given the low CPI of the spin-lock loop the the fetch bandwidth is the main source of interaction between actives threads and the spin-lock threads. Because of that we propose different variants of the Linux spin-lock loop that require less fetch bandwidth.

To achieve this, 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 needs of the spinning thread. We use 3 different instructions as we try to evaluate the most important properties which would make this 'delay' instruction the most effective.

These 'delay' instructions are:

- *FDIV* uses division pipeline of Floating Point unit of the core and has latency of 33 cycles.
- *CASX* always goes to the same address in L2 cache and takes 20-30 cycles to finish
- *L2miss* is a simple load but done on the array initialized for 'pointer chasing' in that manner that the load misses every time in L2 cache. Its latency is 220 cycles.

As may be observed in Figure 3 with this, easy to implement, approach we reduced the overhead caused by a spin-lock from 42%

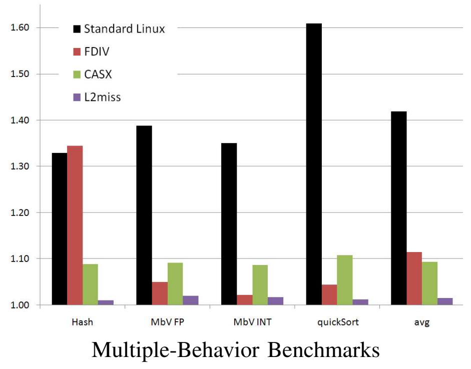

Fig. 3. Effect of 3 instances the same spin-lock loop on different active threads when run on the same core and execution pipe on T2

on average (61% in the worst case) to only 1.5% in average (2% in the worst case). This is a reduction of 38 percentage points with respect to the default Linux spin-lock loop. The price we pay for this improvement is the worse responsiveness of the spin-lock thread - the time that passes from the moment the lock is released till some of spinning threads acquires it. We measure this effect indirectly by calculating the maximum number of cycles that may pass between releasing and acquiring when we have one spin-locked thread. For *L2miss* it is around 230 cycles, for other two implementations is close to 40 cycles versus 10 cycles in default Linux spin-lock. Even with the worst delay, the use of spin-lock gives much faster response than for example context switch, so we think that its use is completely justified.

E. Conclusions

Summing up, the main contributions of this paper are the following:

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 is, the more the performance of active threads is. For example, the execution of three instances of the default Linux spin-lock loop causes a slowdown on the realistic applications we used in this paper of up to 61% and 42% on average for T2.

We show that the fetch bandwidth is the most critical shared resource and it is the responsible of the performance loss (slowdown).

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, it effectively reduces the contention in the fetch stage of the processor. Our best spin-lock loop reduces the overhead on the active threads down to 2% in the worst case and 1.5% on average. This is a reduction of 40% percentage points with respect to the default implementation of the spin-lock loop in Linux.

References

[1] *Logical Domains (LDoms) 1.0.1 Administration Guide*, 2007.
[2] *Netra Data Plane Software Suite 2.0 Reference Manual*, 2007.
[3] *Netra Data Plane Software Suite 2.0 User's Guide*, 2007.
[4] 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.
[5] 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.