Lock Contention Aware Thread Migrations

Kishore Kumar Pusukuri, Rajiv Gupta, Laxmi Narayan Bhuyan
University of California, Riverside
{kishore, gupta, bhuyan}@cs.ucr.edu

Abstract

On a cache-coherent multicore multiprocessor system, the performance of a multithreaded application with high lock contention is very sensitive to the distribution of application threads across multiple processors. This is because the distribution of threads impacts the frequency of lock transfers between processors, which in turn impacts the frequency of last-level cache (LLC) misses that lie on the critical path of execution. Inappropriate distribution of threads across processors increases LLC misses in the critical path and significantly degrades performance of multithreaded programs. To alleviate the above problem, this paper overviews a thread migration technique, which migrates threads of a multithreaded program across multicore processors so that threads seeking locks are more likely to find the locks on the same processor.

Categories and Subject Descriptors D.4.1 [*Process Management*]: Scheduling, Threads

Keywords Multicore, Lock contention, LLC miss, Migration, processor

1. Introduction

On a multicore multiprocessor system, the performance of a multithreaded application with *high lock contention* is highly sensitive to the distribution of threads across processors. In this paper, we demonstrate that the time spent on acquiring locks, as experienced by competing threads, can increase greatly depending upon the processors on which they are scheduled. When a lock is acquired by a thread, the time spent on acquiring it is longer if lock currently resides in a cache line of a remote processor as opposed to the processor on which the acquiring thread is running. In addition, once the thread acquires the lock, access to shared data protected by the lock is also likely to trigger LLC misses. This causes the thread to spend longer time in the critical section. When the competing threads of a multithreaded application are distributed across multiple processors, the above situation arises often and it increases lock transfers between processors. Frequent lock transfers between processors significantly increases long latency LLC misses that fall on the *critical path*. The larger the number of threads, the higher is the lock contention problem which causes LLC misses caused by inter-processor transfer of locks and the shared data they protect. Since the latency of LLC miss is high, the increase of LLC misses in the critical path increases both lock acquisition latency and critical section processing time, which leads to significant performance degradation.

To address the above problem, we propose a thread migration technique, which aims to reduce the variance in the lock arrival times of the threads scheduled on the same processor. The lock arrival times are the times at which threads arrive at the critical section just before they successfully acquire the lock. *By scheduling threads whose arrival times are clustered in a small time interval so that they can all get the lock without losing the lock to a thread on another processor*. Therefore, the thread migration technique ensures that once a thread releases the lock it is highly likely that another thread on the same processor will successfully acquire the lock and LLC misses will be avoided. Consequently, it reduces the lock acquisition time and speeds up the execution of shared data accesses in the critical section, ultimately reducing the execution time of the application.

2. Overview

The overview of the thread migration technique is provided in Algorithm 1. It is implemented by a daemon thread which executes throughout an application's lifetime repeatedly performing the following three steps: monitor threads; form thread groups; and affect thread grouping. The first step monitors the behavior of threads in terms of the percentage of elapsed time they spend on waiting for locks (i.e., lock times). The second step forms groups of similarly behaving threads using the lock times collected during the monitoring step. Finally the third step migrates threads across processors to ensure that threads belonging the same thread group are all moved to the same processor.

PPoPP '14, February 15–19, 2014, Orlando, Florida, USA.

ACM 978-1-4503-2656-8/14/02.
http://dx.doi.org/10.1145/10.1145/2555243.2555273

Algorithm 1: Lock Contention Aware Thread Migrations

Input: N: No. of threads; C: No. of processors.

repeat
  i. Monitor Threads -- sample lock times of N threads.
  if *lock times exceed threshold* **then**
    ii. Form Thread Groups -- sort threads according to lock times and divide them into C groups.
    iii. Affect Grouping -- migrate threads to establish newly computed thread groups.
  end
until *application terminates*;

The thread migration technique is designed to simultaneously reduce inter-processor lock transfers and preserve load balance across the processors – when some threads are migrated to a processor, others are migrated away from it. It is different from thread clustering [3], where contending threads are scheduled on the same processor. However, clustering of threads on the same processor reduces the cost of acquiring locks by sacrificing parallelism due to resulting load imbalance.

Operating system schedulers (e.g., Solaris) dynamically distribute threads across cores to balance the load across the cores but, as shown, they do not handle lock contention. In contrast, several works employ one-thread-per-core *binding* model for completely avoiding thread migrations and to maximizing performance of multithreaded programs running on multicore systems. However, on multiprocessor systems with a large number of cores, multithreaded programs that involve high lock contention exhibit poor performance with binding [2].

3. Preliminary Evaluation

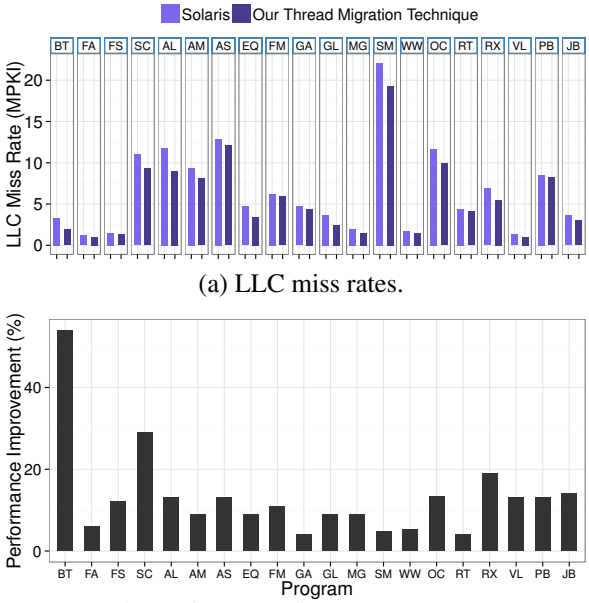
We implemented the thread migration technique on a 64-core, 4-processor machine running Oracle Solaris 11 (each processor has 16 cores). We evaluated the thread migration technique with 20 multithreaded programs including SPEC jbb2005, PBZIP2, and programs from PARSEC, SPEC OMP2001, and SPLASH2 suites. Table 1 lists the programs and their short names. Figure 1 (a) shows LLC miss rates of the 20 programs. LLC miss rate is defined as last-level cache misses per thousand instructions (MPKI). As we can see, our thread migration technique reduces LLC miss rates compared to Solaris and thus improves performance. As Figure 1 (b) shows, our thread migration technique improves performance up to 54% (average 13%).

The key contributions of our work are as follows:

- We identify the important reasons for why modern OSes fail to achieve high performance for multithreaded applications with high lock contention running on multicore multiprocessor systems.
- We develop a thread migration technique, which orchestrates migration of threads between processors with the goal of simultaneously maintaining load balance and reducing lock transfers between processors to reduce LLC misses on the critical path.
- Our thread migration technique does not require any changes to the application source code or the OS kernel.

Table 1: Programs and their short names.

PARSEC: bodytrack (BT), fluidanimate (FA), facesim (FS), streamcluster (SC); SPEC OMP2001: applu (AL), ammp (AM), apsi (AS), equake (EQ), fma3d (FM), gafort (GA), galgel (GL), mgrid (MG), swim (SM), wupwise (WW); SPLASH2: ocean (OC), raytrace (RT), radix (RX), volrend (VL); PBZIP2 (PB); SPEC jbb2005 (JB).



(a) LLC miss rates.

(b) Performance improvement (%).

Figure 1: Our thread migration technique reduces LLC misses and thus improves performance.

4. Acknowledgements

This work is supported by NSF grants CNS-1157377, CCF-0963996, and CCF-0905509.

References

[1] L. Jean-Pierre, D. Florian, T. Gaël, L. Julia and M. Gilles. Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In *USENIX ATC*, 2012.

[2] K.K. Pusukuri and D. Johnson. Has one-thread-per-core binding model become obsolete for multithreaded programs running on multicore systems. In *USENIX HotPar*, 2013.

[3] F. Xian, W. Srisa-an, and H. Jiang. Contention-aware scheduler: unlocking execution parallelism in multithreaded java programs. In *OOPSLA*, 2008.