

Brief Announcement: Evaluating Synchronization Techniques for Light-weight Multithreaded/Multicore Architectures

Srinivas Sridharan
CSE Department
University of Notre Dame
Notre Dame, IN 46556
ssridhar@nd.edu

Arun Rodrigues
Sandia National Labs*
PO Box 5800
Albuquerque, NM 87185
afrodi@sandia.gov

Peter Kogge
CSE Department
University of Notre Dame
Notre Dame, IN 46556
kogge@cse.nd.edu

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming–Parallel Programming;

General Terms: Design, Performance

Keywords: Light-Weight Multithreading, Multicore processors, Processing-In-Memory, Scalable Lock and Barrier Synchronization Techniques, Full-Empty Bits

1. INTRODUCTION

This paper deals with architectures that expose novel concurrency models by using light-weight multithreading *and* support high computation to memory ratio by leveraging technologies such as Processing-In-Memory (PIM), Embedded DRAM, or Stacked Memory [1, 2]. In this paper we explore the idea of implementing key parallel processing functions such as synchronization locks and barriers for one such architecture, namely the *Light-Weight Processing* (LWP) architecture (Figure 1).

Each LWP core is highly multithreaded, allowing a large number of hardware threads (order of 16-32 threads per core) to be concurrently in execution within both the LWP and the larger *Light-Weight Processing Chip* (LPC). The minimal state of these *Light-Weight Threads* (LWTs) make it possible to efficiently manage them in hardware. Support for large number of hardware threads needs to be backed by low overhead synchronization capability. The LWP architecture uses *Full/Empty Bits* (FEB) [1, 2] in memory to support fast producer-consumer communication.

FEB append an extra bit to each word in memory and can exist in either Full or Empty state. This bit can be (re-)set atomically when performing a load/store instruction hence providing conditional access to the memory work depending on the state of the bit. This allows threads to block on memory words when the FEB status is not the same as expected. Unlike atomic memory operations, FEBs do not lock the entire memory bank but instead only lock the memory location under consideration. Multiple producer-consumer synchro-

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

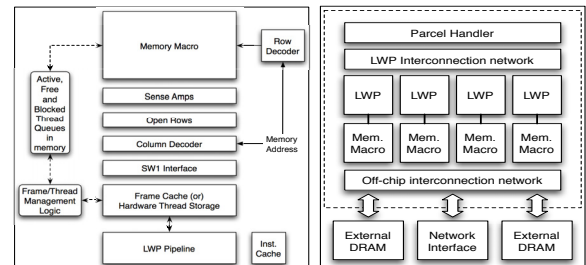


Figure 1: Single LWP core and 4-core LPC

nization on a single memory location is currently handled by a hardware mechanism that maintains a queue of threads.

2. METHODOLOGY

The synchronization algorithms used in this paper were originally mentioned in [3]. These algorithms were designed to *spin-wait* on local/remote variables to check the status of pending synchronization operations. We have modified these algorithms appropriately to use FEBs to *block* threads that are waiting for pending synchronization events.

We used the Structural Simulator Toolkit (SST) which supports a modified version on the C library (LibC). We instrumented the Pthreads synchronization routines (`pthread_barrier*`, `pthread_mutex*` and `pthread_cond*`) to use scalable mutex and barrier algorithms based on FEBs. Significant care has been taken to implement the synchronization algorithms to use single producer-consumer semantics instead of invoking the queuing mechanism required by multiple producers/consumers.

We assume that each Pthread thread gets directly mapped to a hardware LWT. The “main” thread executes on a “master” LWP and assigns child threads to the other LWP cores in a round-robin fashion. After thread assignment is complete, the main thread waits for the child threads to complete. Pthread threads have large stacks and are persistent, making LWTs have large execution state and run for longer time periods. This model does not reflect the actual hardware specification where LWT have minimal state. We are currently working on making this model better reflect the correct specification.

We developed seven microbenchmarks [2] that test synchronization mechanisms and implemented them in C language using Pthreads. The seven microbenchmarks differ

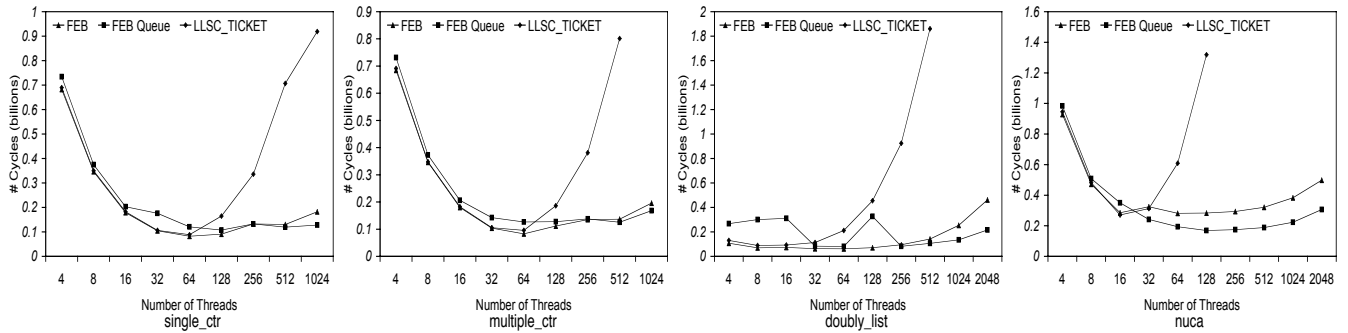


Figure 2: Lock Microbenchmarks: 8-core LPC. The x -axis represents the number of threads ranging from 4 to 2048 and the y -axis represents the total execution time (in billions of clock cycles).

Synchronization Algorithm	SMP	LWP (LLSC/FEB)
Central Barrier	LLSC	Both
Dissemination Barrier	LLSC	Both
Tournament Barrier	LLSC	Both
Tree Barrier	LLSC	Both
Test-&-Set (TS)	LLSC	LLSC
Ticket Lock	LLSC	LLSC
FEB Lock	n/a	FEB
Queue Lock	n/a	FEB

Table 1: Synchronization algorithms evaluated

on a wide range of characteristics including the number of locks, the access patterns of shared data structures with the critical section, the complexity of the critical section code. All microbenchmarks perform significant amounts of random work outside the critical section code. This is to ensure fairness in acquiring the lock and performing critical section operations.

3. EXPERIMENTAL RESULTS

Table 1 lists the various synchronization algorithms we tested. The table also lists whether FEB (blocking) or LLSC (spin-wait) was used to implement the algorithm. We compared the LWP implementations using FEB against LWP or SMP implementations using LLSC instructions. For the SMP/CMP configurations tested (4 to 16 processors), Ticket locks (with proportional back-off) had the best overall performance (in terms of clock cycles). We present only a subset of the results in this paper.

Figure 2 shows the performance of the lock algorithms (FEB Lock, Queue FEB lock and LLSC Ticket Lock) on one LPC with 8 LWP cores. In general, the FEB lock is comparable to LLSC Ticket lock for small number of threads, but greatly outperforms the LLSC Ticket lock for larger number of threads. Some of the graphs for 128 threads and above do not show the LLSC based ticket lock for this reason. We measure *speedup* as the improvement of the FEB implementations over the LLSC implementation for the same number of threads.

FEB based techniques scale better as we increase the number of threads to large values like 2048. On average, we get a speedup of around 5X for 512 threads or more and a best case speed of 13X for the *doubly_list* benchmark. For small

numbers of threads both FEB and LLSC-Ticket locks outperform the FEB Queue locks. However this trend changes as we increase the number of threads as the overheads of the queuing algorithm is less than the contention for the lock. This is clearly visible for the *Nuca* (1024 threads and above) and *Doubly_list* (128 threads and above) microbenchmarks.

4. CONCLUSIONS AND FUTURE WORK

This paper evaluates FEB implementation of scalable lock and barrier algorithms for the LWP architecture, through a traditional synchronization API necessary for supporting legacy applications. In general, the scalable FEB algorithms were useful and provided significant performance improvements over conventional spin-wait mechanisms.

This work opens a lot of opportunities for exploring lightweight multithreading and scalable synchronization mechanisms. Though not discussed in this paper, our simulation infrastructure already supports both SPLASH2 and NAS-OpenMP benchmarks [1, 2]. We are currently investigating these benchmarks and trying to characterize their behavior in terms of synchronization overheads. A number of possible enhancements to SST will make it more useful for investigating large scale LWP architectures. This includes supporting a more classical definition of light weight multithreading, migration capabilities for threads and data distribution schemes.

5. ACKNOWLEDGMENTS

This work is based in part upon earlier work supported by the Defense Advanced Research Projects Agency (DARPA) under its Contract No. NBCH3039003.

6. REFERENCES

- [1] A. Rodrigues, S. Sridharan, S. Thoziyoor, J. Brockman, K. Underwood, and P. Kogge. Enhancing Price/Performance for OpenMP using Processing-In-Memory. In *1st Workshop on Programming Models for Ubiquitous Parallelism (PMUP)*, held in conjunction with the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT), Sep 2006.
- [2] Srinivas Sridharan. Implementing Scalable Locks and Barriers on Large-Scale Light-Weight Multithreaded Systems. M.S CSE Thesis, University of Notre Dame, July 2006.
- [3] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions Computing Systems*, 9(1):21–65, 1991.