A Locked Cache-Based Synchronization Protocol for CMP

Ihab Ismail[1]
Computer Science Department
The American University in Cairo [AUC]

Khaled El-Ayat
Computer Science Department
The American University in Cairo [AUC]

Muhamed Mudawar[2]
Computer Engineering Department
King Fahd University of Petroleum & Minerals [KFUPM]

Abstract - *CMP processors are already replacing complex single core superscalar processor architectures. They offer better performance per watt and area. This is especially true in TLP rich server and web applications. Process / thread synchronization is important since CMP consists of multiple processor cores sharing cache resources including shared data structures.*

This work proposes a locked cache-based shared memory technique suitable for CMP synchronization. A proposed cache coherence protocol, called Lock-based Cache Coherence Protocol (LCCP) was designed and its performance was compared with well known synchronization primitives (LL, SC) using MESI cache coherence protocol. Experiments were performed on the modified MP_Simplesim simulator to implement current proposal. Simulation results show that LCCP outperforms the MESI protocol on the benchmark programs

Keywords: CMP, Chip Multiprocessing, Cache protocols, synchronization, multithreading

1 Introduction

Chip Multi-Processor (CMP) [1] are already becoming the design choice for future servers, desktops and even notebook computers as seen by dual core processors from IBM [13], Intel [22] and AMD. The reasons have been widely reported. Performance gains for Superscalars have reached diminishing returns [2], wire delays may be limiting future performance gains [4]. Although large delays can be managed by pipelining techniques, timing uncertainty will be a problem for designers.

Perhaps the most critical consideration is power consumption in future micro-architectures. This strongly favors simpler multi-core architectures running at reduced frequencies compared with high frequency complex superscalar architectures. *The International Technology Roadmap for Semiconductors* (ITRS) [3] projects that multi-billion transistor chips will be designed by the end of this decade with feature size around 50 nm and clock frequencies around 10 GHz. All of the above clearly favors CMPs.

Processor cores in CMP architecture can communicate together either through shared on-chip caches, through an on-chip bus in shared L2 cache architectures or through an interconnection network in case of private L1 and L2 caches.

Current chip multiprocessors (CMP) such as the Stanford Hydra, IBM Power4 or Power5, Intel's PentiumD or Pentium extreme use different architectures to interconnect the processing cores. Bus interconnect is suitable for small number of cores per CMP whereas mesh or crossbar interconnect fabric is needed for large number of cores.

1.1 Cache Hierarchy Alternatives for CMP

Cache hierarchy and cache coherence require important consideration in CMP design. There are several alternatives for building the cache architecture for a CMP. One alternative is shown in Figure 1a where each processor has its own private L1 caches with a shared L2 cache. This architecture offers good scalability as well as low access latency. The main drawback of this approach is cache coherence resulting from duplicate copies in different caches. Cache coherence protocols are needed to maintain consistency. A second architecture shown in Figure 1b provides private L1 and L2 caches. This approach is suitable for a CMP with a large number of cores. The current CMP proposal assumes private L1 caches and shared L2 cache.

1.2 Motivation

Synchronization is needed in any multiprocessing environment to maintain correct order of events. This is true for Chip Multiprocessors (CMP). Currently synchronization is performed by means of the operating system with hardware support.

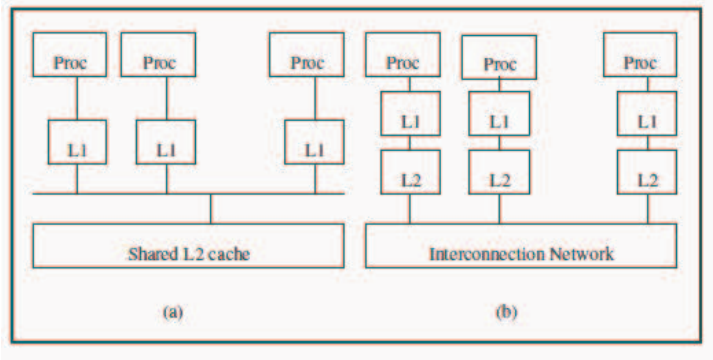

Figure 1: Cache hierarchy alternatives for CMP

This is typically implemented with Spin-locks with high overhead. Spin-locks also suffer from increased communication between the multiple processors.

A common hardware primitive to implement Locks in a synchronization event is the *Load-Linked and Store-Conditional* (LL/SC) synchronization primitive.

The LL/SC synchronization primitive is implemented by loading the synchronization variable from a memory location or cache block into a register, raising a *lock flag* and placing the address of the block in a *lock address* register.

Incoming invalidations are checked against the lock address register and if successful the lock flag is reset. The store-conditional checks the lock flag. If set the write to the synchronization variable occurs, otherwise it fails. The processor releases the lock after a successful write operation to the lock variable. For LL/SC to be successful, both operations must be successful.

This primitive is used to implement Locks, spin locks and barriers. Spin-locks suffer from high overhead as well as from increased unnecessary communication between the multiple processors

This work contributes to the efficiency of locked synchronization by elimination of the high unnecessary overhead associated with spin locks. This is accomplished by enhancing the LL/SC synchronization primitive with support within the cache coherence protocol. The motivation is to increase the successful rate of store-conditional command in the CMP design. A Lock-based Cache Coherence Protocol (LCCP) is proposed to address above concerns.

2 Cache coherence

Shared memory multiprocessors have the advantage of sharing code and data structures among the processors comprising the parallel application. This sharing can result in several copies of a shared block in one or more caches at the same time. To maintain a coherent view of the memory, these copies must be consistent. This is the *cache coherence problem.*

Cache coherence schemes include protocols that maintain coherence in hardware, and software policies that prevent the existence of copies of shared, writable data in more than one cache at the same time.

Hardware coherent protocols include snoopy cache protocols, directory cache protocols and cache-coherent network architectures [6].

Snoopy cache coherence protocols are further divided into *write-invalidate* protocols such as Goodman, Synapse and Berkeley protocols [6], MSI , MESI and MOESI protocols [7] and *write-update* Firefly protocol [8].

Full-map directory, limited-directories and chained-directories are examples of directory-based cache coherence protocols [9].

It should be noted that cache coherence protocols are optimized for data access and not optimized for locked synchronization needs. The excess traffic between cores and unnecessary cache invalidations motivated this work. Synchronization is supported within the cache coherence protocol to decrease the inter-processor traffic within the CMP.

3 Chip Multiprocessors (CMP)

Perhaps the earliest CMP was the Stanford Hydra [1]. It integrates four MIPS-based cores with their primary caches on a single chip together with a shared secondary cache. Processor cores communicate together by two buses. Hydra uses the LL/SC synchronization primitive.

The Compaq Piranha CMP [11], designed for parallel commercial workloads, consists of 8 CPUs. The IBM POWER4 [12] consists of two processor cores per chip, each core having its own instruction

and data L1 caches. The two processor cores share the on-chip L2 cache through a crossbar called core interface unit (CIU). The POWER5 [13] also from IBM is similar but also supports 2-way SMT.

Another MIPS based CMP is MIT's Raw microprocessor [14]. It consists of 16 tiles where each tile contains an eight stage in-order single-issue MIPS-style processor, a static communication router, and two dynamic communication routers. The Raptor CMP [15] is SPARC based and has four cores per CMP.

Other commercial dual-core CMPs include Intel's PentiumD, Pentium extreme., Montecito, Core duo CMP for notebook computers and AMD's Athlon X2 and Opteron chips.

4 Lock-based Cache Coherence Protocol (LCCP)

Lock-based Cache Coherence Protocol (LCCP) implements synchronization by adding a LOCK state to the MESI cache coherence protocol. LCCP uses cache invalidate method to maintain coherence among the individual L1 caches and uses memory write-back policy to reduce the traffic between the L2 cache and the external memory.

One problem with LL/SC is that store-conditional may send invalidations or updates if it fails. In that case two processors may keep invalidating or updating each other and failing. This situation is called ***Livelock*** situation.

Livelock in an invalidation-based cache coherence system is caused when all processors attempt to write to the same memory location at the same time. When the cache block is loaded into the cache in the modified state for store, and before the processor is able to complete its store, the block may be invalidated by another processor attempting to load the same cache block for store. The first processor's store attempt will miss and it will load the cache block again. This situation can repeat indefinitely.

A new LOCK state has been added to the MESI cache coherence protocol to resolve Livelock. On detecting a Load-Linked command, the processor reads the cache block in the LOCK state so that no other processor can read or modify it until the lock is released after a store-conditional. Requests from other processors to access the locked block will be stored in a wait table until the lock is released.

4.1 Finite State Machine diagram

Figure 2 shows the finite state machine (FSM) diagram of LCCP

The solid line in the diagram is considered as an action by the processor, while a dashed line is an action caused by the bus.

The FSM has 5 states; Invalid, Shared, Exclusive, Modified and Locked. The first four states are the same states of the standard MESI protocol.

Locked (L):

This is the new state where the processor loads a cache line that contains a critical section by load-linked (LL) instruction.

The processor issues four types of requests: reads (PrRd), writes (PrWr), loads-locked (PrRdL) and writes-conditional (PrWrC). The reads and writes could be to a memory block that exists in the cache or to one that does not. In the latter case, a block currently in the cache will have to be replaced by the newly requested block. The bus allows the following transactions:

- **Bus Read (BusRd):**

  This transaction happens when a processor misses a read by PrRd and the processor expects data as a result. The cache controller puts the address on the bus and asks for a copy to read. Another cache or memory will provide that copy.

- **Bus Read Exclusive (BusRdX):**

  This transaction is generated when a processor misses a PrWr to a cache block. The cache controller puts the address on the bus and asks for an exclusive copy to modify. All other caches are invalidated.

- **Bus Read Locked (BusRdL):**

  This transaction is generated when a processor misses a PrRdL for a cache block. The cache controller puts the address on the bus and asks for a copy to lock until it writes to it. All other caches are invalidated.

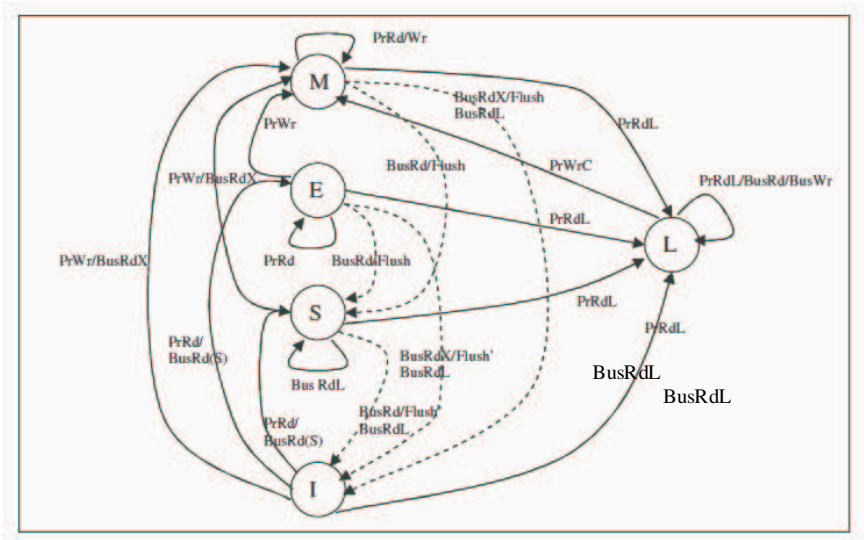

Figure 2: State machine diagram for the Lock-based cache coherence protocol. PrRdL stands for Processor-Read-Locked and PrWrC stands for Processor-Write-Conditional

4.2 Cache coherence details

a) Read Miss (no other copy is found):

In case of a read-miss in the Level 1 cache (L1$) of one of the processor cores and no other copy is found in the other L1 caches, the processor will check if the line is found in the L2$. If there is a copy in L2$, it will be provided to the requesting L1$. If not, the copy will be provided by memory. The copy in L1$ will in EXCLUSIVE state and L2$ in SHARED state.

b) Read Miss (one or more copies are found):

In case a cache miss occurs in one of the L1 caches and there is only one copy, this copy will be provided to the requesting cache and both lines will be in SHARED state. If there is more than one copy, one processor will provide the copy in SHARED state. L2$ will have its copy in the SHARED state.

c) Write hit:

In case there is a write hit in the L1$ and the status of the line is EXCLUSIVE, it will be changed to MODIFIED. A copy will be written through to the L2$ with the state MODIFIED. If the state of the line is MODIFIED, the L2 copy will be updated with the MODIFIED state. If the line is in the SHARED state, the other copies will be invalidated and the status of the line is changed to MODIFED. The line in the L2$ will be updated and the status will be changed to MODIFIED.

d) Read-Modify-Write miss:

In case of a read miss in a read-modify-write operation in both the L2$ and the L1$, the L1$ gets a copy from main memory in the LOCKED state. If the cache block is in the SHARED or MODIFIED states in the L2$, the L2$ will broadcast invalidation signal to all the processors. L2$ will provide the cache block to the requesting L1$ which will read it in the LOCKED state and the L2$ will invalidate its copy and go to INVALID state.

The cache block will be modified and then a write hit occurs for the store-conditional. After the write the cache block changes its state from LOCKED to MODIFIED and a copy will be written through to the L2$ in the MODIFIED state.

If a read request for the locked block comes from another processor, it will find that the block is in the LOCKED state and will wait for the release of the block in a wait-table. Figure 3 shows a schematic of the wait-table. Each entry in the wait-table consists of a valid bit, line tag and a processor ID

Valid bit	Line Tag	Processor ID
Valid bit	Line Tag	Processor ID
Valid bit	Line Tag	Processor ID
Valid bit	Line Tag	Processor ID

Figure 3: Architecture of the lock-release wait-table

Figure 4 shows the structure of the *Processor ID* field in an entry in the wait-table.

1	**0**	**1**	**……**	**0**

Figure 4: Structure of the Processor ID field

As shown in Figure 4, P1 and P3 are waiting to acquire the lock variable. The length of the Processor ID is equal to P bits where P is the number of processor cores on the CMP. The size of the wait-

table is equal to the size of the L1 cache of a single processor core.

4.3 Architecture of the CMP with LCCP

Figure 5 shows the architecture of the CMP implementing the LCCP. Each processor core has its own L1 caches (Instruction & Data) and all the cores share the on-chip unified L2 cache. The wait-table is also implemented on-chip. The processor cores, the shared L2 cache and the wait-table are all connected by an internal bus that can transfer a cache block in one clock cycle.

When a processor successfully finishes an SC instruction, the processor checks with wait-table controller whether another processor is requesting that line tag. If yes, it provides the copy to the requesting processor and L2 cache. The state of the line remains in the LOCK state. If not the processor changes the state of the line from the LOCK state to the MODIFIED state.

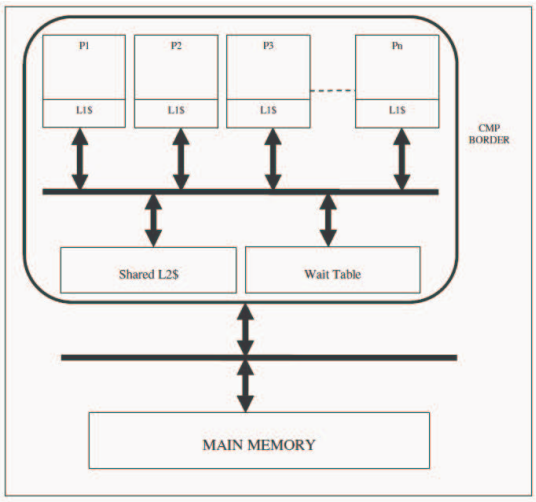

Figure 5: Architecture of the CMP implementing the Lock-based Cache Coherence Protocol

5 Experimental methodology

A functional multiprocessor simulator SS_CMP was developed based on the MP_Simplesim simulator [17]. The original MP_Simplesim was written by Naraig Manjikian at Queens University in Canada. MP_Simplesim is a multiprocessor simulator that is based on the Simplescalar simulator [18].

The instruction-set of MP_Simplesim was extended by adding the load-linked (LL) and store-conditional (SC) instructions. The architecture was modified to include shared L2 cache and the wait-table.

Benchmarks from the SPLASH2 benchmark suite [19] [20] were used to test the above architecture in our simulations. The benchmarks included the Ocean non-contiguous-partitions, Ocean contiguous-partitions and Barnes-Hut applications.

The following performance parameters were monitored to measure the performance of LCCP:

a) Total execution time of the benchmark. (sec)

b) Execution rate of instructions in (Ins/sec).

c) Number of invalidations that were responded to by other processors

6 Simulation results

6.1 Execution Time

The execution time of the benchmarks was measured on 2 configurations:

a) MP_Simplesim with shared L2 cache using MESI and spin-locks for synchronization.

b) MP_Simplesim with shared L2 cache using LCCP as the cache coherence protocol including support for synchronization.

The execution time represents the total simulation time of all the processor cores in a given simulation run and not the time taken by each individual processor to finish its simulation. The individual time of each processor core is not available from the simulator.

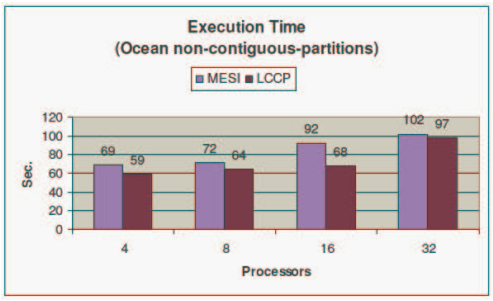

Figure 6: Execution time of Ocean non-contiguous-partitions benchmark

It was observed from Figures 6 and 7 that execution times for LCCP showed improvement of 26% and 21 % for 16 processors running Ocean and Barnes-Hut benchmarks respectively. For 4 - 8 processors a smaller improvement in performance was observed.

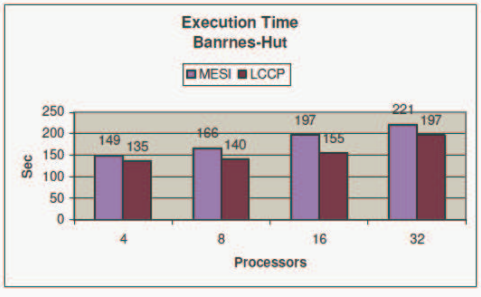

Figure 7: Execution time of Barnes-Hut benchmark

For 16 processors, synchronization demand and overhead resulted in significant performance improvement.

For 32 processors, both execution times of LCCP and MESI increased due to the increase in the instances of the program which led also to the increase in the copies of the lock variables that need to be synchronized and this appeared in the decrease in the percentage of execution time improvement of LCCP compared to MESI.

It was observed that execution times increased as the number of processors increased, contrary to expectation. This is due to Simulator limitations.

6.2 Execution rate

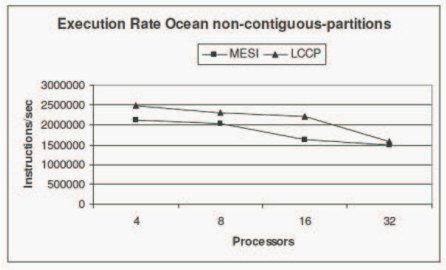

Figure 8: Execution rate Ocean non-contiguous-partitions benchmark

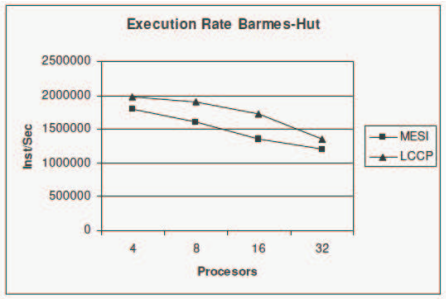

Figure 9: Execution rate Barnes-Hut benchmark

Figures 8 and Figure 9 show the execution rate of the benchmarks in instructions per second. Execution rate for the benchmarks running LCCP was higher compared with that of MESI..

6.3 Acknowledged Invalidations

The number of acknowledged invalidations is an important parameter as it reflects the effect of avoiding the *Livelock* condition when one processor wants to load a cache line in the lock state and another processor has another copy of the requested cache line. When the cache line is locked the requesting processor knows that the owner processor is in the critical section and will not invalidate its copy but will be spinning waiting for the line in the wait table.

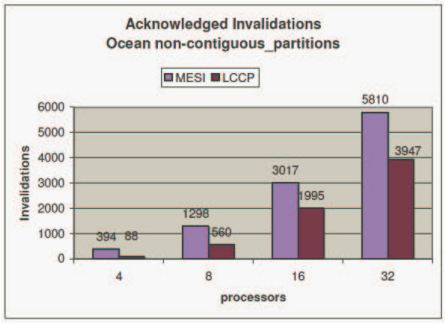

Figure 10: Number of acknowledged invalidations - Ocean non-contiguous-partitions

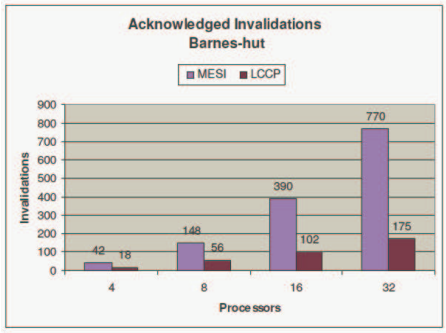

Figure 11: Number of acknowledged invalidations – Barnes-Hut

As observed from Figures 10 and Figure 11, there is significant reduction in the number of acknowledged invalidations in favor of LCCP compared with MESI.

7 Conclusions

Process synchronization between threads running on different processor cores if not properly supported, results in large synchronization overhead as well as unnecessary cache block invalidations.

This research contributes to the problem of synchronization between processor cores on CMP by introducing a new state to the MESI cache coherence protocol and on chip wait-table. The new cache coherence protocol is called lock based cache coherence protocol (LCCP). It was implemented and simulated using MP_Simplesim as the base simulator

modified to support the LCCP protocol and shared L2 cache mode. Simulation experiments using the SPLASH2 suite were conducted. The results show that LCCP outperformed MESI due to the elimination of the spin-lock overhead. LCCP also resulted in significant reduction in the number of acknowledged invalidations compared to MESI protocol. This is due to the wait-table architecture which saved unnecessary invalidations that could happen during spin-waiting.

8 Future work

As the number of cores per chip increase in the future, it would be interesting to investigate the scalability of this technique versus other lock free synchronization techniques.

References

[1] Lance Hammond, et al., "The Stanford Hydra CMP," *IEEE MICRO*, March-April 2000.

[2] Jaehyuk Huh, Doug Burger and Stephen W. Keckler, "Exploring the Design of Future CMPs," *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'01),* 2001.

[3] http://public.itrs.net

[4] Benini L. and De Micheli G., "Networks on Chip: A New Paradigm for Systems on Chip Design." *Proceeding of the 2002 Design, Automation and Test Conference and Exhibition (DATE'02)*, IEEE 2002.

[5] Pierre Guerrier and Alain Greiner, "A Generic Architecture for On-Chip Packet-Switched Interconnections," *Proceedings of the conference on Design, automation and test in Europe,* pp. 250-256, 2002.

[6] Per Stenström, "A Survey of Cache Coherence Schemes or Multiprocessors," *Computer*, Vol. 23, No. 6, June 1990, pp. 12-24.

[7] David E. Culler and Jaswinder Pal Singh, "Parallel Computer Architecture, A Hardware/Software Approach," *Morgan Kaufmann Publishers Inc*. 1999.

[8] James Archibald and Jean-Loup Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Transactions on Computer Systems*, Vol. 4, Num. 4, November 1986, pp. 273-298.

[9] David Chaiken, et al., "Directory-Based Cache Coherence in Large-Scale Multiprocessors." *IEEE Computer*, pp. 49-58, June 1990.

[10] Theo Ungerer, Borut Robic and Jurij Silc, "A Survey of Processors with Explicit Multithreading," *ACM Computing Surveys*, Vol. 35, No. 1, pp. 29-63, March 2003.

[11] Luiz André Barroso, et al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[12] Joel M. Tendler, et al., "POWER4 System Microarchitecture." *Technical White Paper*, October 2001.

[13] Ron Kalla, Balaram Sinharoy and Joel M. Tendler, "IBM POWER5 chip: A Dual-Core Multithreaded Processor", *IEEE MICRO*, pp. 40-47, March-April 2004.

[14] Michael Bedford Taylor et al., "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs." *IEEE MICRO*, pp. 25-35, March-April 2002.

[15] Sang-Won Lee et al. "RAPTOR: A Single Chip Multiprocessor," *The First IEEE Asia Pacific Conference on ASICs*, pp. 217--220, 1999.

[16] Eric H. Jensen, Gary W. Hagensen and Jeffrey M. Broughton, "A New Approach to Exclusive Data Access in Shared Memory Multiprocessors," *Technical Report UCRL-97663, Lawrence Livermore National Library, Livermore CA*, November 1987.

[17] Naraig Manjikian, "Multiprocessor Enhancements of the SimpleScalar Tool Set," *ACM Computer Architecture News*, vol. 29, no. 1, pp. 8-15, March 2001.

[18] D. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0 Tech. Report 1342, Computer Sciences Department, University of Wisconsin-Madison, June 1997.

[19] J.P. Singh, W. Weber and A. Gupta, "SPLASH: The Stanford Parallel Applications for Shared Memory," *Computer Architecture News* 20 (1): 5-44, 1992.

[20] www.flash.stanford.edu/apps/SPLASH

[21] M. Herlihy and J. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures", *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993

[22] L. Hammond, et al., "Transactional Memory Coherence and Consistency," *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004

[1] The Author would like to thank the Computer Science Dept. at AUC for providing the tools used in conducting that research.

[2] The author would like to acknowledge KFUPM for its support in the preparation and presentation of this paper at PDPTA'06.