

Appears in the “XVI Jornadas de Paralelismo”. Granada, Spain, September 2005

KIMP: Multicheckpointing Multiprocessors

Enrique Vallejo¹, Marco Galluzzi², Adrián Cristal², Fernando Vallejo¹, Ramón Beivide¹, Per Stenström³, James E. Smith⁴ and Mateo Valero²

¹*Grupo de Arquitectura de Computadores, Universidad de Cantabria*

²*Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya*

³*Dept. of Computer Science and Engineering, Chalmers University of Technology*

⁴*Dept. of Electrical and Computer Engineering, University of Wisconsin-Madison*

Abstract

Multiprocessors are coming into wide-spread use in many application areas, yet there are a number of challenges to achieving a good tradeoff between complexity and performance. For example, while implementing memory coherence and consistency is essential for correctness, efficient implementation of critical sections and synchronization points is desirable for performance.

The multi-checkpointing mechanisms of Kilo-Instruction Processors can be leveraged to achieve good complexity-effective multiprocessor designs. We describe how to implement a Kilo-Instruction Multiprocessor that transparently, i.e. without any software support, uses transaction-based memory updates. Our model not only simplifies memory coherence and consistency hardware, but at the same time, it provides the potential for implementing high performance speculative mechanisms for commonly occurring synchronization constructs.

1. Introduction

Multiprocessors are rapidly becoming the standard for computing platforms, including

simple single-board and even single-chip systems. Many of these designs implement shared memory multiprocessors because it provides a clear programming model where sharing code and data structures is simplified. However, there are several aspects that complicate the design and programming of a shared memory system and limit its performance:

- The *interconnection hardware*, which provides quick access to data held in remote memory - Accessing data from remote memories is much slower than from local memory, increasing the effective memory latency.
- The *coherence protocol*, which manages the correct sharing of memory values among private caches
- The *consistency model*, which manages the correct ordering of memory operations to different memory locations
- The need for data sharing can generate access conflicts, which are solved with *exclusion mechanisms*.
- Finally, *synchronization operations* which ensure that different program threads can cooperate with each other

A way to solve the above-mentioned problems is to employ **transactional memory systems**. These systems implement memory operations as transactions that are guaranteed to be atomic, and which simplify the design or improve the performance of a multiprocessor system. Much work has been done on this field and we list some important recent work:

- *Transactional Lock Removal* (TLR) [17] detects critical sections, and dynamically substitutes them with transactions, eliding the lock acquisition. Transactions are used as a

This work has been supported by the Ministry of Education of Spain under contracts TIN-2004-07739-C02-01, TIN-2004-07440-C02-01, grants AP2003-0539 (M. Galluzzi) and AP-2004-6907 (E. Vallejo), the HiPEAC European Network of Excellence, and the Barcelona Supercomputing Center. J. E. Smith is partly supported by the NSF grant CCR-0311361. Per Stenström is partly supported by the Swedish Research Council under contract VR 2003-2576.

substitute for critical sections, to augment parallelism when no conflict occurs.

- *Thread-level Transactional Memory (TTM)* [14] is a software-hardware approach that covers different levels of the system. It implements transactional support at thread level, so that the programmer specifies the transactions, with support in lower levels of the operating system and hardware.
- *Transactional Coherence and Consistency (TCC)* [9] proposes a new shared memory model where atomic transactions are the basic unit for communication, simplifying both parallel applications and coherence and consistency hardware. As TTM, this proposal modifies the programming model, forcing the program to be divided into transactions.

In prior work, the memory latency problem has been shown to be significantly reduced by **Kilo-Instruction Processors (KiP)** [1-4]. These processors can hide the memory latency by supporting thousands of in-flight instructions. [5] introduces a multiprocessor composed of KiP, called **Kilo-Instruction Multiprocessor**.

In this paper, we describe a *correct implementation* of a Kilo-Instruction Multiprocessor, which will henceforth be referred as **KIMP**, which takes advantage of the underlying uniprocessor mechanisms to simplify the design and to improve performance. KIMP leverages the multiple checkpointing mechanism of the Kilo-Instruction Processors 1) to perform memory updates in a transactional manner, and 2) to apply speculation mechanisms to execute through critical sections and synchronization points in a straightforward manner.

Performing memory updates in KIMP as if they are transactions leads to *implicit transactions*; i.e., they are orchestrated in hardware, without any software support and are transparent to the programmer. Consequently, the proposed KIMP comprises all the advantages of a transactional memory system, including the implementation of sequential consistency, in a natural and efficient manner. On the other hand, having the ability to speculate through critical sections and synchronization points can improve system performance when such constructs are conservatively coded. Hence, the KIMP design is a high-quality complexity-effective option for future multiprocessor systems.

2. Background

2.1. Kilo-Instruction Processors

Kilo-Instruction Processors [1] hide large latencies, specifically due to memory accesses, because the processor allows thousands of instructions to be in-flight at the same time. However, in order to increase the number of in-flight instructions we must increase the capacities of several resources, the most important ones being the re-order buffer, the instruction queues, the load/store queues and the physical registers. Unfortunately, simply up-sizing these structures, is not feasible. In order to overcome these difficulties, Kilo-instruction processors employ a number of different techniques, to arrive at an overall implementation. The main technique consists of multi-checkpointing long latency instructions instead of having a large ROB [1].

Multi-checkpointing. At specific instructions during program execution, generally branches and long latency instructions like loads that miss in the L2 cache, a checkpoint is taken. The checkpoint is a snapshot of the processor state. If an exception or branch misprediction occurs the state is rolled back to the closest checkpoint prior to the excepting instruction.

As the execution advances, in-flight instructions corresponding to the different checkpoints are executed. When these speculatively executed instructions finish, they remain in the processor queues if they are memory operations, otherwise are flushed from the processor. Only when all the instructions corresponding to the oldest checkpoint are finished, the processor commits the checkpoint atomically, consequently removing memory operations from the load or store queue and committing all the results speculatively calculated. Then, with this action, the speculative instructions become globally performed. Figure 1 shows an example of the multiple checkpointing mechanism, where oldest instructions are to the left. Black instructions are finished, but their results are not committed. In case a), instructions from three consecutive checkpoints are in flight. In case b), the second checkpoint is finished, as all the instructions within it are finished, but it can not commit as it is not the oldest checkpoint in the pipeline. In case c), the first checkpoint (the oldest

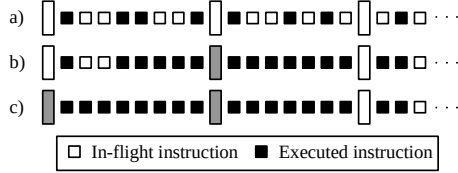


Figure 1. Multiple checkpoints.

one) finishes, so it and the second checkpoint can commit, making the third checkpoint to be the new oldest checkpoint.

2.2. Kilo-Instruction Multiprocessors

A Kilo-Instruction NUMA multiprocessor configuration [5], has been shown to effectively hide large latencies coming from both local and remote memory accesses, including latencies due to the interconnection network. Figure 2 shows the reduction in the execution time for different benchmarks from the Splash2 suite, when using 1024 in-flight instructions as compared with 64; the memory access time is 500 cycles. However, in [5] only an evaluation of the performance achieved by the system is done, leaving out any kind of architectural detail. Describing the system architecture in more detail is one of the goals of this paper.

3. KIMP overview

The KIMP design is based on a small-scale SMP interconnected via a snoopy bus. Our proposal takes advantage of the multiple checkpointing, so the cost of implementing a KIMP can be very low. Different processors execute different portions of code, taking different checkpoints as the execution advances, and being able to roll back execution to a certain checkpoint in case of finding a problem.

All the in-flight instructions remain speculative until their corresponding checkpoint commits. This means that memory instructions remain in the processor queues and do not modify the local cache or the global memory, and they are subject to a rollback. In Figure 3, for example, P_3 can commit checkpoint Chk_{31} , when all the instructions up to Chk_{32} have been completed. In the KIMP system, this commit is followed by the

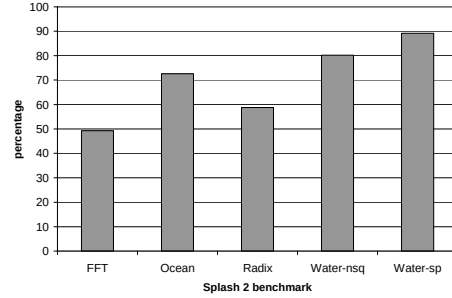


Figure 2. Performance of Kilo-Ins. Multiprocessors.

atomic broadcast of all the cache tags that the processor has modified during the checkpoint execution, these are, the pending memory updates. By “atomic” we mean that after the processor gets access to the bus it does not release the bus until all the tags have been broadcast.

Remote processors snoop the memory updates searching for a conflict with the loads they have in their load queues, which are speculatively executed and are not already committed. In case of a conflict, the remote processor is forced to roll back, because it has speculatively used data that, at this point, is discovered to be “previously” modified. Thus, as long as conflicts are not found, speculatively executed instructions are not discarded. Finally, if no rollback happens, the speculatively executed instructions are globally

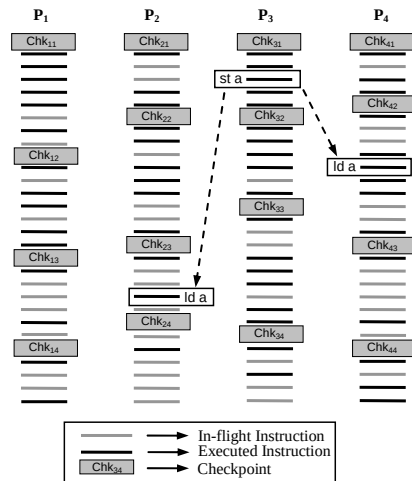


Figure 3. Execution flow for 4 processors.

performed when the checkpoint commits. In the example from Figure 3, the broadcasting of a store to a given memory location “a” conflicts with two other processors that have already speculatively loaded from location “a”, but the loads have not been already committed. In this example, P_2 is rolled back to Chk_{23} and P_4 to Chk_{42} , forcing its newest instructions to be discarded.

Furthermore, multi-checkpointing yields improved performance if simple hardware modifications are made to support the concurrent execution of critical sections and speculative execution beyond synchronization points. These improvements are described in section 7.

3.1. Implicit transactions

In the KIMP system, we define a *transaction* as the atomic execution of the instructions included between a committing checkpoint and the next one. We say the proposed system executes *implicit transactions*. They are *implicit* because they are automatically hardware delimited, by means of the checkpointing mechanism, and the programmer does not need to know that the system provides such transactional behavior. Therefore, the ISA does not need any change, and current binaries can be directly executed. Note that this idea differs from the concept of transactions in previous works [9][14] where transactions are considered as programming constructs that normally need a hardware support.

The transactional behavior is provided by the speculative execution (which provides isolation between different transactions) and the atomic in-order validation (providing consistency and atomicity), together with the described conflict detection mechanism.

The system does not care about conflicts that can exist during the execution of transactions because they are not revealed (not released to the memory hierarchy) until one transaction commits, this is, they are isolated.

Therefore, KIMP provides a correctness substrate, constituted of speculative execution, atomic validation, and the rollback mechanism, which ensures that code is executed correctly.

3.2. Adaptive transaction length

We propose that transaction lengths should be adaptive, decreasing in case of frequent rollbacks and increasing as long as no rollbacks occur. In case of frequent consistency violations, the length of the transactions decreases, also decreasing the number of violations and the number of instructions that are discarded in case of a rollback. This is valid as the correctness substrate ensures correct work independently of the instruction where a checkpoint is taken.

4. Memory coherence

In KIMP, a snoopy bus based multiprocessor, the overall coherence protocol is simplified. Lines are not maintained in “shared” or “exclusive” state, as dictated by MESI-like cache coherence protocols. Our broadcast-based approach works as either a snoopy write invalidate or write update protocol as follows.

Basic Operation. During a transaction, the cache is not modified by local stores, only speculative loads are performed. Once a transaction commits, the pending stores are atomically performed and the broadcast mechanism transmits the changes to the rest of processors, updating or invalidating the remote cache lines. This way, when the stores from a transaction need to be broadcast, the corresponding processor will get control of the bus and release it only when all its memory updates are globally performed. This operation prevents other processor from broadcasting memory updates simultaneously and protects the memory system from coherence and consistency problems.

Finally, if there is an update or invalidation of a cache line, the corresponding processor will update or invalidate those cache lines matching a snooped address, and the execution will roll back to the previous checkpoint.

Broadcast Information. The contents of the broadcast message will determine the snoopy type: if the packet contains the written data, the protocol will behave as write update. Else, if the packet only contains the updated cache line addresses, remote processors will invalidate those lines, and the protocol will work as write invalidate.

5. Memory consistency

5.1. Consistency models

The memory consistency model of a shared-memory multiprocessor determines how the system can overlap or reorder memory operations. Different models offer a trade-off between programming simplicity and performance:

- Sequential Consistency (SC) [11], the most restrictive model, guarantees that interleaved memory operations from different processors appear to execute in program order, at the cost of a generally lower performance [18].
- On the other hand, less restrictive models, such as Release Consistency (RC) [6] provide a higher performance, at the cost of not ensuring strict ordering of operations.
- Other consistency models provide intermediate performance and restrictions, such as Processor Consistency [7].

Sequential Consistency is the most desirable model, as it is the simplest model to understand and it provides the most intuitive programming interface. A basic implementation of SC requires a processor to delay each memory access until the previous one is completed, what is simple but clearly leads to a low performance. However, recent proposals preserve SC without compromising performance. Some of these are:

- SC++ [8] makes use of hardware speculation for both load and store operations, and preserves SC by rolling back when a consistency violation is found. This way the system can rely on reordering and overlapping memory operations for performance similar to that achieved with the RC model.
- TCC [9] proposes a parallel model based on software-delimited transactions, with a sequential ordering between them.

5.2. KIMP maintains SC

We can extend the definition of sequential consistency to transactions, and require only transactions from each processor to be in order. The resulting global order will be an arbitrarily interleaved succession of transactions that will also meet the basic definition of SC since it corresponds with one of the possible sequentially consistent global orderings. We show an example in Figure 4; the third column shows that respecting the program order for transactions will

also respects program order for memory operations.

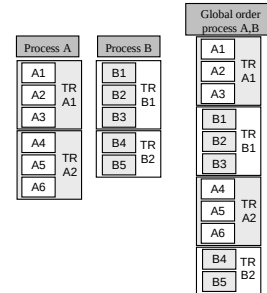


Figure 4. Sequentially consistent reordering of checkpoints from 2 processors.

Furthermore, the KIMP environment allows the execution of memory operations out-of-order. This does not compromise the correctness of the SC model because:

1. All the memory operations are executed speculatively, the stores are delayed until the transaction can commit.
2. All the speculative loads that match the address of a previous pending store receive the correct value thanks to the usual store-forwarding mechanism.
3. The snooping of memory updates from the bus ensures that the values speculatively loaded are valid unless an address match is found, which would produce a rollback of the transaction.
4. Finally, the memory updates from the transaction are atomically broadcast only when the transaction commits, making the pending stores globally performed at this moment.

Thus the result of a transaction is the same, independently of the instructions execution order.

6. Locks and barriers

Lock structures control the access to critical sections by allowing only one process, the lock owner, to enter and to read and modify shared variables. A typical critical section, using load-linked and store-conditional instructions, is shown in Figure 5.

Critical sections ensure exclusive access to the lock owner, forcing any other threads desiring to enter the critical section to stall until the lock is released. Sometimes, this stall is unnecessary, as some threads may not actually modify any data, or

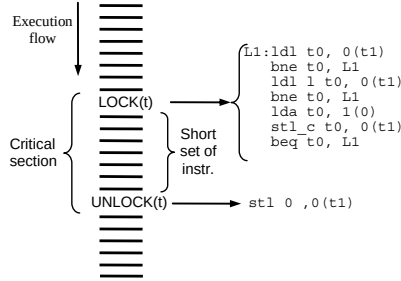


Figure 5. A typical critical section.

they modify different fields of a shared data structure. In these cases, parallel execution could be allowed, avoiding the stall of threads waiting for the lock.

The parallel execution of the critical section eliminates the dependence of other threads on the lock owner, which can sometimes generate very long waits, for example if the lock owner is suspended. The problem of critical sections is even more important in these cases.

On the other hand, Flags and barriers are used to synchronize different threads. A correct execution should make all threads wait for the barrier to open, before continuing execution. Similar to the critical section case, this stall can be avoided in those cases, in which there is no real data interaction between different threads.

6.1. Related work

In [16-17] Rajwar and Goodman propose SLE, a hardware approach that detects lock constructions and avoids acquiring them, leaving the critical section open and thus allowing several instances of the same critical section to execute concurrently. Correctness is preserved by detecting data collisions. All the data used during a speculative section is kept in the local cache, and a remote request to write that address forces the execution to roll back to the checkpoint. Thus, if no collision is detected, several instances of the critical section can be executed in parallel, and if a collision happens, only one of them advances.

In [13] Martínez and Torrellas propose Speculative Synchronization. It preserves one safe thread, which actually acquires the lock, and multiple speculative threads, that detect the busy lock and execute speculatively. When the lock

owner releases it, the rest of the threads commit their state or compete for ownership of the lock. Relating barriers, when such a construct is detected, execution is continued further, in a speculative state. The instructions after a barrier remain speculative, waiting for the barrier to open before validating.

In [19] SLR, an improvement to the previous ideas is proposed. All the speculative threads execute their critical section, even though they may have a conflict. An arbiter dictates the order in which they commit, minimizing collisions.

In TCC [9] every access to shared data has to fall in the same transaction as the computation and the update of the data. Software locks are replaced with special instructions that mark a transaction change. This naturally allows the parallel execution of critical sections, and detects conflicts in case of data collision, without the need of using control variables. Flags and barriers are substituted with phase numbers assigned to each transaction. This way, transactions with the same phase numbers can commit in any order, whereas transactions with consecutive phase numbers are ensured to commit sequentially. Speculation is implemented naturally, as transactions after the phase change can be executed, but not committed.

6.2. KIMP and critical sections

The correctness substrate ensures valid operation, including critical sections execution. In this section, we study and propose some improvements that allow KIMP to execute critical sections in parallel, in those cases where there is no real data collision. As the code from a critical section is executed different checkpoints are taken. In Figure 6, cases a) to c) show different possible checkpointing schemes.

Basic KIMP behavior. When a processor reaches the lock, it checks the value of the lock and acquires it if it is free. This acquire operation remains in speculative state in the processor queues. If a new checkpoint is taken inside the critical section, as in Figure 6 a), the validation of T1 will globally acquire the lock, forcing remote processors inside the critical section to rollback, due to the lock variable invalidation. After that, remote processors will not enter the critical section. The validation of T2 unlocks the section and remote processors can start executing it. The

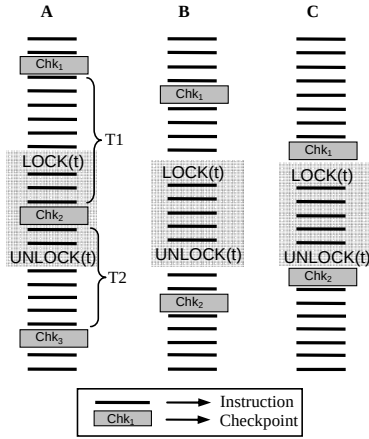


Figure 6. Different checkpointing schemes.

same invalidation happens if no checkpoint is taken until unlocking the critical section, as the lock variable is also written. This case is shown in Figure 6 b), and it should be the most frequent case, due to the short nature of critical sections.

Thus, the validation of a transaction that has executed a critical section will cause any other processor that is executing it to rollback, which means that only a single valid processor stays inside a critical section. These examples show that basic KIMP ensures the correctness of code, wherever checkpoints are taken.

Enhancing KIMP. If the lock variable is the only interaction between different processors accessing the same critical section, parallel execution is feasible. In case b), the store of the lock variable writes an “open” value in a memory position that already contains that value, which constitutes a silent store and unnecessarily forces remote processors to rollback. We propose the use of two mechanisms that allow KIMP to naturally speculate through critical sections.

Firstly, a *silent store detection and removal* mechanism, which detects that the value does not globally modify memory, and removes that update from the update packet. When applied to a lock variable, the lock is removed from the update packet and this would allow a checkpoint configuration such as Figure 6 b) to be executed in parallel, avoiding processor stalls.

Additionally, to make no checkpoint taken within a critical section, we make use of a *lock and barriers detection mechanism*, which

dynamically detects typical Test&Test&Set lock constructs and lock release. This hardware forces a new checkpoint to be taken just before the lock acquire and after the release, so that the lock remains open for the rest of the processors at the beginning of the new transaction. This makes the transaction length equal to the critical section, and avoids unnecessary rollbacks. This example is shown in Figure 6 c).

6.3. Silent stores and store merging

An ordinary *store merging* mechanism reduces the number of stores to the same address within a transaction to only one, and *silent store removal* avoids broadcasting the remaining stores. A silent store [12] is defined as a memory write that does not change the system state. Figure 7 shows an example of use of these mechanisms.

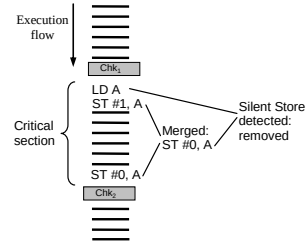


Figure 7. Silent store elimination.

Store merging consists of removing a store when a younger store is to the same address. After performing the *store merging*, we carry out the *silent store removal*. To detect silent stores our mechanism searches older loads before executing a store, and if an address and value match is found, the store is removed.

6.4. KIMP speculation past barriers

KIMP can be adapted to speculate after barriers, in a similar manner to [13]. As in the previous section, there is the need to detect the barrier code and take a new checkpoint just prior to it. Speculative execution starts after the barrier, as shown in Figure 8. All the transactions after this barrier remain fully speculative, meaning that none of them can be validated, as long as the barrier remains closed. This is ensured by setting a “pure speculative mode” in the processor.

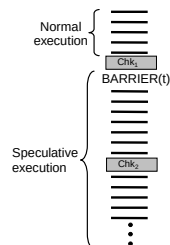


Figure 8. A speculated barrier.

To determine the moment in which the barrier opens, the processor tracks the cache line containing the barrier variable, waiting for a cache event (an invalidation or an update of the line) to check the value again. When the barrier opens, the “pure speculative mode” is disabled, and the processor can start committing all the transactions in the pipeline.

7. Conclusions

This paper introduces KIMP, a framework that makes Kilo-instruction Processors capable of executing parallel code in a transactional fashion, similar to the TCC model, but modifying neither the code nor the programming methodology. Our model maintains Sequential Consistency with a low hardware cost and a high performance potential. The hardware requirements are low, as most of the mechanisms are already proposed for KiP, and the processor model is simplified thanks to the transactional behavior.

Speculative execution in critical sections, together with the advantages of transactional behavior, will provide high performance with no required code modifications.

References

- [1] A. Cristal, M. Valero, J. Llosa, and A. González, “Large virtual ROB’s by processor checkpointing”, TR. UPC-DAC-2002-39, UPC, Spain, July 2002
- [2] A. Cristal, J. F. Martínez, J. Llosa, and M. Valero, “A Case for Resource-conscious Out-of-order Processors”, In IEEE TCCA Comp. Architecture Letters, 2, October 2003.
- [3] A. Cristal, D. Ortega, J. Llosa, and M. Valero, “Out-of-Order Commit Processors”, In Proc. of the 10th HPCA, February 2004.
- [4] A. Cristal et al., “Kilo-instruction Processors: Overcoming the Memory Wall”, To be published on IEEE Micro Magazine, Vol. 25, N. 3, May/June 05.
- [5] M. Galluzzi et al., “A First Glance at Kilo-Instruction based Multiprocessors”, In Proc. of the 1st Conf. on Computing Frontiers, pp. 212-221, Ischia, Italy, April 2004.
- [6] K. Gharachorloo et al., “Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors”, In Proc. of the 17th ISCA, 1990.
- [7] J. R. Goodman, “Cache Consistency and Sequential Consistency”, Tech.Rep. no.61, SCI Committee, March 1989
- [8] C. Gniady, B. Falsafi, and T. N. Vijaykumar, “Is SC + ILP = RC?”, In Proc. of the 26th ISCA, 1999
- [9] L. Hammond et al., “Transactional Memory Coherence and Consistency”, In Proc. of the 31st ISCA, Germany, June 2004.
- [10] T. Karkhanis and J.E. Smith, “A Day in the Life of a Data Cache Miss”, In Proc. of the 2nd WMPI, 2002.
- [11] L. Lamport, “How to make a Multiprocessor Computer that Correctly Executes Multiprocess Programs”, IEEE Transactions on Computers, C-28(9):690-691, 1979.
- [12] M. H. Lipasti, K. M. Lepak, “On the Value Locality of Store Instructions”, In Proc. of 27th ISCA, 2000
- [13] J. Martínez, J. Torrellas, “Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications”, In Proc. of the 10th ASPLOS, Oct. 02
- [14] K. E. Moore, M. D. Hill and D. A. Wood, “Thread-Level Transactional Memory”, TR1524, Comp. Science Dept. UW Madison, March 31, 2005
- [15] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton, “An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors”, In Proc. Of the 7th ASPLOS, Oct. 1996
- [16] R. Rajwar, and J. R. Goodman, “Speculative Lock Elision: Enabling Highly-Concurrent Multithreaded Execution”, In Proc. of 34th Intl. Symp. on Microarchitecture, pp.294-305, Dec. 2001.
- [17] R. Rajwar, and J. R. Goodman, “Transactional Lock-Free Execution of Lock-Based Programs”, In Proc. of the 10th ASPLOS, Oct. 02.
- [18] P. Ranganathan, V.S.Pai, and S. Adve, “Using Speculative Retirement and Larger Instruction Window to Narrow the Performance Gap Between Memory Consistency Models”, In Proc. of the 9th SPAA. June, 1997.
- [19] P. Rundberg, and P. Stenström, “Speculative Lock Reordering”, In Proc. of IPDPS, April 2003.