Implicit Transactional Memory in Chip Multiprocessors

Marco Galluzzi[1], Enrique Vallejo[2], Adrián Cristal[1], Fernando Vallejo[2], Ramón Beivide[2], Per Stenström[3], James E. Smith[4] and Mateo Valero[1]

[1]*Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya*
[2]*Grupo de Arquitectura de Computadores, Universidad de Cantabria*
[3]*Dept. of Computer Science and Engineering, Chalmers University of Technology*
[4]*Dept. of Electrical and Computer Engineering, University of Wisconsin-Madison*

Abstract

Chip Multiprocessors (CMPs) are an efficient way of designing and use the huge amount of transistors on a chip. Different cores on a chip can compose a shared memory system with a very low-latency interconnect at a very low cost. Unfortunately, consistency models and synchronization styles of popular programming models for multiprocessors impose severe performance losses. Known architectural approaches to combat these losses are too complex, too specialized, or not transparent to the software.

In this article, we introduce "implicit transactional memory" as a generalized architectural concept to remove such performance losses. We show how the concept of implicit transactions can be implemented at a low complexity by leveraging the multi-checkpoint mechanism of the Kilo-Instruction Processor. By relying on a general speculation substrate, it supports even the strictest consistency model – sequential consistency – potentially as effectively as weaker models and it allows multiple threads to speculatively execute critical sections, beyond barriers and event synchronizations.

1. Introduction

Chip Multiprocessors (CMPs) are becoming widespread for a variety of applications; they are widely used for both scientific and commercial computing, and, they are making their way into desktop systems. However, CMPs present some design problems, inherent to shared memory systems, that do not occur in the uniprocessor systems they are replacing. Of particular importance is efficient and correct data sharing among different threads executing in different cores. This means maintaining a coherent view of memory among concurrently executing cores, even though they may have copies of memory data in their private caches. It also means that accesses to shared data should occur in an order that is consistent with architected rules that the programmer relies on. These concepts are represented by the coherence protocol

and the consistency model used in the design.

In some cases critical sections are used to ensure mutual exclusive access to shared data structures. In a conservative implementation a closed lock forces threads to stall waiting for the lock owner to finish its work. This leads to a significant performance loss, especially when different data is accessed by different threads so that they could have executed the critical section in parallel.

A proposed method for efficient and accurate data sharing is the use of *transactional memory*. A transactional memory system allows the programmer to use *transactions*, which are blocks of code, to be executed as if they were atomic. If the system can not ensure this atomicity, for example due to a data race with another transaction in a remote core, the transaction is typically aborted and restarted. Critical sections substituted by transactions are naturally parallel executed in case of no real data races. In recent proposals [1][11][13][24], transactions are explicitly conveyed to the hardware through instructions with special semantics, that is, they are *explicit transactions*. Transactional memory provides a simple way to obtain a good performance.

This article introduces the concept of *implicit transactions*, i.e. transactions composed and orchestrated solely in hardware, without the need for explicit instructions and programmer support. Each core automatically divides execution into implicit transactions, executing several ones concurrently, and validating them atomically and in thread-program order. This architectural proposal simplifies the processor design by leveraging the scalable properties of the multi-checkpoint mechanism in the Kilo-Instruction Processor [2][4]. Further, it is expected to provide a high performance, while at the same time maintaining the most restrictive consistency model, Sequential Consistency. Finally, the proposal can host a number of optimizations, such as concurrent speculative execution of parallel sections and silent store detection and deletion, all of them with a low design cost.

The remainder of the paper proceeds as follows. In Section 2 we give an overview of the main proposals and the background related to our work. Section 3 introduces our implicit transaction proposal. Sections 4 and 5 show the implications of the selected design for cache coherence and memory consistency respectively. Section 6 introduces some optional improvement to the base system presented in Section 3: traffic reduction via silent stores elimination, speculation on critical sections and barriers and explicit transaction support. Lastly, Section 7 gives some conclusions and future directions of our work.

2. Related work

Transactional memory systems execute groups of memory operations atomically as single transactions. The use of such transactional behavior is intended to provide: (1) an intuitive model to the programmer and (2) scope for system optimizations. Recently there have been many proposals for transactional memory systems. All the proposals assume some degree of explicit software interaction when forming transactions. Depending on the degree of software interaction, existing methods are considered to be *hardware* or *software* solutions.

Hardware transactional memory systems are largely hardware implementations which provide a software interface. Some examples are:

- *Transactional Memory* [13] provides to the programmer primitive instructions, similar to database primitives, that enable lock-free accesses to shared data structures. The programmer then explicitly identifies such accesses that should be grouped into a transaction.
- *Transactional Lock Removal* (TLR) [21] provides lock-free access to shared data structures in a way that is transparent to the programmer. It is a method that detects critical sections, and dynamically substitutes them with transactions, eliding the lock acquisition.
- *Transactional Coherence and Consistency* (TCC) [11] proposes a new shared memory model where atomic transactions are the basic unit of operation, simplifying both parallel applications and coherence and consistency hardware. In this case, the programmer is given new high level programming constructs and has to conform to a corresponding parallel programming model.

Software transactional memory systems are software implementations with little or no hardware support. Some examples are *Software Transactional Memory* [24] or *Dynamic Software Transactional Memory* [14].

Our work differs from the previous approaches because it defines transactions as the basic operation unit for memory accesses, like TCC, but it is transparent to the programmer, like the TLR proposal, which only applies transactions to critical sections.

The memory consistency model establishes the rules that must be followed when the system overlaps or reorders memory operations performed by the cores. The different consistency models offer a trade-off between programming simplicity and performance. *Sequential Consistency* (SC) [16], which is generally considered to be the most natural programming model, is also the most restrictive. It guarantees that interleaved memory operations from different

processors appear to execute in program order with respect to each thread. A basic implementation of SC normally requires a processor to delay each memory access until the previous one is completed – while simple, this approach clearly leads to a low performance. More relaxed models, such as *Release Consistency* (RC) [9] provide higher performance, at the cost of not ensuring strict ordering of memory operations in hardware; the programmer has to assure correct shared memory accesses.

Recent proposals preserve the SC model without compromising performance, by relying on hardware speculation. In these schemes, SC is preserved by storing a valid system state (with a checkpoint or some sort of buffering) and rolling back when a consistency violation is found. Specific proposals are *Speculative Retirement* [22], which allows hardware speculation of loads, alleviating the impact of the store-to-load constraint imposed by the SC model, or *SC++* [10], which makes use of hardware speculation for both load and store operations, enabling the reordering and overlapping of memory operations for performance similar to that achieved with the RC model.

Our proposal, as we will see in section 5, allows reordering and overlapping of loads and stores similarly to SC++, thus intuitively performing better than a direct SC implementation.

Speculating on critical sections. Lock structures control the access to critical sections by allowing only one process, the lock owner, to enter and execute instructions that read and modify shared variables. 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 actually not modify any data, or modify different fields of a shared structure. In these cases, parallel execution could be allowed without compromising correctness, avoiding the stall of threads waiting for the lock. Some examples from the SPLASH-2 benchmark suite and cited in [20] are shown in Figure 1.

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. There are a number of approaches for reducing the performance loss due to these unnecessary stalls.

In [20] it is showed that some locks are too conservative and true contention for shared data occurs only under certain conditions. Thus, Speculative Lock Elision (SLE) is proposed, a hardware approach that detects a typical Test&Test&Set lock idiom and avoids actually acquiring it, leaving the critical section open and thus allowing several processes to execute the same critical section concurrently. Correctness is preserved by detecting data races for shared data. All the data used during a speculative critical section are kept in the local cache, and a remote update forces the execution to roll

```
a)      LOCK(locks->error_lock)
        if (local_error > multi->err_multi)
                multi->err_multi = local_err;
        UNLOCK(locks->error_lock)

b)      Thread 1
                LOCK(hash_tbl.lock)
                var = hash_tbl.lookup(X)
                if (!var)
                        hash_tbl.add(X);
                UNLOCK(hash_tbl.lock)

        Thread 2
                LOCK(hash_tbl.lock)
                var = hash_tbl.lookup(Y)
                if (!var)
                        hash_tbl.add(Y);
                UNLOCK(hash_tbl.lock)
```

Figure 1: Critical sections that, in most cases, admit parallel execution.

back to a checkpoint. Thus, if no data race is detected, several instances of the same critical section can be executed in parallel, and if a data race happens, only one of them advances, forcing the other ones to restart with new data. Transactional lock removal (TLR) [21] improves this mechanism by appending a timestamp to write requests, which avoids process starvation, ensuring forward progress of the oldest thread.

In [19] Speculative Synchronization is presented. With architected lock instructions that denote the critical section entrance, this work proposes the execution of one safe thread, which actually acquires the lock, and multiple speculative threads, which detect the busy lock and execute speculatively. When the lock owner releases it, the rest of the threads commit their state if they have successfully finished their critical section, otherwise they compete for ownership of the lock.

Speculative lock reordering (SLR) [23] improves the previous ideas. All the speculative threads execute their critical section, even though they may have a conflict. When all of the speculative threads are ready to commit, an arbiter dictates the order in which they do so, minimizing data races.

Transactional memory coherency and consistency (TCC) [11] explicitly partitions code into transactions. Every access to shared data must belong to the same transaction as computation and update of the data. Software locks are replaced with special instructions that mark a transaction change. This method naturally allows the parallel execution of critical sections, and detects conflicts in case of data races, without the need for control variables.

Speculating on barriers and flags. Flags and barriers are mechanisms that synchronize multiple threads. A correct execution forces all threads to wait for the barrier to open or the flag to clear, before continuing execution. Similar to the case with locks, this stall can be avoided in some cases, e.g., when there is no real data interaction between synchronizing threads.

Speculative Synchronization [19] deals with flags and barriers. When such a construct is detected, execution is continued speculatively. The instructions after a barrier remain speculative, waiting for the barrier to open before validating all the work. Barrier detection is carried out with the aid of some software libraries. Hence, this method is not transparent to the programmer.

TCC [11] replaces flags and barriers by assigning a phase number to each transaction. A processor can not commit a transaction if there is a remote pending transaction with a lower phase number. Transactions with the same phase numbers can commit in any order, whereas transactions with different numbers commit sequentially. Speculation is naturally implemented, as transactions after a phase change can be executed, but not committed. Again, this method is not transparent to the programmer.

Kilo-Instruction Processors [2][4] provide the basic mechanisms that are leveraged in this work. Designed to allow thousands of instructions in-flight at the same time, Kilo-Instruction Processors can hide large latencies. They also have been showed to efficiently hide the larger memory latencies that are present in multiprocessors [8]. However, in order to increase the number of in-flight instructions the designer must increase the capacities of several resources, the most important ones being the re-order buffer or ROB, the instruction queues, the load/store queues and the physical registers. Unfortunately, simply up-sizing conventional designs for these structures is not feasible with current and near-term technology.

In order to overcome the difficulties of up-sizing critical structures, Kilo-instruction processors employ a number of different techniques, based on the fact that critical resources are underutilized in present out-of-order processors [5][15]. The main technique consists of maintaining multiple checkpoints instead of having a large ROB [3]. It allows instructions to be committed in an out-of-order fashion, freeing more resources than in normal processors. Other techniques are the Slow Line Instruction Queue [6] and Ephemeral Registers [18].

During program execution, checkpoints are taken at certain critical points. If an exception or branch misprediction occurs, instead of using a ROB to restore the state by flushing all instructions following the excepting one, the state is rolled back to the closest checkpoint prior to the excepting instruction and execution can resume at that point. This fact together with a pseudo-ROB [6] that maintains the youngest in-flight instructions and allows precise recovery of these instructions makes possible to use a relatively small set of checkpoints.

As the instructions in the pipeline advance, in-flight instructions corresponding to the different checkpoints are speculatively executed. Memory operations remain in the processor's queues, and any other instructions are removed

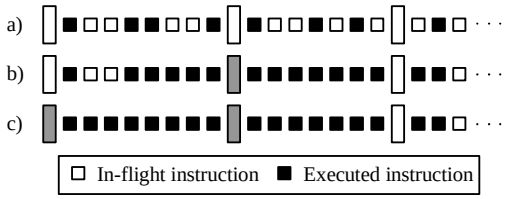

Figure 2: Multiple checkpoints.

from the processor when they commit, placing results in their destination registers. Only when all the instructions corresponding to the oldest checkpoint are finished, the processor commits the checkpoint, consequently removing memory operations from the load and store queues and committing all the results speculatively calculated. With this action, the speculative instructions become globally performed. Figure 2 shows an example of this mechanism, with the oldest instructions on the left. Black instructions are finished, but not committed. In case a), instructions from three consecutive checkpoints are in flight. In b), the second checkpoint is finished, because all the instructions within it are finished, but it can not commit because it is not the oldest one in the pipeline. In c), the first checkpoint (the oldest one) finishes, so it and the second checkpoint can commit, and the third checkpoint then becomes the oldest checkpoint.

3. Checkpointing and implicit transactions

The concept of *implicit transactions* can be applied to different multiprocessor configurations, including multiple chips systems. We consider here a shared memory CMP where the cores implement the previous multiple checkpointing mechanism interconnected via a snoopy bus, with private L1 caches. With this implementation all the memory access instructions between two checkpoints behave as a single memory transaction, which means that the associated memory updates (the **store** instructions) are kept as a group and are globally and atomically performed when the corresponding checkpoint commits.

We call this kind of transactions **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, both the programming model and the ISA are unaffected, and current binaries can be directly executed. Note that this idea differs from the concept of transaction that normal transactional memory systems rely on, where transactions are considered as programming constructs with hardware support. In our model every memory access instruction is, by default, part of some transaction.

In the following subsections we explain the basic system operation and describe a possible heuristic for taking checkpoints.

3.1. Basic operation. Correctness substrate

All the in-flight instructions remain speculative until their corresponding checkpoint commits. Memory instructions remain in the processor queues and do not modify the local cache or the global memory, as they are subject to a rollback. After a checkpoint commits, all stores belonging to the checkpoint are broadcast over the bus, validating all the speculative memory updates in the checkpoint. The broadcast packet is snooped by remote processors, and addresses are compared with their speculative memory accesses. If an address-match is found, the remote processor rolls back to a checkpoint prior to use of the address matching data because the data cannot be considered up to date.

Figure 3 shows an example of the execution flow among four processors, P_1 to P_4, and their respective checkpoints. The processors execute different portions of code, taking different checkpoints as the execution advances, and are able to roll back execution to a certain checkpoint in case of an exception, branch misprediction or memory consistency violation as we will see below.

The oldest checkpoint in the processor can commit when all of its corresponding instructions, i.e. those that come after that checkpoint and before the next one, are finished. In Figure 3, for example, P_3 can commit checkpoint Chk_{31}, when all the instructions up to Chk_{32} have been completed. This commit is followed by acquiring the write grant and a logically

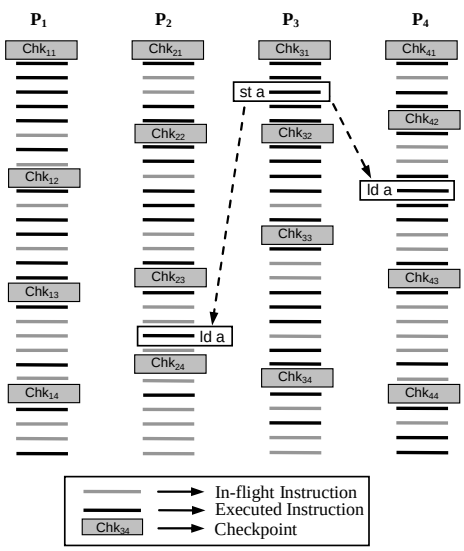

Figure 3: Execution flow for 4 processors.

atomic broadcast of all the cache tags that the processor has modified during the checkpoint execution; these are the pending memory updates. By “logically atomic” we mean that once the processor gets the grant to write on the bus it does not release it until all the tags have been broadcast.

Remote processors snoop the memory update addresses, searching for a conflict with any loads they have in their load queues, which have been speculatively executed but have not been committed. In case of a conflict, the remote processor in question is forced to roll back, because it has speculatively used data that, at this point, is discovered to be “previously” modified from a logical perspective. Conversely, if no conflicts are found, speculatively executed instructions are not discarded. Finally, when the broadcast finishes, the checkpoint commits and the speculatively executed instructions are considered to be globally performed. In the example of Figure 3, the broadcast of a store to memory location “a” conflicts with two other processors that have already speculatively loaded from location “a”. In this example, P_2 is rolled back to Chk_{23}, causing instructions from Chk_{24} to Chk_{23} to be discarded. Also P_4 rolls back to Chk_{42}.

The behavior proposed, consisting of speculative execution, atomic validation, and the rollback mechanism, constitutes a correctness substrate that ensures correct execution of any code. However, some optimizations are proposed in sections 6 to improve the performance of the system.

3.2 Adaptive transaction length

We propose that the transaction length should be adaptive in order to deal with frequent rollback scenarios. As an example, a heuristic would dictate starting with a fixed checkpoint length, decreasing in case of frequent rollbacks and increasing as long as no rollbacks occur up to a maximum length. This way, in case of frequent consistency violations, the length of the transactions decreases, also decreasing the probability of a violation to occur again and making the process to advance.

Even though this example will work, any heuristic should take into account a number of issues:

- The total number of instructions.
- The memory operations, because they use specific processor resources and they can make the processor rollback due to a consistency violation.
- The branch instructions, since they can make the processor rollback due to a misprediction.
- Synchronization constructs such as locks and barriers, as we will see in Section 6.

To conclude, the system works correctly independently of the instructions where checkpoints are taken, because of the correctness substrate indicated previously.

4. Cache coherence protocol

Our initial approach to the coherence protocol is similar to the one in TCC [11]. This approach does not consider different states for a cache line. However, any conventional MESI-like protocol, even a directory, might be used as long as there is some mechanism to ensure the atomicity of validations.

The broadcast-based approach works as part of either a snoopy write-invalidate or write-update protocol. Once a transaction commits, the broadcast mechanism transmits the changes to the rest of processors, updating or invalidating the remote cache lines. When the stores from a transaction need to be broadcast, the core will get the bus write permission and release it only when all its memory updates are globally performed. This operation prevents other cores from broadcasting memory updates simultaneously and protects the memory system from coherence and consistency problems.

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. In case it contains the updated addresses, remote processors will invalidate the lines, and it behaves as write invalidate.

5. Memory consistency

Our implicit transaction design naturally maintains the sequential consistency model, which is the most desirable model for programmers since it is closer to the way of thinking when programming. In the following subsections, we first explain how the SC model is preserved and then we discuss some performance issues.

5.1 Implicitly preserving SC

Sequential consistency requires the result of any execution to be the same as if the memory accesses executed by each processor were kept in order and with the accesses among different processors (arbitrarily) interleaved. Figure 4 shows an example of a sequentially consistent global ordering of memory operations from two processors. The third column shows a global order that respects the program orders from processors A and B.

In our system we extend the definition of sequential consistency to implicit transactions, and require only transactions

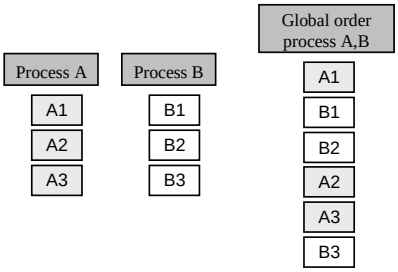

Figure 4: Sequentially consistent reordering of memory operations from 2 processors.

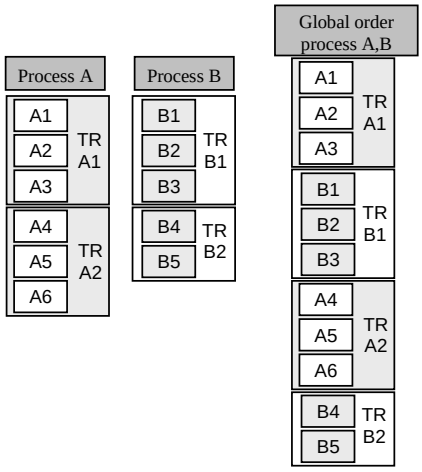

Figure 5: Sequentially consistent reordering of checkpoints from 2 processors.

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 to one of the possible sequentially consistent global orderings. Figure 5 shows an example where we group instructions into transactions, labeled [TR_A1, TR_A2] for processor A, and [TR_B1, TR_B2] for processor B. The third column shows that respecting the program order for these transactions also respects it for memory operations.

However, in our design, we allow the execution of memory operations out-of-order within a processor, and therefore in Figure 5 instructions in transaction “TR_A1” could be reordered, for example, as “A2, A3 and A1”. This does not affect the consistency model because:

- All the memory operations are speculatively executed.
- Speculative loads that match the address of a previous store receive the correct value thanks to the usual store-forwarding mechanism, that is, they are correct.
- The snooping of updates from the bus ensures that the values speculatively loaded remain valid.
- Finally, the memory updates from the transaction, issued out-of-order, are atomically and in-order broadcast only when the transaction commits, making the pending stores globally performed.

5.2 Performance issues related to consistency

Different works try to meet better performance levels in SC by speculation. Speculating on memory operations implies having the capability to reorder and overlap, what is precisely what makes the system hide the latency of cache misses.

One of the latest contributions is SC++ [10] that achieves speculative reordering of both load and store instructions with very good performance results.

The implicit transactions proposal also allows reordering and overlapping of memory operations between loads and stores, from the same or different transactions of the same processor. Therefore, based on these previous papers, we can state that our proposed design most likely performs better than a direct implementation of SC and probably as good as SC++.

6. System improvements

Previous sections have shown the correctness of a system based on implicit transactions. This section shows some techniques aimed at boosting performance, while preserving the previous correctness.

6.1. Reducing bus traffic with silent stores removal

A silent store [17] is defined as a memory write that does not modify the system state. In [17] it is shown that a non-trivial percentage of the stores are silent. Temporally silent stores [12] are sets of stores that a processor performs to the same address which, globally, leave the same initial value. One example is the pair composed of the acquire and release of a critical section lock variable. This lock silent property is used in other works, such as SLE [20], which implements silent store detection specifically designed for those silent stores of a critical section lock.

In our design, we propose the use of two mechanisms to remove silent stores within a single transaction:

- An ordinary *store merging* mechanism reduces the number of stores to the same address within a transaction to

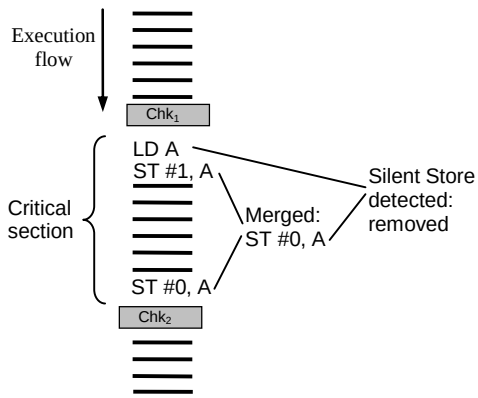

Figure 6: Silent store elimination.

only one, similar to that implemented in the Alpha 21164 [7].

- *Silent store removal* removes a store if the position had been previously read containing the same value.

Figure 6 shows an example of these mechanisms applied to the case of a critical section, similarly to SLE.

6.2. Speculating through critical sections

The correctness substrate ensures valid operation, including the execution of critical sections and barriers. However, as studied in previous works, if no real dependence exists between different threads trying to execute the same critical section, the threads can progress in parallel. The silent removal feature proposed in the previous point naturally allows our system to speculate on critical sections, and data races will be the ones that control rollbacks, in case of speculation being incorrect.

As critical sections are short in nature, we will initially suppose that a critical section completely falls within a single transaction as in figure 7B. The first processor to execute it detects the lock as open, speculatively acquires it, and proceeds to the critical section. If another thread reached the same critical section, it would find the lock open, as the acquire operation remains speculative in the first processor's queues, and would proceed inside.

When the first processor leaves the critical section, the lock is speculatively released, which returns to the initial open state. When this transaction commits, the silent store removal mechanism will remove the stores associated with the lock variable, as they do not modify its value globally. The broadcast will only consist of the values modified inside the critical section. Thus, the second processor will not be forced to rollback unless a real data race exists, detecting that the first

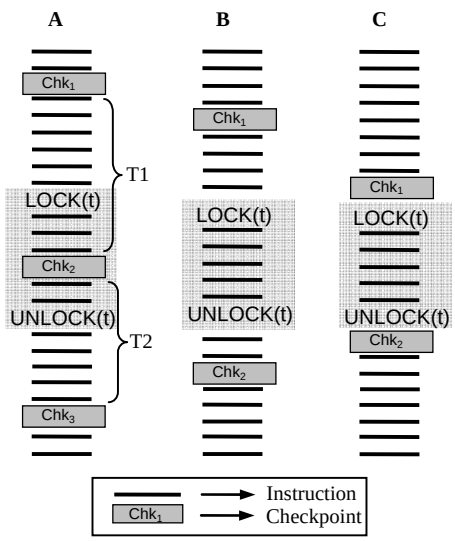

Figure 7: Different checkpointing schemes.

processor has modified a variable that the second has speculatively read.

This will work as long as critical sections fit into a single transaction. Figure 7A and 7B show two cases where this speculation would not work and work, respectively. The final proposal is a lock detection mechanism (similar to the one proposed in SLE), that forces a new checkpoint to be taken just before entering a critical section and just after leaving it. This leads to the case shown in fig. 7C, which naturally allows the parallel execution of critical sections with no real data races.

6.3. Speculating past flags and barriers

The proposed design can be adapted to speculate after barriers, similarly to Speculative Synchronization [19]. 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. However, in this case the transactions after this barrier can not be committed as long as the barrier remains closed. 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 to determine when the barrier opens. A remote invalidation of the tracked line does not force a rollback, but makes the processor check the variable again. This case is more complicated as there are several barrier implementations.

7. Conclusion and future work

This paper introduces a framework that makes multiprocessors capable of executing parallel code in a transactional fashion, similar to the TCC model, but modifying neither the code nor the programming model. This approach is oriented specifically to chip-multiprocessors, as it takes advantage of the fast interconnect between different cores. Our model maintains sequential consistency with a low hardware cost, a high performance potential and a reduced bus overhead. The hardware requirements are low, as most of the mechanisms leverage the mechanisms in kilo-instruction processors, and the processor model is simplified thanks to the transactional behavior.

Our model considers silent store elimination, speculative execution in critical sections and barriers, reducing as much as possible the performance loss that these elements cause in parallel programs. This, together with the advantages of being able to handle thousands of in-flight instructions, and the transactional behavior, will provide high performance with no required code modifications.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, “Unbounded Transactional Memory”, In Proc. of the 11th HPCA, California, pp 316-327, Feb. 2005.

[2] A. Cristal, O. Santana, Francisco 1, M. Galluzzi, T. Ramirez, M. Pericas, M. Valero. "Kilo-Instruction Processors: Overcoming the Memory Wall," *IEEE Micro*, vol. 25, no. 3, pp. 48-57, May/June 2005.

[3] A. Cristal, M. Valero, J. Llosa, and A. González, “Large virtual ROBs by processor checkpointing”, Tech. Rep. UPC-DAC-2002-39, UPC, Spain, July 2002.

[4] A. Cristal, O. Santana, M. Valero, J. F. Martínez, “Toward Kilo-instruction Processors”, In ACM Transactions on Architecture and Code Optimization, V. 1, No. 4, Dec. 04.

[5] 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.

[6] A. Cristal, D. Ortega, J. Llosa, and M. Valero, “Out-of-Order Commit Processors”, Proc. of the 10th HPCA, Feb. 2004.

[7] J. H. Edmondson, et al., “Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor”, Digital Technical Journal, Vol. 7, No. 1, 1995, pp. 119--135.

[8] 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.

[9] K. Gharachorloo et al., “Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors”, In Proc. of the 17th ISCA, 1990.

[10] C. Gniady, B. Falsafi, and T. N. Vijaykumar, “Is SC + ILP = RC?”, In Proc. of the 26th ISCA, 1999.

[11] L. Hammond et al., “Transactional Memory Coherence and Consistency”, In Proc. of the 31st ISCA, Germany, June 2004.

[12] K. M. Lepak and M. H. Lipasti. “Temporally silent stores”, In Proc. of the 10th Symposium on Architectural Support for Programming Languages and Operating Systems, Oct. 2002.

[13] Maurice Herlihy and J. Eliot B. Moss. “Transactional Memory: Architectural Support for Lock-Free Data Structures”. In Proceedings of the 20th ISCA, pages 289-300, May 1993.

[14] Maurice Herlihy, Victor Luchangco, Mark Moir, and William Scherer III. “Software Transactional Memory for Dynamic-Sized Data Structures”. In 22nd ACM Symp. on Principles of Distributed Computing, Massachusetts, 2003.

[15] T. Karkhanis and J.E. Smith, “A Day in the Life of a Data Cache Miss”, In Proc. of the 2nd WMPI, 2002.

[16] L. Lamport, “How to make a Multiprocessor Computer that Correctly Executes Multiprocess Programs”, IEEE Transactions on Computers, C-28(9):690-691, 1979.

[17] M. H. Lipasti, K. M. Lepak, “On the Value Locality of Store Instructions”, In Proc. of the 27th ISCA, 2000

[18] J. F. Martínez, A. Cristal, M. Valero, and J. Llosa, “Ephemeral Registers”, Technical Report CSL-TR-2003-1035, Cornell Computer Systems Lab, 2003.

[19] J. Martínez, J. Torrellas, “Speculative Synchronization: Applying Thread-Level Speculation to Explicitily Parallel Applications”, In Proc. of the 10th ASPLOS, Oct. 02

[20] R. Rajwar, and J. R. Goodman, “Speculative Lock Elision: Enabling Highly-Concurrent Multithreaded Execution”, In Proc. of 34th Int. Symp. on Microarchitecture, pp.294-305, Dec. 2001.

[21] R. Rajwar, and J. R. Goodman, “Transactional Lock-Free Execution of Lock-Based Programs”, In Proc. of the 10th ASPLOS, Oct. 02.

[22] 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 Symposium on Parallelism in Algorithms and Architectures. June, 1997.

[23] P. Rundberg, and P. Stenström, “Speculative Lock Reordering”, In Proc. of IPDPS, April 2003.

[24] N. Shavit and D. Touitou. “Software Transactional Memory”. In PODC ’95, pp. 204-213, August 1995.