Implicit Transactional Memory in Kilo-Instruction 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

Although they have been the main server technology for many years, multiprocessors are undergoing a renaissance due to multi-core chips and the attractive scalability properties of combining a number of such multi-core chips into a system. The widespread use of multiprocessor systems will make performance losses due to consistency models and synchronization styles of popular programming models even more evident than they already are. Known architectural approaches to combat these losses are generally too complex, too specialized, or not transparent to software.

In this article, we introduce *implicit transactional memory* as a generalized architectural concept to remove unnecessary performance losses caused by consistency models and synchronization styles. We show how the concept of *implicit transactions* can be implemented with low complexity by leveraging the multi-checkpoint mechanism of the Kilo-Instruction Processor. By relying on a general speculation substrate, this method 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

Over the last decades, multiprocessors have played a key role in servers for a wide range of computational problems ranging from scientific/engineering tasks to information processing tasks, for example in database and web servers. Their importance will likely increase further as the available transistors on a chip will soon allow a midrange server of the 90s to fit on a single chip. Building large-scale multiprocessors will be a matter of connecting a limited number of such chips, for example in a non-uniform-memory-access (NUMA) configuration.

One reason for the success of shared-memory multiprocessors, as opposed to other parallel architecture styles, is their intuitive programming style (and relatively low parallelization effort). For example, parallel threads can be managed via a work queue, with inter-thread communication taking place through shared data structures. While this approach greatly reduces the effort required to achieve good load balance, the challenge for the programmer is to assure that data dependences among threads are respected. This is achieved by using *synchronization primitives*, e.g., locks and barriers. Regardless of the primitives chosen, synchronization typically leads to two orthogonal sources of performance loss: *memory access ordering* and *serialization* losses.

Regarding *memory access ordering*, event synchronization in many legacy codes may use regular loads and stores that cannot be distinguished from other non-synchronizing loads and stores. This approach will work correctly as long as the underlying machine supports a strict consistency model such as sequential consistency. Unfortunately, supporting a strict consistency model has performance and/or complexity implications because the architecture has to guarantee that individual loads and stores are *performed* in the order specified by the program of each thread to establish a consistent global execution order. While naïve implementations may stall the processor until each load or store completes, research

[5][12] has shown that it is possible to allow a thread to have multiple outstanding memory accesses as long as reordering violations can be detected and their effects can be "undone". Conventional speculation support in wide-issue processors can be extended to accomplish this, but this approach will suffer from scalability problems. Specifically, the length of speculation must be extended to cover the entire memory access latency, approaching hundreds of processor cycles, so it would be necessary to support a thousand or more instructions simultaneously in flight.

It is well-known that memory access ordering losses can be reduced or eliminated by relaxed memory consistency models. Indeed, if synchronization primitives can be identified by the programmer or compiler and if this information can be passed to the hardware via special synchronization operations, one can allow reordering of non-synchronizing memory accesses without any speculation support. Apart from not being compatible with some legacy codes, such relaxed consistency models cannot eliminate the second source of performance loss: *serialization loss*. Serialization loss occurs when threads stall at a synchronization point, waiting for other threads to arrive. For example, a thread cannot enter a critical section until another thread releases the lock, or, a thread cannot execute beyond a barrier until all other threads have reached the barrier. This serialization loss can sometimes be avoided if the programmer spends more effort exploiting the inherent concurrency in the program. An alternative approach that removes the burden from the programmer is to let threads speculatively execute past synchronization points [7][10][11][13]. For example, it is semantically correct to allow multiple threads to execute a critical section as long as there are no data races, i.e., accesses from multiple threads to the same variable with at least one access being a store. Apart from relying on special-purpose speculation mechanisms, these proposals [7][10][11][13] assume that critical section boundaries can be identified at the hardware level, giving the approach limited appeal.

In order to avoid memory access ordering and serialization losses, there is a need for a general speculation mechanism that has attractive scalability properties. Further, in order to be applicable to a broad software base, it should not require that synchronization primitives be explicitly identified by the programmer or compiler. In this article, we propose *implicit transactions*, i.e. sequences of memory operations executed in hardware as atomic units, without any software support and transparent to the programmer. Within an implicit transaction, a processor's memory operations can be performed in any order (subject to sequential program semantics). Each transaction begins with a checkpoint, i.e. a point to which architected state can be rolled back, if necessary. Then a thread executes speculatively beyond a checkpoint until it successfully arrives at the next checkpoint, which ends the implicit transaction. Then, the thread commits all its changes to the memory state by making its stores visible to the other threads (e.g. via a snoopy bus). At that time, if another thread detects a data race with the committing thread, then the current transaction of the conflicting (non-committing) thread will fail. At that point, the conflicting thread must roll back to its previous checkpoint and begin re-execution.

While other recent transactional memory proposals also rely on a speculation mechanism, they are either specialized for certain software idioms or otherwise rely on software to mark the start and end of a transaction explicitly [6][10][13], or they are restricted to a specific programming task, e.g. critical sections [11]. In contrast, we show that implicit transactions can be naturally integrated into superscalar processors by leveraging the multi-checkpoint mechanism of the *Kilo-Instruction Processor* [1]. As we have shown in previous work, the complexity with which Kilo-Instruction Processors can manage thousands of in-flight instructions is remarkably low and is sufficient to hide long memory access latencies in multiprocessors [3].

The remainder of the article proceeds as follows. In Section 2, we present the basic mechanisms in Kilo-Instruction Processors that support implicit transactions. Then, Section 3 explains how checkpointing is used in our design to create implicit transactions. Section 4 shows the implications of the selected design for cache coherence and memory consistency. Finally, in Section 5, we describe the application of speculation mechanisms for executing beyond a locked critical section or a closed barrier.

2. Kilo-Instruction Processors

The implicit transactional memory concept leverages some of the key mechanisms in the Kilo-Instruction Processor, especially its multi-checkpoint mechanism. Therefore, we first provide a summary of the Kilo-Instruction Processor.

Kilo-Instruction Processors [1] have a demonstrated ability to hide large latencies, specifically due to memory accesses, because the processor allows thousands of instructions to be in-flight simultaneously. In order to increase the number of in-flight instructions, however, the designer must increase the capacities of several resources, the most important ones being the re-order buffer (ROB), the instruction issue queues, the load/store queues and the physical registers. Unfortunately, simply up-sizing these structures is not feasible with current or near-term technology.

In order to overcome the difficulties of up-sizing critical structures, Kilo-instruction processors employ a number of techniques to arrive at a complexity-effective implementation. Such an approach is possible because critical resources are underutilized in present out-of-order processors [9]. The main technique consists of multi-checkpointing long latency instructions instead of using a very large ROB. This way, instructions can release resources in an out-of-order fashion, thereby requiring fewer resources than in a ROB-based implementation. The second technique is the Slow Line Instruction Queue that employs a secondary instruction queue to which long-latency instructions can be off-loaded. This mechanism allows the regular instruction queue to remain small and fast. The third technique is called Ephemeral Registers which is an aggressive register recycling mechanism that combines delayed register allocation and early register recycling and, in conjunction with multi-checkpointing and Virtual Tags, allows the processor to non-conservatively de-allocate resources.

During program execution, a checkpoint is taken at certain instructions, generally at branches that depend on a load miss or load instructions that are likely to miss in the L2 cache. 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. Using a relatively small set of checkpoints for long flight-time instructions assures safe points of return and reduces ROB requirements considerably. Although effective, this may lead to a longer recovery time than with a conventional ROB. Therefore, to minimize the exception and misprediction penalty a reduced structure called a pseudo-ROB can be used. The pseudo-ROB only maintains the youngest in-flight instructions and allows quick misprediction recovery for these instructions in a manner similar to a conventional ROB. Because exceptions and branch mispredictions occur most frequently within these youngest instructions, the average recovery time is effectively reduced.

Given that multiple checkpoints are taken during program execution, the mechanism works as follows. As the instructions in the pipeline advance, in-flight instructions corresponding to the different checkpoints are executed. When these speculatively executed instructions finish, memory operations remain in the processor queues; otherwise they are completed by the processor, leaving results in their destination registers. When all the instructions corresponding to the oldest checkpoint are finished, the processor commits the checkpoint atomically, by removing memory operations from the load and store queues and committing all the results speculatively calculated. With this action, the results of speculative instructions may be observed by other processors and become globally performed.

3. Implicit Transactions

Given the multi-checkpoint mechanisms in the Kilo-Instruction Processors, we will now see how they naturally support implicit memory transactions. To simplify the discussion, we assume a multiprocessor system with Kilo-Instruction Processor nodes interconnected by a bus using a snoopy cache protocol. Other interconnection architectures will be briefly discussed later. Specifically, implicit transactions cause the instructions between two checkpoints to appear to the rest of the system as a single memory transaction. In particular, memory updates (the *store* instructions) associated with a transaction are managed as a group and are globally and atomically performed when the corresponding checkpoint commits.

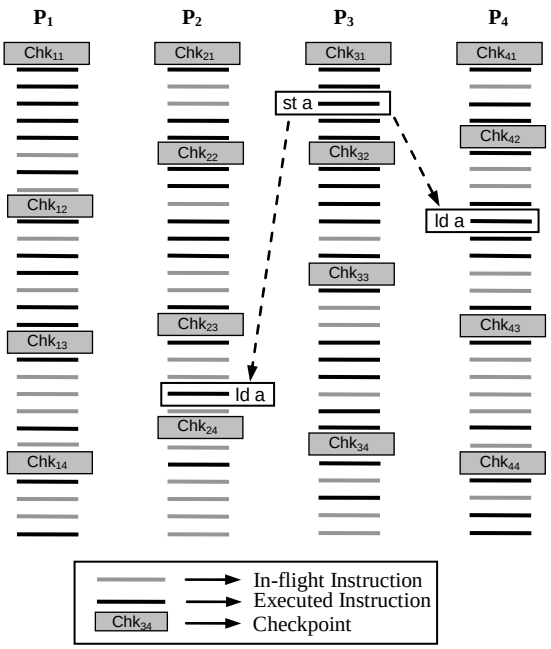

Figure 1: Execution flow for 4 processors.

We call these *implicit transactions* because they are automatically and transparently hardware delimited, by means of the checkpoint mechanism; a key point is that *every* instruction belongs to some implicit transaction. The programmer does not need to know that the system provides transactional behavior. Therefore, both the programming model and the instruction set are unaffected, and legacy binaries can be directly executed. Note that this idea differs from the concept of transaction that known transactional memory systems rely on [6][7][10][13], where transactions are considered as programming constructs that normally depend on instruction set support. At the same time, implicit transactions can support TCC-based programming models [6] where all the code is executed using transactions, by detecting a few special transaction boundary-marking instructions. This property allows the execution of traditional, as well as TCC-based programming models, using the same underlying hardware.

Now that the close relationship between a checkpoint and an implicit transaction in our system has been established, we will henceforth use the term *checkpoint* and the term *implicit transaction* (or just *transaction*) interchangeably. In the following we first describe a straightforward embodiment that implements implicit transactions and then discuss possible performance/complexity optimizations.

3.1 Basic Scheme

As in the multiple checkpoint mechanism for a single Kilo-Instruction Processor, 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 may be correctly rolled back. In a snoopy bus based system, after a checkpoint commits, all the stores are broadcast over the bus as a packet, thereby validating all the speculative memory updates in the checkpoint. The broadcast packet is snooped by remote processors, and if an address-match is found the remote processor rolls back to the checkpoint previous to its data use because it may have used a stale value.

Figure 1 shows an example of the execution flow for 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. The oldest checkpoint for a given processor can commit when all of its corresponding instructions, i.e. the ones that come after the checkpoint and before the next checkpoint,

are finished. In Figure 1, for example, P_3 can commit checkpoint Chk_{31}, when all the instructions up to Chk_{32} have been completed. Then, part of the commit process is an atomic bus broadcast of all stores, and in particular the cache tags that the processor has modified during the checkpoint execution; these are, the pending memory updates. "Atomic" simply means that after the processor gets access to the bus it does not release it until all the tags have been broadcast.

Remote processors snoop the memory updates searching for a conflict with load instructions in their load queues; by definition these are speculatively executed because they are not yet committed. In case of an address match, the remote processor is forced to roll back to the appropriate checkpoint because it has used data from a shared location that, logically speaking, has been over-written by a "previous" store instruction. Conversely, if no such address matches are found, the remote processor continues execution, uninterrupted. In the example from Figure 1, the broadcast of a store to a given memory location "a" conflicts with two other processors that have already speculatively loaded from location "a", and the loads have not yet committed. 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}, forcing its newest instructions to be discarded.

With such a checkpoint-based system, forward progress of the parallel application is always guaranteed because roll-backs, other than normal exceptions or branch mispredictions, occur only when one processor is committing a checkpoint and another one has a conflict. Therefore, at least one processor, the one committing, must make forward progress. The system is not concerned with conflicts that can exist during the execution of a transaction because they are not globally visible until the transaction commits.

This completes the explanation of how our design provides a correctness substrate, although it may be non-optimal from a performance point of view. The design is comprised of speculative execution, atomic validation, and the roll-back mechanism, which collectively ensure that the code is executed correctly.

3.2 Performance/Complexity Optimizations

The length of a transaction has a significant effect on the performance/complexity tradeoff. On the one hand, longer transactions allow a higher degree of memory re-ordering, but on the other hand longer transactions may cause more roll-backs due to data races. Thus, we propose that transaction length should be adaptively adjusted in order to give good performance and avoid frequent roll-back scenarios. One could start from a fixed transaction length and adaptively shorten or lengthen it as more/fewer data races are observed. In case of frequent consistency violations, the length of the transactions decreases, also decreasing the probability of violations recurring. We are currently investigating heuristics to strike a good compromise between performance and resources needed. Note that having a mechanism to adaptively adjust the transaction length will also promote fairness among threads.

Our proposed baseline implementation of implicit transactions assumes that one can handle a very large number of in-flight memory operations. While the Kilo-Instruction Processor aims at providing this capability, more attention is needed with regard to the scalability of load/store queues. While encouraging solutions exist (see e.g., [4]), we are currently exploring solutions that better match our implicit transaction framework.

4. Implicit Transactional Cache Coherence and Memory Consistency

This section demonstrates how cache coherence is maintained and that strong memory consistency models can be supported at high performance under the implicit transactional model.

4.1 Cache Coherence

A baseline snoopy cache coherence protocol works as follows. During the execution of a transaction, the local cache is not modified by local stores, only speculative loads are performed including loads that miss in the caches. Once a transaction commits, the pending stores are atomically performed by broadcasting 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 corresponding processor will be granted access to write on 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 guarantees that a transaction is globally performed in an atomic fashion. Finally, if there is an update or invalidation of a remote cache line, all remote cache lines matching the snooped address will be updated or invalidated and the associated processors will roll back to the previous checkpoint. Note, however, that while transactions have to be committed one by one, individual nodes can still gain access to the bus to service cache misses. While a processor is validating the different stores of a transaction, other processors can interleave read requests without affecting correctness, as no update is performed on a read.

With implicit transactions, cache coherence protocols can be greatly simplified. For example, the basic protocol does not need to maintain cache lines in "shared" or "exclusive" state, as dictated by MESI-like cache coherence protocols. While this inevitably can result in higher traffic, simulation results in [6], assuming a similar transactional framework supported in software, show that the traffic is manageable. Further, there is room for improvement using a *write cache* [2] to coalesce multiple modifications of individual words in a cache line and using *silent store elimination* to be discussed later.

Since the basic protocol exploits the atomicity that a bus provides, generalizing it to a non-broadcast environment appears problematic. However, we are exploring extensions to scalable protocols. For example, a directory protocol should be extended with an arbitration mechanism. Arbitration would decide which processor can send its atomic update or invalidate packet to the rest of processors, avoiding the interleaving problem. Arbitration could be implemented, for instance, using a token-based mechanism. Briefly, the simplification of a normal coherence protocol under this approach would be the same as for a bus: no coherence state is potentially needed for each memory address. However, it would be good for performance to maintain the list of sharers on each directory entry in order to reduce the number of messages.

4.2 Memory Consistency

One of the most important properties of implicit transactional memory is that sequential consistency is simply enforced, potentially without performance loss due to memory ordering constraints and in a manner that is transparent to the software. The key observation is that a globally consistent transaction execution order is established, where transactions from a single thread respect the program order of that thread.

In Figure 2 we give an example of a sequentially consistent global order of memory operations from two different processors; the operations are labeled {A1, A2, A3} for processor A, and {B1, B2, B3} for processor B. The third column shows a global order that respects the program orders from both processors.

Our proposed implementation groups memory accesses from each processor into *implicit transactions* by taking checkpoints. Thus, we can extend the definition of sequential consistency to such transactions, and require only transactions from each processor to be in order. The resulting global order will be an interleaved sequence of transactions that meets the basic definition of SC because it corresponds to one of the possible sequentially consistent global orders. Figure 3 provides 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 program order for these transactions will also respect program order for memory operations.

It is important to note that memory operations can be executed out-of-order, and therefore in Figure 3 it would be possible for instructions in transaction "TR_A1" to be reordered, for example, as "A2, A3 and A1" instead of the order "A1, A2 and A3" as shown. This avoids memory ordering performance losses without violating sequential consistency, as the speculative character of these operations ensures their validity, and they are committed atomically in a block. Because the Kilo-Instruction Processor can support hundreds of in-flight memory instructions with a constant checkpoint recovery time, this approach is expected to be significantly more scalable than the proposed SC++ implementation [5] that relies on a tracking buffer with a variable recovery time whose size increases with the number of instructions tracked.

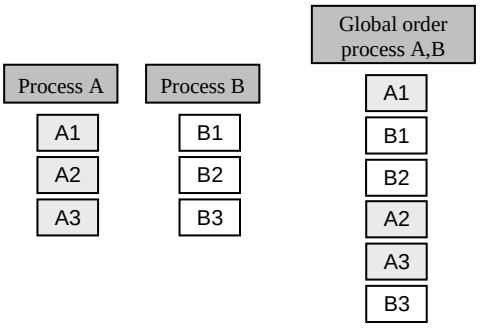

Figure 2: Sequentially consistent reordering of memory operations from 2 processors.

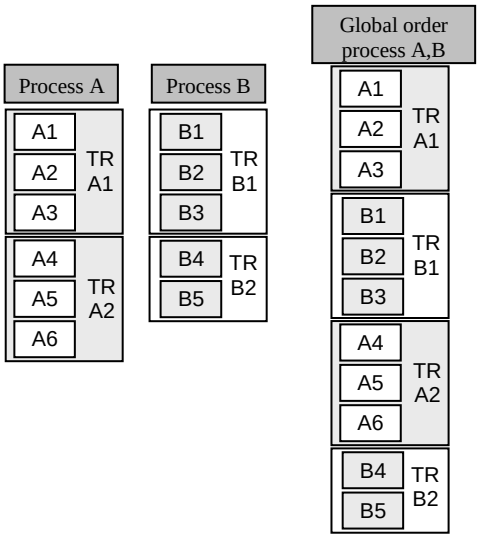

Figure 3: Sequentially consistent reordering of checkpoints from 2 processors.

5. Synchronizations and Implicit Transactions

We now turn our attention to ways that implicit transactions can avoid costly serialization losses by allowing multiple threads to speculatively execute beyond synchronization points. We first show a basic scheme that focuses on correct concurrent execution of critical sections and then discuss opportunities for optimizations of the basic scheme.

5.1. Basic Scheme

Figure 4 shows an example in which a processor executes a lock-protected critical section under three checkpointing scenarios A, B, and C. When a processor reaches the lock, it checks the value of the lock and acquires it if it is free; otherwise it spins on the lock variable. If the lock is free, the store to the lock variable, i.e. the acquiring operation, remains in a speculative state in the processor queues until the store can commit. Meanwhile, a new checkpoint can be taken inside the critical section (Figure 4a). Committing the transaction that finishes inside the critical section will validate the lock store instruction, thereby globally acquiring the lock, and forcing remote processors inside the critical section to roll back due to the invalidation of the lock variable. After that, remote processors will be inhibited from entering the critical section, due to the acquired lock. The validation of transaction T1 will force any other thread executing inside the critical section to roll back, as the lock variable is in the remote processor read set. When the processor validates transaction T2, the critical section gets unlocked and remote processors can start executing it.

The same invalidation happens if no checkpoint is taken until the critical section is unlocked (Figures 4b and 4c). In this case, the lock variable is also written. Thus, the validation of a transaction that has entirely executed a critical section will cause any other processor that is speculatively executing it to roll back, which means that only a single valid processor stays inside a critical section. This should be the most frequent case, due to the short nature of critical sections.

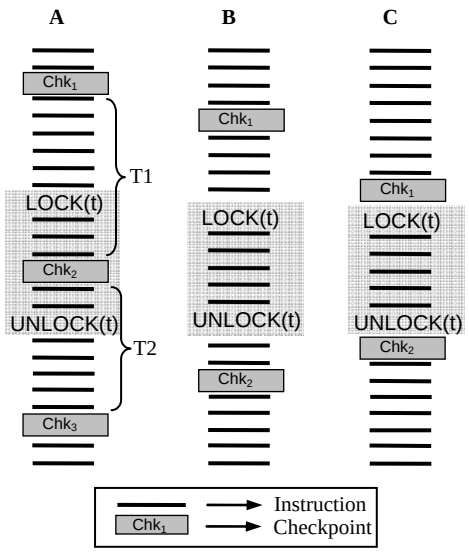

Figure 4: Different checkpointing scenarios.

Finally, it is obvious that the system behaves correctly in the presence of flags and barriers. In effect, processors that reach a closed flag keep spinning on it, ensuring that no code is executed past the flag. These examples show that the correctness substrate ensures correct execution in presence of critical sections and barriers, independently of where checkpoints are taken.

5.2. Performance Optimizations

Roll-backs will sometimes happen even if lock variable accesses are the only sources of data races. In theory, this leads to an unnecessary performance loss. When one processor has acquired a lock, the release the lock variable will write the same value to the lock variable that is already in this memory position – a *temporally silent store* [8]. We propose the use of two optional mechanisms that allow the system to detect these cases and naturally speculate through critical sections, thereby improving performance and reducing network traffic.

Store merging and silent store removal. The first mechanism detects and removes silent stores by detecting that a store does not globally modify memory, and removes that update from the update packet. 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. Store merging therefore removes a store when a younger store is to the same address. If we apply this mechanism when all the instructions of the checkpoint are finished, we need not bother with race conditions such as load instructions between two merged stores. In such a case, the load instruction would have received the correct value via store forwarding mechanisms internal to the processor. The next step is to remove silent stores by leveraging the store-forwarding logic used in current processors. Store-forwarding searches the older stores before executing a load, and if an address match is found, the value of the store is forwarded to the load. In contrast, our mechanism searches older loads before executing a store, and if an address and value match is found the store is removed because it is silent. The mechanism is simple because it is only necessary to look for silent stores within a single transaction, with all the loads queued waiting to be committed and the stores queued waiting to be broadcast to the memory hierarchy. This mechanism, when applied to the checkpoint scenario B in Figure 4, would remove the lock variable from the update packets and allow different instances of the same critical section to run in parallel avoiding processor serialization stalls, in the absence of other data races. For typically small critical sections, scenario B is likely to be the common case.

Lock detection mechanism. If locks are detected at run-time, we can ensure that no checkpoint is taken within a critical section, to enforce scenario B in Figure 4. We make use of a *lock detection mechanism*, which dynamically detects typical Test&Set lock constructs. To maximize concurrency, the lock detection hardware forces a new checkpoint to be taken just before the lock, so that the lock remains open for the rest of the processors at the beginning of the new transaction. In addition, the hardware could detect the lock release, which is a simple store to the lock variable, and take a new checkpoint just after it. This makes the transaction length equal to the critical section, and avoids unnecessary roll-backs due to data races on data outside the critical section as in scenario C in Figure 4.

Finally, we note that the proposed method preserves correctness independently of the length of the critical section. In contrast, TCC [6], which is a similar transaction approach, locally buffers all the memory updates corresponding to a certain transaction. In case of a buffer overflow, the processor in TCC must acquire the bus and not release it until the end of the transaction, thus blocking the rest of the system. In case of such an overflow, the system takes a new checkpoint and waits for the resources to free from previous checkpoints before continuing execution. Thus, in such a case, the behavior would be similar to scenario A in Figure 4, which is correct but does not enable parallel execution.

5.3. Speculating Beyond Flags and Barriers

The proposed design can be adapted to speculate after barriers, in a similar fashion as [10]. Here, too, there is a need to detect the barrier code and take a new checkpoint just prior to it. Speculative execution starts after the barrier. All the transactions after the 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.

To determine the time at 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) before checking 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. Note that a remote invalidation of the speculatively marked line does not force a roll-back, but makes the processor re-check the variable. Of course, all the speculative execution done before opening the barrier variable has no effect on the consistency model because the commit only happens after the real barrier opens up, and the correctness substrate ensures that the cache contents remain valid up to the commit instant.

The expected performance improvement of this scheme depends on the average time the processors wait at a barrier, while in the previous case the processor can commit the critical section and continue execution. Thus, if the waiting time does not exceed the time needed for the pipeline to fill up and stall, performance will improve. As Kilo-instruction Processors are designed to support thousands of in-flight instructions, this good-case scenario will occur frequently. Furthermore, in case of a data race forcing a roll-back, this mechanism will prefetch the needed data, possibly reducing memory latencies encountered later on.

6. Concluding Remarks

This paper introduces a framework that makes Kilo-instruction Processors capable of executing parallel code in a transactional fashion, similar to the TCC model, but assumes no modification of the code nor the programming model. 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 are already proposed for Kilo-Instruction Processors, and the processor model is simplified thanks to the transactional behavior.

Our model also enables speculative execution in critical sections and beyond barriers, reducing performance losses that these constructs cause in parallel programs. This, together with the advantages of transactional behavior, will provide high performance with no required code modifications.

Acknowledgments

This work has been supported by the Ministry of Education and Science of Spain under contracts TIN-2004-07739-C02-01 and 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.

References

[1] A. Cristal et al., "Kilo-instruction Processors: Overcoming the Memory Wall", In *IEEE Micro Magazine*, Vol. 25, No. 3, pp. 48-57, May/June, 2005.

[2] F. Dahlgren, M. Dubois, and P. Stenström, "Combined Performance Gains of Simple Cache Protocol Extensions", In *Proc. of 21st Int'l Symp. on Computer Architecture (ISCA'94)*, pp. 187-197, April 1994.

[3] M. Galluzzi et al., "A First Glance at Kilo-Instruction based Multiprocessors", In *Proc. of the 1st Conf. on Computing Frontiers*, pp. 212-221, April 2004.

[4] A. Gandhi et al., "Scalable Load and Store Processing in Latency Tolerant Processors", In *Proc. of the 32nd Int'l Symp. on Computer Architecture (ISCA'05)*, pp. 446-457, June 2005.

[5] C. Gniady, B. Falsafi, and T. N. Vijaykumar, "Is SC + ILP = RC?", In *Proc. of the 26th Int'l Symp. on Computer Architecture (ISCA'99)*, May 1999.

[6] L. Hammond et al., "Transactional Memory Coherence and Consistency", In *Proc. of the 31st Int'l Symp. on Computer Architecture (ISCA'04)*, pp. 102-113, June 2004.

[7] Maurice Herlihy and J. Eliot B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures", In *Proc. of the 20th Int'l Symp. on Computer Architecture (ISCA'93)*, pp. 289-300, May 1993.

[8] K. M. Lepak, and M. H. Lipasti, "Temporally Silent Stores", In *Proc. of the 10th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, October 2002.

[9] T. Karkhanis, and J.E. Smith, "A Day in the Life of a Data Cache Miss", In *Proc. of the 2nd Workshop on Memory Performance Issues (WMPI 2002)*, May 2002.

[10] J. Martínez, and J. Torrellas, "Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications", In *Proc. of the 10th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, October 2002.

[11] R. Rajwar, and J. R. Goodman, "Transactional Lock-Free Execution of Lock-Based Programs", In *Proc. of the 10th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, October 2002.

[12] 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 Symp. on Parallelism in Algorithms and Architectures (SPAA'97)*, pp. 199-210, June 1997.

[13] P. Rundberg, and P. Stenström, "Speculative Lock Reordering", In *Proc. of Int'l Parallel and Distributed Processing Symp. (IPDPS'03)*, April 2003.