Solving Multiprocessor Drawbacks with Kilo-Instruction Processors

Abstract

Nowadays, a good multiprocessor system design has to deal with many drawbacks in order to achieve a good tradeoff between complexity and performance. For example, while solving problems like coherence and consistency is essential for correctness the way to solve processor stalls due to critical sections and synchronization points is desirable for performance. And none of these drawbacks has a straightforward solution.

We show in our paper how the multi-checkpointing mechanism of the Kilo-Instruction Processors can be correctly leveraged in order to achieve a good complexity-effective multiprocessor design. Specifically, we describe a Kilo-Instruction Multiprocessor that transparently, i.e. without any software support, uses transaction-based memory updates. Our model simplifies the coherence and consistency hardware and gives the potential for easily applying different desirable speculative mechanisms to enhance performance when facing some synchronization constructs of current parallel applications.

1. Introduction

Currently, the relevance of multiprocessors is increasing, especially when focusing on simple on-board or even on-chip systems with few processors or cores. Many of these designs implement shared memory multiprocessors since they provide, in a cost-effective way, a clear programming model where sharing code and data structures is simpler.

However, there are several aspects that complicate the design and programming of a shared memory system beyond the single processor architecture, and limit its performance. Among these constricts or difficulties, we can cite:

- The interconnection hardware, which should provide a quick access to remote data and memory. Accessing data from remote memories is much slower than from local memory, increasing the effective memory latency. This latency can be increased even more by the overhead that the coherence protocol imposes.
- The coherence protocol, which determines how different memory values can be shared between private caches of different processors, ensures that the same global value for a given memory position is observed. Basic protocols are based on a memory directory or a broadcast bus, though much work has been done in this field, trying to make simpler protocols and improve performance [ref. Token, ref. Sohi-Burger].
- The consistency model, which determines how the system overlaps or reorder memory operations. Sequential Consistency (SC) [ref SC] is the most desirable model as it provides the most intuitive programming model, but it requires that memory operations from each program have to be executed in-order what limits the performance of the system. Other models [ref. S.AdveConsistency] achieve a

higher performance by relaxing these constraints, at the cost of a more complex programming framework.

- The need for sharing data between different processes can generate access conflicts, which are solved with exclusion mechanisms. Locks ensure sequential access to shared data, hence stalling different processors that try to access the same critical section simultaneously. Sometimes critical sections are created in a conservative manner what can degrade the performance. Previous works [ref. Martinez, ref. TLR, ref. SLE, ref. TCC, ref. Stenstrom] have proposed the speculative execution of critical sections, improving performance in case of no real data contention.
- Finally, synchronization operations are used to ensure that different program threads can communicate each other and cooperate. These operations, such as barriers or flags, can also stall processors since a synchronization wait is needed, sometimes in a conservative manner. Some proposals [ref Martínez, ref. TCC] deal with this problem, with the same speculative execution idea.

A way to solve the problems above mentioned, to some extent, is to employ **transactional memory systems**. These systems employ software or hardware transactions, which are expected to be atomic, in order to communicate with the memory, and try to simplify the design or improve the performance of a multiprocessor environment. Much work has been done on this field and we list some important recent works:

- Transactional Lock Removal (TLR) [ref. TLR] is a method that detects critical sections, and dynamically substitutes them with transactions, eliding the lock acquiring. In this approach, transactions are used only to substitute critical sections, with the aim of augmenting parallelism if no real contention occurs.
- Thread-level Transactional Memory (TTM) [ref. TTM] is a software-hardware approach that covers different levels of the system. They propose to build a transactional support at thread level, so that the programmer can start and finish transactions, with support in lower levels of the operating system and hardware.
- Transactional Coherence and Consistency (TCC) [ref. TCC] proposes a new shared memory model where atomic transactions are the basic unit for communication, simplifying parallel applications and the coherence and consistency hardware. Similarly to TTM, this proposal modifies the programming model, forcing the programmer to divide manually the program into transactions.

However, the memory latency problem, in particular, has been best showed to be solved by the **Kilo-Instruction Processors**. They try to hide the memory latency by holding thousands of in-flight instructions. This has been proved to be efficient both in single processors [ref. Adrian] and in multiprocessor environments [ref. Marco]. Among the different mechanisms proposed to enable this amount of in-flight instructions, the main mechanism is a multiple checkpointing [ref. OOOcommit] which easily facilitates speculative execution.

This paper proposes the use of the Kilo-Instruction Processors as the base for a transparent transaction-based multiprocessor, which presents the interesting property of not needing any software support. Our proposal comprises all the advantages of a transactional memory system, what includes preserving sequential consistency in a natural and efficient manner. The multiple checkpointing mechanism, already incorporated in Kilo-Instruction Processors, makes the proposed system a high-quality complexity-effective design option, since it is leveraged to automatically mark transactions boundaries and to straightforwardly apply speculation mechanisms to improve performance on critical sections and synchronization points.

2. Background

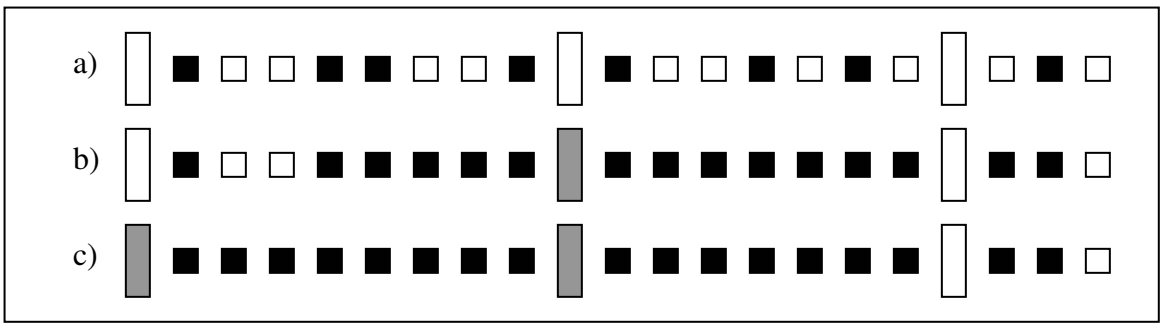
2.1. Kilo-Instruction Processors

First works on single Kilo-Instruction Processors [ref Adrian] demonstrated in detail their ability for hiding large latencies, specifically due to memory accesses, since the processor allows thousands of instructions to be in-flight at the same time. Different mechanisms need to be applied for supporting such amount of in-flight instructions without simply up-sizing not-scalable structures such as the re-order buffer (ROB).

The ROB is substituted with a reduced structure called pseudo-ROB, which only maintains the youngest in-flight instructions, and a multi-checkpoint mechanism. A checkpoint is a snapshot of the processor state, which is taken at a specific instruction of the program being executed. When an exception or branch misprediction occur, if the excepting instruction is found in the pseudo-ROB a normal flush up to the excepting instruction is done, otherwise the state will be rolled back to the closest checkpoint prior to the excepting instruction, making the processor suffering a longer recovery time. Fortunately, the former case is supposed to happen more frequently than the latter one. Therefore, using a relatively small set of checkpoints for long flight time instructions assures safe points of return and reduces ROB requirements considerably.

Multi-checkpointing

The multi-checkpointing mechanism works as follows. As the pipeline advances, in-flight instructions corresponding to different checkpoints are executed. Those instructions remain speculative, as long as there are pending instructions in the same checkpoint. When all the instructions corresponding to the older checkpoint are finished, the processor validates the checkpoint atomically. With this action, the speculative instructions become globally executed, and they are removed from the queues of the processor. Figure XX shows an example of the multiple checkpointing mechanism, where oldest instructions are in the left part. In case a), instructions from three consecutive checkpoints are in flight. Black instructions are finished, but not committed. 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 oldest checkpoint finishes, so it can commit, followed by the second one, which becomes the oldest after the commit operation.



The multi-checkpointing mechanism comes for free with Kilo-Instruction Processors and helps our system in two interesting ways. First, a checkpoint is used as transaction delimiter, what makes any software support unnecessary unlike the TCC approach. Second, checkpointing implicitly allows speculative execution of instructions and, if needed, recovery from violations, what permits processes to go through locked critical sections and barriers.

2.2. Kilo-Instruction Multiprocessors

This previous work by A. Cristal et al. and the increased latency problem due to remote accesses in multiprocessors is what motivated the idea of Kilo-Instruction Multiprocessors where we place together a number of Kilo-Instruction Processors for building small-scale non-uniform memory access (NUMA) multiprocessors. The new multiprocessor configuration, firstly published in [ref CFmarco], has been shown to effectively hide large latencies coming from both local and remote memory accesses, thus hiding also latencies coming from traversing the interconnection network. Figure 1 shows the reduction in the execution time for different benchmarks from the splash2 suite, when using 1024 in-flight instructions instead of 64, and having a memory access time of 500 cycles.

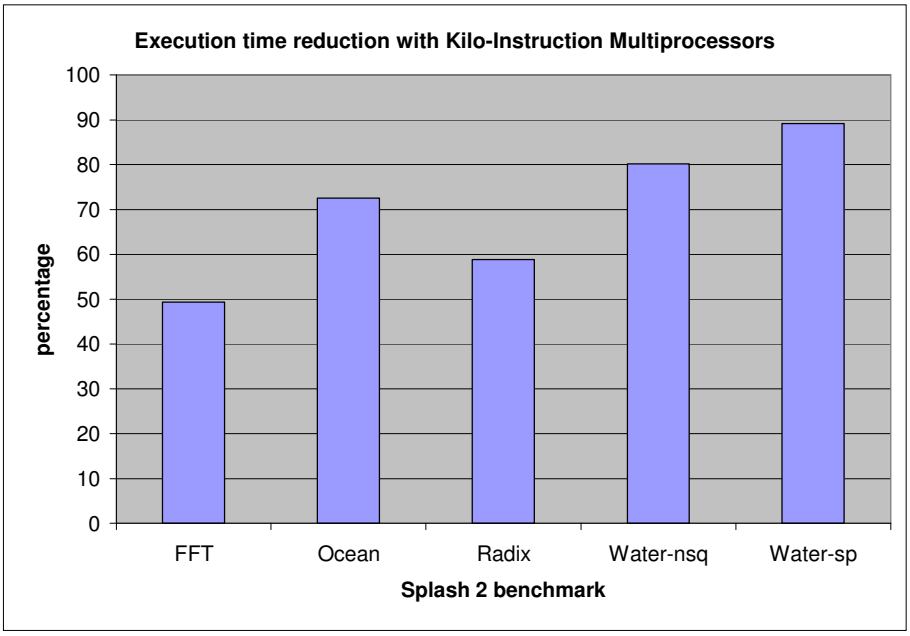


Figure 1: Execution time reduction for a 16-processor system with 500c memory access time

The work [ref CFmarco] explores for the first time multiprocessors based on processors that uses checkpointing as the basic mechanism for improving performance, while other previous works like [ref SafetyNet] make use of checkpointing just for fault tolerance. However, in [ref CFmarco] only an evaluation of the performance achieved by the system is done, leaving out any kind of architectural detail. Describing in detail the architecture of such systems is exactly what we partly want to solve with this paper, showing one possible design for checkpointing-based multiprocessors.

3. System Overview

We center our proposal on shared-memory multiprocessors because they are a cost-effective option for multiprocessors. In particular, we propose as the basic system a small-scale multiprocessor where the nodes are Kilo-Instruction Processors interconnected via a snoopy bus. Kilo-Instruction Processors give to the multiprocessor environment the interesting ability of hiding latencies coming from local or remote memory accesses as shown in [ref Marco].

However, our proposed system goes further on and takes advantage of the main mechanism of the Kilo-Instruction Processors, i.e. the multi-checkpointing. We propose storing the memory updates from a checkpoint and release them to the memory hierarchy just before committing the checkpoint, updating or invalidating other caches. Remote processors simply snoops the memory updates searching for a conflict with current loads. Only when a conflict is found a rollback to a previous checkpoint is needed for the conflicting remote processor, what is expected to occur infrequently.

This way our system acts like a transaction-based system, similarly to the proposed TCC [ref TCC] but without any software support. This behavior allows the simplification of the coherence protocol, since multiple states for a shared line are not needed, and the natural support for sequential consistency memory model, which is the easiest way for programmers to think about the ordering of memory operations.

Section 4 explains in detail the multiple checkpointing mechanism. The following sections, 5 and 6, make some coherence and consistency considerations.

Furthermore, multi-checkpointing consents to improve performance by adding trivial hardware modifications and support the concurrent execution of critical sections and the speculative execution beyond synchronization points.

In order to concurrently have different threads executing a critical section we ensure that a checkpoint is taken before a lock construct by detecting the corresponding read-modify-write instruction. Not locking the entrance to a critical section is what makes different threads to enter concurrently. Since the updating of the lock variable would conflict with remote speculative executions, we also apply a “silent stores elimination” mechanism

avoiding unnecessary rollbacks if no actual data conflicts exist in the critical section. Adding a mechanism for detecting read-modify-writes instructions is straightforward. The detection of silent stores is also easy to implement and will aid the system reducing the number of stores released to the memory hierarchy. Section 7 details the mechanisms that we propose.

Speculating beyond synchronization points is also easy, since we can have a mechanism to detect simple barriers constructs. Once a barrier is detected a checkpoint is taken and the cache line corresponding to the flag variable is marked and changed to the expected value. This way the processor is able to execute following instructions in a pure speculative mode, which means that no checkpoint can be committed. The speculated instructions can be rolled back if a memory conflict is found, otherwise they will wait for an update of the line associated to the flag variable indicating that the expected and speculated value has been reached what disables the pure speculative mode. Of course, any modification to the flag variable that does not correspond to the expected value will be ignored and will not produce a rollback. More details are given in section 8.

The recovery from violations for such mechanisms is included for free in Kilo-Instructions Processors since they already restore the register state from the previous checkpoint of an excepting instruction.

4. Transactions

We define a transaction as the instructions included between a taken checkpoint, to which it belongs, and the next one. The stores of a transaction are kept as a group, and atomically released to the memory hierarchy only when the associated checkpoint is committed, updating the main memory and the other remote caches.

4.1. Basic Operation

In our snoopy bus based shared memory system, the stores or memory updates are packaged and in-order broadcasted through the network after a checkpoint validation, and the update broadcast packet is snooped by the remote caches. This action validates all the speculative memory updates corresponding to a certain checkpoint, its write set, and allows the remote processors to check if the data they are using speculatively, its read set, has been modified. Thus, remote caches compare received modified addresses with their tags, and if there is a match, and the data has been used, the processor rolls back to the checkpoint just before the data use. All the speculatively executed instructions are valid as long as a processor does not detect such a collision, and when all the instructions within two consecutive checkpoints finish, they can safely commit.

4.2. Adaptive Transaction Length

The number of outstanding transactions and the number of instructions included in a single transaction will directly depend on the number of possible checkpoints and on the instant in which a checkpoint is taken. Previous experiences with a single Kilo-

Instruction Processor dictates that a few checkpoints are enough, and that a checkpoint has to be preferably taken when long latency instructions appear.

In this multiprocessor transactional system, we propose that checkpoint length should be adaptive, decreasing in case of a rollback and increasing as long as no rollbacks occur, up to a maximum value. This ensures that, in case of frequent collisions, the length of the transactions decrease, also decreasing the number of collisions. This is valid as our system, based in traditional code not oriented to transactions, maintains correctness independently of the instruction where the checkpoint is taken. Even more, the processor system can consider the type of memory accesses, since no access to shared memory pages would not force a rollback due to data collision. In order to speculate through critical section and synchronization point we also add a mechanism to take a checkpoint when a lock or barrier construct is found, what will be deeper explained in sections 7 and 8.

5 Coherence Mechanism

Our broadcast-based approach works as a snoopy write invalidate or write update protocol. Any processor can modify a cache line, and the in-order broadcast mechanism described in the previous chapter will notify the change to the rest of processors when the transaction commits. Multiple modified cache lines can coexist in different processors, as long as they are speculative. The update of one of them invalidates any other cache lines, which ensures that only a valid copy exists.

The contents of the commit 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 addresses, remote processors will invalidate those lines, and the protocol will work as write invalidate.

This coherence mechanism can be implemented over a shared bus or a network, as long as it ensures in-order broadcast. Obviously, such a mechanism can be applied to small scale systems, up to a limit of about 32 processors. For bigger systems, a hierarchical protocol (such as [ref. coherencia jerárquica]) should be used, as the broadcast mechanism would soon saturate the network.

6. Consistency

6.1. Consistency Models

The memory consistency model of a shared-memory multiprocessor determines how the system can overlap or reorder memory operations, offering a trade-off between programming simplicity and high performance. Less restrictive models, such as Release Consistency (RC) [ref] provide a higher performance, at the cost of not ensuring the validity of the order of memory operations. On the other hand, Sequential Consistency (SC), the most restrictive model, guarantees that memory operations appear to execute in program order, which is more intuitive for the programmer, at the cost of a generally

lower performance [ref]. Some other consistency models provide intermediate performance, such as Processor Consistency [ref] or Total Store Ordering [ref].

The proposal TCC [ref], solve the problem of the consistency by imposing a new programming model and offering a parallel model based in software transactions. With TCC just imposing a sequential ordering between transactions commits is needed.

Sequential Consistency is the most desirable model, as it is the simplest model to understand and it provides the most intuitive programming interface. With this model memory operations from each processor appear to be executed atomically and in order and this is what programmers expect when building parallel applications because of its similarity to the familiar uniprocessor memory system. A basic implementation of SC just require a processor to delay the next memory access until the previous one is completed, what is simple but evidently leads to a low performance.

6.2. Our proposal

Our proposal, based on the already explained transactional behaviour, provides SC support in a natural manner: instead of assuring that single memory operations follow an ordered execution, we require transactions to be in order. Transactions are the basic unit of execution in our system. The instructions within a transaction are executed speculatively and validated atomically, when all of them are finished. Requiring an order for transactions inside a single program, then, is trivial since the rollback mechanism itself, in case of a collision, makes different transaction behave as if they were consecutive.

Next, we explain with an example why we maintain SC. Sequential consistency requires that the result of any execution be the same as if the memory accesses executed by each processor were kept in order and the accesses among different processors were arbitrarily interleaved. In other words, we can have different global orders were we interleave memory operations from the different processors, but each of these global orders must maintain each individual program order. In table 1 we have en example of a sequentially consistent global order of memory operations from two different processors, labeled [A1, A2, A3] for processor A, and [B1, B2, B3] for processor B. The third column shows that this global order respects exactly the program orders from processors A and B.

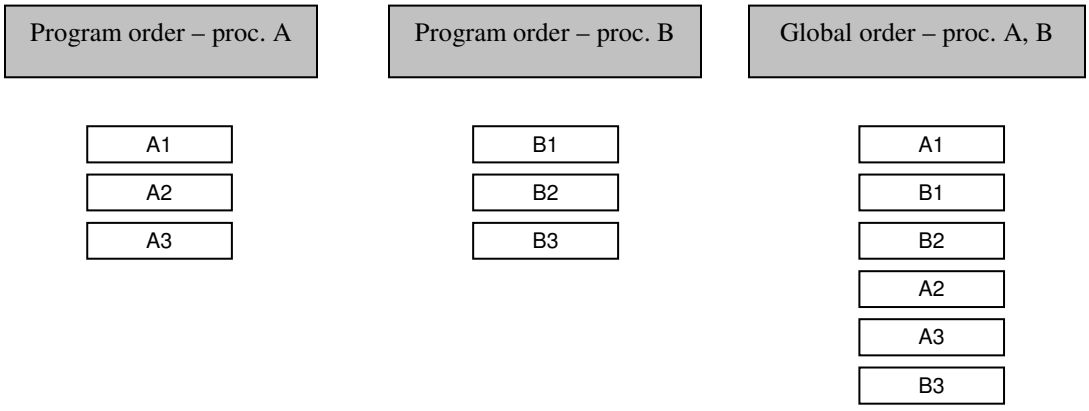


Figure 1. Example of a sequentially consistent reordering of memory operations from two processors

In our proposal we group memory accesses from each processor by taking checkpoints. Thus, we can extend the definition of sequential consistency to checkpoints, and just requiring checkpoints from each processor to be in order. The resulting global order will be an arbitrarily interleaved succession of checkpoints that will also meet the basic definition of SC since it corresponds with one of the possible sequentially consistent global orders for memory operations. We show an example in table 2 where we group instructions in checkpoints, labeled [Chk_a1, Chk_a2] for processor A, and [Chk_b1, Chk_b2] for processor B. The third column shows that respecting the program order for checkpoints will also respect program order for memory operations.

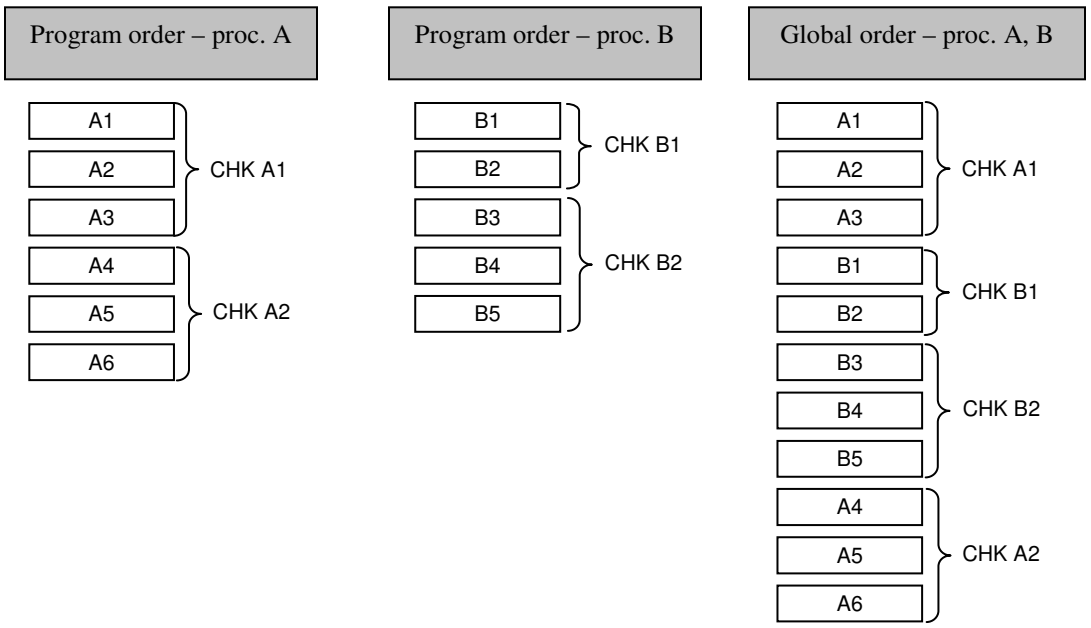


Figure 2. Example of a sequentially consistent reordering of checkpoints from two processors

Within a transaction, different instructions can finish execution out of order. This does not constitute a consistency problem. Within the transaction, the store-forwarding mechanism ensures that all load instructions receive the correct value, even if it has been modified by another instruction in the transaction. The validation of the write set (i.e., all the different store instructions in the checkpoint) is made in exclusive mode, this is, no other processor can validate any other checkpoint at the same time. Thus, the whole validation is sent atomically, and the global result is the same independently of the order of execution of instructions, which is valid due to the store-forwarding. Thus, the validation of the transaction executed out of order is globally equivalent to the validation of the same transaction in which all instructions are executed in order, ensuring SC.

SC has traditionally provided a lower performance respect to other consistency methods, such as Release Consistency [ref], due to the constraints that it imposes to the hardware.

However, the behaviour of our transactional system can reduce these constraints. As the global memory is not remotely modified during the execution of the instructions of a single transaction, there is no need to follow a strict order in the load or store operations, as happens in SC. Even more, previous studies [ref. S. Adve, ref SC+ILP=RC] show that a long instruction window narrows the performance gap from SC to other consistency methods. Thus, the expected performance will be higher that typical SC implementations.

7. Locks

Lock structures control the access to critical sections by allowing only one process, the lock owner, to access a critical where shared variables are read and modified. A typical critical section can be seen in fig. X

```
lock(c)
        short set of instructions
unlock(c)
```

Fig. X

Critical sections ensure exclusive access to the lock owner, forcing the rest of the threads to stall up to the lock release. Sometimes, this is unnecessary, as different threads might not modify any data, or modify different fields of a shared object. In these cases, parallel execution could be allowed, avoiding the stall of the threads waiting for the lock.

7.1. Related work

Previous works have tried to reduce the performance loss due to unnecessary waits for lock releases. In [ref SLE] Rajwar and Goodman showed that some locks are too conservative, and the collision in the access to shared data can occur in only certain conditions. Thus, they proposed SLE, a hardware approach that detects typical Test&Test&Set lock construction and avoids acquiring it, 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 same critical section can be executed in parallel, and if a collision happens, only one of them advances, forcing the other ones to restart with new data. This method was improved in [ref TLR] by adding a timestamp to the write requests, which avoids process starvation, ensuring the forward progress of the oldest thread.

In [ref Martínez] Martínez and Torrellas proposed Speculative Synchronization. With a similar lock detection mechanism, they propose the existence of a safe thread, which really 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 (if they have already finished their critical section) or compete for ownership of the lock.

In [ref Stenström] an improvement to the previous ideas is proposed. All the speculative threads execute their critical section, even though they have any conflict. When all of the speculative threads can commit, an arbiter dictates the order in which they do, so that collisions are minimized.

TCC [ref TCC] proposes a transaction based system that avoids the use of locks at critical sections. Instead, code is divided into transactions, and a hardware mechanism ensures that they behave as if their execution was atomic. Instead of using critical sections, every access to shared data falls within the same transaction. Thus, there is no need for software locks, but a transaction change instruction. This method naturally allows the parallel execution of critical sections, and detects conflicts. The main problem is that code has to be re-written and compiled with transactions in mind

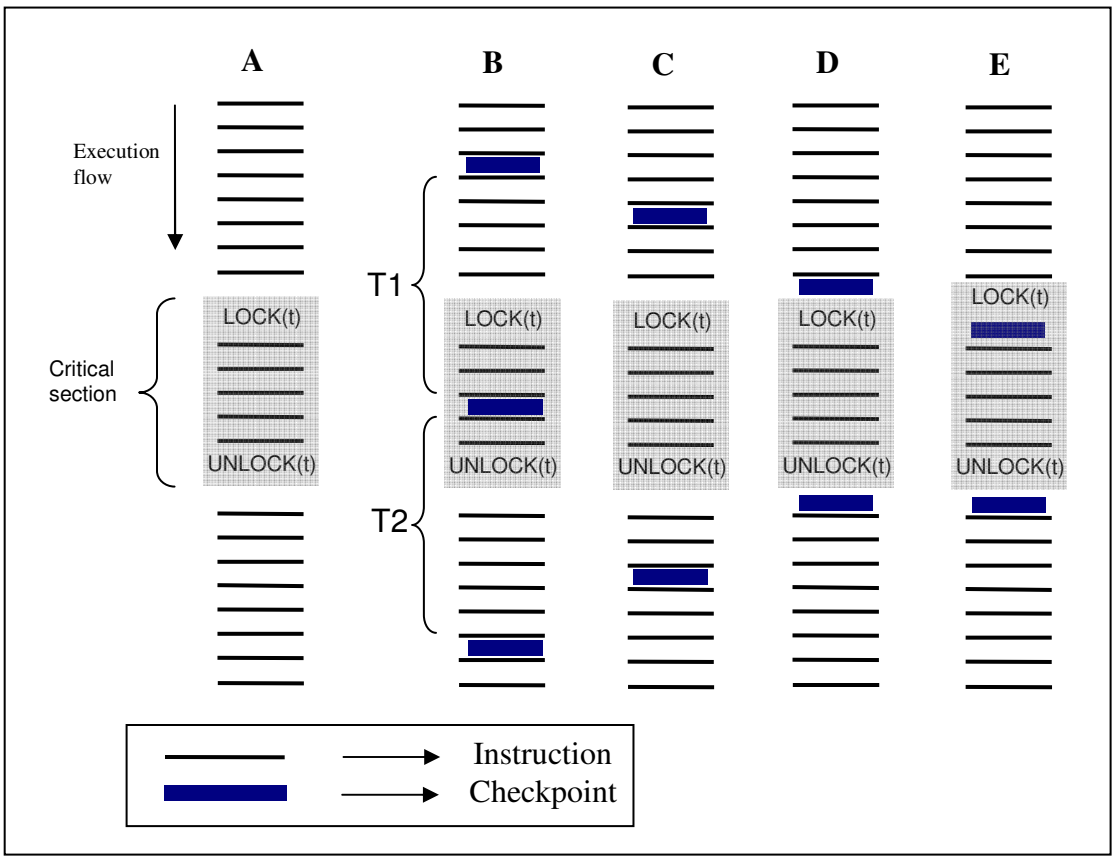
7.2. Our proposal

Our proposal makes use of kilo-instruction multiple checkpointing to execute code in portions, similar to the transactions in TCC, which behave as if they were atomic. We will see how the different hardware components that we propose influence the system behaviour. Figure X a) represents a portion of code with a critical section, bounded by the lock and unlock operations. When a processor reaches the lock, it checks the value of the lock and speculatively acquires it, in case it is free.

The multiple checkpointing with the invalidation mechanism ensure correct code execution, including critical sections. The lock variable is read and speculatively modified. If a new checkpoint is taken inside the critical section, the validation of the transaction that finishes inside the critical section will acquire the lock, forcing remote processors inside the critical section to rollback, due to the lock variable invalidation. After that, remote processors would not enter the critical section, due to the acquired lock. This case is shown in figure X b), where a checkpoint is taken inside the critical section. The validation of transaction T1 would force any other thread inside the critical section to rollback.

The same happens if no checkpoint is taken until the unlock of the critical section, as the lock variable is also written, even though with the original open value. Thus, the validation of a transaction that has entered inside a critical section would cause any other processor that is executing it speculatively to rollback, which ensures that only a single valid processor inside a critical section. This case is shown in figure X c), and it would be the most frequent one, due to the short nature of critical sections. However, if no real interaction there was within different threads, in this case parallel execution could be

As the code is executed different checkpoints are taken. Cases b) to e) show different possible checkpointing schemes. In case b), a new checkpoint is taken within the critical section. The validation of transaction T1 would globally close the lock, forcing remote processors within the critical section to rollback, as they snoop the invalidation of the lock variable. When the processor validates transaction T2, the critical section gets unlocked and remote processors can execute it.



Case c) shows an execution in which no checkpoint is taken within the critical section. A critical section is usually short, so this is likely to happen, and in this case several processors could execute the critical section in parallel if no real data interaction there was between them. Such a data interaction is determined by the values that each processor modifies when it atomically validates the transaction. The lock variable is acquired and released atomically within each transaction, and if it was included in the update packet, it would force remote processors to rollback, though it remains with the same value. The silent stores detection and removal mechanism is used to detect this behaviour, at least for the lock variable, and it will remove it from the update packet. Thus, if different threads have no real interaction within the critical section, their execution can progress forward in parallel, as in [ref TLR Rajwar]. Else, the first thread to finish will modify some data, forcing the rest of the threads to rollback with new data.

As it seems desirable that no checkpoint is taken within a critical section, we make use of the lock and barriers detection unit, which analyzes code and detects the lock, with the typical Test&Test&Set construction. To maximize parallelism, the lock detection hardware forces a new checkpoint to be taken just before the lock, so that it remains open for the rest of the processors at the beginning of the new transaction. The hardware keeps analyzing the code to detect the lock release, and takes a new checkpoint just after it. This makes the transaction length equal to the critical section, and avoids unnecessary rollbacks due to collisions on data outside the critical section. This example is shown in figure X d). Anyway, it should be noted that in absence of the silent store removal and

the lock detection mechanisms, though with a lower expected performance, the multiple checkpointing with the invalidation mechanism would ensure correct execution.

In case of multiple consecutive conflicts, a processor can force the others to wait by just taking a checkpoint just after acquiring the lock, as shown in figure X e). This prevents the rest of them from entering the critical section, thus ensuring safe operation for the owner of the lock, and avoiding starvation. This behaviour is determined by the lock detection hardware, which tracks the number of times that a critical section execution is rolled back. In this case, the rest of the processors could speculatively execute the critical section, waiting for the lock to be released, as proposed in [ref Speculative Sync. Martínez], but the previous multiple collisions suggest that this would be inefficient, possibly having to roll back.

7.3 Silent Stores and Store Merging

A recent work on value locality introduced the term *silent stores* [ref ISCA2K]. A silent store is defined as a memory write that does not change the system state, and this work shows how a non-trivial percentage of the stores is silent. There are other works that make use of this program characteristic in their proposals, but we will just mention the speculative lock elision (SLE) [ref SLE] which is of our interest. The SLE proposal, previously explained in the paper, try to dynamically remove locks to make critical section being executed concurrently. Nevertheless, in order to achieve that lock removal, SLE needs to detect and elide the silent store pairs associated with the modification of the flag variable of the lock and unlock constructs of parallel applications. Therefore, SLE implements some kind of silent stores detection which is specifically designed for those silent store pairs of a critical section, without taking care of the rest of possible silent stores.

In our proposal, we also need to remove these silent store pairs since, similarly to SLE, we remove the lock from critical sections to concurrently execute them. However, the mechanism we propose do not just remove this kind of silent stores. We intend to remove all the possible silent stores that appear during a single transaction, which of course includes these silent stores associated with locks because we ensure a critical section to fall within a single transaction as explained in previous subsection.

The mechanism we propose is quite simple as we can leverage the existent hardware used for store-forwarding [ref]. Store-forwarding search between older stores before executing a load, and if an address match is found, the value of the store is forwarded to the load. Our mechanism instead, searches between older loads before executing a store, and if an address and value match is found, the store is removed since it is a silent store. Our silent stores mechanism is this simple since we just look for silent stores during a single transaction, and during a single transaction we have all the loads queued waiting to be committed and the stores queued waiting to be broadcasted to the memory hierarchy. Note that this silent store detection will be performed just before broadcasting the stores of a transaction.

Another mechanism that our system provides and can be easily implemented is a *store merging* similar to that implemented in the Alpha 21164 [ref]. With store merging we mean removing a store before executing it when a younger store matches the same address. This way when two stores to the same address are found only the younger one will be executed. If we apply this mechanism just before broadcasting the stores of a transaction, and before searching for silent stores, we do not have to take care about race conditions as in [ref ALPHA]. The Alpha 21164 need to ensure, for instance, that between these two stores there is not a load to the same address, but in our system, at the point we perform the store merging, all the loads of the transaction are supposed to be executed. Although, it is also true that our mechanism is simpler because, unlike the Alpha 21164, only stores of a single transaction are supervised.

8. Flags and Barriers

Flags and barriers are used to synchronize different threads. A correct execution must make all threads wait for the barrier to open, before continuing execution. Similarly to the previous case, this stall could be avoided in some cases, in which there is no real data interaction between different threads. Thus, barriers

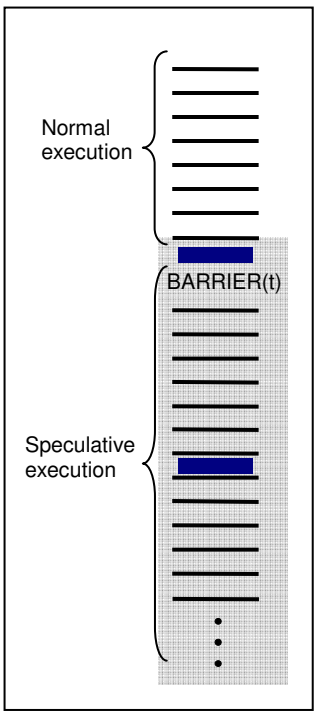
8.1. Related Work

Speculative Synchronization [ref] also deals with flags and barriers. When such a construct is detected, the execution is continued further, in speculative state. The code after a barrier remains speculative, waiting for the barrier to open before validating all that work.

TCC substitutes flags and barriers with software phase numbers. Each transaction is labelled with a phase number, and each processor tracks the phase of the others. 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 consecutive phase numbers are ensured to commit sequentially. Speculation is implemented naturally, as transactions after the phase change can be executed, but not committed.

8.2 Our proposal

Our proposal can easily be adapted to speculation after barriers, in a similar manner to [ref Martínez]. The lock and barrier detection unit would detect the barrier code, as in the previous case, and takes a new checkpoint just before it. A speculation control unit sets the speculative cache bit (bit S in figure XX) and starts speculative execution after the barrier, as shown in figure XXX, All the transactions after this barrier remain speculative, meaning that none of them can be validated, as long as the barrier remains closed. This is ensured by setting a “speculation mode” in the processor, avoiding any checkpoint validation corresponding to forward execution.



The speculative control unit waits for a cache event on the cache line marked as speculative (an invalidation or an update of the line) to check the barrier value again. When the barrier gets open, the "speculation mode" is disabled, and the processor can start committing all the transactions in the pipeline. Note that a remote invalidation of the speculative marked line does not force a rollback, but makes the speculative control unit check the variable again.

This speculation has not effect on the consistency model. All the execution has been made before the modifying of the barrier variable, which does not respond to SC. However, the rollback mechanism ensures that the cache contains are not externally modified, and the validation after the barrier opening makes those transactions behave as if they were executed after it, at the commit point.

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, in this case all the forward execution will have to wait for the barrier to actually open before committing. Thus, if the waiting time does not exceed the time needed for the pipeline to get full and stall, we could have performance improvements. As the Kilo-instruction Processors are designed to have thousands of in-flight instructions, it can occur frequently. Furthermore, in case of a conflict forcing a rollback, this mechanism would prefetch needed data, similarly to [ref run-ahead execution], possibly reducing following memory latencies.

9. Conclusion

This paper shows a framework that makes Kilo-instruction Processors capable to execute code in a transaction way, similar to the TCC model, modifying neither the code nor the

programming methodology. Our model maintains Sequential Consistency with low hardware constrains, and a high expected performance. The hardware requirements are low, as most of the mechanisms are already proposed for kilo-instruction processors.

Our model considers speculative execution in critical sections and barriers, eliding as far as possible the performance loss that these constructs cause in parallel programs. This, together with the advantages of transactional behaviour, will provide a high performance with no code modifying.

References

[S. Adve] P. Ranganathan, V.S.Pai, 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.

[S. Adve consistency] V. S. Pai, P. Ranganathan, S. V. Adve, T. Harton, "An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors", In Proc. Of the 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, October 1996

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

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

[TLR] R. Rajwar, J. R. Goodman, "Transactional Lock-Free Execution of Lock-Based Programs". In Proc. of the 10th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, Oct. 02.

[SLE] R. Rajwar, J. R. Goodman, "Speculative Lock Elision: Enabling Highly-Concurrent Multithreaded Execution". In Proc. of the

[TCC] K. Olukotun *et al*, "Transactional Memory Coherence and Consistency" In Proc. of the 31st Annual International Symposium on Computer Architecture, Germany, June 2004.

[Martínez] J. Martínez, J. Torrellas, "Speculative Synchronization: Applying Thread-Level Speculation to Explicitily Parallel Applications", In Proc. of the 10th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, Oct. 02

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

[silent stores] Mikko H. Lipasti, Kevin M. Lepak, "On the Value Locality of Store Instructions", In Proc. of the 27th ISCA, 2000

[SC+ILP=RC] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar, "Is SC + ILP = RC?", In Proc. of the 26th ISCA, 1999

[TTM] Kevin E. Moore, Mark D. Hill and David A. Wood, "Thread-Level Transactional Memory", TR1524, Comp. Science Dept. UW Madison, March 31, 2005

[Token Coh.] Milo M. K. Martin, Mark D. Hill and David A. Wood, "Token Coherence: Decoupling Performance and Correctness", In Proc. of the 30th ISCA, 2003

[Sohi-burger] J. Huh, J. Chang, D. Burer, G. S. Sohi, "Coherence Decoupling: Making Use of Incoherence", In Proc. Of the 11th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, October 2004

[Stenström] P. Rundberg and P. Stenström, "Speculative Lock Reordering", In Proc. Of the Int. Parallel and Distributed Procesing Symposium, April 2003

[ref SafetyNet] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. "SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery", In Proceedings of the 29th Annual International Symposium on Computer Architecture, June 2002.