Analysing Data Access Strategies in Cache Coherent Architectures

Karin Genther and Rüdiger Reischuk

Inst. für Theoretische Informatik, Med. Universität zu Lübeck
Wallstraße 40, D–23560 Lübeck
{genther, reischuk}@tcs.mu-luebeck.de

TR A-98-25

December 1999 (revised version)

Abstract. The conception of multiprocessor systems based on shared memory is to provide logically uniform data access, even in systems with many processors. Since data requests that cannot be satisfied physically within the processor environment suffer from high latencies, these systems employ memory hierarchies including private caches. Two powerful techniques that reduce and hide latency in memory hierarchies are buffering and pipelining of data accesses. In order to utilize these techniques in global shared-memory architectures, new memory consistency models have been defined. In this paper, we discuss the cache coherence problem and three consistency models that are commonly used to enhance performance. We develop a uniform parameterized model for different parallel architectures and estimate the latency for data access in each consistency model abstracting from the details of hardware mechanisms. The goal of our approach is to obtain realistic predictions of running-times for the various models.

1 Introduction

New multiprocessor architectures based on the shared-memory paradigm have emerged during the last years, beside the familiar multiprocessor systems with distributed memory already known and investigated for a long time. In the most general case, such systems provide a non-uniform memory access (NUMA) because access paths to memory modules might have different properties depending on processor and memory addresses. As memory accesses that are not satisfied within the processor environment suffer from long latencies, these systems employ memory hierarchies in order to achieve better performance behaviour. Within this setting, the simple techniques known from bus-based architectures for insuring cache coherence had to be extended. New memory consistency models have been introduced and evaluated using simulation techniques.

Until now, no general analytical models have been established for NUMA architectures that may help to answer questions concerned with effective running times of parallel algorithms or practically efficient adaptation of algorithms to a specific architecture. So far, we know such theoretical models for synchronous algorithms, e.g. the BSP model introduced by Valiant in 1990 [15], or the LogP model developed by Culler et al. in 1996 [3]. They use a few parameters to describe the performance of networks of workstations.

Such a model would facilitate describing the efficiency of a parallel architecture using only parameters for representing latency when accessing different parts of the memory hierarchy and thereby abstracting from the details of the network topology. In contrast to the results obtained by simulation, above all the model would aid in specifying the exact performance gains of the different memory consistency models.

In this paper, we review the main aspects of cache coherence and the three most important memory consistency models (i.e. *sequential consistency* – SC, *total store ordering* – TSO, and *partial store ordering* – PSO) in Sect. 2. We emphasize on separating the aspects belonging to coherence mechanism from those belonging to consistency because coherence seems to be an indispensable characteristic of a memory system in order to develop correct programs, whereas the choice of a consistency model only influences the programming style and the speed of the execution. In Sect. 3 we develop architectural models for NUMA systems which help to separate those parts of the coherence protocols which originate in the mechanism of the hardware from those techniques implemented to insure coherence and the desired consistency model. For that, we focus on two-level memory hierarchies, i.e. on a system with individual caches, which are near to and private for each processor, and one level of global memory with shared address space that is equally far from all processors. We show in Sect. 4 how to measure the amount of time needed for single data accesses in different architectures and thereby differentiating between the three consistency models. Finally, in Sect. 5 we examine one application where the consistency models used in this paper exhibit a very different behaviour as far as performance is concerned. Section 6 provides a summary and directions for future research.

2 Cache Coherence and Memory Consistency

Following the well-known notion from uniprocessor machines, we intuitively call a memory system *coherent* if every read returns the most recently written value of the same data item, thus obeying program order. But as far as multiprocessor machines are concerned, this definition is too simplistic. In fact, it comprises two different aspects of memory system behaviour, namely *what* values can be returned by a read and *when* a written value will be returned by a read. The first aspect is called *cache coherence*, the second *memory consistency*.

Before we give more precise definitions of the two terms, let us remember a few concepts from uniprocessors that are also important in this context. A *memory operation* is either a single *read* (or *load*) or a single *write* (or *store*) of a processor accessing memory (we assume uniprocessor memory operations to be atomic). A memory operation is *initiated* if the processor has sent the request and can no further influence its execution or termination. An initiated memory operation is *issued* if it is presented to a part of the memory hierarchy (e.g. cache, write buffer or main memory); it is only through issuing memory operations that a processor may observe the state of the memory system. For this reason, we consider a memory operation to be *performed* if a processor observes it as being completed. A write operation by P_{i} (i.e. processor i) is called being *performed with respect to* P_{k} if a following read to the same address by P_{k} returns the value defined by the write (or any subsequent write). A read operation by P_{i} is called being *performed with respect to* P_{k} if subsequent write operations to the same address by P_{k} cannot influence the value returned by the read.

Cache Coherence

Memory operations in uniprocessor systems are unequivocally ordered because of program order, so the notion of "most recently" has a well defined meaning. In multiprocessor systems we encounter the difficulty of having not only one program order, but many of them, and all these program orders interact with the memory system through their memory accesses. Moreover, a memory operation may be in one of a few intermediate states before being completely performed. If the memory system does not consist of a memory hierarchy then each memory operation will be performed with respect to all processors at the moment it accesses main memory, i.e. main memory would impose a serial order on all accesses to the same physical address. Within this serial order, all memory operations must be seen in the order determined by the program order of each individual process.

Transferring this concept to a multiprocessor machine results in an interleaving of the memory operations from the various processes where individual program orders are preserved. Obviously, there is no sense in making any assumptions about the interleaving of independent operations from different processes; every interleaving that preserves each of the individual program orders is called a *legal interleaving*. Using this intuitive definition, the notion of the "most recent operation" describes the immediate predecessor in this hypothetical ordering of all memory operations of the system. One additional important aspect is that all write operations to the same memory address must be observable in the same ordering by all processes.

Collecting all aspects described informally so far leads us to the following definition of a coherent system:

A multiprocessor system is *coherent* if the result of any program execution is such that for each memory address a hypothetical serial sequence of all memory operations accessing this single memory address can be constructed which is consistent with the program execution and for which the following holds:

1. In this sequence, the operations of any individual process appear in the same ordering in which the process has issued them.
2. For all read operations, the value returned by such an operation is the one written by the previous write operation to this memory address.

Implicit in this definition is the property of *write serialization* which enforces that all processes see write operations to the same memory location in the same order. A multiprocessor system not satisfying this minimal requirement would render it difficult to make precise statements about e.g. the correctness of multiprocessor programs. (This property is explicitly discussed in [9] where it is called *cache consistency*.)

The *result* of a program execution are the values of all read operations. From this result, we cannot deduce the effective sequence of operations in the system, but we will see in the context of consistency models that it is only the observable sequence of memory operations that matters.

A memory operation in multiprocessor systems is said to be *complete* if it is performed with respect to all processors.

Memory Consistency

We need coherence in program situations where information has to be transfered between processes through main memory, i.e. if several processes read the value written by some other process. A coherent system guarantees that eventually a value written is made visible to all processes in the system. But this guaranty does not extend to the point of time when this happens. One of the problems arising from this delay is illustrated in Fig. 1.

Processor 1	**Processor 2**
`WriteBuffer();`	
`flag = 1;`	`wait until (flag==1);`
	`ReadBuffer();`

Fig. 1. Coherence vs. consistency

In this example the data item `flag` (assumed to be set to zero initially) is used to prevent processor 2 from reading elements from an empty buffer. More precisely, we even expect the filling of the buffer to be completed before the change on `flag` will become visible for processor 2. The definition of coherence only guarantees that the new value of `flag` will at some time be observable by processor 2, but possibly before the change in the buffer is visible.

Clearly, we need something more than coherence to give a multiprocessor system with a shared address space a precise semantics. This semantics defines a model for the ordering on operations that can be used by programmers to reason about possible execution results and the correctness of programs. A *memory consistency model* therefore specifies restrictions on legal interleavings of memory operations in a multiprocessor system. These restrictions apply to accesses of one or several processors to the same memory location as well as to different memory locations. Hence, consistency subsumes coherence.

In the following, we describe three different memory consistency models which are the basis for the performance analysis in the main part of this paper.

The first memory consistency model is the straightforward transfer of the sequential semantics of memory operations in a uniprocessor. This yields the intuitive model formalized as *sequential consistency* (SC) and defined as follows in [12]: [A multiprocessor is sequentially consistent if]

> the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Implementing SC requires that the system preserves the two constraints described in this definition. First, the program order of each individual process must be obeyed in the sequential order. Second, the sequential order observed must be the same to all processes, i.e. memory operations of each process must execute atomically. Whereas coherence says that write operations to the same memory location must be visible to all processes in the same order, SC requires that writes to all memory locations must be seen in the same order by all processes.

With the definition of SC it becomes obvious that the ordering of memory operations of one process relative to subsequent reads and writes is relevant for the semantics of consistency models. In weaker consistency models, one or more of the four orderings between reads and writes are relaxed. As we still want to obtain a coherent system, these relaxations will only apply as long as pairs of data accesses to different memory locations are concerned.

We use two of these possible relaxations in this paper, the most meaningful ones. The first relaxation allows reads to be performed before a preceding write operation is completed. This model is called *Total Store Ordering* (TSO), it originates in the specification of the SUN Sparc processor[1]. This specification includes a mechanism called *read own write early* which insures program order for the issuing processor. An example illustrating this model is given in Fig. 2 on the left.

In this example, assuming $\mathtt{A} = \mathtt{B} = 0$ initially, under SC the result $(\mathtt{A}, \mathtt{B}) = (0, 0)$ is impossible. But using the consistency model TSO, $(\mathtt{A}, \mathtt{B}) = (0, 0)$ might be observed because performing the read operations before the writes have been completed is no longer prohibited.

The second relaxed model additionally allows writes to be performed before a preceding write operation has been completed. This model is called *Partial Store Ordering* (PSO), it also originates in the specification of the SUN Sparc processor and also includes *read own write early.* An example illustrating this model is given in Fig. 2 on the right.

[1] Providing a correct specification for TSO is a non-trivial task. The instruction set of the Sparc processor does not exactly match the semantics of TSO (L. Higham, personal communication).

Proc. 1	**Proc. 2**
`A = 1;`	`B = 1;`
`read B;`	`read A;`

(a) TSO

Proc. 1	**Proc. 2**
`A = 1;`	`wait until (flag==1);`
`B = 1;`	`u = A;`
`flag = 1;`	`v = B;`

(b) PSO

Fig. 2. Examples for TSO and PSO

Again assuming all variables to be set to zero initially, under PSO any result with $\mathtt{u}, \mathtt{v} \in \{0, 1\}$ is possible because the write to `flag` (i.e. `flag = 1`) can either pass by the write to `A` (`A = 1`) or the write to `B` or both of them and eventually `B = 1` happens before `A = 1`. Under the restrictions of SC and TSO the result $(\mathtt{u}, \mathtt{v}) = (1, 1)$ is the only possible one because write operations must strongly obey program order in the interleaving of all write operations.

Through the possible rearrangement of observations of memory operations of a single processor, both TSO and PSO permit more execution orderings than SC which leads to a larger number of legal interleavings of parallel programs. In fact, the interleavings that are observed by the processors may even differ between processors. This results in faster program executions.

Further relaxations of orderings among memory operations are the consistency models *weak ordering* [4] and *release consistency* [7] and variants of them. These very far going relaxations will not be considered here. A tutorial on shared-memory consistency models can be found in [1].

3 Architectural Models and Implementation

As a basis for our abstraction of a multiprocessor system, we take a general multiprocessor machine which is sufficient to model all relevant aspects of the architecture. This machine model consists of a number of processors, each equipped with a private cache. Through a network (or crossbar) the processors are connected to the shared global memory which possibly is divided into different memory modules. Our basic architectural model is shown in Fig. 3. The processors perform blocking reads. A two-level memory hierarchy is used which consists of a shared global memory and write back caches. In a system with write back caches (as opposed to write through caches), on a write request data items are changed only in the cache. The modified data items are copied back to the global memory when they are replaced or on demand from other processors. We abstract from the details of the interconnecting network, they will be adequately modelled by the parameters for the delay of data accesses.

The next section will give evidence that for implementing the consistency models introduced in the previous section different variants of this general hardware architecture are needed. Accordingly, the first variant extends the cache with a write buffer, this is shown in Fig. 4. This write buffer is advantageous in situations where the consistency model allows overlapping of memory operations

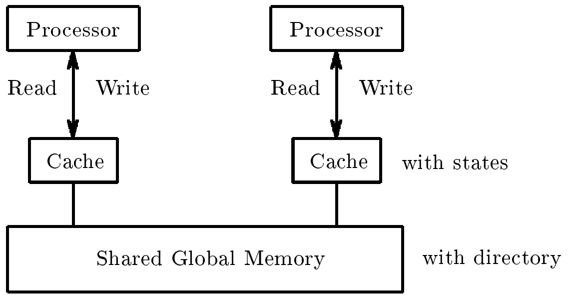
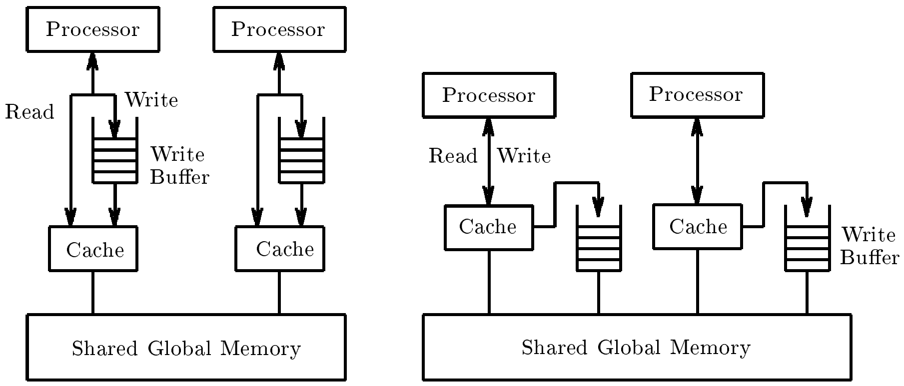


Fig. 3. Basic model



(a) serial

(b) parallel

Fig. 4. Model with write buffers

or where the processor might be engaged in internal computation while a write operation is being performed. As far as hardware implementation is concerned, the buffer can either be inserted between processor and cache or between cache and the network as long as the buffer is only used privately by the processor the cache belongs to.

In the second variant, the caches are substituted by so called non-blocking caches (Fig. 5). At the moment of a cache miss, in a non-blocking cache the status of the memory operation causing the miss is entered into a register and the cache is free to accept further requests from the processor while the outstanding operation is serviced by the MSHR. The register storing the status of the outstanding operation is called MSHR (miss status holding register). The concept of non-blocking caches originates in [11], further details of the implementation of this techniques may be found either in this original paper or in modern textbooks as e.g. [10].

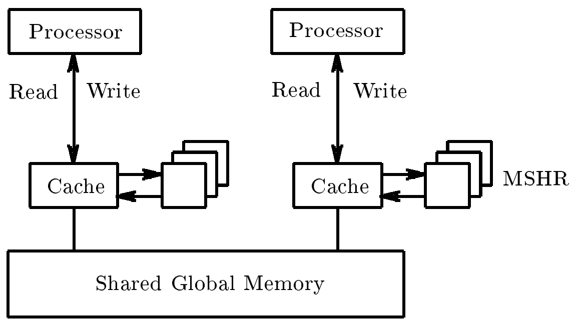


Fig. 5. Model with non-blocking caches (MSHR)

Directory-Based Coherence Protocols

As our goal is to model modern large-scale shared-memory machines, we assume that the system is equipped with a directory that keeps the state of each block of memory that may be cached. This directory supplies the foundation for an efficient implementation of coherence protocols because in large-scale architectures bus-based mechanisms for keeping cache coherence are no longer appropriate. At this point, we will only give a short sketch of a possible directory implementation, but this should suffice to have an impression of the underlying mechanisms.

In typical directory implementations, an entry in the directory is associated with each memory block. At any time during program execution, a memory block may be present in any part of the memory hierarchy. So one part of the directory rests in global memory: We assume a set of `Present`-pointers to be associated with each block in the global memory which point to those caches that own a copy of that block. Additionally for each block a `Modified`-flag announces whether one of the caches is the exclusive owner of that block and has modified it. The other parts of the directory are distributed over the caches: Each block in a cache is marked with a `Valid`- and a `Private`-flag. The `Valid`-flag tells whether the contents of that block in the cache is up-to-date, the `Private`-flag is set if the cache is the exclusive owner of that block. This set of flags and pointers provides enough information to have a precise general view of the system status at every point during program execution. Especially, a correct decision can be made whether an access to the cache is a *hit* or a *miss*. Furthermore, write serialization (important for coherence) can be enforced on the level of memory blocks. For more details on directory implementations, consult [2, 13] or [10].

We differentiate between the two classes of invalidation and write update coherence protocols because these are distinct designs which exhibit strongly different performance behaviour.

Invalidation protocols (INV) ensure that a processor has exclusive access to a data item before it writes to that item. The protocol invalidates all cached copies of the block containing that data item, so that there are no other readable or writable copies of that block in the memory system when the write is performed.

Write update protocols (UPD) update all cached copies of a data item. For this class of protocols the bandwidth requirements for broadcasting blocks on every modification are much higher. Intuitively, these protocols are only advantageous when reading from a previously modified block, this tradeoff will be examined in detail later.

On principle, cache coherence protocols may be combined freely with memory consistency models. In order to get the desired semantics of a consistency model, the coherence protocol has to be implemented adequately together with the available hardware architecture. To reduce and hide the latency occuring when data items are accessed that do not reside close to the processor, modern multiprocessor architectures use buffers for temporarily storing data and pipelines to overlap memory accesses. The main difference of the protocol implementations then is revealed in the design and the usage of those buffers and pipelines. More on this topic will be shown in the next section.

In our examinations in the following section, we measure the delays that occur when memory operations are executed. We use two parameters to keep analyses manageable. The delays are modelled from the viewpoint of the processor. For that we distinguish the time needed to access a data item in the cache (denoted t_{c} in Fig. 6), to access blocks in global memory (denoted t_{m}) or to broadcast information for the protocol (e.g. acknowledgements, also denoted t_{m}). The time units t_{c} and t_{m} both include the time for requesting the data

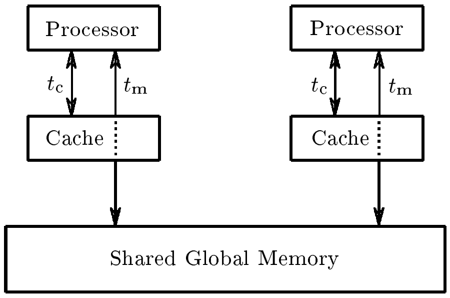


Fig. 6. Time units to access data items in a multiprocessor architecture

and getting it delivered. As accessing blocks in the global memory is always through the cache (and the cache protocol), we shall assume t_{c} to be part of t_{m} in those cases, therefore incurring no additional costs. Of course t_{c} and t_{m} vary with the hardware of the architecture, i.e. with the organization of memory or the topology of the network. Typical values are 1-2 clock cycles for t_{c} and 8-66 clock cycles for t_{m} [10]. By $\tau = t_{\mathrm{m}}/t_{\mathrm{c}}$ we abbreviate the ratio between the two parameters.

4 Efficiency of Data Accesses in SC, TSO and PSO

In this section, we study the effects the various implementations of the consistency models SC, TSO and PSO within the architectural variants may produce with respect to the delay for accessing data items by read or write operations. This way the aspects incurred by the hardware architecture are separated from the aspects incurred by the consistency model itself. Our analysis at this point does not take into account possibilities for increasing performance by hiding latency, i.e. by overlapping memory operations within longer access sequences. These would only be effective when sequences of operations are examined where memory operations may overlap each other or overlap with internal computations of a processor.

Sequential Consistency

Obviously, the consistency model SC does not allow many optimizations with respect to performance because every memory operation has to be completed before the next one can be initiated.

Let us first have a closer look at the basic model in Fig. 3. If a process has issued a memory operation, it must await the servicing of its request until completion. The process is blocked, no difference whether the operation is a read or a write. Therefore, the constraints of SC are automatically met, a memory operation is only issued if the previous one is completed.

The amount of time a processor is blocked depends on the memory operation itself and whether the coherence protocol is invalidate or write update. On a *read hit* (i.e. the requested item is valid in the cache) and on a *write hit* (i.e. the processor is the exclusive owner of that item valid in the cache), both protocol types complete the request within time t_c. Servicing a *read miss* in an invalidate protocol needs either delay t_m (item has not been modified in any copy) or delay $2t_m$ (item has been modified in some copy and global memory has to be updated before the read request can be answered). With a write update protocol, a read miss can always be answered with delay t_m because the global memory holds an up-to-date copy at every time. Servicing a *write miss* in an invalidate protocol needs time $2t_m$ (invalidating other copies, waiting for acknowledgements and reading block with the requested item from global memory). With a write update protocol, a write miss needs delay $3t_m$ because in this case a two phase protocol (see e.g. [8]) must be used with SC. The following table sums up these considerations.

SC for the basic model	Read Hit	Read Miss min.	Read Miss max.	Write Hit	Write Miss
INV protocol	t_c	t_m	$2t_m$	t_c	$2t_m$
UPD protocol	t_c	t_m		t_c	$3t_m$

Let us introduce the following notation. c_R denotes the percentage of read operations among all memory accesses and $c_W = 1 - c_R$ the percentage of writes. c_{RH} denotes the percentage of read hits among all read operations and $c_{RM} =$

$1-c_{\mathrm{RH}}$ the percentage of read misses, analogously c_{WH} and c_{WM} for write hits and misses among all write operations. c_{RU} denotes the percentage of read misses among all read misses that access a modified value.

Proposition 1. *The average delay $T_{\mathrm{SC-INV}}$ for memory accesses under SC with an invalidate protocol is*

$$T_{\mathrm{SC-INV}} = c_{\mathrm{R}}\left(c_{\mathrm{RH}}t_{\mathrm{c}} + c_{\mathrm{RM}}\left[c_{\mathrm{RU}}2t_{\mathrm{m}} + (1-c_{\mathrm{RU}})t_{\mathrm{m}}\right]\right) + c_{\mathrm{W}}\left[c_{\mathrm{WH}}t_{\mathrm{c}} + c_{\mathrm{WM}}2t_{\mathrm{m}}\right] .$$

The difference between $T_{\mathrm{SC-INV}}$ and the average delay $T_{\mathrm{SC-UPD}}$ for memory accesses with a write update protocol is

$$T_{\mathrm{SC-INV}} - T_{\mathrm{SC-UPD}} = t_{\mathrm{m}}\left[c_{\mathrm{R}}c_{\mathrm{RM}}c_{\mathrm{RU}} - (1-c_{\mathrm{R}})c_{\mathrm{WM}}\right] .$$

Proof. As explained in the text above, servicing a write hit under SC with an invalidate protocol needs delay t_{c}, servicing a write miss needs delay $2t_{\mathrm{m}}$. Therefore, the overall amount of time a processor is blocked for memory accesses that are write operations sums up to $c_{\mathrm{W}}\left[c_{\mathrm{WH}}t_{\mathrm{c}} + c_{\mathrm{WM}}2t_{\mathrm{m}}\right]$ (where c_{W}, c_{WH} and c_{WM} denote percentages as introduced above). If the memory access is a read operation, the delay for a hit with an invalidate protocol is t_{c}, whereas a miss is either delayed time t_{m} (in case of unmodified values) or $2t_{\mathrm{m}}$ (in case of modified values). Therefore, the overall amount of time a processor is blocked for read operations sums up to $c_{\mathrm{R}}\left(c_{\mathrm{RH}}t_{\mathrm{c}} + c_{\mathrm{RM}}\left[c_{\mathrm{RU}}2t_{\mathrm{m}} + (1-c_{\mathrm{RU}})t_{\mathrm{m}}\right]\right)$.

Thus, the average delay $T_{\mathrm{SC-INV}}$ for memory accesses under SC with an invalidate protocol is

$$T_{\mathrm{SC-INV}} = c_{\mathrm{R}}\left(c_{\mathrm{RH}}t_{\mathrm{c}} + c_{\mathrm{RM}}\left[c_{\mathrm{RU}}2t_{\mathrm{m}} + (1-c_{\mathrm{RU}})t_{\mathrm{m}}\right]\right) + c_{\mathrm{W}}\left[c_{\mathrm{WH}}t_{\mathrm{c}} + c_{\mathrm{WM}}2t_{\mathrm{m}}\right] .$$

Analogously, the average delay $T_{\mathrm{SC-UPD}}$ for memory accesses under SC with a write update protocol is

$$T_{\mathrm{SC-UPD}} = c_{\mathrm{R}}\left(c_{\mathrm{RH}}t_{\mathrm{c}} + c_{\mathrm{RM}}t_{\mathrm{m}}\right) + c_{\mathrm{W}}\left(c_{\mathrm{WH}}t_{\mathrm{c}} + c_{\mathrm{WM}}3t_{\mathrm{m}}\right) .$$

This yields the following difference between the average delay $T_{\mathrm{SC-INV}}$ and $T_{\mathrm{SC-UPD}}$:

$$T_{\mathrm{SC-INV}} - T_{\mathrm{SC-UPD}} = t_{\mathrm{m}}\left[c_{\mathrm{R}}c_{\mathrm{RM}}c_{\mathrm{RU}} - (1-c_{\mathrm{R}})c_{\mathrm{WM}}\right] .$$

□

As an example, consider the case where $c_{\mathrm{WM}} = c_{\mathrm{RM}}$. Then SC with an invalidate protocol has a smaller average delay if $(1 - c_{\mathrm{R}} - c_{\mathrm{R}}c_{\mathrm{RU}}) > 0$.

If we use the model with write buffers from Fig. 4 for SC, this buffer must obey the FIFO principle. A write operation must not be deleted from the buffer until its completion. A processor issuing a write operation pushes this into the buffer and immediately proceeds with its program, it only has to block if the buffer is full. The disadvantage of this model occurs at read operations: In this case, a processor has to wait until all outstanding writes from the buffer have been completed, only after that it is allowed to issue the read and wait for its completion. The influence on the amount of time needed for the operations is obviously determined by the number of elements in the write buffer, it therefore depends on the specific application.

Implementing SC in an architecture with non-blocking caches does not yield any advantage in performance because no outstanding operations are allowed.

Total Store Ordering

The consistency model TSO allows the completion of write operations to be passed by successive reads, so this model cannot be implemented in the basic model. We will therefore start describing TSO implemented for an architecture with write buffers as in Fig. 4.

For this model, under TSO read operations no longer have to wait for writes from the write buffer to be completed. Reads may pass by these outstanding writes, but special care has to be taken to implement *read own write early*, i.e. if a read request is to an item in the write buffer, this value has to be returned for the read operation. For this, the write buffer not only has to obey FIFO as under SC, it also needs some extra address logic in hardware.

Hence, on a read the processor initiates the operation and waits for its completion, the read may overlap with outstanding writes taking care of *read own write early*. On a write, the processor pushes this write into the buffer and immediately proceeds with its program, it only needs to block if the buffer is full.

If we compare these access delays with SC (implemented with write buffers), we observe benefits when reads follow write operations. One negative side effect of this model may be that the chance of a full write buffer is higher, but as is shown in [6] through simulations, this case occurs rather seldom in practical applications.

Next, we describe how the consistency model TSO works with an architecture with non-blocking caches like in Fig. 5. In order to support outstanding requests, a cache must supply mechanisms for mixing multiple outstanding requests to the same cache line, for recognizing read requests to an outstanding write and for buffering written items until they can be merged with the answers to outstanding requests. For this, (cache) protocols have to be extended and hardware must be equipped with additional registers and buffers (e.g. as explained in [11]).

In an architecture with non-blocking caches, a read is not only a hit if the requested data item resides in the cache, but also if the read operation is to the address of an outstanding write. This may increase the chance of read hits. As far as write requests are concerned under TSO, the use of more than one MSHR must be treated with special care because write requests are not allowed to overlap each other. Either one allows only one MSHR or the MSHRs are organized into a FIFO queue where the following write request must not be started until the previous one has been completed.

As an example for the necessary protocol extensions, we explain how a write miss is handled with non-blocking caches. If there is no free MSHR to store the write request, the processor blocks and waits for an empty MSHR. If there is a free MSHR at the write request then a cache line is reserved for this request. Attention must be payed to cache lines with outstanding request, they must not be evicted. After a successful reservation of a cache line, the new data item is written into the block at the corresponding offset, the MSHR is initiated with the block address and a bit is set that indicates which data item in the block belongs to this request. This bit vector makes mixing of outstanding requests possible if this is allowed under the consistency model. Following this initialization, loading the corresponding block from global memory and obtaining the status for an exclusive write can be started and the processor proceeds with its program execution. The next table represents the delays necessary for data accesses under TSO in this model with MSHRs.

TSO with MSHR	Read Hit	Read Miss		Write Hit	Write Miss	
		min.	max.		min.	max.
INV protocol	t_c	t_m	$2t_m$	t_c	t_c	$2t_m$
UPD protocol	t_c	t_m		t_c	t_c	$3t_m$

In the following, $c_{\text{MA}}^{\text{TSO}}$ denotes the percentage of those write misses among all write misses for which an empty MSHR is available.

Proposition 2. *The difference between $T_{\text{SC-INV}}$ and the average delay $T_{\text{TSO-INV}}$ for memory accesses under TSO with an invalidate protocol is*

$$T_{\text{SC-INV}} - T_{\text{TSO-INV}} = c_{\text{W}} c_{\text{WM}} c_{\text{MA}}^{\text{TSO}} (2t_{\text{m}} - t_{\text{c}})\,.$$

Proof. For the average delay $T_{\text{SC-INV}}$ for memory accesses under SC with an invalidate protocol see proposition 1.

Under TSO, only the delay of write misses differs from SC. As explained above, when an empty MSHR is available at the time of a write miss, servicing a write request only needs time t_{c}. If there is no empty MSHR available servicing the request suffers the usual delay known from SC. Thus, the average delay $T_{\text{TSO-INV}}$ for memory accesses under TSO with an invalidate protocol is

$$\begin{aligned} T_{\text{TSO-INV}} = {} & c_{\text{R}} \left(c_{\text{RH}} t_{\text{c}} + c_{\text{RM}} \left[c_{\text{RU}} 2t_{\text{m}} + (1 - c_{\text{RU}}) t_{\text{m}}\right]\right) \\ & + c_{\text{W}} \left(c_{\text{WH}} t_{\text{c}} + c_{\text{WM}} \left[c_{\text{MA}}^{\text{TSO}} t_{\text{c}} + (1 - c_{\text{MA}}^{\text{TSO}}) 2t_{\text{m}}\right]\right)\,. \end{aligned}$$

This yields the following difference between the average delay $T_{\text{SC-INV}}$ and $T_{\text{TSO-INV}}$:

$$T_{\text{SC-INV}} - T_{\text{TSO-INV}} = c_{\text{W}} c_{\text{WM}} c_{\text{MA}}^{\text{TSO}} (2t_{\text{m}} - t_{\text{c}})\,.$$

□

Partial Store Ordering

Exactly as under TSO, the use of the basic model without write buffers or non-blocking caches does not allow to implement the performance gains permitted by the consistency model PSO. Therefore we start our examination with the architecture with write buffers in Fig. 4.

Again the write buffer must support the mechanism of *read own write early* in order to provide program order in cases where a read refers to a previously written value which still resides in the write buffer. But under PSO, a write need no longer wait for the acknowledgements of previous writes because write operations passing by other writes are allowed, so the write buffer may forward writes to the global memory with a higher rate. Attention must be paid to those cases when a write is to an address with an outstanding write because writes to the same memory location must always obey program order. If the write buffer is to solve this task then mechanisms for combining accesses to the same address or for blocking the buffer have to be provided.

Hence, on a read the processor initiates the read and waits for its completion, potentially passing by outstanding writes. On a write, the processor pushes the write operation into the buffer, it only has to block if the buffer is full. The write buffer does not have to await acknowledgements of previous write operations.

The delays for isolated data accesses do not change compared with TSO. A performance gain may be achieved by the faster forwarding of writes from the buffer to global memory. As a side effect the chance that the write buffer runs full is reduced.

Using the model with non-blocking caches from Fig. 5 for PSO, it is possible to employ a number of independent MSHRs because writes are allowed to pass by outstanding writes. The MSHRs again must supply a data structure that contains the addresses of memory blocks and a bit vector for combining writes to different parts of a cache line. The delays for single data accesses are the same as for TSO, but the number of misses which exhibit a long latency – as seen from the processor's view – is reduced.

On a read the processor initiates the read and waits for its completion, potentially passing by outstanding writes. On a write, the processor passes the write operation to the cache, it only has to block if there is no empty MSHR available. The delays for accessing data items are the same as for TSO (see table on page 13).

As above, $c_{\mathrm{MA}}^{\mathrm{PSO}}$ denotes the percentage of those write misses among all write misses for which an empty MSHR is available.

Proposition 3. *The difference between $T_{\mathrm{TSO-INV}}$ and the average delay $T_{\mathrm{PSO-INV}}$ for memory accesses under PSO with an invalidate protocol is*

$$T_{\mathrm{TSO-INV}} - T_{\mathrm{PSO-INV}} = c_{\mathrm{W}} c_{\mathrm{WM}} (c_{\mathrm{MA}}^{\mathrm{PSO}} - c_{\mathrm{MA}}^{\mathrm{TSO}})(2t_{\mathrm{m}} - t_{\mathrm{c}})\,.$$

Proof. For the average delay $T_{\mathrm{TSO-INV}}$ for memory accesses under TSO with an invalidate protocol see proposition 2.

Under PSO, the delays for single data accesses are the same as under TSO. The only difference between the two models TSO and PSO is that under PSO more than one MSHR may be used independently. Therefore, the percentage $c_{\mathrm{MA}}^{\mathrm{PSO}}$ of those write misses for which an empty MSHR is available might be greater than $c_{\mathrm{MA}}^{\mathrm{TSO}}$.

This yields the following difference between the average delay $T_{\mathrm{TSO-INV}}$ and $T_{\mathrm{PSO-INV}}$:

$$T_{\mathrm{TSO-INV}} - T_{\mathrm{PSO-INV}} = c_{\mathrm{W}} c_{\mathrm{WM}} (c_{\mathrm{MA}}^{\mathrm{PSO}} - c_{\mathrm{MA}}^{\mathrm{TSO}})(2t_{\mathrm{m}} - t_{\mathrm{c}})\,. \qquad \square$$

5 Bursting Writes and General Access Sequences

Taking a closer look at the results of the previous section for the delays of isolated data accesses, we notice that this analysis does not exhibit the full power of the relaxed consistency models. Obviously this is because the advantages of the relaxations are only visible in measuring performance if we consider overlapping operations. In this section we identify examples where relaxed models clearly outperform SC in the architectural model with non-blocking caches and an invalidate-based cache coherence protocol.

Consider the following scenario: A processor writes values to consecutive elements of a field A, e.g. write $A(0), A(j), A(2j), A(3j), \ldots$. The offset j between two of these elements is assumed to be greater than the block size of the cache. Hence, after some initial phase each write is a miss because the cache is filled up with previously written elements (one per block) and the corresponding block with the next element has to be loaded from global memory.

Proposition 4. *Let k denote the number of MSHRs and n the length of the sequence of writes, then the total time to execute this sequence is $kt_{\mathrm{c}} + (n-k)2t_{\mathrm{m}}$. After the k MSHRs are filled up, the processor will be blocked under SC and TSO.*

The following result shows that PSO has the ability to tolerate bursting writes.

Theorem 1. *For the consistency model PSO the following holds: If the architecture (with invalidate-based cache coherence protocol) supplies at least 2τ MSHRs, then any sequence of write operations does not block a processor. Using the notation above, this means $c_{\mathrm{MA}}^{\mathrm{PSO}} = 1$.*

Proof. Assume $2\tau+1 = (2t_{\mathrm{m}}/t_{\mathrm{c}})+1$ to be the length of the sequence of successive write operations and 2τ to be the number of MSHRs provided by the hardware.

Look at some arbitrary moment T while processing the operations in the sequence. All write operations that have been started before time $T - 2t_{\mathrm{m}}$ are completed at this moment (a write miss needs delay $2t_{\mathrm{m}}$ under PSO).

At time $T - 2t_{\mathrm{m}}, T - 2t_{\mathrm{m}} + t_{\mathrm{c}}, \ldots, T - t_{\mathrm{c}}$ a number of 2τ write operations are started, these occupy 2τ MSHRs (within a time interval of t_{c}, MSHRs can be initialized). Hence, at the moment T under consideration, the write operation started in point $T - 2t_{\mathrm{m}}$ is completed, the MSHR is released and can be used for the next write operation – numbered $2\tau + 1$. At the moment $T + t_{\mathrm{c}}$, the next MSHR gets released and is ready to receive the write operation numbered $2\tau + 2$, etc. For each following write a MSHR will be available. □

Corollary 1. *For PSO with 2τ MSHRs, write misses do not suffer any delay even in the worst case. It does not make sense to increase the number of MSHRs further. PSO outperforms SC and TSO up to a factor of 2τ.*

Whereas the previous results show how much performance can be gained for a special sequence of accesses, namely a large number of consecutive writes, the next theorem will give us insight into the advantage of TSO and PSO for a more general class of access sequences.

Theorem 2. *Let σ be an access sequence consisting of read and write operations. Let $t_{\mathrm{WM}} = n_{\mathrm{WM}} \cdot 2t_{\mathrm{m}}$ denote the time needed for n_{WM} write miss operations in σ and t_{rem} be the time needed to execute all remaining accesses from σ.*

If $t_{\mathrm{rem}} = \alpha t_{\mathrm{WM}}$ holds with $\alpha > 1$, then the advantage of TSO and PSO on σ relative to SC with an invalidate protocol is at most

$$\mathrm{Adv}_{\mathrm{TSO}}(\tau) = \mathrm{Adv}_{\mathrm{PSO}}(\tau) = \frac{1 - 1/2\tau}{1+\alpha} .$$

Proof. Under SC no overlapping of memory operations is possible, therefore the time to execute σ sums up to

$$T_{\mathrm{SC}}(\tau) = (1+\alpha) t_{\mathrm{WM}} .$$

With TSO and PSO, the processor can utilize the delay incurred by a blocked MSHR while waiting for the execution of a write miss to execute some of the remaining operations. To achieve the bound under TSO, the distance between two consecutive write miss operations in σ must be greater or equal to $2t_{\mathrm{m}} - t_{\mathrm{c}}$. Under the assumption that the blocking times are totally exploited for the execution of remaining operations, the time needed to execute σ is

$$\begin{aligned} T_{\mathrm{TSO}}(\tau) = T_{\mathrm{PSO}}(\tau) &= n_{\mathrm{WM}} \cdot t_{\mathrm{c}} + t_{\mathrm{rem}} = n_{\mathrm{WM}} \frac{t_{\mathrm{m}}}{\tau} + \alpha t_{\mathrm{WM}} \\ &= \frac{1}{2\tau} t_{\mathrm{WM}} + \alpha t_{\mathrm{WM}} = (\frac{1}{2\tau} + \alpha) t_{\mathrm{WM}} . \end{aligned}$$

This yields an advantage of TSO and PSO on σ relative to SC of

$$\begin{aligned} \mathrm{Adv}_{\mathrm{TSO}}(\tau) = \mathrm{Adv}_{\mathrm{PSO}}(\tau) &= \frac{T_{\mathrm{SC}}(\sigma) - T_{\mathrm{TSO}}(\tau)}{T_{\mathrm{SC}}(\tau)} \\ &= \frac{(1+\alpha) - (\frac{1}{2\tau} + \alpha)}{1+\alpha} = \frac{1 - 1/2\tau}{1+\alpha} . \end{aligned}$$

□

Using this formula, we may predict the quantitative gain in performance for TSO and PSO for known access sequences as compared to SC.

6 Concluding Remarks

In this paper, we have clearly differentiated between cache coherence and three commonly used memory consistency models. We have worked out how the logical view on the different orderings of memory operations in the consistency models SC, TSO and PSO corresponds to the technical details of an implementation on a shared-memory multiprocessor architecture. We have identified the most important mechanisms, namely buffering and pipelining of data accesses, and we have determined the amount of time needed in the various consistency models to handle isolated read and write operations. The small differences exhibited in these examinations do not explain the performance differences between the consistency models that have been measured in simulations. The reason for this is that performance gains of various consistency models only become apparent if memory operations are overlapped with each other or with internal computations of the processor. This effect is shown by our results in the last section where we have shown how efficiently TSO and PSO may handle general access sequences.

The models presented in this paper may form the basis for further investigations into the different memory hierarchy levels of CC-NUMA architectures. A number of interesting questions arises, i.e. to search for classical applications where relaxed models are of advantage for efficiency, and to verify the results on the advantage of TSO and PSO for applications on real-life parallel shared memory computers. Within this context, it would also be helpful to extend the method proposed in [5] to multiprocessor systems.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29:66–76, December 1996.

[2] Lucien M. Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[3] David E. Culler, Richard M. Karp, David Patterson, Abhijit Sahay, Eunice Santos, Klaus Erik Schauser, Ramesh Subramonian, and Thorsten von Eicken. LogP: A Practical Model of Parallel Computation. *Communications of the Association for Computing Machinery*, 39(11):78–85, November 1996.

[4] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13^{th} Annual International Symposium on Computer Architecture*, pages 434–442, 1986.

[5] Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems.* PhD thesis, Universität des Saarlandes, September 1997.

[6] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Proceedings of the 4^{th} International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, 1991.

[7] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17^{th} Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[8] David B. Glasco, Bruce A. Delagi, and Michael J. Flynn. Update-Based Cache Coherence Protocols for Scalable Shared-Memory Multiprocessors. In *Proceedings of the 27^{th} Hawaii International Conference on System Sciences*, volume I, pages 534–545, 1994.

[9] James R. Goodman. Cache Consistency and Sequential Consistency. Technical Report 61, IEEE Scalable Coherence Interface Working Group, March 1989.

[10] John L. Hennessy and David A. Patterson. *Computer Architecture. A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 2 edition, 1996.

[11] David Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8^{th} Annual International Symposium on Computer Architecture*, pages 81–87, 1981.

[12] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[13] C.K. Tang. Cache system design in the tightly coupled multiprocessor system. In *Proceedings of the National Computer Conference*, pages 749–753, 1976. Also appears in [14].

[14] Milo Tomašević and Veljko Milutinović. *The Cache-Coherence Problem in Shared-Memory Multiprocessors: Hardware Solutions.* IEEE Computer Society Press, Los Alamitos, California, 1993.

[15] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the Association for Computing Machinery*, 33(8):103–111, 1990.