Memory Consistency Models of Modern CPUs

Georgios Georgopoulos
Embedded Systems Seminar
georgopu@rhrk.uni-kl.de
Contents

1 Introduction ........................................ 1

2 Related Work ......................................... 2

3 Memory Consistency .................................... 3
   3.1 Memory Consistency Models .......................... 4
   3.2 Weak Consistency ................................... 5
   3.3 Release ............................................. 6
   3.4 Memory Barriers ..................................... 7

4 Processors with Relaxed Memory Models ............... 8
   4.1 IBM 370 ............................................. 8
   4.2 SPARC V8 Total Store Ordering (TSO) ............... 9
   4.3 x86 TSO ........................................... 10
   4.4 SPARC V8 Partial Store Ordering (PSO) .............. 11
   4.5 ALPHA ............................................ 12
   4.6 SPARC V9 Relaxed Memory Order (RMO) .............. 12
   4.7 IBM PowerPc ....................................... 13
   4.8 Intel Architecture 64 ................................ 14

5 Comparisons ........................................... 14

6 Conclusion ............................................ 15
Abstract

In modern parallel systems that are used nowadays almost in every computing area, the basic problem lies in the distribution of system resources and memory. System processors share the common memory and try to retrieve it, in order to read and write memory cells.

Programs that are running parallel in the processors should give at the end of processing the desired outcome as it is expected by the programmer and at the same time to be efficient and correct. This is achieved throughout a kind of a deal between the processors, the compilers and the memory and it is called memory consistency model. Memory consistency model follows specific rules concerning the program order, in such a way for the processor and the compiler to be allowed or not many optimizations.

The most intuitive memory consistency model is called sequential and limits down the multiprocessor environment benefit, resulting in a low performance optimization since it requires the sequential execution of memory instructions. Multiprocessor systems introduced memory models, capable of utilizing processor and compiler ability to reorder the memory instructions, the well known relaxed memory models which have the ability to allow the out of order program execution. Specifically, based on the limitations of each model, reorder are allowed between memory instructions and at the same time consistent memory view is achieved at the end of the program. The main problem of these systems is portability, since many memory models can not be executed in every processor and the most processor manufacturers follow their own memory models.

Moreover, in order to achieve an expected view in memory and even more when there is a need for maintaining the program order and of course disallow memory reordering between the program instructions, memory consistency models provide fence instructions which are placed between the desired instructions and force the specific instructions in an order execution. Thanks to these memory fences or memory barriers, memory models can be transformed and adapted from other processors and run efficiently in them. It can also achieve a sequential view of memory by just placing memory fences between every pair of memory instructions and not allow reordering between them at all.

Mainly, in this report the relaxed memory models will be presented, which are used by modern multiprocessor systems and share the same main memory. First, memory models will be introduced in general and their memory barriers. Afterwards, the benefits and optimizations that they provide, due to the reordering of the instructions. Furthermore, relaxed memory models will be presented in detail and the allowable memory reordering and their memory fences will be defined.

1 Introduction

The major part of this work deals with the relaxed memory models, which are provided by each manufacturer as well as the necessary methods in order to present a global consistency view to the programmer.

An overview of the memory consistency, of all models, which will be further presented as well as an overview of the necessity of memory barriers is given in the third chapter. In that way, with a particular focus on memory consistency, its importance in modern multi-processor systems will be explained and will be also introduced a variety of the different models. Beginning with the sequential consistency and continuing with more relaxed memory models, the advantages and disadvantages of the different models
will be presented. Finally, focusing on memory barriers their functionality will be explained, how they can change these particular models and concluding how they are adopted by different manufacturers.

The most important part of study will be presented in chapter 4, regarding the shared memory consistency models, which are used in modern multi-processors. Afterwards, we will see a categorization of these models based in common characteristics that they provide, referring to their memory instructions reordering. Based on this categorization we will first consider the most strict relaxed models and continue with the next models, which permit more relaxations. Finally we consider the most relaxed memory models, which allow the reordering of memory instructions of all instruction pairs.

In the last section we will see some useful comparisons of the different memory consistency models explained and explore possible optimizations which can be achieved. Also we will see that with some modifications these memory models could provide portability. Some advantages and disadvantages between the different modes will be explained. Also, due to memory fences it will be shown that there are models which may implement the same program with the same results. Obviously, the models which could execute the same program resulting to a consistent view in memory, except from the memory barriers, should also adapt a consistency model not so different from the model that already runs the specific program.

2 Related Work

There is a wide variety of papers dealing with memory consistency models which are used in modern multiprocessors.

The most relevant paper which is used for this document writing is published by Kourosh Gharachorloo[3] and explains briefly the memory consistency models as well as their adoption by modern shared-memory multiprocessor systems. The specific paper introduces the memory consistency models and why they are necessary to modern architectures. Specifying firstly, the sequential consistency as an extension from uniprocessor instruction execution to multi-processor systems defines the relaxed memory consistency models and their barriers intending to give a view from the strictest to the most relaxed memory consistency model and how the compiler succeeds through these memory instruction reordering to optimize the performance to the modern architectures.

A document published also by Kourosh Gharachorloo, Shared Memory Consistency Models: A Tutorial[9], categorizes the relaxed models to its common instruction reordering and explains models with or without caches and their functionality.

Paul E. McKenney published a document[6], Memory Ordering in Modern Microprocessors, which was one of the major document explaining the memory barriers and their usage in modern processors. It is briefly explained why it is required from the modern processors to reorder their memory operations and it is also explained in detail the barriers that Linux provides. Furthermore, a wide variety of processors, their relaxations and their barriers, are introduced. It is also explained in detail the memory reordering that each processor permits.

Paul E. McKenney’s document[7], Memory Barriers: a Hardware View for Software Hackers, instead of the above information which is included in the previous document, it is also introduced the cache coherence, important for the models which use cache coherence in addition to the memory consistency. It
is specially referring to MESI model and its functionality and it is briefly explaining with many examples, how the MESI model is used to give a coherent view of the many processor caches which are retrieving the same main memory.

In the document published by Scott Owens Susmit Sarkar Peter Sewell[10], A Better x86 Memory Model: x86-TSO, it is explained the model that x86 uses, the TSO model and how this model provides to compiler the ability to reorder the memory instructions and the optimizations which are arising due to the memory reordering. Moreover, the memory barriers of this model are described, providing to the programmer a useful tool for program writing.

In Fences in Weak Memory Models (Extended Version)[4] published by Jade Alglave, Luc Maranget, Susmit Sarkar and Peter Sewell, the document deals with the memory barriers quoting examples of weak memory models and their barrier.

3 Memory Consistency

Memory reordering by compilers or processors renders the existence of memory consistency models necessary in modern computer systems. Compilers, as the memory accesses are still expensive, are trying to reorder the memory instructions and to execute them out of order with a purpose of optimizing the performance of the system. So, with the many shared memory multiprocessor systems, there is a need for implementing a memory consistency model in order to secure a consistent view in main memory. In that way, memory consistency models are the solution for an execution of a program by a multiprocessor system that accesses the memory and how these writes and reads to memory, affect the view which a programmer expects to be placed in memory.[3]

Figure 1 shows the architecture of a such a system, a shared memory multiprocessor system, in which each processor keeps a copy of the memory in its cache as well as its connection to the main memory and the inputs-outputs.

![Figure 1: Shared-Memory Multiprocessor Architecture][8]
3.1 Memory Consistency Models

In uniprocessor systems, the program order is followed by the processor ensuring that the next memory instruction of the program has access to the last write in order to read it or to modify it. In multi-processor systems, it is more complicated because of the many processors which are involved in the execution of a program. This may happen because the processors may keep a copy of the main memory in their caches and even more different processors of the system are willing to modify a memory cell and ask for access to the main memory to store their results. In this case, a conflict may arise due to the many modified copies of the memory held by the processors and trying to store them in memory. Due to this undesirable situation, strict rules must be specified in order to ensure a consistent view in the main memory.

Nevertheless, an extension to the uniprocessor memory model, which follows the program order and executes the memory instructions in order could be adapted to the needs of the multi-processor systems. This extended model succeeds to present a global serialization of the instruction of all in the system involved processors, resulting in the same memory view independent from the sequence of the instructions which have to be executed.[3]

\[
\begin{align*}
& P1 & & P2 \\
& a1: A=1 & & a2: B=1 \\
& b1: x=A & & b2: u=B
\end{align*}
\]

Figure 2: Sequential Consistency Execution

Figure 2 shows an example of sequential consistency model in which the only possible outcome is the \((x,u)=(1,1)\). Obviously, it is the case because it doesn’t matter which processor executes its instructions first but the reads which follow the write operations will see definitely the above write results.

As already mentioned, the sequential memory consistency model is an adaption from the uniprocessor to multi-processor systems that obeys in two laws as Lamport[5] introduced:

Definition: A multiprocessor system 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.[5]

According to the above definition, the necessary conditions in order to categorize a model as a sequential consistency memory model, are as follows:

1. Each processor instruction which will be processed by its processor has to follow the program order, i.e. to execute in order without permitting any re-orderings between the instructions[5] and
2. a sequential serialization, including all processor’s instructions, must be defined and the result in memory should be the same independent from the instruction sequence which will be followed. [5]

The sequential consistency model is very helpful because it provides a consistent view in memory, since it follows each processor’s program order. Although, the most modern processors do not follow the sequential consistency model, mainly because of the restricted performance optimization, which could be achieved by the compiler due to the reordering of the instruction pairs. Because of this restriction many consistency models have been developed and are characterized as relaxed memory consistency models and they do not obey to these restrictions of sequential consistency, i.e. the memory instructions reordering of a program resulting the out of order execution, taking advantage of compiler optimizations in order to increase the performance of the multi-processor system.

A wide range of relaxed memory models has been developed, specially adjusted to each processor architecture, which uses them and they are distinguished from each other regarding the relaxation they provide and reorder the instructions. For instance, there are four types of reordering the instruction pairs that may arise in relaxation models:

1. a read followed by a read memory instruction

2. a read followed by a write memory instruction

3. a write followed by a read memory instruction

4. a write followed by a write memory instruction

Due to these instruction’s reordering, safety nets, also known as memory barriers, have been introduced in order to avoid the reordering between a pair of instructions and follow the sequential execution of the program order. In the chapter 3.4 memory barriers will be briefly explained and in the next chapter memory barriers will be connected to the memory models that introduce them.

Figure 3 shows a list of processors which they adapt different relaxed memory consistency models and they will be explained briefly in the next chapter presenting the relaxations they permit. Y depicts the relaxation in each model when it permits the particular instruction reordering as well as the earlier read of other processor’s writes.[6]

3.2 Weak Consistency

Weak consistency model uses synchronization operations in order to ensure the program order between the memory instructions. Thus, the ability to reorder the memory instructions between the synchronization operations is provided to the compiler, which optimizes the performance and hides the write latency. Weak ordering demands the preceding instructions to be completed before the execution of a synchronization operation and allows the reordering of the operations that are placed between these syn-
chronization instructions, considering also the particular model that follows the weak ordering and its restrictions. Moreover, the accesses to synchronization instructions are sequentially consistent.[3]

In the following two examples of figure 4, 4.a shows that it is possible under weak consistency because of the two reads of P2 and P3 will read the writes of P1 since there is not a synchronization operation in P2 and P3. Respectively, example 4.b is not weak consistent because there is a synchronization operation and the read tries to access an older value of the first processor. Thus, it should read the value b and not the value a.

### 3.3 Release

Release consistency is similar to weak consistency but it is even more relaxed memory model than the previous one. Instead of a synchronization barrier, Release consistency provides two synchronization barriers namely the Acquire and Release operations. Acquire acts like a lock operation and gains access in a shared data segment while Release operation acts like an unlock which leaves the critical section and continues with executing the next to the Release operations.

The restrictions of the Release model refer to the operations following the Acquire operation that wait for its completeness and analogously the execution of these instructions have to be completed before the execution of the Release operation. The difference to the weak memory model is up to the preceding to the Acquire instructions and these which follow the Release instruction. Acquire does not have to wait for the preceding instructions to complete its execution and also the instructions that follow the Release do not have to wait for the Release to be completed in order to start their execution. Nevertheless, if there are instructions where there is the need for waiting the Release operation to execute, must be protected.
with an Acquire operation which has to be placed before them in order to force them to wait for the completion of the Release.[3]

Figure 5 shows an example of release consistency. The writes of processor 1 execute after the acquire operation and before the release while the read of processor 2 reads correct the value b because it is protected from an acquire instruction. Finally, read operation of processor 3 reads correct the value a since it is not protected by an acquire operation.

3.4 Memory Barriers

As already mentioned above, because of the reordering of the memory instructions in a multi-processor system in relaxed memory models in order to increase the performance, it is also possible to result in an undesirable view in the main memory. Due to this undesirable behavior in different processors systems which will be presented below, every processor architecture provides special instructions, known as fence instructions, memory fences or barriers. Their purpose is to prevent the reordering between their above and their following instructions and forces them to sequential execution. So, the previous instructions to a placed in program instruction have to be completed before the following start to execute. In that way, it is ensured the execution of memory instructions in program order, as the programmer expects, and an out of order execution is no more allowable which could cause undesirable executions and a not expected view of the main memory[4].

In relaxed memory consistency models, each processor defines its unique memory barriers. In the next chapter, I will introduce a range of processors and their memory barriers. Memory fences may vary
Figure 6: Memory Barriers Example[2]

according to the different relaxation that they prevent. For instance, read barriers may prevent a read and its following instructions from reordering while other write barriers may not allow a write to reorder with its following operations. In addition to these barriers, there are also barriers which may not allow all reorderings between memory instructions such as read and write and demand from these to be completed before the following instructions start to execute.

In the next chapter, the range of memory barriers will be considered according to the processor consistency model which uses them.

Figure 6 shows how a memory fence reacts by preventing the preceding and the next to the fence instructions from reordering. Also, it shows that memory operations within these fence operations may be reordered according to the restrictions that each memory consistency model defines.

4 Processors with Relaxed Memory Models

In the next chapters will be discussed several commercial memory consistency models which are introduced by different processors. All next models, are relaxed memory models and despite that some of them have small differences, it is not possible to execute the same program in these models without the existence of unexpected memory results. In some models, even if they have different restrictions may the results be the same and there is the possibility of running a program to different memory consistency models.

These implementations by each memory model require the knowledge by the programmer to specific models in order to present a program with the desired results. Next, the memory models will be presented, beginning from the most strict relaxed model and moving on with more relaxed models, their optimizations and their restrictions defined by each model.

4.1 IBM 370

The IBM 370 processor memory consistency model is the first one we consider as a relaxed memory consistency model. In this particular model, we first consider the reordering between a pair of memory
instructions and specifically a reordering between a write followed by a read. All other re-orderings follow the sequential consistency model as well as a write operation which is followed by a read in the same memory location[3].

As mentioned before, IBM 370 reorders the write with the followed read operation when they refer to a different memory location and not to the same but this is not the only restriction. The model of IBM provides to the programmer specific fence instructions which cause sequential execution between the instructions even in the case a write followed by a read as has been mentioned above and restrict this relaxation when this is what the programmer wishes.

Fence instructions should be placed between the desired instructions in order to enforce the memory instructions to execute in order. These instructions, called serialization instructions in IBM 370 model i.e. compare and swap or special branch instructions. They are placed between the write instruction and the followed read instruction and they demand the execution of the instructions before them. The instructions that follow the fence instruction can only be executed if the previous to this fence instruction are already finished[9].

The relaxation that IBM 370 model provides may increase the performance in comparison to the sequential consistency but still is not enough in order to provide the compiler and the processor the ability for more reorderings.

4.2 SPARC V8 Total Store Ordering (TSO)

Considering the next model, the SPARC V8 Total Store Ordering presented by Sun Microsystems, is a kind of improvement to the previous IBM 370 model. SPARC V8 processor approaches the relaxation of a write instruction followed by a read, using the store buffers which provides the SPARC V8 model. In that way, store instructions may be buffered in a buffer and remain there until they will be stored to the memory. The read operation which follows the write in the program order may here execute first and search to the buffer for the last write. This procedure is allowed even if the write is still not globally visible to other processors[3].

All the other memory operations, as in IBM 370 model also, follow the program order as Figure 3 shows. SPARC V8 allows a read instruction to search the buffer in order to find the last write but there is a restriction to other processor’s writes. It is not allowed by a read to return a write instruction which is not included in the buffer of its processor instructions. Other processor’s writes must be first global visible before a read from another processor returns their value.

SPARC V8 memory model in order to protect a reordering within the write operation and its following read, read-modify-write instructions must be implemented by replacing the write or the read instruction by these read-modify-write instructions. In that way, the model avoids the out of order execution between the above described instructions and provides a sequential view of execution[3].

In Figure 7 is presented an example which runs in two processors in order to understand the differences between sequential consistency, IBM and SPARC V8 TSO consistency model. Below, it is briefly explained the execution order and the reordering of the instructions that may be applied by each memory consistency model.

Example 7.a shows a program execution in two processors P1 and P2. The first processor writes the
value 1 to A and next reads the value of the other’s processor write. Respectively, the same procedure does the processor P2 by writing the value 1 to B. Under sequential consistency the outcome \((x,u) = (0,0)\) is not possible while this same outcome is possible under IBM 370 and TSO model of SPARC architectures because of reordering within a write which is followed by a read and they access different memory locations. Example 7.b is similar to the previous example with the addition of a read to the same memory location after a write operation to each processor. A possible outcome \((x,u,v,w) = (2,2,0,0)\) is not allowed by sequential consistency and IBM 370 models but it is possible under TSO model. This is achieved because of the reordering that may be happened within a write instruction and a following read that access the same memory location. Reads can access the store buffer in order to find the last write and at the end of the program to present the right value.

4.3 x86 TSO

Memory model introduced by x86-TSO is similar to the previous SPARC V8 TSO model providing the same relaxations. As in SPARC’s model it is not allowed memory reordering when a write followed by a write as well as a read which is followed by a read or a write. The relaxation that is allowed as in SPARC V8 is a reordering between a write followed by a read in case that they are accessing or not the same memory location. A difference to the SPARC’s model refers to memory barriers and even more to the lock instructions provided by x86[10].

Each processor that is willing to access a critical code section, has to check first if another processor has locked this section and if it is not the case, then it can lock the particular section and gain exclusive access to the memory. Meanwhile, no other processor can execute memory instructions. On the other hand, if one processor has locked the critical section and another is willing to lock the section, then it must wait until the processor unlocks the critical section in order to lock the section and continue its execute.

As for the fence instructions provided by x86 model can be categorized as below:

1. Ifence instructions, where loads cannot be reordered

2. sfence instuctions, where stores cannot be reordered
3. mfence instructions, where neither writes nor reads can be reordered

Thus, by using these fence instructions it is possible even a sequential execution of the program by placing them between each pair of operations.

![x86-TSO Operational Model](image)

Figure 8: x86-TSO Operational Model[1]

Figure 8 shows an operational view of x86-TSO model which also shows, besides from each processor store buffers, the lock operations which each processor may use for gaining access to a critical code section.

### 4.4 SPARC V8 Partial Store Ordering (PSO)

The next memory consistency model, presented by SPARC, is the second model of the V8 architecture and improves even more than the previous TSO model. In addition to the reordering of a write followed by a read operation, it also permits another optimization which provides the processor and the compiler with a great advantage to reorder a write instruction which is followed by another write. The restriction in order to reorder these instructions is up to the location of the memory that the two writes refer to. It allows a write instruction to be reordered with it’s following write in case that the two writes are trying to access a different memory location. In case of trying to access the same memory location, the reordering between them is disallowed and the two writes follow the program order. The same is followed by all other pairs of memory instructions and they are executed in program order as in sequential consistency model. Of course, as in previous models, fence instructions are provided in order to prevent the reordering of the above mentioned instructions[3].

The fence instruction provided by PSO consistency model called STBAR and it is introduced to prevent the reordering of two following writes. It enforces the in order execution between the writes and it should be placed between the writes in order to force the previous to the fence instruction write to complete its
execution before the fence instruction. In the same way, the following write should start its execution after the fence instruction and of course, if the previous write has been fully executed. So, the PSO model under these fence instructions can be transformed to a TSO model[6].

4.5 ALPHA

The memory model of Alpha processor is the first one that we consider and allows all the combinations of read and write memory instructions to execute out of order. In case of these instructions refer to the same memory location it is not allowed an out of order execution and the model respects the program order even in the case of a read followed by a read. It is also disallowed in this particular model to perform an out of order execution, when a fence instruction is placed between a write and a read instruction of a program and it enforces the instructions to follow the program order execution in such a way that reordering of memory operations is not allowed[3].

The fence instructions that the alpha model uses distinguish analogously between all operations and the write operations. In case the program has to execute the write operations in program order a fence instruction, called write memory barrier WMB, must be placed between the write operations, which must be executed in program order. Moreover, there is another fence instruction provided by Alpha and can enforce all operations to follow the program order between all read and write operations. It is obvious, that by placing these fence instructions between each pair of the memory instructions the current model can easily obey the sequential consistency rules.

In the next models, in addition to the relaxations that alpha memory model provides, the read to read reordering restriction is also permitted and a read followed by a read in the same location is allowed to execute out of order.

4.6 SPARC V9 Relaxed Memory Order (RMO)

An extended version of the TSO and PSO models previously discussed is introduced by the Sun and used in Spark’s V9 processor. This model extends the PSO model and moreover permits the out of order execution of a read followed by a read and a write followed by a read memory operation. On the other hand, the operations of a read followed by a write as well as a write followed by a write that refer to the same memory location execute in program order. So, this model relaxes even more the reordering between the memory operations and provides the processor as well as the compiler a big advantage, by this reordering to optimize mainly the performance.

As in the previous models presented above, if there are fence instructions between the memory operations, it also disallows the reordering of memory instructions and constrains the in order execution between the previous from the fence instruction and the instructions that follow the fence instruction. In the current model of the relaxed memory ordering, it is provided a single fence instruction which with a specified combination in a four-bit opcode is able to prevent the instructions from executing out of order in all combinations[3]. For instance, a specific combination disallows the out of order execution between a read followed by a write and another one disallows a read reorders with a followed read. Two other combinations of this fence instruction does not allow analogously a write followed by a read or a write to execute out of order.
Thereby, there is the first time in this document that is introduced a fence instruction which has to be set in a combination in order to avoid the out of order execution between followed instructions.

4.7 IBM PowerPc

Another relaxed memory model presented by IBM, the PowerPc memory consistency model, relaxes all pair of operations and also allows, in comparison to other relaxed models, the out of order execution of a read followed by a read or a write operation. All other memory operations, such as a read followed by a write and a write followed by a read or a write, can be reordered in case that they refer to different memory locations. An additional advantage that PowerPc model provides is that a processor’s read may have access to other processor’s writes even in the case that these are still not globally visible. Thus, write latency can be hidden due to this optimization and permits to other processor’s reads to access write operations[3].

IBM PowerPC uses barrier operation, known as sync operation, which is similar to mb memory barrier of Alpha model and disallows the reordering of all instructions except from the read to read reordering. In case that a sync operation is placed between two read memory instructions even with the existence of the barrier the read operations may be reordered, as figure 9 shows. The outcome (x,y)=(2,0) could be possible by PowerPC. In order to avoid this reordering, another barrier, the lwsync barrier has been introduced which prevent the read reordering and force them to an in order execution.

\[
\begin{align*}
\text{P3} & : a1: A=2 \\
\text{P4} & : a2: x=A \\
& : b2: \text{sync} \\
& : c2: y=A
\end{align*}
\]

Figure 9: Program Execution in IBM’s PowerPc Model

IBM model is the most relaxed model discussed so far in this document and this is up to the relaxation that provides, allowing reads access other processor’s writes earlier, optimizing the performance while the compiler has the chance to optimize the code by reordering the memory instructions.

Figure 9 shows an example where an outcome (x,y)=(2,0) is possible under PowerPc even with the fence instruction within the two reads. As already mentioned above, it is not sufficient in IBM’s PowerPc model just to place the sync instruction to prevent a reordering between two reads but a lwsync memory barrier must be placed to force the program to in order execution of the memory instructions.
4.8 Intel Architecture 64

IA64 model introduces the Acquire and Release instructions with the purpose of protecting from undesired instruction reordering. As mentioned above, all relaxations between memory instructions are allowed if and only if Acquire and Release operations are not placed between them.

In this model reordering of instructions is allowed when they are placed between an Acquire and a Release operation with the restriction that they have to complete their execution before the Release operation executes. Respectively, the Acquire operation has to complete its execution before the next to it operations execute. A further relaxation provided by IA64 is that Acquire does not have to wait for the completion of the previous instructions as well as the following from Release instructions do not have to wait for Release to execute before execute themselves. In case that there is the need of waiting for the Release to complete, next instructions must be protected with an Acquire operation.

Furthermore, IA64 it is provided with a further memory barrier, MB barrier, that may be placed between two instructions and demands the completeness of the preceding instructions in order to execute. Thus, it is achieved a correct view of values to the following instructions while all preceding operations to MB fence instruction are completed.

5 Comparisons

As it is already described, different memory consistency models have implemented by several processor architectures, each of them adopts a memory model and optimize it to its needs. The mentioned processor architectures despite their differences could be summarized in three big categories beginning from the most strict memory models to the most relaxed:

1. Relaxing the write operation which is followed by a read

2. Relaxing the write operation which is followed by a read and a write

3. Relaxing all program orders

In the first category, IBM 370, SPARC V8 TSO and x86-TSO models can be grouped because of the write to read relaxation that they provide. IBM’s model reorders the write to read but restricts this reordering in case that both instructions refer to the same memory location while the next two TSO models allow the reordering even in the case of accessing the same memory location. Although, TSO models manage to present the correct value of the last write by using the store buffers.

The second category includes only the SPARC’s V9 PSO model and optimizes the previous category models by allowing the writes to different memory locations to execute out of order. Furthermore, the PSO model could with the addition of the STBAR memory fence to transform in TSO model.

The last category includes all other models which have been described in this document and relaxes all program orders. Alpha, SPARC V9 RMO, IBM’s PowerPC and IA64 are included in this category.
relaxing the program orders, other by adapting weak consistency and other release consistency model. There are some differences within these models regarding the manner of relaxing their program orders but in general, they can be part of this category.

6 Conclusion

This document presented the memory consistency models in modern shared-memory multiprocessors. Because of the shared data, it is important to keep the memory consistent and avoid an undesirable view after a program execution. This problem is solved by introducing memory consistency models in order to help the programmer to understand each processor’s model and write correct and efficient programs.

Performance is always the programmer’s desire in order to exploit the computing system’s abilities. The relaxed models by reordering the memory instructions help the compiler, i.e. to hide the write latency and optimize the performance of a program by executing it out of order and using store buffers.

The problems which have arisen due to different models impact the portability of the program which may not have the same behavior to each processor memory model. Because of the different memory reordering that processors provide, unexpected problems may arise, makes the program writing difficult to the programmer. Thus, a good knowledge of several memory models is helpful and gives the advantage to the programmer to expect each model’s behavior and avoid critical errors in code writing.

Although the memory models define different rules according to their relaxations, memory barriers have been introduced to help programmers write code which could be executed in several machines. Using memory barriers, and specifically the correct memory barriers provided by each architecture, reordering of the memory operations could be avoided and even more processors may execute their programs under sequential consistency. In that way, portability of a code segment is possible and hundreds of programs may be executed with the same results with a consistent view in memory.

Concluding, a big disadvantage of relaxed memory models is the complexity of programming due to the optimizations from the compiler and the memory barriers that have to be placed in program to ensure the behavior expected by the programmer. As at most memory models react differently, programming must be careful in order to present a program that can be executed in many models with the same results.

References


