# The OpenMP Memory Model

Jay P. Hoeflinger[1] and Bronis R. de Supinski[2]

[1] Intel, 1906 Fox Drive, Champaign, IL 61820
jay.p.hoeflinger@intel.com
http://www.intel.com
[2] Lawrence Livermore National Laboratory, P.O. Box 808, L-560,
Livermore, California 94551-0808*
bronis@llnl.gov
http://www.llnl.gov/casc

**Abstract.** The memory model of OpenMP has been widely misunderstood since the first OpenMP specification was published in 1997 (Fortran 1.0). The proposed OpenMP specification (version 2.5) includes a memory model section to address this issue. This section unifies and clarifies the text about the use of memory in all previous specifications, and relates the model to well-known memory consistency semantics. In this paper, we discuss the memory model and show its implications for future distributed shared memory implementations of OpenMP.

## 1 Introduction

Prior to the OpenMP version 2.5 specification, no separate OpenMP Memory Model section existed in any OpenMP specification. Previous specifications had scattered information about how memory behaves and how it is structured in an OpenMP program in several sections: the parallel directive section, the flush directive section, and the data sharing attributes section, to name a few. This has led to misunderstandings about how memory works in an OpenMP program, and how to use it.

The most problematic directive for users is probably the `flush` directive. New OpenMP users may wonder why it is needed, under what circumstances it must be used, and how to use it correctly. Perhaps worse, the use of explicit flushes often confuses even experienced OpenMP programmers.

Indeed, the SPEC OpenMP benchmark program ammp was recently found to be written with assumptions that violate the OpenMP version 2.0 (Fortran) [1] specification. The programmer apparently assumed that a full-memory flush was implied by acquiring and releasing an OpenMP lock. The OpenMP Fortran 2.0 specification is largely silent on the issue of whether a flush is implied by a lock acquire, probably creating the confusion that led to the error. One must go to the OpenMP C/C++ 2.0 [2] specification to find language that addresses the flush operation in relation to OpenMP locks, and even that language is ambiguous.

* This work was partially performed under the auspices of the U.S. Department of Energy by University of California LLNL under contract W-7405-Eng-48. UCRL-ABS-210774.

M.S. Mueller et al. (Eds.): IWOMP 2005/2006, LNCS 4315, pp. 167 – 177, 2008.

The proposed OpenMP 2.5 specification unifies the OpenMP Fortran and C specifications into a single document with a single set of rules, as much as possible. The OpenMP language committee has tried to provide a coherent framework for the way OpenMP relates to the base languages. One of the basic parts of this framework is the OpenMP memory model.

Up to now, the lack of a clear memory model has not made much difference. In general, compilers have not been very aggressive with code re-ordering optimizations and multiprocessors have been fairly simple in structure. Programs that did not follow the memory model would still usually work. But optimizing compilers are getting more sophisticated and aggressive. OpenMP implementations and machine architectures are getting more complicated all the time. Multi-core processors, NUMA machines and clusters of both are becoming more prevalent, making it all the more important that the nature of the memory behavior of OpenMP programs be clearly established.

In this paper, we describe the OpenMP memory model, how it relates to well-known memory consistency models, and the implications the model has for writing parallel programs with OpenMP. In section 2, we describe the OpenMP memory model, as it exists in the proposed OpenMP 2.5 specification. In section 3, we briefly discuss how the memory usage was addressed in previous OpenMP specifications, and how this has led to programmer confusion. In section 4, we show how the OpenMP memory model relates to existing memory consistency models. Finally, section 5 discusses the implications of using the OpenMP memory model to address distributed shared memory systems for OpenMP.

## 2 The OpenMP Memory Model

OpenMP assumes that there is a place for storing and retrieving data that is available to all threads, called the *memory*. Each thread may have a *temporary view* of memory that it can use instead of memory to store data temporarily when it need not be seen by other threads. Data can move between memory and a thread's temporary view, but can never move between temporary views directly, without going through memory.

Each variable used within a parallel region is either shared or private. The variable names used within a parallel construct relate to the program variables visible at the point of the parallel directive, referred to as their "original variables". Each shared variable reference inside the construct refers to the original variable of the same name. For each private variable, a reference to the variable name inside the construct refers to a variable of the same type and size as the original variable, but private to the thread. That is, it is not accessible by other threads.

There are two aspects of memory system behavior relating to shared memory parallel programs: coherence and consistency [3]. Coherence refers to the behavior of the memory system when a single memory location is accessed by multiple threads. Consistency refers to the ordering of accesses to different memory locations, observable from various threads in the system.

OpenMP doesn't specify any coherence behavior of the memory system. That is left to the underlying base language and computer system. OpenMP does not guarantee anything about the result of memory operations that constitute data races within a program. A data race in this context is defined to be accesses to a single variable by at least two threads, at least one of which is a write, not separated by a synchronization operation. OpenMP *does* guarantee certain consistency behavior, however. That behavior is based on the OpenMP *flush* operation.

The OpenMP flush operation is applied to a set of variables called the *flush set.* Memory operations for variables in the flush set that precede the flush in program execution order must be firmly lodged in memory and available to all threads before the flush completes, and memory operations for variables in the flush set, that follow a flush in program order cannot start until the flush completes. A flush also causes any values of the flush set variables that were captured in the temporary view, to be discarded, so that later reads for those variables will come directly from memory.

A `flush` without a list of variable names flushes all variables visible at that point in the program. A `flush` with a list flushes only the variables in the list.

The OpenMP flush operation is the only way in an OpenMP program, to guarantee that a value will move between two threads. In order to move a value from one thread to a second thread, OpenMP requires these four actions in exactly the following order:

1. the first thread writes the value to the shared variable,
2. the first thread flushes the variable.
3. the second thread flushes the variable and
4. the second thread reads the variable.

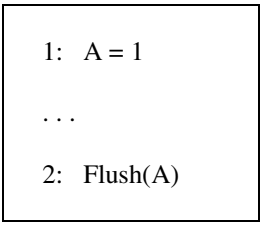


**Fig. 1.** A write to shared variable A may complete as soon as point 1, and as late as point 2

The flush operation and the temporary view allow OpenMP implementations to optimize reads and writes of shared variables. For example, consider the program fragment in Figure 1. The write to variable A may complete as soon as point 1 in the figure. However, the OpenMP implementation is allowed to execute the computation denoted as "…" in the figure, before the write to A completes. The write need not complete until point 2, when it must be firmly lodged in memory and available to all other threads. If an OpenMP implementation uses a temporary view, then a read of A during the "…" computation in Figure 1 can be satisfied from the temporary view, instead of going all the way to memory for the value. So, flush and the temporary view together allow an implementation to hide both write and read latency.

A flush of all visible variables is implied 1) in a barrier region, 2) at entry and exit from parallel, critical and ordered regions, 3) at entry and exit from combined parallel work-sharing regions, and 4) during lock API routines. The flushes associated with the lock routines were specifically added in the OpenMP 2.5 specification, a distinct change to both 2.0 specifications, as discussed in the following section. A flush with a list is implied at entry to and exit from atomic regions, where the list contains the object being updated.

The C and C++ languages include the `volatile` qualifier, which provides a consistency mechanism for C and C++ that is related to the OpenMP consistency mechanism. When a variable is qualified with `volatile`, an OpenMP program must behave as if a flush operation with that variable as the flush set were inserted in the program. When a read is done for the variable, the program must behave as if a flush were inserted in the program at the sequence point prior to the read. When a write is done for the variable, the program must behave as if a flush were inserted in the program at the sequence point after the write.

Another aspect of the memory model is the accessibility of various memory locations. OpenMP has three types of accessibility: shared, private and threadprivate. Shared variables are accessible by all threads of a thread team and any of their descendant threads in nested parallel regions.

Access to private variables is restricted. If a private variable `X` is created for one thread upon entry to a parallel region, the sibling threads in the same team, and their descendant threads, must not access it. However, if the thread for which X was created encounters a new parallel directive (becoming the master thread for the inner team), it is permissible for the descendant threads in the inner team to access X, either directly as a shared variable, or through a pointer. The difference between access by sibling threads and access by the descendant threads is that the variable `X` is guaranteed to be still available to descendant threads, while it might be popped off the stack before siblings can access it. For a threadprivate variable, only the thread to which it is private may access it, regardless of nested parallelism.

```
#pragma omp parallel private(x) shared(p0,p1)
Thread 0                                  Thread 1

x = …;                                    x = …;
p0 = &x;                                  p1 = &x;
/* references in the following line are not allowed: */
…*p1 …                                    … *p0 …

#pragma omp parallel shared(x)
Thread 0      Thread 1      Thread 0      Thread 1
… x …         … x …         … x …         … x …
…*p0 …        … *p0 …       … *p1 …       … *p1 …

/* the following are not allowed:  */
… *p1 …       … *p1 …       … *p0 …       … *p0 …
```

**Fig. 2.** Access to a private variable by name or through a pointer is allowed only on the thread to which the variable is private, and its descendant threads

## 3 Memory Usage Descriptions in Previous Specifications

OpenMP specifications prior to OpenMP 2.5 barely addressed the OpenMP memory model. In the 2.0 C/C++ spec, the memory model was discussed in a paragraph in the execution model section, and in some text in the description of the flush directive. The 2.0 Fortran spec includes similar text in the description of the flush directive. It has no equivalent paragraph in the execution model section, although a paragraph in the section on the shared clause serves this purpose. The "data sharing attribute clauses" section in the C/C++ 2.0 spec and the "data scope attribute clauses" section of Fortran 2.0 describe the affects of the shared and private clauses.

The scattered text of the 2.0 specifications collectively gives an impression of memory behavior without being comprehensive. Nowhere in the 2.0 or earlier specs was there a mention of a temporary view of memory, but processor registers were mentioned. The proposed 2.5 specification has made this temporary view more general, which allows other forms of temporary memory.

Another issue related to the memory model is whether flushes are implied by the OpenMP lock API routines. The Fortran 2.0 spec is silent on the issue. However, those routines are not mentioned in the list of places where a flush is implied, so it is clear that the intention was that the lock routines do not imply flushes. The C/C++ 2.0 spec is likewise silent, but says "There may be a need for flush directives to make the values of other variables consistent."

The lack of a clear statement in previous specs with respect to flushes for the lock API routines has caused significant confusion. A very common mistake made by programmers is to forget to insert appropriate flushes when locks are being used.

| **Thread 0** | **Thread 1** |
|---|---|
| `omp_set_lock(lockvar);`<br>`count++;`<br>`omp_unset_lock(lockvar);` | |
| | `omp_set_lock(lockvar);`<br>`count++;`<br>`omp_unset_lock(lockvar);` |

**Fig. 3.** Threads cooperating through locks to increment a shared variable `count`

Consider the example in Figure 3. Most programs are written in this fashion, but without an implied flush in the `omp_set_lock` or `omp_unset_lock` routines, this program may not work as expected. This is because OpenMP semantics do not require a read of `count` from memory before the increment operation, or a flush of `count` to memory after it. Both threads are allowed to operate only on their temporary view of `count`. Even worse, the compiler might very well in-line the calls and reorder the update of `count` such that it is no longer in the locked region since there is no dependence between the calls and the variable `count`.

| Thread 0 | Thread 1 |
|---|---|
| `omp_set_lock(lockvar);`<br>`#pragma omp flush(count)`<br>`count++;`<br>`#pragma omp flush(count)`<br>`omp_unset_lock(lockvar)` | |
| | `omp_set_lock(lockvar);`<br>`#pragma omp flush(count)`<br>`count++;`<br>`#pragma omp flush(count)`<br>`omp_unset_lock(lockvar);` |

**Fig. 4.** A failed attempt to correctly use variables inside a locked region

| Thread 0 | Thread 1 |
|---|---|
| `omp_set_lock(lockvar);`<br>`#pragma omp flush(count,lockvar)`<br>`count++;`<br>`#pragma omp flush(count,lockvar)`<br>`omp_unset_lock(lockvar);` | |
| | `omp_set_lock(lockvar);`<br>`#pragma omp flush(count,lockvar)`<br>`count++;`<br>`#pragma omp flush(count,lockvar)`<br>`omp_unset_lock(lockvar);` |

**Fig. 5.** A correct way to write a locked update according to OpenMP 2.0

Including flushes of `count` inside the locked region, as in Figure 4, ensures that the most recent value for `count` is read, and that memory is updated with the write. However, it still does not address the compiler reordering problem. Essentially, these flushes on `count` do not ensure any ordering with operations on `lockvar`. The compiler is still free to reorder the call to omp_set_lock with respect to the flushes and the increment of `count` because they don't refer to the same variables.

The programmer would need to write the code as in Figure 5 to both prevent reordering with respect to the lock calls, and to keep the global value of `count` up to date. That is, the programmer must ensure ordering between the two variables by including both in the flush list.

A no-list flush is implicit for the lock API routines in the proposed 2.5 spec. Thus, code written in the natural manner of Figure 3 will work as most programmers expect. As mentioned above, the SPEC OpenMP code ammp was written in this manner (see Figure 6).

```
#ifdef _OPENMP
      omp_set_lock(&(a1->lock));
#endif
      a1fx = a1->fx;
      a1fy = a1->fy;
      a1fz = a1->fz;
      a1->fx = 0;
      a1->fy = 0;
      a1->fz = 0;
      xt = a1->dx*lambda +a1->x - a1->px;
      yt = a1->dy*lambda +a1->y - a1->py;
      zt = a1->dz*lambda +a1->z - a1->pz;
#ifdef _OPENMP
      omp_unset_lock(&(a1->lock));
#endif
```

**Fig. 6.** SPEC OpenMP benchmark ammp source code that demonstrates failure to use flush directives with OpenMP locks (incorrect prior to specification version 2.5)

## 4 The OpenMP Memory Consistency Model

OpenMP provides a relaxed consistency model that is similar to *weak ordering* [7][8]. Strong consistency models enforce program order, an ordering constraint that requires memory operations to appear to execute in the sequential order specified by the program. For example, a memory model 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" [3]. The OpenMP memory model specifically allows the reordering of accesses within a thread to different variables unless they are separated by a flush that includes the variables. Intuitively, the temporary view is not always required to be consistent with memory. In fact, the temporary views of the threads can diverge during the execution of a parallel region and flushes (both implicit and explicit) enforce consistency between temporary views.

Memory consistency models for parallel machines are based on the ordering enforced for memory accesses to different locations by a single processor. We denote memory access ordering constraints by using the "→" (ordering) notation applied to reads (R), writes (W), and synchronizations (S). For instance, for reads preceding writes in program execution order, constraining them to maintain that order would be denoted R →W. Sequential consistency requires all memory accesses to complete in the same order as they occur in program execution, meaning the orderings R→R, R→W, W→R, and W→W. It also requires the effect of the accesses by all threads to be equivalent to performing them in some total (i.e., sequential) order.

Sequential consistency is often considered difficult to maintain in modern multiprocessors. The program order restriction prevents many important compiler optimizations that reorder program statements [4]. Frequently, sequentially consistent multiprocessors do not complete a write until its effect is available to all other processors.

Relaxed consistency models remove the ordering guarantees for certain reads and writes, but typically retain them around synchronizations [4][5][6]. There are many

types of relaxed consistency. The OpenMP memory model is most closely related to weak ordering. Weak ordering prohibits overlapping a synchronization operation with any other shared memory operations of the same thread, while synchronization operations must be sequentially consistent with other synchronization operations. Thus, the set of orderings guaranteed by weak ordering is the following: {S→W, S→R, R→S, W→S, S→S}.

Relaxed consistency models have been successful because most memory operations in real parallel programs can proceed correctly without waiting for previous operations to complete. Successful parallel programs arrange for a huge percentage of the work to be done independently by the processors, with only a tiny fraction of the memory accesses being due to synchronization. Thus, relaxed consistency semantics allows the overlapping of computation with memory access time, effectively hiding memory latency during program execution.

OpenMP makes no guarantees about the ordering of operations during a parallel region, except around flush operations. Flush operations are implied by all synchronization operations in OpenMP. So, an optimizing compiler can reorder operations inside a parallel region, but cannot move them into or out of a parallel region, or around synchronization operations. The flush operations implied by the synchronization operations form memory fences. Thus, the OpenMP memory model relaxes the order of memory accesses except around synchronization operations, which is essentially the definition of weak ordering.

The programmer can use explicit flushes to insert memory fences in the code that are not associated with synchronization operations. Thus, the OpenMP memory consistency model is a variant of weak ordering.

The OpenMP memory model further alters weak ordering by allowing flushes to apply only to a subset of a program's memory locations. The atomic construct includes an implied flush with a flush set consisting of only the object being updated. An optimizing compiler can reorder accesses to items not in the flush set with respect to the flush. Further, no ordering restrictions between flushes with empty flush set intersections are implied. In general, using a flush set implies that memory access ordering is only required for that set. The correct use of flush sets can be very complicated and we urge OpenMP users to avoid them in general.

The ordering constraint of OpenMP flushes is modeled on sequential consistency, similar to the restrictions on synchronization operations in weak ordering and lazy release consistency [7][8][9]. Specifically, the OpenMP memory model guarantees:

1. If the intersection of the flush-sets of two flushes performed by two different threads is non-empty, then the two flushes must be completed as if in some sequential order, seen by all threads;
2. If the intersection of the flush-sets of two flushes performed by one thread is non-empty, then the two flushes must appear to be completed in that thread's program order;
3. If the intersection of the flush-sets of two flushes is empty, then the threads can observe these flushes in any order.

If an OpenMP program uses synchronization constructs and flushes to avoid data races, then it will execute as if the memory was sequentially consistent.

## 5 Future Implications of the OpenMP Memory Model

Messaging latency to remote nodes in a modern computational cluster is hundreds or thousands of times higher than latency to memory for modern processors (see Figure 7). This latency makes the traditional method of enforcing sequential consistency (requiring a thread that issues a write of a shared variable to wait for the value to be visible to all threads) prohibitively expensive for OpenMP clusters. Fortunately, the OpenMP memory consistency model allows latency hiding of memory operations. This freedom is useful for a hardware shared memory (HSM) OpenMP implementation, but it is essential for a distributed shared memory (DSM) version of OpenMP, which simply has more memory latency to hide.

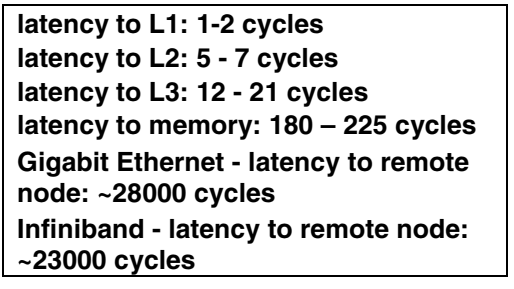


**Fig. 7.** Itanium® latency to caches compared with latency to remote nodes

So, we claim that the relaxed memory model of OpenMP, with its ability to do cheap reads and hide the latency of writes, enables DSM OpenMP implementations. Without the ability to hide a cluster's enormous memory latency, DSM OpenMP implementations might only be useful for embarrassingly-parallel applications.

Even taking advantage of latency hiding, a DSM OpenMP implementation may be useful for only certain types of applications. In an Intel® prototype DSM OpenMP system, called Cluster OMP, we have found that codes in which certain characteristics dominate are very difficult to make perform well, while codes with other dominant characteristics can have good performance.

Codes that use flushes frequently tend to perform poorly. This means that codes that are dominated by fine-grained locking, or codes using a large number of parallel regions with small amounts of computation inside, typically have poor performance. Frequent flushes emphasize the huge latency between nodes on a cluster, since they reduce the frequency of operations that can be overlapped.

Codes dominated by poor data locality are also unlikely to perform well with DSM OpenMP implementations. Poor data locality for the Cluster OMP system means that memory pages are being touched by multiple threads. This implies that more data will be moving between threads, over the cluster interconnect. This data movement taxes the cluster interconnection network more than for a code with good data locality. The more data being moved, the more messaging overheads will hurt performance.

On the other hand, in experiments with the Cluster OMP system, we have observed that certain applications achieved speedups that approach the speedups obtained with

an HSM system (see Figure 8). We have seen that computation with good data locality and little synchronization dominates the highest performing codes.

The applications we tested were gathered from Intel® customers who were participating in a technology preview of the prototype system. We can't reveal details of the codes, but the application types were:

1. a particle simulation code
2. a magneto-hydro-dynamics code
3. a computational fluid dynamics code
4. a structural simulation code
5. a graph processing code
6. a linear solver code
7. an x-ray crystallography code

Figure 8 shows the performance results we obtained for these codes. The speedup is shown for both the OpenMP and Cluster OMP versions of each code. In addition, the ratio of those speedups is shown, in the form of the Cluster OMP speedup as a

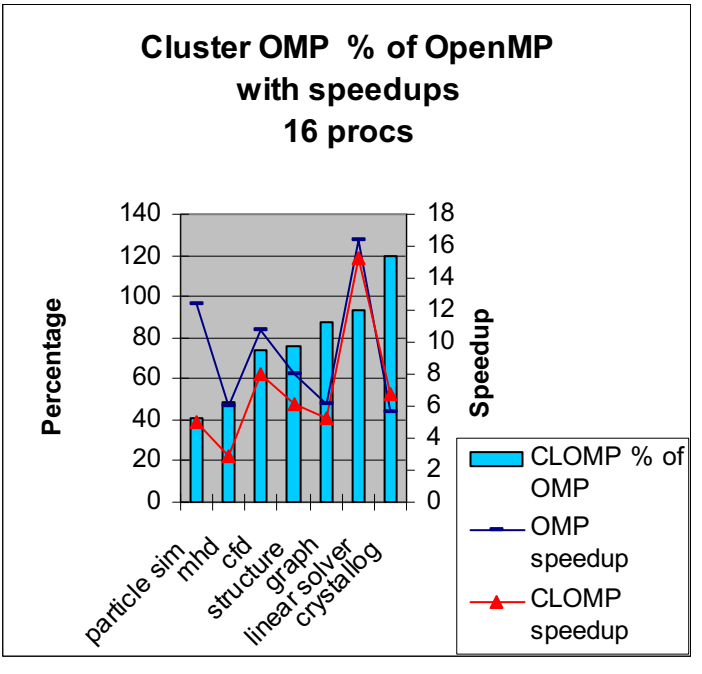


**Fig. 8.** Raw speedup of Cluster OMP on a cluster and OpenMP on a hardware shared memory machine, plus speedup percentage of Cluster OMP versus OpenMP for a set of codes

percentage of the OpenMP speedup. For these seven codes, five of them achieved a Cluster OMP speedup that was more than 70% of the performance of the OpenMP performance.

We have found that the applications for which Cluster OMP works well are usually those that have a large amount of read-only shared data and a small amount of read-

write shared data. If, in addition, the data access patterns are irregular, then the applications would be more difficult to write using a direct messaging API, such as MPI.

These characteristics are typical of an emerging class of applications, known as RMS workloads (Recognition, Mining, and Synthesis) [10]. These workloads involve applications that typically use massive amounts of input data, such as pattern recognition, parallel search, data mining, visualization, and synthesis. We speculate that a DSM implementation of OpenMP may be useful for applications of this type.

## 6 Conclusion

The proposed OpenMP 2.5 spec unifies and clarifies the OpenMP memory model. The refined memory model description alleviates some of the confusion over the use of the `flush` directive and simplifies the correct use of OpenMP. Further, the proposed OpenMP 2.5 spec has added an implicit no-list flush to the lock API routines, making their use more intuitive.

OpenMP enforces a variant of weak ordering, as clearly demonstrated in the memory model description. This has performance implications for programs run on HSM systems, because it allows the compiler to apply optimizations and reorder code in a program. It also has important implications for reasonable performance of OpenMP on future DSM systems.

## References

1. OpenMP Architecture Review Board: OpenMP Fortran Application Program Interface, Version 2.0. OpenMP Architecture Review Board (2000)
2. OpenMP Architecture Review Board: OpenMP C and C++ Application Program Interface, Version 2.0. OpenMP Architecture Review Board (2002)
3. Lamport, L.: How to Make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs. IEEE Transactions on Computers 28(9), 690–691 (1979)
4. Adve, S.V., Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial. IEEE Computer 29(12), 66–76 (1996) (Also: WRL Research Report 95/7, Digital Western Research Laboratory, Palo Alto, California, 1995)
5. Hennessy, J.L., Patterson, D.A.,, C.A.: Computer Architecture A Quantitative Approach, 2nd edn. Morgan Kaufman Publishers, Inc., San Francisco (1996)
6. Gharachorloo, K.: Memory Consistency Models for Shared-Memory Multiprocessors. PhD thesis, Stanford University (1995)
7. Dubois, M., Scheurich, C., Briggs, F.: Memory Access Buffering in Multiprocessors.
8. Adve, S.V., Hill, M.D.: Weak Ordering – A New Definition. In: Proceedings of the 17th Annual International Symposium on Computer Architecture, May 1990, pp. 2–14 (1990)
9. Keleher, P.: Lazy Release Consistency for Distributed Shared Memory. PhD thesis, Department of Computer Science, Rice University (January 1995)
10. `http://www.intel.com/technology/computing/archinnov/teraera/`