A simple and efficient memory model for weakly-ordered architectures

Raúl Silvera
rauls@ca.ibm.com

Michael Wong
michaelw@ca.ibm.com

Paul McKenney
paulmck@us.ibm.com

Bob Blainey
blainey@ca.ibm.com

Document number: N2153=07-0013

Date: 2007-01-12

Project: Programming Language C++, Evolution Working Group

Reply-to: Raul Silvera (rauls@ca.ibm.com)

Revision: Version 1.00

Abstract

This paper will propose modifications to the current ISO C++ Memory Model [ISOMM] to efficiently support a wider group of machine architectures, in particular those that support relaxed memory consistency models. Our model provides three forms of standalone memory fences which, when combined with unordered atomic operations, allow the programmer to represent arbitrarily complex ordered atomic operations. One of the design goals of this model is to lessen interference with traditional compiler optimizations; unnecessary constraints limit the precision of program analysis and can have a significant detrimental impact on performance. We will describe some use cases to demonstrate the usability of this model, and compare it with the current [ISOMM] model. We will present some empirical results to show the overhead of the ordering constraints on atomic operations. The large overhead of these primitives is part of the motivation for providing a fine granularity of ordering constraints.

1. Introduction

Weakly-ordered processor architectures [Hennessy] provide a relaxed view of the memory subsystem, where different processors may have different views of shared storage. One of the motivations for having weak storage ordering is to allow storage subsystem optimizations, which enable better scaling of the memory nest design. It is important to ensure that modern programming models do not artificially constrain the scalability of these systems, which would ultimately undermine their success.

The structure of this paper is to first define a simple memory model that can be described in natural language, and has sufficient expressive power to precisely describe the memory synchronization requirements of many algorithms. This memory model is based on the current practice on the IBM C/C++ parallelizing compilers for PowerPC-based systems[XLC].

One of the main design goals of this model is to allow the programmer to precisely convey the ordering requirements of an algorithm, so that implementations can avoid introducing unnecessary synchronization. Unnecessary synchronization will affect performance of both parallel and sequential applications, as the hardware primitives needed to implement it on weakly ordered architectures typically have a significant runtime cost, even on sequential execution.

We will then discuss some use cases where this model is advantageous over the current ISOMM model proposal, and will provide experimental results to quantify the performance impact on current IBM hardware. Finally, we will present a prioritized list of recommendations for the current ISOMM model.

2. A simple memory model

Visibility, atomicity and ordering are separate concepts, which together define a memory model. Visibility defines the circumstances under which a thread can observe the effects of memory operations performed by another thread. Atomicity determines whether a single memory operation will become visible to other threads only on its entirety, or whether intermediate states not defined by the programmer may be visible to other threads. We treat isolation, whether a store into a defined memory location may affect the value of neighboring locations, as part of atomicity. Ordering is concerned with possible observed orders of memory operations with respect to other threads.

These concepts are very closely related; high-level locking primitives that provide visibility, atomicity and ordering guarantees are very common in parallel programming. However, many hardware implementations and some modern parallel programming languages[1] provide primitive operations that do not combine them. In many cases, being able to precisely define the memory consistency requirements of an algorithm is crucial to achieve high performance.

This model requires all stores that modify the same memory location to be totally ordered. When restricted to operations performed by a single thread, this order is consistent with program order. All threads are guaranteed to observe any subset of those stores in an order that is consistent with the total order of stores.

2.1. Atomicity

This model defines no atomicity guarantees on unmarked storage. As in ISOMM, explicit type qualifiers are to be used to mark specific integral variables as atomically updatable. Also, it provides isolation on all storage, with some exceptions related to bitfields, which we will not define in detail.

2.2. Visibility

This model provides no visibility guarantees on unmarked or atomic storage, other than what is implied by the ordering guarantees.

2.3. Ordering

As in ISOMM, this model defines operations on atomic storage with specific memory ordering guarantees. These *atomic operations* are composed of a set of native memory operations (regular load and stores) and an optional set of ordering guarantees.

Each atomic operation defines three sets of memory operations:

A: Memory operations preceding the atomic operation in program order[2], plus any memory operations from other threads performed with respect to this thread before the atomic operation.

B: Memory operations implied by the atomic operation.

C: Memory operations following the atomic operation in program order, plus any memory operations performed by other threads that have observed the result of a store in C.

A store is performed with respect to a thread when any subsequent loads of that memory location from that thread return the stored value, or the value stored by a later store in the total order of stores.

A load is performed with respect to a thread when no subsequent instructions from that thread can affect the value returned by that load.

An ordering guarantee defines how operations in these sets are performed with respect to each other. We say that a set of operations is performed *before* another set of operations if the operations on the first are

---

[1] OpenMP [OpenMP2.5] includes both a standalone memory fence (the OMP FLUSH directive) and unordered atomic updates (the OMP ATOMIC directive).

[2] Program order refers to the defined order of evaluation defined by the underlying language, which for C++ is not a total order.

performed with respect to another thread before any of the operations of the second set are performed with respect to that thread.

2.3.1. Ordering atomic operations

This model defines three forms of ordering atomic operations.

- An atomic operation with *acquire* semantics ensures that all memory operations in set B are performed before any memory operation in set C.
- An atomic operation with *release* semantics ensures that all memory operations in set A are performed before any memory operation in set B.
- An atomic operation with *ordered* semantics has both acquire and release semantics. That is, all memory operations in set A are performed before any memory operation in set B, and all memory operations in set B are performed before any memory operation in set C.

The terms acquire and release are evocative of the lock acquire and lock release operations. Typically a lock acquisition requires a load of the lock variable with acquire semantics, and a lock release requires a store to the lock variable with release semantics.

As in ISOMM, this model defines only a subset of all possible combinations of operations/orderings:

	load	Store	Compare-and-swap[3]
unordered (raw)	✓	✓	✓
acquire	✓	×	✓
release	×	✓	✓
ordered	✓	✓	✓

It should be highlighted that according to their definition, *ordered* atomic operations provide two separate ordering guarantees. On weakly-ordered architectures, the cost of this operation may be significantly larger than individual *acquire* or *release* operations.

No ordering exists between atomic memory operations other than what is provided by their definitions this section. In particular, atomic operations do not guarantee sequential consistency. In cases where atomic variables with sequential-consistent behavior are desirable, they are available to the programmer at a cost in performance by manipulating them exclusively with atomic operations with ordered semantics.

2.3.2. Standalone memory fences

While the atomic operations defined in the previous section are adequate in many cases to precisely represent the ordering requirements of an algorithm, there are situations where they are insufficient. The fundamental issue is that a memory model can only provide a fixed set of ordering constructs, while some algorithms may require ordering guarantees on arbitrarily complex sequences of memory operations. If no other mechanisms are available, the programmer is forced to use sequences of the ordering atomic operations provided, potentially overspecifying the ordering requirements and incurring unnecessary costs. Given the non-local nature of the ordering guarantees, it will frequently be impossible to optimize away any redundancy through program analysis.

Having the ability to separately specify atomicity and ordering is particularly important on weakly-ordered architectures that provide mechanisms to implement these guarantees. However, even on architectures that do not provide such explicit mechanisms, the reduced synchronization burden may still have an impact to performance as it exposes optimization opportunities to the compiler that otherwise might be unsafe.

In current practice, the programmer uses non-portable mechanisms to take advantage of these hardware facilities; notably, the Linux kernel defines a comprehensive set of explicit memory fences [McKenney], which are mapped via preprocessor macros to non-portable mechanisms for each of its host architectures.

---

[3] Other atomic update operations may be included as well, but for the purpose of this document, they are analogous to compare-and-swap.

The omission of some of these mechanisms will force developers to continue to use platform-specific idioms to achieve maximum performance, defeating the purpose of a standard memory model.

This model provides three forms of standalone memory fences which, when combined with unordered atomic operations, allows the programmer to define arbitrarily complex sets of ordered atomic operations. We have named these fences according to their intended usage. Following the set definitions on the previous section, a memory fence is an ordering atomic operation with an empty set B.

- An ordered memory fence ensures that all memory operations in set A are performed before any memory operation in set C.
- An acquire memory fence ensures that all loads in set A are performed before any memory operation in set C. From the terminology on the JSR-133 cookbook for compiler writers [JSR133C], this is a LoadLoad;LoadStore barrier.
- A release memory fence ensures that all memory operations in set A are performed before any store in set C. This is a LoadStore;StoreStore barrier.

2.3.3. Other variants of ordering atomic operations

There are some situations where it is useful to provide some variants of the primitives introduced so far, for use on specific situations:

- The *acquire_address_base* ordering: This is identical to acquire, except that it only affects the ordering of memory operations in set C that are address-dependent on the value returned by the atomic operation. We say that a memory operation is address-dependent on a value if the computation of the address being manipulated by the operation is affected in a non-trivial way by that value.
- The *compiler_ordered* memory fence: this is identical to the ordered memory fence, except that it doesn't trigger the introduction of any hardware ordering primitives. This fence is to be used in place of an ordered fence in situations where synchronization instructions are not needed; for example, when all the threads manipulating the storage in question are known to share the same view of memory.

3. Impact to compiler optimization

One of the design goals of this model is to lessen interference with traditional compiler optimizations. Unnecessary constraints limit the precision of program analysis and can have a significant detrimental impact on performance. Compilers frequently operate on a limited program scope, so any restrictions introduced by the model would likely affect both sequential and parallel sections of a program as well as shared code that is intended for use in both sequential and parallel contexts.

There are only two restrictions uniquely associated with this model:

- Write speculation and invention is forbidden. That is, a write to a variable from a thread cannot be observed by other threads unless that write appears in the flow of control of the program given the current inputs.
- Atomic loads cannot be replicated. Multiple uses of a value returned by an atomic operation cannot be reloaded from storage; this prevents conflicts in cases where the value reloaded from storage is modified.

Under these constraints and after considering any specified ordering guarantees, the compiler may treat atomic operations as regular memory operations. The relaxed visibility guarantees provided by this model allow the compiler to freely reorganize and remove atomic operations within the boundaries of the preceding acquire operation and the next release operation.

Note that while atomic loads cannot be replicated, it is permitted for them to be coalesced by the compilation system. For example, the expression load_raw(x) + load_raw(x) can be replaced by 2 * load_raw(x). Situations that depend on the visibility of stores from other threads, such as busy waiting loops, require the specification of ordering constraints to ensure that the value is reloaded from memory.

4. Motivating examples

In this section we will describe some use cases to demonstrate the usability of this model, and compare it with the current ISOMM model.

4.1. Mailbox inspection: acquire fences

There are situations where the ordering guarantees needed for an atomic load depend on a value loaded. In these cases, ISOMM requires the use of a load_acquire operation, which is unnecessarily strong.

In this model, the standalone acquire fence allows the program to decide whether the ordering constraints are needed, depending on the value being loaded.

One example situation is when a slave thread is polling a set of mailboxes for messages:

```
for (i=0; i< num_mailboxes; i++) {
  if (mailbox[i].load_raw() == my_id) {
    acquire_fence();    // Prevents speculation of memory
    do_work(i);         // accesses in do_work
 }
}
```

In this example, the algorithm requires the memory accesses inside do_work to be performed after the corresponding load is executed. Using a load_acquire on each mailbox check would be correct, but would introduce unnecessary ordering constraints between loads from the mailbox. These constraints will increase the latency of dispatch on weakly ordered machines, and would be unnecessary if a mailbox did not contain a message for the current thread.

Also, those unnecessary ordering constraints may affect compiler transformations. In this example, if the mailboxes are contiguous in memory, it would be possible to load multiple mailboxes at a time using a wide (SIMD) load instruction, but this may be disallowed by the ordering constraints implied by the load_acquire operation.

4.2. Multiple lock release: release fences

The standalone release fence is useful when multiple signals need to be sent after an operation has been completed. One such example is when releasing multiple locks[4]:

```
do_work(A,B);
release_fence();   // Ensures memory accesses in do_work
                   // are visible before releasing the lock
lock_A.store_raw(LOCK_UNLOCKED);
lock_B.store_raw(LOCK_UNLOCKED);
```

For this example, ISOMM only provides the store_release operation, which would introduce an unnecessary ordering specification between the stores to the lock variables.

Again, this will introduce a significant penalty on weakly-ordered machines. Even on architectures with stronger memory ordering, performance may be affected as store_release operations would prevent the two stores from being combined or reordered by the compiler if that was found legal through program analysis.

4.3. Reference counting

Another common example is when using reference counting to deallocate an object once all threads have finished working on it. On this example, an object contains a counter indicating how many threads are accessing it; after each thread finishes working with that object, the reference count in decremented. The

[4] This example assumes that the lock release ordering is not important. If the algorithm being implemented requires a certain release ordering, then the intermediate memory fences are needed. However, the common use of multiple locks does not require a specific lock release ordering.

last thread to finish working with the object will release the object. In this memory model, that would be coded like this:

```
do_work(object);
if (fetch_and_add_release(ref_count,-1) == 0) {
   acquire_fence(); // Ensures that the destruction of the
                    // object is not speculated ahead of
                    // the ref_count reaching zero
   recycle(object);
}
```

In this case two orderings are required. One release before decrementing the reference count, to ensure that the counter is decremented after all the uses of the object have been performed, and one acquire after the counter has reached zero, to ensure that the object is recycled after its reference count has been set to 0.

The important point is that while all threads must follow release semantics, only the last one to update the counter requires acquire semantics. For this example, ISOMM only provides the fetch_and_add_ordered operation, which would introduce an unnecessary acquire fence on each thread, increasing the latency of the operation.

Also, the use of an ordered atomic operation will prevent unrelated loads to be speculated ahead of the fetch_and_add operation. With the finer granularity of this example, it is possible for the hardware or compiler to speculate those loads for all threads except for the last one.

5. Use cases from the Linux kernel

The Linux kernel is a great example of a modern parallel application that is performance sensitive and has been ported to a wide variety of architectures. In this section we present some relevant techniques used on the Linux kernel to minimize the cost of synchronization and discuss how they could be implemented under this memory model.

A more detailed examination of reference counting on the Linux kernel is presented in [McKenney2].

5.1. Per-Thread Split Counters

Per-thread split counters are used heavily in operating systems and server applications for purposes of statistical counting in cases where updates are much more frequent are readouts. The reason such counters are heavily used is that they impose minimal overhead on high-frequency critical-path operations such as networking transmission and reception, while still providing data critical to systems management, administration, and troubleshooting.

Each thread (or, in the case of the Linux kernel, each CPU) is assigned its own sub-counter, so that the counter value is obtained by summing up all threads' sub-counters. Each such sub-counter is aligned to the appropriate machine boundary so that normal loads and stores will be atomic, that is, a load from a given sub-counter will return either the initial value of that sub-counter, or the value stored by some store to that sub-counter.

A given thread can then update the counter via normal arithmetic operations, with no memory barriers or atomic instructions required. Code for this idiom is as follows:

```
/* Define the per-CPU counter. */
DEFINE_PER_CPU(unsigned long, mycounter) = {0};

/*
 * Modify a per-CPU counter.  In the Linux-kernel
 * implementation, this would have to be a C-preprocessor
 * macro.
 */
void counter_add(unsigned long *cp, unsigned long v) {
  __get_cpu_var(cp) += v;
}

/*
 * Return the aggregate value of a per-CPU counter.
 * Again, in the Linux kernel, this would have to be
 * a C-preprocessor macro.
 */
unsigned long counter_value(unsigned_long *cp) {
  int cpu;
  unsigned long sum;

  for_each_possible_cpu(cpu) {
    sum += per_cpu(cp, cpu);
  }
  return sum;
}
```

Note that there are cases where the values returned by counter_value() on different CPUs might violate sequential consistency. This is a feature, not a bug. To see this, imagine that "mycounter" was tracking the total number of bytes received via TCP/IP over all interfaces and connections on a machine with three Ethernet adapters. Suppose that these three adapters concurrently receive packets whose lengths are 700, 1100, and 1300 bytes. If these are the first three packets received by this machine, then there are six possible sequences for the cumulative number of bytes received:

1. 0, 700, 1800, 3100
2. 0, 700, 2000, 3100
3. 0, 1100, 1800, 3100
4. 0, 1100, 2400, 3100
5. 0, 1300, 2000, 3100
6. 0, 1300, 2400, 3100

If each packet is being processed by a different CPU, and each of six other CPUs are concurrently and repeatedly executing counter_value(), then it is entirely possible that each of these six CPUs will see a different sequence of values – even on machines implementing the TSO memory ordering, the tightest such ordering that we are aware of in high-volume commercial microprocessors. But this is inherent in the reality of the situation: since the packets are being received concurrently, any ordering assigned to them will by definition be arbitrary. Furthermore, such a situation is at odds with the use case itself, which specified infrequent readout of the counters. There is thus no justification for any high-overhead code sequence that would impose the arbitrary and meaningless ordering that would be required for sequential consistency.

This important usage case illustrates the need for atomic loads and stores in absence of ordering guarantees, and also illustrates a situation where sequential consistency is inherently unnecessary.

5.2. Hash Tables With Lockless Readers

Operating systems contain many read-mostly data structures, such as those representing the hardware and software configuration of the machine and of the environment in which it resides. The contents of these data structures rarely change, but could do so at any time, and they are accessed quite frequently, for example, routing tables are accessed on each packet transmission or directory/file caches are accessed on each I/O. For such structures, it is useful to reduce access overhead to the bare minimum, eliminating

memory barriers and atomic instructions from that code path, even at the expense of a significant increase in update overhead.

For simplicity, this example focuses only on hash-table insertion to the exclusion of removal. Removal can be handled easily, but doing so adds nothing to this example. Also for simplicity, this hash table stores unadorned integers as opposed to the more complex structures that tend to be stored in Linux kernel code using this approach.

```
struct foo {
  struct foo *next;
  int key;
};
struct foo *hashtable[NUM_BUCKETS]
DEFINE_SPIN_LOCK(foo_lock);

int find_foo(int key) {
  struct foo *p;
  int retval;

  rcu_read_lock();
  p = rcu_dereference(hashtable[foo_hash(key)]);
  while (p != NULL && p->key < key)
    p = rcu_dereference(p->next);
  retval = p != NULL && p->key == key;
  rcu_read_unlock();
  return retval;
}

int insert_foo(int key) {
  struct foo *newp, *p, **plast;
  int retval = 0;

  spin_lock(&foo_lock);
  plast = &hashtable[foo_hash(key)];
  p = *plast;
  while (p != NULL && p->key < key) {
    p = p->next;
    plast = &p->next;
  }
  if (p == NULL || p->key != key) {
    newp = kmalloc(sizeof(*newp, GFP_KERNEL);
    if (newp != NULL) {
      newp->key = key;
      newp->next = p;
      rcu_assign_pointer(*plast, newp);
      retval = 1;
    }
  }
  spin_unlock(&foo_lock);
  return retval;
}
```

The rcu_dereference() primitive ensures that its argument is fetched before any subsequent load or store (in program order) that depends on that argument. (There are some indications that rcu_dereference() also needs to prevent compiler optimizations that result in its argument being fetched multiple times, but the implementation currently in the Linux 2.6.19 kernel does not have this effect.) All CPUs except DEC Alpha enforce ordering of dependent loads, so rcu_dereference() evaluates to its argument. On Alpha, p=rcu_dereference(head) is equivalent to:

```
        p = head;
        smp_mb();
```

Thus, only on Alpha, rcu_dereference() prevents the multiple-fetch compiler optimizations described above.

The rcu_assign_pointer() ensures that any prior stores dereferencing the pointer (second argument) are completed before the store of the pointer into the first argument. On many CPUs, rcu_assign_pointer(a,b) is equivalent to the following:

```
        smp_wmb();
        a = b;
```

In principle, only prior assignments that depend on the value of "b" need be affected by rcu_assign_pointer(), but in practice a full store barrier is used.

On non-Alpha CPUs, the above search and insertion functions allow searching without any special atomic instructions, memory barriers, or communication cache misses, permitting extremely low search overheads, as is appropriate for a data structure that is searched frequently and seldom (if ever) modified. However, for this to work correctly, the "next" pointer in "struct foo" must be atomically accessed by normal loads and stores. This example thus demonstrates the need for variables and structure fields that are atomically accessed by normal loads and stores, but without other compiler-generated overhead.

It is important to note that rcu_dereference() is not required in insert_foo(). This is because insert_foo() holds the lock, preventing any other thread from modifying the hash chain in question.

5.3. Communication With Interrupt/Exception Handlers

On all modern multiprocessor-capable CPUs, a given CPU sees its own accesses as occurring in program order, (thankfully) trivializing memory-ordering concerns in single-threaded code. However, suppose that code running in a given thread that must interact with an interrupt or exception handler that runs in the context of that same thread. In this case, reordering done by the CPU is transparent, since both the thread and the handler runs on the same CPU. However, such code cannot ignore the possibility of reordering due to compiler optimizations.

For example, consider NMI-based profiling. Such profiling might well make use of a dynamically allocated buffer that contained fields indicating the buffer size in addition to an array comprising the profiling buckets themselves, as caricatured below:

```
struct profile_buf {
        unsigned long size;
        int count[0];
} *pb;

int start_profile(int size) {
  struct profile_buf *p;
  p = kzalloc(sizeof(*p) + size * sizeof(p->size),  GFP_KERNEL);
  if (p == NULL) return 0;
  p->size = size;
  barrier();
  pb = p;
}

void nmi_prof(unsigned long pc) {
  struct profile_buf *p;

  p = rcu_dereference(pb);
  if (p == NULL) return;
  if (pc > p->size) return;
  p->count[pc]++;
}
```

In this example, the barrier() primitive makes use of a gcc extension to forbid the compiler from reordering memory references, so that an NMI handler will either see pb==NULL or see a properly initialized struct profile_buf. A full memory barrier is not required here, because the NMI handler is guaranteed to execute on the CPU being profiled.

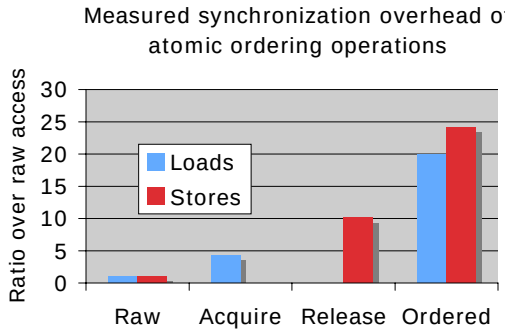
This example illustrates the need for some reliable way of preventing optimizations that would reorder memory references. The earlier examples illustrate this need as well, but indirectly. For example, the rcu_assign_pointer() primitive must also prevent the compiler from engaging in optimizations that would reorder memory references across this primitive – otherwise, the compiler could prevent this primitive from doing its job. This example also illustrates the need to address compiler optimizations independently of CPU reordering.

6. Performance evaluation

In this section we present some empirical results to quantify some of the performance advantages of this model over other alternatives under current PowerPC hardware.

6.1. Overhead of ordering constraints

In this experiment we create a simple loop that iterates 10 billion times performing a single atomic memory operation. We generated the sequences necessary to implement different ordering constraints on this memory operation. The code was compiled with basic optimization on, but the atomic variable was made volatile, to prevent any memory operations from being removed by the compiler.



This experiment shows the overhead of the ordering constraints on atomic operations. The large overhead of these primitives is part of the motivation for providing a fine granularity of ordering constraints. They will allow the user to specify the precise ordering requirements of his algorithm, and avoid unnecessary ordering constraints and their negative effect in performance.

Making atomic operations follow sequential consistency will also cause additional ordering constraints to be introduced. Basically, it will increase the overhead of all atomic operations at least to the level of ordered operations. This is the rationale for the model not providing sequential consistency in the presence of acquire, release or raw operations.

7. Conclusions

In this paper we have presented a memory model that can be implemented efficiently on weakly ordered machines, and has sufficient expressive power to describe the ordering requirements of a wide variety of parallel algorithms. Since IBM is one of the hardware and software vendors with extensive experience dealing with weakly-ordered shared memory architectures, we believe we are uniquely positioned to provide feedback on improving the memory model for the C++ standard.

This is a prioritized list of differences of this model versus ISOMM:

1. Provide standalone memory fences. This is crucial to avoid the introduction of unnecessary ordering constraints. Our proposal has been to introduce only three forms of ordering constraints,

but an alternative is to include all possible fences (LoadLoad, LoadStore, StoreLoad & StoreStore) to allow exploitation on hardware architectures that provide such primitives.

2. Do not require ordering on atomic operations over what is specified by their ordering constraints. In particular, allow them to be freely reordered within a region bound by an acquire constraint and a release constraint.
3. Allow atomic operations to be removed if they are found to be redundant based on sequential program analysis.
4. Define a mechanism to enable ordering of dependent memory operations, both through control flow or data flow dependences.
5. Define a mechanism to allow ordering of atomic operations without introducing any hardware primitives.

We believe these improvements will increase the flexibility of this memory model and provide greater expressiveness while avoiding unnecessary performance penalties.

Acknowledgements

We would like to recognize and show our gratitude for significant contributions to this paper from Michael Maged, Vijay Saraswat, Christopher von Praun, Kevin Stoodley and Cathy May.

References

[Hennessy] John Hennessy, David Patterson, Computer Architecture, a Quantitative Approach, Second Edition, Morgan Kaufman, 1996.

[ISOMM] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2138.html

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2052.htm

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2047.html

[JSR133C] http://g.oswego.edu/dl/jmm/cookbook.html

[McKenney] http://www.rdrop.com/users/paulmck/scalability/paper/ordering.2006.08.21a.pdf

[McKenney2] Overview of Linux-Kernel Reference Counting N2167=07-0027

[OpenMP2.5] http://www.openmp.org/drupal/mp-documents/spec25.pdf

[PowerPC] http://www-128.ibm.com/developerworks/eserver/articles/archguide.html

[PRISM] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2075.pdf

[XLC] http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/index.jsp