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: N2237=07-0097

Date: 2007-05-06

Project: Programming Language C++, Evolution Working Group

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

Revision: Version 2.00

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 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

This model provides no ordering guarantees on unmarked storage. Ordering guarantees are provided on specific operations on atomic storage.

The guarantees provided are based on the principle that the programmer normally requires is acquire and release operations, that permit cross-thread communication of unmarked storage through signalling on atomic storage.

No implicit ordering is provided by this model, other than what is described on the next section for the specific operations. The only additional principle is that operations on atomic storage must follow cache-consistency; that is, stores to a *single* atomic memory location must be globally ordered, and all threads must observe those stores in a consistent order.

3. Memory ordering rules

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 (ordinary 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

---

[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.

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 after they 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 in the first are performed with respect to any given other thread before any of the operations in the second set are performed with respect to that thread.

3.1. Ordering atomic operations

This model defines three forms of ordering atomic operations.

- An atomic operation with *acquire* semantics ensures that all loads 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 store in set B.
- An atomic operation with *ordered* semantics fully orders the three sets. 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 the meaningful subset of all possible combinations of operations/orderings:

	load	Store	Compare-and-swap[3]
unordered (raw)	YES	YES	YES
acquire	YES	NO	YES
release	NO	YES	YES
ordered	YES	YES	YES

It should be highlighted that according to their definition, *ordered* atomic operations provide two separate ordering guarantees, and it is a proper superset of the union of acquire and release orderings. 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 in 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.

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

---

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

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. 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, allow 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.

3.3. Dependence based ordering

A frequent situation where ordering is required is between a load and a subsequent memory accesses that depends on it. That is, where the address of the memory location accessed by the second operation depends on the value returned by the first operation.

Many architectures provide mechanisms to order such memory accesses with significantly reduced overhead. Thus it makes sense to define a special case of ordering constraint to exploit these mechanisms.

One simple mechanism to enable this operation would be a new ordering constraint, to be applied on loads[4]. We have named this ordering constraint "*dependence_acquire*"; it basically provides a subset of the guarantees provided by the acquire constraint defined on section 3.1. Thus, a trivial implementation of this mechanism would be to implement all dependence_acquire operations as if they were acquire operations.

This new ordering constraint is defined identically to acquire (section 3.1), except that the set C is restricted to the following:

C: Memory operations following the atomic operation in program order, whose *effective* address depends directly on the value returned by the load operation.

An important concern that has been voiced previously against introducing such a mechanism on the standard refers to the potential for compiler optimizations to hide dependences from the hardware.

There are several possible ways to deal with these concerns:

---

[4] Doug Lea and Cliff Click have proposed a similar mechanism for inclusion on the HotSpot JVM. Their mechanism is a fence (postLoadperObjectFence) which identifies the variable that contains the address returned by the first memory operation.

- Do not deal with this issue on the standard. An implementation could choose to avoid performing optimizations that can potentially break dependences, upgrade all dependence_acquire operations to acquire operations, or selectively do so when static analysis cannot guarantee that all dependences are maintained. Implementations could also introduce non-portable mechanisms, such as compilation flags, to override this behavior and allow potential dependence-breaking optimizations to occur freely.
- Provide mechanisms to allow the programmer to indicate which dependences must be maintained by the compilation subsystem. This would allow the compiler to break unmarked dependences without regard for their impact to dependence-based ordering constraints. One such proposal is to introduce a separate storage class modifier, which would ensure that optimizations do not break dependences implied by accesses to this variable. Such a mechanism would be simple to implement by optimizing compilers by following a subset of the current behavior for volatiles.
- Exclude from the set C operations whose address cannot be affected by the value loaded by the first operation. This would allow the compilation system to optimize away computations that are canceling, such as addressing of the form `array[x-x]`.

On section 6.5 we further discuss some specific snippets of code (taken from N2176), and how they could be addressed by implementations.

4. 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 unnecessary 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. This applies to both ordinary and atomic loads.
- 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 ordinary 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.

5. 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 proposal.

5.1. 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 in this case 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.

One alternative that has been proposed under the current ISOMM model for this example is to use a dummy load_acquire of the mailbox in place of the acquire fence. It is unclear whether that would be sufficient under the current model, since its intent is to allow the compiler to fully eliminate dead acquire loads. In any case, the need of such dummy acquire loads goes against the programmability and teachability goals of the memory model.

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 that would be disallowed by the ordering constraints implied by the load_acquire operation.

5.2. 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[5]:

```
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.

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

---

[5] 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.

```
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. Similar to the mailbox example in section 5.1, one alternative that has been proposed under the current ISOMM model is to use a load_acquire of ref_count in place of the acquire fence. It is unclear whether that would be sufficient under the current model, since its intent is to allow the compiler to fully eliminate dead acquire loads. Again, the need of such dummy acquire loads certainly goes against the programmability and teachability goals of the memory model.

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 ordering granularity of the fence version of this program, it is possible for the hardware or compiler to speculate those loads for all threads except the last one.

6. 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 in 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].

6.1. 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 may not be possible under a sequentially-consistent execution. This is a feature, not a bug. To see this, imagine that "mycounter" was tracking the total number 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.

6.2. 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.

Additional examples of Linux-kernel RCU use are shown in Section 6.5.

6.3. 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, consider code running in a given thread that must interact with an interrupt or exception handler which 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 Non-Maskable Interrupt (NMI) based profiling. Such profiling might 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 characterized 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.

6.4. Additional Linux-Kernel RCU Use Cases

This section summarizes read-side RCU use cases from a memory-dependency viewpoint. In all cases, the general read-side pattern is as follows:

```
rcu_read_lock();
do_something();
p = rcu_dereference(gp);
do_something_with(p);
rcu_read_unlock();
```

The Linux 2.6.20 kernel uses six generic patterns of memory dependency in its read-side RCU critical sections as follows:

Pattern	# Uses	Expansion of do_something_with()
field	229	Dereferences a field: p->b
field-list	49	Traverses multiple structures: p->q->a
refcnt	47	Acquires a reference: atomic_inc(&p->refcnt)
field-array	38	Selects a field that is an array: p->a[i]
lock	10	Acquires a per-structure lock: spin_lock(&p->lock)
case	6	Casts the pointer: ((struct foo*)p)->f

The number of uses column sums to more than the number of rcu_dereference() primitives in the kernel due to the fact that some RCU read-side critical sections combine multiple patterns.

Each pattern is described in one of the sections below.

6.4.1. Pattern “field”

This is the canonical use of RCU. Insertion is performed as follows:

```
p1->a = 1;
smb_wmb();
gp = p1;
```

Reading is performed as follows:

```
rcu_read_lock();
p = rcu_dereference(gp);
x = p->a;
rcu_read_unlock();
```

Note that the following pattern covers linked lists of RCU-protected elements:

```
rcu_read_lock();
count = 0;
p = rcu_dereference(gp);
while (p != NULL) {
	count++;
	p = rcu_dereference(p->next);
}
rcu_read_unlock();
```

The key difference between this linked-list traversal and the +field-list pattern covered below is that the above requires an rcu_dereference on each step through the list.

6.4.2. Pattern “field-list”

This quite similar to the +field pattern, the only difference is that a multi-linked structure is treated as a single object from RCU's viewpoint. Insertion is performed as follows:

```
p1->a = 1;
q1->b = 1;
p1->q = q1;
smb_wmb();
gp = p1;
```

Reading is performed as follows:

```
rcu_read_lock();
p = rcu_dereference(gp);
x = p->q->b;
rcu_read_unlock();
```

Because the two structures are initialized and inserted as a unit, rcu_dereference() is required only on the first pointer traversal into the multi-struct object.

6.4.3. Pattern “refcnt”

Reference counts are often used to protect slow paths. For example, consider an oversimplified cache that must be refilled by a function that can contain quiescent states:

```
struct cache {
	struct cache *next;
	int value;
	atomic_t refcnt;
} *head

int peek(void) {  /* peek function */
	struct cache *p;

	rcu_read_lock();
	p = rcu_dereference(head);
	if (p->value == 0) {
		atomic_inc(&p->refcnt);
		rcu_read_unlock();
		refill_cache(); /* too slow for RCU readers */
		rcu_read_lock();
		if (atomic_dec_and_test(&p->refcnt)) {
			rcu_read_unlock();
			cache_free(head, p);
			return 0;
		}
	}
	rcu_read_unlock();
	return p->value;
}
```

6.4.4. Pattern “field-array”

RCU-protected arrays are used to implement hash tables, tries, and, more recently, B-trees in the Linux kernel. In all current cases, the size of the array is either non-unity or unknown to the compiler.

```
struct mapped_to_object {
	int data;
};

struct bucket {
	int size;
	struct mapped_to_object *p[0];
} *entries;

int lookup(int id) { /* lookup function */
	struct bucket *ep;
	struct mapped_to_object *q;
	int value;

	rcu_read_lock();
	ep = rcu_dereference(entries);
	q = rcu_dereference(ep[id]);
	if (q == NULL)
		value = -1;
	else
		value = q->data;
	rcu_read_unlock();
	return value;
}
```

Note that the first rcu_dereference() is needed only if the hash table can grow or shrink. The second rcu_dereference() is needed only if entries can be added to a hash table without simultaneously growing or shrinking it.

6.4.5. Pattern “lock”

This example combines array access and locking, as is done in the Linux kernel's System V IPC code.

```
struct mapped_to_object {
	spinlock_t lock;
	int data;
};

struct bucket {
	int size;
	struct mapped_to_object *p[0];
} *entries;

struct mapped_to_object lookup_lock(int id) { /* lookup function */
	struct bucket *ep;
	struct mapped_to_object *q;

	rcu_read_lock();
	ep = rcu_dereference(entries);
	q = rcu_dereference(ep[id]);
	if (q == NULL) {
		rcu_read_unlock();
		return NULL;
	}
	spin_lock(&q->lock);
	rcu_read_unlock();
	return q;
}

void lookup_release(struct mapped_to_object *q) { /* release function */
	spin_unlock(&q->lock);
}
```

Again, the first rcu_dereference() is needed only if the hash table can grow or shrink. The second rcu_dereference() is needed only if entries can be added to a hash table without simultaneously growing or shrinking it.

6.4.6. Pattern "cast"

Casting is used in the Linux kernel where a container structure is used for a variety of different types of elements. The primary example is the forwarding information base trie, which is used in the networking protocol stacks. This usage of casting is straightforward, so no example is given.

6.5. RCU Dependency Ordering Summary

Dependency ordering is used heavily by RCU within the Linux kernel. It might be tempting to dismiss this as specific to operating-system kernels, but the fact is that the core kernel code (excluding device drivers and architecture-specific functionality such as booting) faces many of the same concerns as do multithreaded applications and middleware. In addition, the ordering provided by dependencies is highly intuitive, as one of the authors (Paul) can attest, having been the one informing several groups of the fact that Alpha does not respect dependency-based ordering.

We therefore cannot in good conscience ignore the need for dependency-based ordering.

6.5.1. Dependency-Ordering Examples From N2176

N2176's first example presents simple ordering of a dereference operation:

```
r1 = x.load_dependent_acquire();
r2 = *r1;
```

Here the dependency chain begins with a load_dependent_acquire() member function and is entirely contained within the same function scope, so an implementation could determine statically all the dependent uses and avoid full acquire semantics.

N2176's second example generates an artificial data dependency in order to force ordering:

```
r1 = x.load_dependent_acquire();
r3 = &a + r1 - r1;
r2 = *r3;
```

N2176 notes that standard optimizations would result in the following, which would violate the dependency:

```
r1 = x.load_dependent_acquire();
r3 = &a;
r2 = *r3;
```

However, the fact that this dependency chain begins with a load_dependent_acquire() would prohibit this optimization in this case. Contrast this with the following, where “a” is a non-atomic variable:

```
r1 = x.load_dependent_acquire();
r3 = &a + r1 - r1 + a - a;
r2 = *r3;
```

The compiler would be permitted to optimize away the “a-a”, but not the “r1-r1”. If the programmer wishes the “r1-r1” to be optimized away, the aforementioned intrinsic that marks the end of a dependency chain could be provided.

N2176's third example shows that an innocent-seeming transformation might convert a dependency chain that would be recognized by a given system into a form that might not be:

```
r1 = x.load_dependent_acquire();
if (r1 == 0)
  r2 = *r1;
else
  r2 = *(r1 + 1);
```

The innocent transformation might result in the following:

```
r1 = x.load_dependent_acquire();
if (r1 == 0)
  r3 = r1;
else
  r3 = r1 + 1;
r2 = *r3;
```

The authors of N2176 do not specify what hardware recognizes the first but not the second dependency chain, but assuming that there is such hardware, either the optimization must be prohibited or the backend for the offending machine must either emit explicit memory-barrier instructions to enforce the needed ordering or manufacture the dependencies required by the hardware. This decision is left in the hands of the compiler writers, who would presumably consider how common the offending hardware was and how useful the optimization in question was. If desired, the manufacturer of the offending system could provide tools or documentation to help locate sections of code that exceeds the system's ability to track dependencies.

N2176's fourth example concerns duplicate code, as might be produced by inline functions or C-preprocessor macros:

```
r1 = x.load_dependent_acquire();
if (r1) {
    r2 = y.a;
} else {
    r2 = y.a;
}
```

In absence of the load_dependent_acquire() member function, compiler might reasonably collapse the above as follows, losing the dependency:

```
r1 = x.load_dependent_acquire();
r2 = y.a;
```

The load_dependent_acquire() member function prohibits this transformation, or, alternatively, causes the backend to emit an explicit memory barrier in order to enforce the ordering.

N2176's fifth example also concerns duplicate code, but where the dependency chain spans compilation-unit boundaries:

```
r1 = x.load_dependent_acquire();
if (r1) {
    f(&y);
} else {
    g(&y);
}
```

In this case, the dependency chain will be preserved only if the declarations and definitions of f() and/or g() specify dependency-preserving attributes, in which case the compiler will know to preserve dependencies when compiling and when invoking f() and/or g(). This same analysis applies to the examples given in the "why this seriously breaks optimizations" section in N2176.

N2176's sixth example concerns optimizations explicitly for the purpose of breaking dependency chains:

```
r2 = x.load_dependent_acquire();
r3 = r2 -> a;
```

However, the fact that the dependency chain begins with a load_dependent_acquire() member function prohibits the problematic optimization to the following:

```
r2 = x.load_dependent_acquire();
r3 = r1 -> a;
if (r1 != r2) r3 = r2 -> a;
```

As before, tagging function declarations and definitions with dependency-preserving attributes provides the compiler the information that it needs to correctly compile dependency chains that span compilation-unit boundaries.

N2176's seventh example concerns accesses to single-element arrays:

```
r1 = x.load_dependent_acquire();
r2 = a[r1 -> index % a_size];
```

If a_size is known to the compiler to be zero, this could be optimized to the following, destroying the dependency chain:

```
r1 = x.load_dependent_acquire();
r2 = a[0];
```

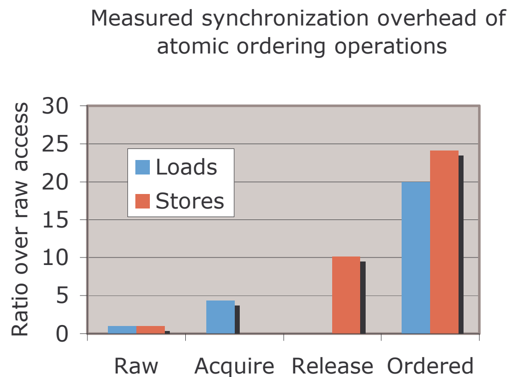
However, the load_dependent_acquire() member function would prohibit this optimization (or, alternatively, require that an explicit memory barrier be supplied between the two optimized statements).

7. 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.

7.1. 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 enabled, 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.

8. 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 full reordering of raw atomic operations, and allow load-acquire operations to be reordered ahead of preceding store-release operations.
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 dependencies.
5. Define a mechanism to allow ordering of atomic operations without introducing any hardware primitives.

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

9. 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.

10. 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

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