Boost.Atomic

Helge Bahmann

Table of Contents

Introduction

Presenting Boost.Atomic

Boost.Atomic is a library that provides `atomic` data types and operations on these data types, as well as memory ordering constraints required for coordinating multiple threads through atomic variables. It implements the interface as defined by the C++11 standard, but makes this feature available for platforms lacking system/compiler support for this particular C++11 feature.

Users of this library should already be familiar with concurrency in general, as well as elementary concepts such as "mutual exclusion".

The implementation makes use of processor-specific instructions where possible (via inline assembler, platform libraries or compiler intrinsics), and falls back to "emulating" atomic operations through locking.

Purpose

Operations on "ordinary" variables are not guaranteed to be atomic. This means that with `int n=0` initially, two threads concurrently executing

```
void function()
{
  n ++;
}
```

might result in `n==1` instead of 2: Each thread will read the old value into a processor register, increment it and write the result back. Both threads may therefore write `1`, unaware that the other thread is doing likewise.

Declaring `atomic<int> n=0` instead, the same operation on this variable will always result in `n==2` as each operation on this variable is *atomic*: This means that each operation behaves as if it were strictly sequentialized with respect to the other.

Atomic variables are useful for two purposes:

- as a means for coordinating multiple threads via custom coordination protocols
- as faster alternatives to "locked" access to simple variables

Take a look at the examples section for common patterns.

Thread coordination using Boost.Atomic

The most common use of **Boost.Atomic** is to realize custom thread synchronization protocols: The goal is to coordinate accesses of threads to shared variables in order to avoid "conflicts". The programmer must be aware of the fact that compilers, CPUs and the cache hierarchies may generally reorder memory references at will. As a consequence a program such as:

int x = 0, int y = 0;

```
thread1:
  x = 1;
  y = 1;

thread2
  if (y == 1) {
    assert(x == 1);
  }
```

might indeed fail as there is no guarantee that the read of x by thread2 "sees" the write by thread1.

Boost.Atomic uses a synchronisation concept based on the *happens-before* relation to describe the guarantees under which situations such as the above one cannot occur.

The remainder of this section will discuss *happens-before* in a "hands-on" way instead of giving a fully formalized definition. The reader is encouraged to additionally have a look at the discussion of the correctness of a few of the examples afterwards.

Enforcing *happens-before* through mutual exclusion

As an introductory example to understand how arguing using *happens-before* works, consider two threads synchronizing using a common mutex:

```
mutex m;

thread1:
  m.lock();
  ... /* A */
  m.unlock();

thread2:
  m.lock();
  ... /* B */
  m.unlock();
```

The "lockset-based intuition" would be to argue that A and B cannot be executed concurrently as the code paths require a common lock to be held.

One can however also arrive at the same conclusion using *happens-before*: Either thread1 or thread2 will succeed first at `m.lock()`. If this is be thread1, then as a consequence, thread2 cannot succeed at `m.lock()` before thread1 has executed `m.unlock()`, consequently A *happens-before* B in this case. By symmetry, if thread2 succeeds at `m.unlock()` first, we can conclude B *happens-before* A.

Since this already exhausts all options, we can conclude that either A *happens-before* B or B *happens-before* A must always hold. Obviously cannot state *which* of the two relationships holds, but either one is sufficient to conclude that A and B cannot conflict.

Compare the spinlock implementation to see how the mutual exclusion concept can be mapped to **Boost.Atomic**.

happens-before through `release` and `acquire`

The most basic pattern for coordinating threads via **Boost.Atomic** uses `release` and `acquire` on an atomic variable for coordination: If ...

- ... thread1 performs an operation A,
- ... thread1 subsequently writes (or atomically modifies) an atomic variable with `release` semantic,
- ... thread2 reads (or atomically reads-and-modifies) the value this value from the same atomic variable with `acquire` semantic and
- ... thread2 subsequently performs an operation B,

... then A *happens-before* B.

Consider the following example

```
atomic<int> a(0);

thread1:
  ... /* A */
  a.fetch_add(1, memory_order_release);

thread2:
  int tmp = a.load(memory_order_acquire);
  if (tmp == 1) {
    ... /* B */
  } else {
    ... /* C */
  }
```

In this example, two avenues for execution are possible:

- The `store` operation by thread1 precedes the `load` by thread2: In this case thread2 will execute B and "A *happens-before* B" holds as all of the criteria above are satisfied.
- The `load` operation by thread2 precedes the `store` by thread1: In this case, thread2 will execute C, but "A *happens-before* C" does *not* hold: thread2 does not read the value written by thread1 through `a`.

Therefore, A and B cannot conflict, but A and C *can* conflict.

Fences

Ordering constraints are generally specified together with an access to an atomic variable. It is however also possible to issue "fence" operations in isolation, in this case the fence operates in conjunction with preceding (for `acquire`, `consume` or `seq_cst` operations) or succeeding (for `release` or `seq_cst`) atomic operations.

The example from the previous section could also be written in the following way:

```
atomic<int> a(0);

thread1:
  ... /* A */
  atomic_thread_fence(memory_order_release);
  a.fetch_add(1, memory_order_relaxed);

thread2:
  int tmp = a.load(memory_order_relaxed);
  if (tmp == 1) {
    atomic_thread_fence(memory_order_acquire);
    ... /* B */
  } else {
    ... /* C */
  }
```

This provides the same ordering guarantees as previously, but elides a (possibly expensive) memory ordering operation in the case C is executed.

happens-before through `release` and `consume`

The second pattern for coordinating threads via **Boost.Atomic** uses `release` and `consume` on an atomic variable for coordination: If ...

- ... thread1 performs an operation A,
- ... thread1 subsequently writes (or atomically modifies) an atomic variable with `release` semantic,
- ... thread2 reads (or atomically reads-and-modifies) the value this value from the same atomic variable with `consume` semantic and
- ... thread2 subsequently performs an operation B that is *computationally dependent on the value of the atomic variable*,

... then A *happens-before* B.

Consider the following example

```
atomic<int> a(0);
complex_data_structure data[2];

thread1:
  data[1] = ...; /* A */
  a.store(1, memory_order_release);

thread2:
  int index = a.load(memory_order_consume);
  complex_data_structure tmp = data[index]; /* B */
```

In this example, two avenues for execution are possible:

- The `store` operation by thread1 precedes the `load` by thread2: In this case thread2 will read `data[1]` and "A *happens-before* B" holds as all of the criteria above are satisfied.
- The `load` operation by thread2 precedes the `store` by thread1: In this case thread2 will read `data[0]` and "A *happens-before* B" does *not* hold: thread2 does not read the value written by thread1 through `a`.

Here, the *happens-before* relationship helps ensure that any accesses (presumable writes) to `data[1]` by thread1 happen before before the accesses (presumably reads) to `data[1]` by thread2: Lacking this relationship, thread2 might see stale/inconsistent data.

Note that in this example, the fact that operation B is computationally dependent on the atomic variable, therefore the following program would be erroneous:

```
atomic<int> a(0);
complex_data_structure data[2];

thread1:
  data[1] = ...; /* A */
  a.store(1, memory_order_release);

thread2:
  int index = a.load(memory_order_consume);
  complex_data_structure tmp;
  if (index == 0)
    tmp = data[0];
  else
    tmp = data[1];
```

`consume` is most commonly (and most safely! see limitations) used with pointers, compare for example the singleton with double-checked locking.

Sequential consistency

The third pattern for coordinating threads via **Boost.Atomic** uses `seq_cst` for coordination: If ...

- ... thread1 performs an operation A,
- ... thread1 subsequently performs any operation with `seq_cst`,
- ... thread1 subsequently performs an operation B,
- ... thread2 performs an operation C,
- ... thread2 subsequently performs any operation with `seq_cst`,
- ... thread2 subsequently performs an operation D,

then either "A *happens-before* D" or "C *happens-before* B" holds.

In this case it does not matter whether thread1 and thread2 operate on the same or different atomic variables, or use a "stand-alone" `atomic_thread_fence` operation.

Programming interfaces

Memory order

The enumeration `boost::memory_order` defines the following values to represent memory ordering constraints:

Constant	Description
`memory_order_relaxed`	No ordering constraint. Informally speaking, following operations may be reordered before, preceding operations may be reordered after the atomic operation. This constraint is suitable only when either a) further operations do not depend on the outcome of the atomic operation or b) ordering is enforced through stand-alone `atomic_thread_fence` operations
`memory_order_release`	Perform `release` operation. Informally speaking, prevents all preceding memory operations to be reordered past this point.
`memory_order_acquire`	Perform `acquire` operation. Informally speaking, prevents succeeding memory operations to be reordered before this point.
`memory_order_consume`	Perform `consume` operation. More restrictive (and usually more efficient) than `memory_order_acquire` as it only affects succeeding operations that are computationally-dependent on the value retrieved from an atomic variable.
`memory_order_acq_rel`	Perform both `release` and `acquire` operation
`memory_order_seq_cst`	Enforce sequential consistency. Implies `memory_order_acq_rel`, but additional enforces total order for all operations such qualified.

See section *happens-before* for explanation of the various ordering constraints.

Atomic objects

`boost::atomic<T>` provides methods for atomically accessing variables of a suitable type *T*. The type is suitable if it satisfies one of the following constraints:

- it is an integer, boolean, enum or pointer type
- it is any other data-type (`class` or `struct`) that has a non-throwing default constructor, that is copyable via `memcpy` and comparable via `memcmp`.

Note that all classes having a trivial default constructor, no destructor and no virtual methods satisfy the second condition according to C++98. On a given platform, other data-types *may* also satisfy this constraint, however you should exercise caution as the behaviour becomes implementation-defined. Also be warned that structures with "padding" between data members may compare non-equal via `memcmp` even though all members are equal.

`boost::atomic<T>` template class

All atomic objects supports the following operations:

Syntax	Description
`atomic()`	Initialize to an unspecified value
`atomic(T initial_value)`	Initialize to `initial_value`
`bool is_lock_free()`	Checks if the atomic object is lock-free
`T load(memory_order order)`	Return current value
`void store(T value, memory_order order)`	Write new value to atomic variable
`T exchange(T new_value, memory_order order)`	Exchange current value with `new_value`, returning current value
`bool compare_exchange_weak(T & expected, T desired, memory_order order)`	Compare current value with `expected`, change it to `desired` if matches. Returns `true` if an exchange has been performed, and always writes the previous value back in `expected`. May fail spuriously, so must generally be retried in a loop.
`bool compare_exchange_weak(T & expected, T desired, memory_order success_order, memory_order failure_order)`	Compare current value with `expected`, change it to `desired` if matches. Returns `true` if an exchange has been performed, and always writes the previous value back in `expected`. May fail spuriously, so must generally be retried in a loop.
`bool compare_exchange_strong(T & expected, T desired, memory_order order)`	Compare current value with `expected`, change it to `desired` if matches. Returns `true` if an exchange has been performed, and always writes the previous value back in `expected`.
`bool compare_exchange_strong(T & expected, T desired, memory_order success_order, memory_order failure_order))`	Compare current value with `expected`, change it to `desired` if matches. Returns `true` if an exchange has been performed, and always writes the previous value back in `expected`.

`order` always has `memory_order_seq_cst` as default parameter.

The `compare_exchange_weak`/`compare_exchange_strong` variants taking four parameters differ from the three parameter variants in that they allow a different memory ordering constraint to be specified in case the operation fails.

In addition to these explicit operations, each `atomic<`*`T`*`>` object also supports implicit `store` and `load` through the use of "assignment" and "conversion to `T`" operators. Avoid using these operators, as they do not allow explicit specification of a memory ordering constraint.

`boost::atomic<`*`integral`*`>` template class

In addition to the operations listed in the previous section, `boost::atomic<`*`I`*`>` for integral types *`I`* supports the following operations:

Syntax	Description
`T fetch_add(T v, memory_order order)`	Add `v` to variable, returning previous value
`T fetch_sub(T v, memory_order order)`	Subtract `v` from variable, returning previous value
`T fetch_and(T v, memory_order order)`	Apply bit-wise "and" with `v` to variable, returning previous value
`T fetch_or(T v, memory_order order)`	Apply bit-wise "or" with `v` to variable, returning previous value
`T fetch_xor(T v, memory_order order)`	Apply bit-wise "xor" with `v` to variable, returning previous value

`order` always has `memory_order_seq_cst` as default parameter.

In addition to these explicit operations, each `boost::atomic<I>` object also supports implicit pre-/post- increment/decrement, as well as the operators `+=`, `-=`, `&=`, `|=` and `^=`. Avoid using these operators, as they do not allow explicit specification of a memory ordering constraint.

`boost::atomic<pointer>` template class

In addition to the operations applicable to all atomic object, `boost::atomic<P>` for pointer types `P` (other than `void` pointers) support the following operations:

Syntax	**Description**
`T fetch_add(ptrdiff_t v, memory_order order)`	Add `v` to variable, returning previous value
`T fetch_sub(ptrdiff_t v, memory_order order)`	Subtract `v` from variable, returning previous value

`order` always has `memory_order_seq_cst` as default parameter.

In addition to these explicit operations, each `boost::atomic<P>` object also supports implicit pre-/post- increment/decrement, as well as the operators `+=`, `-=`. Avoid using these operators, as they do not allow explicit specification of a memory ordering constraint.

Fences

Syntax	**Description**
`void atomic_thread_fence(memory_order order)`	Issue fence for coordination with other threads.
`void atomic_signal_fence(memory_order order)`	Issue fence for coordination with signal handler (only in same thread).

Feature testing macros

Boost.Atomic defines a number of macros to allow compile-time detection whether an atomic data type is implemented using "true" atomic operations, or whether an internal "lock" is used to provide atomicity. The following macros will be defined to `0` if operations on the data type always require a lock, to `1` if operations on the data type may sometimes require a lock, and to `2` if they are always lock-free:

Macro	Description
`BOOST_ATOMIC_CHAR_LOCK_FREE`	Indicate whether `atomic<char>` (including signed/unsigned variants) is lock-free
`BOOST_ATOMIC_SHORT_LOCK_FREE`	Indicate whether `atomic<short>` (including signed/unsigned variants) is lock-free
`BOOST_ATOMIC_INT_LOCK_FREE`	Indicate whether `atomic<int>` (including signed/unsigned variants) is lock-free
`BOOST_ATOMIC_LONG_LOCK_FREE`	Indicate whether `atomic<long>` (including signed/unsigned variants) is lock-free
`BOOST_ATOMIC_LLONG_LOCK_FREE`	Indicate whether `atomic<long long>` (including signed/unsigned variants) is lock-free
`BOOST_ATOMIC_ADDRESS_LOCK_FREE`	Indicate whether `atomic<T *>` is lock-free

Usage examples

Reference counting

The purpose of a *reference counter* is to count the number of pointers to an object. The object can be destroyed as soon as the reference counter reaches zero.

Implementation

```
#include <boost/intrusive_ptr.hpp>
#include <boost/atomic.hpp>

class X {
public:
  typedef boost::intrusive_ptr<X> pointer;
  X() : refcount_(0) {}

private:
  mutable boost::atomic<int> refcount_;
  friend void intrusive_ptr_add_ref(const X * x)
  {
    x->refcount_.fetch_add(1, boost::memory_order_relaxed);
  }
  friend void intrusive_ptr_release(const X * x)
  {
    if (x->refcount_.fetch_sub(1, boost::memory_order_release) == 1) {
      boost::atomic_thread_fence(boost::memory_order_acquire);
      delete x;
    }
  }
};
```

Usage

```
X::pointer x = new X;
```

Discussion

Increasing the reference counter can always be done with `memory_order_relaxed`: New references to an object can only be formed from an existing reference, and passing an existing reference from one thread to another must already provide any required synchronization.

It is important to enforce any possible access to the object in one thread (through an existing reference) to *happen before* deleting the object in a different thread. This is achieved by a "release" operation after dropping a reference (any access to the object through this reference must obviously happened before), and an "acquire" operation before deleting the object.

It would be possible to use `memory_order_acq_rel` for the `fetch_sub` operation, but this results in unneeded "acquire" operations when the reference counter does not yet reach zero and may impose a performance penalty.

Spinlock

The purpose of a *spin lock* is to prevent multiple threads from concurrently accessing a shared data structure. In contrast to a mutex, threads will busy-wait and waste CPU cycles instead of yielding the CPU to another thread. *Do not use spinlocks unless you are certain that you understand the consequences.*

Implementation

```
#include <boost/atomic.hpp>

class spinlock {
private:
  typedef enum {Locked, Unlocked} LockState;
  boost::atomic<LockState> state_;

public:
  spinlock() : state_(Unlocked) {}

  void lock()
  {
    while (state_.exchange(Locked, boost::memory_order_acquire) == Locked) {
      /* busy-wait */
    }
  }
  void unlock()
  {
    state_.store(Unlocked, boost::memory_order_release);
  }
};
```

Usage

```
spinlock s;

s.lock();
// access data structure here
s.unlock();
```

Discussion

The purpose of the spinlock is to make sure that one access to the shared data structure always strictly "happens before" another. The usage of acquire/release in lock/unlock is required and sufficient to guarantee this ordering.

It would be correct to write the "lock" operation in the following way:

```
lock()
{
  while (state_.exchange(Locked, boost::memory_order_relaxed) == Locked) {
    /* busy-wait */
  }
  atomic_thread_fence(boost::memory_order_acquire);
}
```

This "optimization" is however a) useless and b) may in fact hurt: a) Since the thread will be busily spinning on a blocked spinlock, it does not matter if it will waste the CPU cycles with just "exchange" operations or with both useless "exchange" and "acquire" operations. b) A tight "exchange" loop without any memory-synchronizing instruction introduced through an "acquire" operation will on some systems monopolize the memory subsystem and degrade the performance of other system components.

Singleton with double-checked locking pattern

The purpose of the *Singleton with double-checked locking pattern* is to ensure that at most one instance of a particular object is created. If one instance has been created already, access to the existing object should be as light-weight as possible.

Implementation

```
#include <boost/atomic.hpp>
#include <boost/thread/mutex.hpp>

class X {
public:
  static X * instance()
  {
    X * tmp = instance_.load(boost::memory_order_consume);
    if (!tmp) {
      boost::mutex::scoped_lock guard(instantiation_mutex);
      tmp = instance_.load(boost::memory_order_consume);
      if (!tmp) {
        tmp = new X;
        instance_.store(tmp, boost::memory_order_release);
      }
    }
    return tmp;
  }
private:
  static boost::atomic<X *> instance_;
  static boost::mutex instantiation_mutex;
};

boost::atomic<X *> X::instance_(0);
```

Usage

```
X * x = X::instance();
// dereference x
```

Discussion

The mutex makes sure that only one instance of the object is ever created. The `instance` method must make sure that any dereference of the object strictly "happens after" creating the instance in another thread. The use of `memory_order_release` after creating and initializing the object and `memory_order_consume` before dereferencing the object provides this guarantee.

It would be permissible to use `memory_order_acquire` instead of `memory_order_consume`, but this provides a stronger guarantee than is required since only operations depending on the value of the pointer need to be ordered.

Wait-free ring buffer

A *wait-free ring buffer* provides a mechanism for relaying objects from one single "producer" thread to one single "consumer" thread without any locks. The operations on this data structure are "wait-free" which means that each operation finishes within a constant number of steps. This makes this data structure suitable for use in hard real-time systems or for communication with interrupt/signal handlers.

Implementation

```
#include <boost/atomic.hpp>

template<typename T, size_t Size>
class ringbuffer {
public:
  ringbuffer() : head_(0), tail_(0) {}

  bool push(const T & value)
  {
    size_t head = head_.load(boost::memory_order_relaxed);
    size_t next_head = next(head);
    if (next_head == tail_.load(boost::memory_order_acquire))
      return false;
    ring_[head] = value;
    head_.store(next_head, boost::memory_order_release);
    return true;
  }
  bool pop(T & value)
  {
    size_t tail = tail_.load(boost::memory_order_relaxed);
    if (tail == head_.load(boost::memory_order_acquire))
      return false;
    value = ring_[tail];
    tail_.store(next(tail), boost::memory_order_release);
    return true;
  }
private:
  size_t next(size_t current)
  {
    return (current + 1) % Size;
  }
  T ring_[Size];
  boost::atomic<size_t> head_, tail_;
};
```

Usage

```
ringbuffer<int, 32> r;

// try to insert an element
if (r.push(42)) { /* succeeded */ }
else { /* buffer full */ }

// try to retrieve an element
int value;
if (r.pop(value)) { /* succeeded */ }
else { /* buffer empty */ }
```

Discussion

The implementation makes sure that the ring indices do not "lap-around" each other to ensure that no elements are either lost or read twice.

Furthermore it must guarantee that read-access to a particular object in `pop` "happens after" it has been written in `push`. This is achieved by writing `head_` with "release" and reading it with "acquire". Conversely the implementation also ensures that read access to a particular ring element "happens before" before rewriting this element with a new value by accessing `tail_` with appropriate ordering constraints.

Wait-free multi-producer queue

The purpose of the *wait-free multi-producer queue* is to allow an arbitrary number of producers to enqueue objects which are retrieved and processed in FIFO order by a single consumer.

Implementation

```
template<typename T>
class waitfree_queue {
public:
  struct node {
    T data;
    node * next;
  };
  void push(const T &data)
  {
    node * n = new node;
    n->data = data;
    node * stale_head = head_.load(boost::memory_order_relaxed);
    do {
      n->next = stale_head;
    } while (!head_.compare_exchange_weak(stale_head, n, boost::memory_order_release));
  }

  node * pop_all(void)
  {
    T * last = pop_all_reverse(), * first = 0;
    while(last) {
      T * tmp = last;
      last = last->next;
      tmp->next = first;
      first = tmp;
    }
    return first;
  }

  waitfree_queue() : head_(0) {}

  // alternative interface if ordering is of no importance
  node * pop_all_reverse(void)
  {
    return head_.exchange(0, boost::memory_order_consume);
  }
private:
  boost::atomic<node *> head_;
};
```

Usage

```
waitfree_queue<int> q;

// insert elements
q.push(42);
q.push(2);

// pop elements
waitfree_queue<int>::node * x = q.pop_all()
while(x) {
  X * tmp = x;
  x = x->next;
  // process tmp->data, probably delete it afterwards
  delete tmp;
}
```

Discussion

The implementation guarantees that all objects enqueued are processed in the order they were enqueued by building a singly-linked list of object in reverse processing order. The queue is atomically emptied by the consumer and brought into correct order.

It must be guaranteed that any access to an object to be enqueued by the producer "happens before" any access by the consumer. This is assured by inserting objects into the list with *release* and dequeuing them with *consume* memory order. It is not necessary to use *acquire* memory order in `waitfree_queue::pop_all` because all operations involved depend on the value of the atomic pointer through dereference

Limitations

While **Boost.Atomic** strives to implement the atomic operations from C++11 as faithfully as possible, there are a few limitations that cannot be lifted without compiler support:

- **Using non-POD-classes as template parameter to `atomic<T>` results in undefined behavior**: This means that any class containing a constructor, destructor, virtual methods or access control specifications is not a valid argument in C++98. C++11 relaxes this slightly by allowing "trivial" classes containing only empty constructors. **Advise**: Use only POD types.
- **C++98 compilers may transform computation- to control-dependency**: Crucially, `memory_order_consume` only affects computationally-dependent operations, but in general there is nothing preventing a compiler from transforming a computation dependency into a control dependency. A C++11 compiler would be forbidden from such a transformation. **Advise**: Use `memory_order_consume` only in conjunction with pointer values, as the compiler cannot speculate and transform these into control dependencies.
- **Fence operations enforce "too strong" compiler ordering**: Semantically, `memory_order_acquire`/`memory_order_consume` and `memory_order_release` need to restrain reordering of memory operations only in one direction. Since there is no way to express this constraint to the compiler, these act as "full compiler barriers" in this implementation. In corner cases this may lead to worse code than a C++11 compiler could generate.
- **No interprocess fallback**: using `atomic<T>` in shared memory only works correctly, if `atomic<T>::is_lock_free == true`

Porting

Unit tests

Boost.Atomic provides a unit test suite to verify that the implementation behaves as expected:

- **fallback_api.cpp** verifies that the fallback-to-locking aspect of **Boost.Atomic** compiles and has correct value semantics.
- **native_api.cpp** verifies that all atomic operations have correct value semantics (e.g. "fetch_add" really adds the desired value, returning the previous). It is a rough "smoke-test" to help weed out the most obvious mistakes (for example with overflow, signed/unsigned extension, ...).
- **lockfree.cpp** verifies that the **BOOST_ATOMIC_*_LOCKFREE** macros are set properly according to the expectations for a given platform, and that they match up with the **is_lock_free** member functions of the **atomic** object instances.
- **atomicity.cpp** lets two threads race against each other modifying a shared variable, verifying that the operations behave atomic as appropriate. By nature, this test is necessarily stochastic, and the test self-calibrates to yield 99% confidence that a positive result indicates absence of an error. This test is very useful on uni-processor systems with preemption already.
- **ordering.cpp** lets two threads race against each other accessing multiple shared variables, verifying that the operations exhibit the expected ordering behavior. By nature, this test is necessarily stochastic, and the test attempts to self-calibrate to yield 99% confidence that a positive result indicates absence of an error. This only works on true multi-processor (or multi-core) systems. It does not yield any result on uni-processor systems or emulators (due to there being no observable reordering even the order=relaxed case) and will report that fact.

Tested compilers

Boost.Atomic has been tested on and is known to work on the following compilers/platforms:

- gcc 4.x: i386, x86_64, ppc32, ppc64, armv5, armv6, alpha
- Visual Studio Express 2008/Windows XP, i386

If you have an unsupported platform, contact me and I will work to add support for it.