LLVM Atomic Instructions and Concurrency Guide

Introduction

LLVM supports instructions which are well-defined in the presence of threads and asynchronous signals.

The atomic instructions are designed specifically to provide readable IR and optimized code generation for the following:

- The C++11 `<atomic>` header. (C++11 draft available here.) (C11 draft available here.)
- Proper semantics for Java-style memory, for both `volatile` and regular shared variables. (Java Specification)
- gcc-compatible `__sync_*` builtins. (Description)
- Other scenarios with atomic semantics, including `static` variables with non-trivial constructors in C++.

Atomic and volatile in the IR are orthogonal; “volatile” is the C/C++ volatile, which ensures that every volatile load and store happens and is performed in the stated order. A couple examples: if a SequentiallyConsistent store is immediately followed by another SequentiallyConsistent store to the same address, the first store can be erased. This transformation is not allowed for a pair of volatile stores. On the other hand, a non-volatile non-atomic load can be moved across a volatile load freely, but not an Acquire load.

This document is intended to provide a guide to anyone either writing a frontend for LLVM or working on optimization passes for LLVM with a guide for how to deal with instructions with special semantics in the presence of concurrency. This is not intended to be a precise guide to the semantics; the details can get extremely complicated and unreadable, and are not usually necessary.

Optimization outside atomic

The basic `'load'` and `'store'` allow a variety of optimizations, but can lead to undefined results in a concurrent environment; see NotAtomic. This section specifically goes into the one optimizer restriction which applies in concurrent environments, which gets a bit more of an extended description because any optimization dealing with stores needs to be aware of it.

From the optimizer's point of view, the rule is that if there are not any instructions with atomic ordering involved, concurrency does not matter, with one exception: if a variable might be visible to another thread or signal handler, a store cannot be inserted along a path where it might not execute otherwise. Take the following example:

```
/* C code, for readability; run through clang -O2 -S -emit-llvm to get
    equivalent IR */
 int x;
 void f(int* a) {
   for (int i = 0; i < 100; i++) {
     if (a[i])
       x += 1;
   }
 }
```

The following is equivalent in non-concurrent situations:

```
int x;
void f(int* a) {
  int xtemp = x;
  for (int i = 0; i < 100; i++) {
    if (a[i])
      xtemp += 1;
  }
  x = xtemp;
}
```

However, LLVM is not allowed to transform the former to the latter: it could indirectly introduce undefined behavior if another thread can access `x` at the same time. (This example is particularly of interest because before the concurrency model was implemented, LLVM would perform this transformation.)

Note that speculative loads are allowed; a load which is part of a race returns `undef`, but does not have undefined behavior.

Atomic instructions

For cases where simple loads and stores are not sufficient, LLVM provides various atomic instructions. The exact guarantees provided depend on the ordering; see Atomic orderings.

`load atomic` and `store atomic` provide the same basic functionality as non-atomic loads and stores, but provide additional guarantees in situations where threads and signals are involved.

`cmpxchg` and `atomicrmw` are essentially like an atomic load followed by an atomic store (where the store is conditional for `cmpxchg`), but no other memory operation can happen on any thread between the load and store.

A `fence` provides Acquire and/or Release ordering which is not part of another operation; it is normally used along with Monotonic memory operations. A Monotonic load followed by an Acquire fence is roughly equivalent to an Acquire load, and a Monotonic store following a Release fence is roughly equivalent to a Release store. SequentiallyConsistent fences behave as both an Acquire and a Release fence, and offer some additional complicated guarantees, see the C++11 standard for details.

Frontends generating atomic instructions generally need to be aware of the target to some degree; atomic instructions are guaranteed to be lock-free, and therefore an instruction which is wider than the target natively supports can be impossible to generate.

Atomic orderings

In order to achieve a balance between performance and necessary guarantees, there are six levels of atomicity. They are listed in order of strength; each level includes all the guarantees of the previous level except for Acquire/Release. (See also LangRef Ordering.)

NotAtomic

NotAtomic is the obvious, a load or store which is not atomic. (This isn't really a level of atomicity, but is listed here for comparison.) This is essentially a regular load or store. If there is a race on a given memory location, loads from that location return undef.

Relevant standard

This is intended to match shared variables in C/C++, and to be used in any other context where memory access is necessary, and a race is impossible. (The precise definition is in LangRef Memory Model.)

Notes for frontends

The rule is essentially that all memory accessed with basic loads and stores by multiple threads should be protected by a lock or other synchronization; otherwise, you are likely to run into undefined behavior. If your frontend is for a "safe" language like Java, use Unordered to load and store any shared variable. Note that NotAtomic volatile loads and stores are not properly atomic; do not try to use them as a substitute. (Per the C/C++ standards, volatile does provide some limited guarantees around asynchronous signals, but atomics are generally a better solution.)

Notes for optimizers

Introducing loads to shared variables along a codepath where they would not otherwise exist is allowed; introducing stores to shared variables is not. See Optimization outside atomic.

Notes for code generation

The one interesting restriction here is that it is not allowed to write to bytes outside of the bytes relevant to a store. This is mostly relevant to unaligned stores: it is not allowed in general to convert an unaligned store into two aligned stores of the same width as the unaligned store. Backends are also expected to generate an i8 store as an i8 store, and not an instruction which writes to surrounding bytes. (If you are writing a backend for an architecture which cannot satisfy these restrictions and cares about concurrency, please send an email to llvm-dev.)

Unordered

Unordered is the lowest level of atomicity. It essentially guarantees that races produce somewhat sane results instead of having undefined behavior. It also guarantees the operation to be lock-free, so it does not depend on the data being part of a special atomic structure or depend on a separate per-process global lock. Note that code generation will fail for unsupported atomic operations; if you need such an operation, use explicit locking.

Relevant standard

This is intended to match the Java memory model for shared variables.

Notes for frontends

This cannot be used for synchronization, but is useful for Java and other “safe” languages which need to guarantee that the generated code never exhibits undefined behavior. Note that this guarantee is cheap on common platforms for loads of a native width, but can be expensive or unavailable for wider loads, like a 64-bit store on ARM. (A frontend for Java or other “safe” languages would normally split a 64-bit store on ARM into two 32-bit unordered stores.)

Notes for optimizers

In terms of the optimizer, this prohibits any transformation that transforms a single load into multiple loads, transforms a store into multiple stores, narrows a store, or stores a value which would not be stored otherwise. Some examples of unsafe optimizations are narrowing an assignment into a bitfield, rematerializing a load, and turning loads and stores into a memcpy call. Reordering unordered operations is safe, though, and optimizers should take advantage of that because unordered operations are common in languages that need them.

Notes for code generation

These operations are required to be atomic in the sense that if you use unordered loads and unordered stores, a load cannot see a value which was never stored. A normal load or store instruction is usually sufficient, but note that an unordered load or store cannot be split into multiple instructions (or an instruction which does multiple memory operations, like `LDRD` on ARM without LPAE, or not naturally-aligned `LDRD` on LPAE ARM).

Monotonic

Monotonic is the weakest level of atomicity that can be used in synchronization primitives, although it does not provide any general synchronization. It essentially guarantees that if you take all the operations affecting a specific address, a consistent ordering exists.

Relevant standard

This corresponds to the C++11/C11 `memory_order_relaxed`; see those standards for the exact definition.

Notes for frontends

If you are writing a frontend which uses this directly, use with caution. The guarantees in terms of synchronization are very weak, so make sure these are only used in a pattern which you know is correct. Generally, these would either be used for atomic operations which do not protect other memory (like an atomic counter), or along with a `fence`.

Notes for optimizers

In terms of the optimizer, this can be treated as a read+write on the relevant memory location (and alias analysis will take advantage of that). In addition, it is legal to reorder non-atomic and Unordered loads around Monotonic loads. CSE/DSE and a few other optimizations are allowed, but Monotonic operations are unlikely to be used in ways which would make those optimizations useful.

Notes for code generation

Code generation is essentially the same as that for unordered for loads and stores. No fences are required. `cmpxchg` and `atomicrmw` are required to appear as a single operation.

Acquire

Acquire provides a barrier of the sort necessary to acquire a lock to access other memory with normal loads and stores.

Relevant standard

This corresponds to the C++11/C11 `memory_order_acquire`. It should also be used for C++11/C11 `memory_order_consume`.

Notes for frontends

If you are writing a frontend which uses this directly, use with caution. Acquire only provides a semantic guarantee when paired with a Release operation.

Notes for optimizers

Optimizers not aware of atomics can treat this like a nothrow call. It is also possible to move stores from before an Acquire load or read-modify-write operation to after it, and move non-Acquire loads from before an Acquire operation to after it.

Notes for code generation

Architectures with weak memory ordering (essentially everything relevant today except x86 and SPARC) require some sort of fence to maintain the Acquire semantics. The precise fences required varies widely by architecture, but for a simple implementation, most architectures provide a barrier which is strong enough for everything (`dmb` on ARM, `sync` on PowerPC, etc.). Putting such a fence after the equivalent Monotonic operation is sufficient to maintain Acquire semantics for a memory operation.

Release

Release is similar to Acquire, but with a barrier of the sort necessary to release a lock.

Relevant standard

This corresponds to the C++11/C11 `memory_order_release`.

Notes for frontends

If you are writing a frontend which uses this directly, use with caution. Release only provides a semantic guarantee when paired with a Acquire operation.

Notes for optimizers

Optimizers not aware of atomics can treat this like a nothrow call. It is also possible to move loads from after a Release store or read-modify-write operation to before it, and move non-Release stores from after an Release operation to before it.

Notes for code generation

See the section on Acquire; a fence before the relevant operation is usually sufficient for Release. Note that a store-store fence is not sufficient to implement Release semantics; store-store fences are generally not exposed to IR because they are extremely difficult to use correctly.

AcquireRelease

AcquireRelease (`acq_rel` in IR) provides both an Acquire and a Release barrier (for fences and operations which both read and write memory).

Relevant standard

This corresponds to the C++11/C11 `memory_order_acq_rel`.

Notes for frontends

If you are writing a frontend which uses this directly, use with caution. Acquire only provides a semantic guarantee when paired with a Release operation, and vice versa.

Notes for optimizers

In general, optimizers should treat this like a nothrow call; the possible optimizations are usually not interesting.

Notes for code generation

This operation has Acquire and Release semantics; see the sections on Acquire and Release.

SequentiallyConsistent

SequentiallyConsistent (`seq_cst` in IR) provides Acquire semantics for loads and Release semantics for stores. Additionally, it guarantees that a total ordering exists between all SequentiallyConsistent operations.

Relevant standard

This corresponds to the C++11/C11 `memory_order_seq_cst`, Java volatile, and the gcc-compatible `__sync_*` builtins which do not specify otherwise.

Notes for frontends

If a frontend is exposing atomic operations, these are much easier to reason about for the programmer than other kinds of operations, and using them is generally a practical performance tradeoff.

Notes for optimizers

Optimizers not aware of atomics can treat this like a nothrow call. For SequentiallyConsistent loads and stores, the same reorderings are allowed as for Acquire loads and Release stores, except that SequentiallyConsistent operations may not be reordered.

Notes for code generation

SequentiallyConsistent loads minimally require the same barriers as Acquire operations and SequentiallyConsistent stores require Release barriers. Additionally, the code generator must enforce ordering between SequentiallyConsistent stores followed by SequentiallyConsistent loads. This is usually done by emitting either a full fence before the loads or a full fence after the stores; which is preferred varies by architecture.

Atomics and IR optimization

Predicates for optimizer writers to query:

- `isSimple()`: A load or store which is not volatile or atomic. This is what, for example, memcpyopt would check for operations it might transform.
- `isUnordered()`: A load or store which is not volatile and at most Unordered. This would be checked, for example, by LICM before hoisting an operation.
- `mayReadFromMemory()`/`mayWriteToMemory()`: Existing predicate, but note that they return true for any operation which is volatile or at least Monotonic.
- `isStrongerThan` / `isAtLeastOrStrongerThan`: These are predicates on orderings. They can be useful for passes that are aware of atomics, for example to do DSE across a single atomic access, but not across a release-acquire pair (see MemoryDependencyAnalysis for an example of this)
- Alias analysis: Note that AA will return ModRef for anything Acquire or Release, and for the address accessed by any Monotonic operation.

To support optimizing around atomic operations, make sure you are using the right predicates; everything should work if that is done. If your pass should optimize some atomic operations (Unordered operations in particular), make sure it doesn't replace an atomic load or store with a non-atomic operation.

Some examples of how optimizations interact with various kinds of atomic operations:

- `memcpyopt`: An atomic operation cannot be optimized into part of a memcpy/memset, including unordered loads/stores. It can pull operations across some atomic operations.
- LICM: Unordered loads/stores can be moved out of a loop. It just treats monotonic operations like a read+write to a memory location, and anything stricter than that like a nothrow call.
- DSE: Unordered stores can be DSE'ed like normal stores. Monotonic stores can be DSE'ed in some cases, but it's tricky to reason about, and not especially important. It is possible in some case for DSE to operate across a stronger atomic operation, but it is fairly tricky. DSE delegates this reasoning to MemoryDependencyAnalysis (which is also used by other passes like GVN).
- Folding a load: Any atomic load from a constant global can be constant-folded, because it cannot be observed. Similar reasoning allows sroa with atomic loads and stores.

Atomics and Codegen

Atomic operations are represented in the SelectionDAG with `ATOMIC_*` opcodes. On architectures which use barrier instructions for all atomic ordering (like ARM), appropriate fences can be emitted by the AtomicExpand Codegen pass if `setInsertFencesForAtomic()` was used.

The MachineMemOperand for all atomic operations is currently marked as volatile; this is not correct in the IR sense of volatile, but CodeGen handles anything marked volatile very conservatively. This should get fixed at some point.

One very important property of the atomic operations is that if your backend supports any inline lock-free atomic operations of a given size, you should support *ALL* operations of that size in a lock-free manner.

When the target implements atomic `cmpxchg` or LL/SC instructions (as most do) this is trivial: all the other operations can be implemented on top of those primitives. However, on many older CPUs (e.g. ARMv5, SparcV8, Intel 80386) there are atomic load and store instructions, but no `cmpxchg` or LL/SC. As it is invalid to implement `atomic load` using the native instruction, but `cmpxchg` using a library call to a function that uses a mutex, `atomic load` must *also* expand to a library call on such architectures, so that it can remain atomic with regards to a simultaneous `cmpxchg`, by using the same mutex.

AtomicExpandPass can help with that: it will expand all atomic operations to the proper `__atomic_*` libcalls for any size above the maximum set by `setMaxAtomicSizeInBitsSupported` (which defaults to 0).

On x86, all atomic loads generate a `MOV`. SequentiallyConsistent stores generate an `XCHG`, other stores generate a `MOV`. SequentiallyConsistent fences generate an `MFENCE`, other fences do not cause any code to be generated. `cmpxchg` uses the `LOCK CMPXCHG` instruction. `atomicrmw xchg` uses `XCHG`, `atomicrmw add` and `atomicrmw sub` use `XADD`, and all other `atomicrmw` operations generate a loop with `LOCK CMPXCHG`. Depending on the users of the result, some `atomicrmw` operations can be translated into operations like `LOCK AND`, but that does not work in general.

On ARM (before v8), MIPS, and many other RISC architectures, Acquire, Release, and SequentiallyConsistent semantics require barrier instructions for every such operation. Loads and stores generate normal instructions. `cmpxchg` and `atomicrmw` can be represented using a loop with

LL/SC-style instructions which take some sort of exclusive lock on a cache line (`LDREX` and `STREX` on ARM, etc.).

It is often easiest for backends to use AtomicExpandPass to lower some of the atomic constructs. Here are some lowerings it can do:

- cmpxchg -> loop with load-linked/store-conditional by overriding `shouldExpandAtomicCmpXchgInIR()`, `emitLoadLinked()`, `emitStoreConditional()`
- large loads/stores -> ll-sc/cmpxchg by overriding `shouldExpandAtomicStoreInIR()`/`shouldExpandAtomicLoadInIR()`
- strong atomic accesses -> monotonic accesses + fences by overriding `shouldInsertFencesForAtomic()`, `emitLeadingFence()`, and `emitTrailingFence()`
- atomic rmw -> loop with cmpxchg or load-linked/store-conditional by overriding `expandAtomicRMWInIR()`
- expansion to __atomic_* libcalls for unsupported sizes.

For an example of all of these, look at the ARM backend.

Libcalls: __atomic_*

There are two kinds of atomic library calls that are generated by LLVM. Please note that both sets of library functions somewhat confusingly share the names of builtin functions defined by clang. Despite this, the library functions are not directly related to the builtins: it is *not* the case that `__atomic_*` builtins lower to `__atomic_*` library calls and `__sync_*` builtins lower to `__sync_*` library calls.

The first set of library functions are named `__atomic_*`. This set has been “standardized” by GCC, and is described below. (See also GCC’s documentation)

LLVM’s AtomicExpandPass will translate atomic operations on data sizes above `MaxAtomicSizeInBitsSupported` into calls to these functions.

There are four generic functions, which can be called with data of any size or alignment:

```
void __atomic_load(size_t size, void *ptr, void *ret, int ordering)
void __atomic_store(size_t size, void *ptr, void *val, int ordering)
void __atomic_exchange(size_t size, void *ptr, void *val, void *ret, int ordering)
bool __atomic_compare_exchange(size_t size, void *ptr, void *expected, void *desired,
```

There are also size-specialized versions of the above functions, which can only be used with *naturally-aligned* pointers of the appropriate size. In the signatures below, “N” is one of 1, 2, 4, 8, and 16, and “iN” is the appropriate integer type of that size; if no such integer type exists, the specialization cannot be used:

```
iN __atomic_load_N(iN *ptr, iN val, int ordering)
void __atomic_store_N(iN *ptr, iN val, int ordering)
iN __atomic_exchange_N(iN *ptr, iN val, int ordering)
bool __atomic_compare_exchange_N(iN *ptr, iN *expected, iN desired, int success_order,
```

Finally there are some read-modify-write functions, which are only available in the size-specific variants (any other sizes use a `__atomic_compare_exchange` loop):

```
iN __atomic_fetch_add_N(iN *ptr, iN val, int ordering)
iN __atomic_fetch_sub_N(iN *ptr, iN val, int ordering)
iN __atomic_fetch_and_N(iN *ptr, iN val, int ordering)
iN __atomic_fetch_or_N(iN *ptr, iN val, int ordering)
iN __atomic_fetch_xor_N(iN *ptr, iN val, int ordering)
iN __atomic_fetch_nand_N(iN *ptr, iN val, int ordering)
```

This set of library functions have some interesting implementation requirements to take note of:

- They support all sizes and alignments – including those which cannot be implemented natively on any existing hardware. Therefore, they will certainly use mutexes in for some sizes/alignments.
- As a consequence, they cannot be shipped in a statically linked compiler-support library, as they have state which must be shared amongst all DSOs loaded in the program. They must be provided in a shared library used by all objects.
- The set of atomic sizes supported lock-free must be a superset of the sizes any compiler can emit. That is: if a new compiler introduces support for inline-lock-free atomics of size N, the `__atomic_*` functions must also have a lock-free implementation for size N. This is a requirement so that code produced by an old compiler (which will have called the `__atomic_*` function) interoperates with code produced by the new compiler (which will use native the atomic instruction).

Note that it's possible to write an entirely target-independent implementation of these library functions by using the compiler atomic builtins themselves to implement the operations on naturally-aligned pointers of supported sizes, and a generic mutex implementation otherwise.

Libcalls: __sync_*

Some targets or OS/target combinations can support lock-free atomics, but for various reasons, it is not practical to emit the instructions inline.

There's two typical examples of this.

Some CPUs support multiple instruction sets which can be swiched back and forth on function-call boundaries. For example, MIPS supports the MIPS16 ISA, which has a smaller instruction encoding than the usual MIPS32 ISA. ARM, similarly, has the Thumb ISA. In MIPS16 and earlier versions of Thumb, the atomic instructions are not encodable. However, those instructions are available via a function call to a function with the longer encoding.

Additionally, a few OS/target pairs provide kernel-supported lock-free atomics. ARM/Linux is an example of this: the kernel provides a function which on older CPUs contains a "magically-restartable" atomic sequence (which looks atomic so long as there's only one CPU), and contains actual atomic instructions on newer multicore models. This sort of functionality can typically be provided on any architecture, if all CPUs which are missing atomic compare-and-swap support are uniprocessor (no SMP). This is almost always the case. The only common architecture without that property is SPARC – SPARCV8 SMP systems were common, yet it doesn't support any sort of compare-and-swap operation.

In either of these cases, the Target in LLVM can claim support for atomics of an appropriate size, and then implement some subset of the operations via libcalls to a `__sync_*` function. Such functions *must* not use locks in their implementation, because unlike the `__atomic_*` routines used by AtomicExpandPass, these may be mixed-and-matched with native instructions by the target lowering.

Further, these routines do not need to be shared, as they are stateless. So, there is no issue with having multiple copies included in one binary. Thus, typically these routines are implemented by the statically-linked compiler runtime support library.

LLVM will emit a call to an appropriate `__sync_*` routine if the target ISelLowering code has set the corresponding `ATOMIC_CMPXCHG`, `ATOMIC_SWAP`, or `ATOMIC_LOAD_*` operation to "Expand", and if it has opted-into the availability of those library functions via a call to `initSyncLibcalls()`.

The full set of functions that may be called by LLVM is (for `N` being 1, 2, 4, 8, or 16):

```
iN __sync_val_compare_and_swap_N(iN *ptr, iN expected, iN desired)
iN __sync_lock_test_and_set_N(iN *ptr, iN val)
iN __sync_fetch_and_add_N(iN *ptr, iN val)
iN __sync_fetch_and_sub_N(iN *ptr, iN val)
iN __sync_fetch_and_and_N(iN *ptr, iN val)
iN __sync_fetch_and_or_N(iN *ptr, iN val)
iN __sync_fetch_and_xor_N(iN *ptr, iN val)
iN __sync_fetch_and_nand_N(iN *ptr, iN val)
iN __sync_fetch_and_max_N(iN *ptr, iN val)
iN __sync_fetch_and_umax_N(iN *ptr, iN val)
iN __sync_fetch_and_min_N(iN *ptr, iN val)
iN __sync_fetch_and_umin_N(iN *ptr, iN val)
```

This list doesn’t include any function for atomic load or store; all known architectures support atomic loads and stores directly (possibly by emitting a fence on either side of a normal load or store.)

There’s also, somewhat separately, the possibility to lower `ATOMIC_FENCE` to `__sync_synchronize()`. This may happen or not happen independent of all the above, controlled purely by `setOperationAction(ISD::ATOMIC_FENCE, ...)`.