Lowering C11 Atomics for ARM in LLVM

Reinoud Elhorst

***Abstract*—This report explores the way LLVM generates the memory barriers needed to support the C11/C++11 atomics for ARM. I measure the influence of memory barriers on performance, and I show that in some cases LLVM generates too many barriers. By leaving these barriers out, performance increases significantly. I introduce two LLVM passes, which will remove these extra barriers, improving performance in my test by 40%. I believe one of these passes is ready to be upstreamed to LLVM, while the other will need more testing.**

***Index Terms*—C11, C++11, ARM, LLVM, atomics, dmb, memory synchronization, memory barriers**

I. Introduction

Traditionally C and C++ were developed as languages for single-threaded programs. The languages did not contain any support for threads, and where threading was introduced in libraries, for instance with `libpthread`, this leads to subtle problems in programs because of compiler and hardware instruction reordering [1]. Thread support should be written into the language, and cannot be introduced as a library.

As one starts to work with shared memory (as threads do), one has to deal with the memory model. A memory model, in simple terms, describes the constraints that hold, when one thread writes to shared memory and another thread reads from the same memory location[1]. Specifically, when does one thread "see" the writes from another thread, and in which order. Only relatively recently have systems with multiple processing units become mainstream[2], and therefore the memory model is something that mainstream language and program designers only have had to deal with recently.

With the introduction of C11/C++11, these languages acquired official support for multi-threading, solving the problems mentioned in [1]. Many C compilers had already introduced proprietary instructions that allowed the same things, but only with C11/C++11 these became part of the standard. C11/C++11 demands that programmers specify which variables could be part of a *data race* (simultaneous read and write, or two simultaneous writes to the same memory location), by specifying them as `atomic`[3]. It also allows programmers to specify the memory order they wish to use for each separate load from or store to shared memory. By default a "safe" memory order, sequentially consistent, is used, but a programmer can specify more relaxed behaviour in order to produce high performance code.

This report describes the work I did related to the memory model and compilation of C11 code to ARM on LLVM. It shows the influence of memory barriers on performance, the current state of LLVM in dealing with atomics on ARM, and proposes a patch to LLVM, which results in a speed-up of the seqlock algorithm, presented in section II-A, by 40%. As a result, the sequentially consistent version of the algorithm becomes as fast as a version with a more relaxed memory behaviour.

Even though all benchmarks done in this project were written in C11, I expect to find similar results for C++11. In addition, the optimizations proposed in this report apply to steps late in the LLVM tool chain. For that reason other languages compiling through LLVM may benefit from the optimizations as well.

A. Memory model

A memory model describes how writes from one thread should propagate to other threads.

Let's start with a small example. We have 2 variables, `x` and `y`, both initially being `0`.

```
thread 0          thread 1
 y = 1;            x = 1;
 print(x);         print(y);
```

Naively we would argue that the possible printed results for `(x,y)` can be `(1,0)` (if thread 0 runs faster), `(0,1)` (if thread 1 runs faster) or `(1,1)` (if both threads run equally fast). A value of `(0,0)` would not be possible, since if `print(x);` prints `0`, that must mean that thread 1 has not gotten around to executing `x = 1;` yet. As a result, by the time thread 1 gets to do `print(y);`, it must read the value `1`, since thread 0 already did the printing, so must surely already have done the `y = 1;`.

Modern processor architectures have optimizations in the way they deal with reads from and writes to memory. As a result, on modern processors (for instance recent x86 and

R. Elhorst is a student at the Computer Laboratory of the University of Cambridge. Email re302@cam.ac.uk.

[1]I talk about threads here, but the statements are equally valid for separate programs accessing shared memory. The rest of this report however, as well as most solutions proposed, mainly focuses on threads.

[2]This can be either multiple processors, multiple cores per processor, or multiple concurrent instructions per core, as well as a CPU with other devices, such as GPUs or network cards, that may write directly to memory.

[3]One should make a clear distinction between race conditions, and data races. A race condition is the behaviour of a program where the output is dependent on the sequence in which instructions in different threads are executed. Race conditions are present in most multi-threaded programs and are not bad, just something a programmer needs to be aware of. A data race is a situation in C11/C++11 where there can be simultaneous read and write access (or two simulaneous writes) to a variable not defined as `atomic`, and is a program fault. C11/C++11 defines undefined behaviour for programs with data races, some of the reasons for this are explained by [2].

ARM models) the code in the example may very well print `(0,0)`. This is because the architectures use a more relaxed memory model than we assumed.

The "naively" assumed memory model is called *sequential consistency* [3]. Architectures such as x86 and Sparc have a *weaker* memory model, meaning that they give fewer guarantees, ARM and Power have even weaker memory models. The reason for these weaker models is that the processors can execute code faster when dealing with a weaker memory model. This can be easily understood; if after each time the processor does a write to the memory, it has to wait until the value has actually been written to memory, this will slow things down considerably, especially considering that memory typically runs at a much slower speed than the processor. Equally, if a processor can only start reading a variable from memory the moment it needs it, it will waste several clock cycles between starting the read and the moment the variable is actually available[4].

Most programmers will expect to be working with a sequentially consistent memory model, and when one specifies variables as `atomic`, C11/C++11 defines by default a sequentially consistent memory model for these variables. Alternatively a programmer may explicitly specify a memory order on variable access. Sometimes we speak about a program using a more relaxed memory model, in this case we mean that some, most or all memory access specify a more relaxed memory order. A programmer using a more relaxed memory model usually does so in an effort to increase the program's performance. A more relaxed memory model may or may not, depending on the hardware architecture and the compiler, result in faster code (but should never be slower).

It is the compiler's job to make sure that the programmed memory model is also present in the generated machine code. This means that firstly the compiler should not reorder instructions in a way that invalidates the memory model; this requirement is easy for the compiler to adhere to, by only performing instruction reordering in cases when it knows it is allowed to. Secondly the compiler must generate machine code that tells the hardware not to reorder instructions in a way that is not allowed under the requested memory model. This is achieved by inserting *memory barrier instructions* into the machine code. These barriers (or *fences* or *memory synchronizations*) are `mfence`s on x86, `sync`s and `lwsync`s on Power and `dmb`s on ARM (although sometimes a faster result can be achieved by using some other codes, as shown by [4]).

It will go into too much detail to discuss the memory models defined in C11/C++11, some background can be gotten from [5], with formal definitions in [6]. A more practical description can be found in the C++ reference[5]. For the rest of this report it suffices to say that the memory model *sequentially consistent* is stronger than *acquire/release*, which is turn is stronger than *relaxed*.

[4]It is worth noting that this is just an explanation why optimizations in memory reads and writes are necessary. What actually happens when a variable gets written to or read from memory differs per architecture and is often more complex.

[5]`http://en.cppreference.com/w/cpp/atomic/memory_order`

B. Reasoning about memory models

As I mentioned before, the sequentially consistent memory model can be seen as the one that a (naive) programmer would expect to be working with. This is the reason why C11/C++11 states that if no memory model has been specified for access to a variable specified as `atomic`, the sequentially consistent memory model should be used. In addition, most research on algorithms for shared memory systems assumes a sequentially consistent model.

Most programmers find it hard to reason about concurrency in programs, and reasoning about it when choosing a weaker-than-sequentially-consistent model makes the task much harder. It is very hard to test whether your code is correct; if it runs correctly, it is hard to test whether your current compiler, on your current hardware, generated correct code ([4] showed that some behaviour occurs in less than one in a million cases, some conditions that are allowed to occur on an architecture according to the specifications, do not actually happen on specific subarchitectures), it is harder to test this for all compilers for all architectures, and even if you could succesfully show this, it does not mean that the code is correct. There are some large scale projects using relaxed memory models, notably the Linux kernel, but it is not clear whether the implementation is correct ([7] seems to suggest it is not in all cases). For smaller pieces of code, formal proof can be used to show correctness.

If the compiler were able to compile code written with a stronger memory model, to execute just as fast as if written for a weaker memory model, this would take away the need for the programmer to reason about memory models.

C. Environment

In this report I run several benchmarks, and look at the result of several compile steps on LLVM with a clang front end[6]. The most recent LLVM and clang development versions were used, those of svn head rev 198020, 25 December 2013. It was compiled with debug enabled, although debug information was removed in some cases in this report, in order to fit the results on the page. Unless stated otherwise, no changes were made in LLVM or clang, the `-O3` flag was used for compilation, compilation target was `armv7l-unknown-linux-gnueabihf`, and the pthread library was compiled in. Because it was non-trivial to get the C11 `stdatomic.h` library working on the test platform, all code was written using the LLVM-specific atomics notation. Internally the C11 notation and the LLVM notation are equivalent, and all results should be considered equally valid as in the case the C11 notation had been used.

Performance tests were done on an Odroid U2[7], a machine with a quad core ARM processor. All measurements in this report are wall clock time, measured on the Odroid's real time clock. Since the load on the test machine was low, and I used at most two threads, the wall clock time will accurately

[6]LLVM is an open-source compiler which supports multiple source languages and compile targets. Clang is the C/C++ frontend to LLVM. `http://www.llvm.org/`

[7]`http://en.wikipedia.org/wiki/Odroid`

measure performance. When timings are mentioned, these are the average of many runs.

D. Acknowledgements

The project would not have been possible without the help from, discussions with and patient explanations of the intricacies of the memory model on ARM by Mark Batty and Shaken Flur. In addition I thank Renato Golin from Linaro for giving me access to the Odroid U2 machine, so that I could run my benchmarks. Finally thanks goes out to David Chisnall, for suggesting the project in the first place, and making sure I arrived at the starting line with enough knowledge to see it through to the end.

II. Project

A. Sequential lock

The sequential lock (*seqlock*) is a locking mechanism, developed for the linux kernel. For the tests I use a simplified version of the seqlock algorithm, which supports just one writer thread (listing 1). The seqlock algorithm presented makes sure that a call to read will only ever return values for `v1` and `v2` that were written in the same write call. If, for instance, `v1` and `v2` are the top and bottom 32 bits of a 64-bit number, it would be impossible for any reader to end up reading the top and bottom 32 bits from different numbers.

Listing 1: The simplified seqlock algorithm used in the tests in C11

```
atomic int lock=0, x1=0, x2=0;

void write(int v1, int v2) {
    int local_lock = load_relaxed(lock);
    store(lock, ++local_lock);
    store(x1, v1);
    store(x2, v2);
    store(lock, ++local_lock);
}

void read(int *v1, int *v2) {
    int lv1, lv2, local_lock;
    while (true) {
        local_lock = load(lock);
        if (local_lock & 1) continue;
        lv1 = load(x1);
        lv2 = load(x2);
        if (local_lock == load(lock)) {
            *v1 = lv1;
            *v2 = lv2;
            return;
        }
    }
}
```

The algorithm contains one `load_relaxed`, which is a load with memory order relaxed (this is correct because only this thread is writing to the lock variable, and a load will always see writes that happened before in the same thread). All other calls to `load` and `store` are sequentially consistent. Most programmers with a background in concurrency will be able to reason about this code and understand that it will indeed work in the way it is intended.

The seqlock algorithm will also be correct if we interpret all `load`s as having memory model acquire, and all `store`s as memory model release[8]. Because we both have an intuitively correct sequentially consistent version, and a correct version with a weaker memory model, this makes the algorithm ideal for testing. [8] showed that there are more versions of this algorithm using weaker memory models. These versions are not being considered in this report since they will not contribute to the main findings.

The actual tests writes the numbers $(0,0)$, $(1,1)$, ... , $(10^9-1, 10^9-1)$ into $(x1, x2)$. Another thread reads back the values for $(x1, x2)$ and asserts that $x1 == x2$. After a read, this thread enters a short sleep, to simulate a consumer reading some value and doing something with it. The actual test code has the writer and reader in a loop (listing 2). This allows the compiler to move the initialization out of the loop, and the local_lock variable can be local to the function and does not require the relaxed load. One might argue that this makes the test case contrived; on the other hand will it make the inspection of the machine code in the loop later a lot easier, and allow us to see some interesting features. I also argue that the conclusions drawn from these tests remain equally valid in the general case.

Listing 2: version with the seqlock in a loop

```
atomic_int lock=0, x1=0, x2=0;

void writer() {
    int i, local_lock=0;
    for (i=1; i<=MAXI; i++) {
        store(lock, ++local_lock);
        store(x1, i);
        store(x2, i);
        store(lock, ++local_lock);
    }
}

void reader() {
    int v1=0, v2=0, i=0, ii;
    while (v1 < MAXI - 1) {
        i++;
        int lv1, lv2, local_lock;
        while (true) {
            local_lock = load(lock);
            if (local_lock & 1) continue;
            v1 = load(x1);
            v2 = load(x2);
            if (local_lock == load(lock)) {
                break;
            }
        }
        assert(v1 == v2);
        nanosleep(SLEEPTIME, NULL);
    }
}
```

B. Influence of the memory model on performance

The seqlock program is compiled in 3 ways:

- *Implied memory model*. No memory model gets specified. This means that I define macros

  ```
  #define load(x) x
  #define store(x,y) x=y
  ```

[8] A formal proof for this statement can be made, but is not included in this report.

- *Sequential consistency*: Use the sequentially consistent memory model for everything[9].

```
#define load(x) \
    __c11_atomic_load(&x, __ATOMIC_SEQ_CST)
#define store(x,y) \
    __c11_atomic_store(&x, y, __ATOMIC_SEQ_CST)
```

- *Acquire/release*: Use the acquire for the loads and release for the stores.

```
#define load(x) \
    __c11_atomic_load(&x, __ATOMIC_ACQUIRE)
#define store(x,y) \
    __c11_atomic_store(&x, y, __ATOMIC_RELEASE)
```

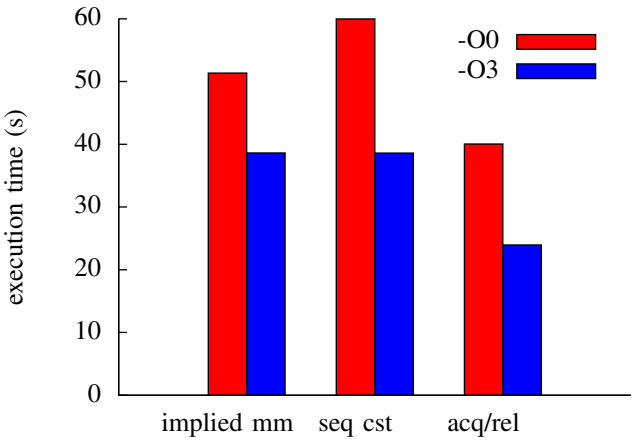

Fig. 1: Seqlock algorithm using different memory models and different optimization settings

According to the C11 standard, the implicit model and the sequentially consistent model are equivalent. One would expect the implicit and sequentially consistent versions of the algorithm to have equal performance, and the acquire/release version to be faster, if atomics are implemented well in LLVM. However figure 1 clearly shows that in the case we do not ask the compiler for optimizations (`-O0`), the sequentially consistent code performs considerably worse than the one with the implied memory model. It is outside the scope of this project to determine where this difference comes from; the LLVM IR shows that each atomic load and store goes through an `atomictmp` or `atomicdst` variable when explicitly specifying that they should be done sequentially consistent. In my opinion it would make sense if both an implicit and explicit memory model load/store were compiled through the same code path.

Interestingly enough in the `-O3` optimization, the sequentially consistent memory model seems to be the faster of the two.

As expected, the algorithm with the weaker memory model runs faster. To understand where this difference comes from, one must look at the generated machine code (listing 3).

[9]Remember that for practical reasons I use the LLVM atomics notation, not the C11 notation. Since the two notations are equivalent in LLVM, all conclusions are valid for the case where C11 atomics were used as well.

Listing 3: Machine code for the loop in the `writer` function, optimization `-O3`, left the sequentially consistent version, right the acquire/release

```
1  .LBB0_1:                                .LBB0_1:
2    dmb     ish                             dmb     ish
3    sub     r4, r1, #1                      sub     r4, r1, #1
4    str     r4, [r2]                        str     r4, [r2]
5    add     r0, r0, #1                      add     r0, r0, #1
6    dmb     ish                             dmb     ish
7    cmp     r0, r3
8    dmb     ish
9    str     r0, [r12]                       str     r0, [r12]
10   dmb     ish                             dmb     ish
11   dmb     ish
12   str     r0, [lr]                        str     r0, [lr]
13   dmb     ish                             dmb     ish
14   dmb     ish
15   str     r1, [r2]                        str     r1, [r2]
16   add     r1, r1, #2                      add     r1, r1, #2
17   dmb     ish
18                                           cmp     r0, r3
19   bne     .LBB0_1                         bne     .LBB0_1
```

Appendix A explains the machine code in more detail, the interesting thing to note is that the two versions only differ in two places: the `cmp` in line (7) is in the acquire/release version in line (18), and the sequentially consistent version has a `dmb` in 4 places — lines (8), (11), (14) and (17) —, where the acquire/release version has none. A similar pattern can be seen in the loop portion of the `reader` function (not displayed in this report). Since a `dmb` is a barrier, it is expected that the sequentially consistent version of the code, which has to maintain a stronger memory model, has more. The hypothesis is that `dmb`s are relatively expensive operations, and that they are responsible for the slower execution in the sequentially consistent version.

C. Double barriers

The machine code in listing 3 was generated by applying mappings between C11 code and ARM machine code, such as those published by Peter Sewell[10]. Although none of these mappings can be made smaller (i.e. every `dmb` is needed for the mapping to be correct), sometimes the resulting code does contain more `dmb`s than are necessary. For instance, a single store operation is mapped to `dmb; str; dmb`. Two stores in a row in this case would result in `dmb; str; dmb; dmb; str; dmb`. The ARM memory model shows that two `dmb`s in a row do not strengthen the memory model more than a single `dmb` [4]. This means that we are allowed to remove the second `dmb`. For completeness I also refer to [9] which described similar work for the x86 architecture, including formal proofs.

The hypothesis from the last section is that `dmb`s are expensive. Listing 3 shows two places in the sequentially consistent code where two `dmb`s appear in a row. For "Optimization A", I remove one of each pair, and make a similiar optimization to the `reader`'s loop. Figure 2 shows that this indeed gives a considerable speedup. A second optimization

[10]`http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html`

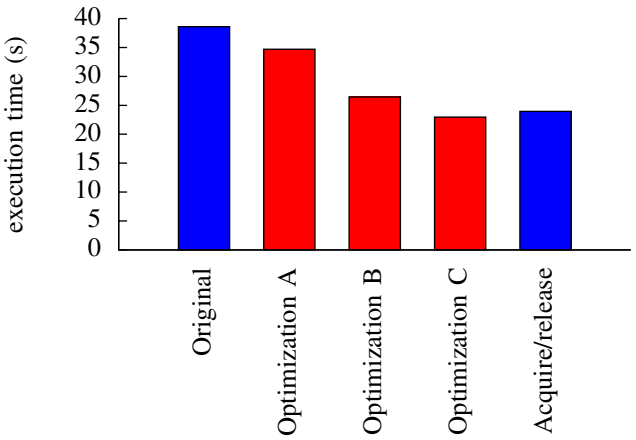

Fig. 2: Different optimizations for the sequentially consistent seqlock algorithm (red) compared to the original and the acquire/release version (blue).

can be made by realizing that the `cmp` in line (7) does not do any memory access, and therefore the state of the memory model is not influenced by it: the `dmb`s in lines (6) and (8) can be considered to be following one another as far as the memory is concerned, and therefore the `dmb` at line (8) can be removed. Again a similar optimization can be made in the `reader`'s loop, this result is shown as "Optimization B".

Finally there is one more double `dmb` that can be eliminated. The loop `.LBB0_1` will loop 10^9 times, before exiting. This means that the `dmb` at line (17) is followed in almost all cases by the `dmb` in line (2) (the `bne` statement in between has the same properties in relation to memory as the `cmp` statement discussed before). As a result the `dmb` statement in line (17) only has to be executed when the loop ends; this can be achieved by placing the statement past the `bne` statement[11]. This optimization (again with a similar one in the `reader`'s loop), is shown as "Optimization C".

Figure 2 shows that the optimizations where I removed double `dmb`s, or moved `dmb`s out of the loop, resulted in considerable speedups, confirming the hypothesis of the last section. In addition listing 4 shows that removing double `dmb`s from the `writer`'s loop gives code that is almost equal to the acquire/release version; the `reader`'s loop is also almost the same. In the tests the optimization C version is actually faster than the acquire/release version; I do not know the reason for this although I give a possible explanation for this in appendix B.

I want to emphasize that even though Optimization C and the acquire/release version are the same for all practical matters (the final `dmb` can even be removed because after it nothing happens but the thread ending, and the `cmp` can be moved in either of them so that they line up), this does not mean that I rewrote the algorithm from a sequentially consistent to an acquire release memory model, nor is this a proof that the acquire/release version of the algorithm is correct. All it shows is that under this specific ARM memory model mapping the two memory models may share the same machine code.

Again I want to point to [8], which shows additional relaxed memory behaviour implementations of the seqlock algorithm, some of which might be faster than the versions shown here; this was not tested.

D. Remove double barriers in an LLVM pass

LLVM runs *passes* at different points during the compilation. Passes can, among other things, be used for optimization. The results of the last section show that a pass which removes double `dmb`s is at least in some cases beneficial, and I am confident that there are not many cases where it is harmful to do so.

Such a pass cannot run against the LLVM IR, since it has optimizations that are specific to the ARM machine code, hence it has to run in the ARM code generation step.

I wrote two passes. Optimization A and B from the last section are being handled by the `ARMRemoveDoubleBarriersPass`[12]. The pass iterates over a basic block, looks for `dmb`s, and removes the `dmb`, if the `dmb` could be moved up until it is next to another `dmb`. In the seqlock example in the last section it was easy to reason why it is correct to move the `dmb` past the `cmp`. In the pass I assume the `dmb` is allowed to move past any instruction which does not have any memory access, according to `MachineInstr::mayLoad()` and `MachineInstr::mayStore()`.

Optimization C is harder to implement. The correct way would be to determine for each basic block whether it starts with a `dmb` (*starts with* here means that it is possible to move a `dmb` in such a way that it is the first instruction), or ends in one as last instruction before branch statements (again, after moving). A control flow graph will then show places where double `dmb`s happen, and one can decide whether these can be removed entirely, or moved to another or to a new basic block. Unlike the pass for optimization A and B, this pass may increase the code size, and the pass should avoid actions that do so in case optimization target `-Os` is chosen. In addition, moving `dmb`s so that they get executed at a different point in the program, may actually slow things down (in Optimizations A and B we always only remove `dmb`s; in Optimization C we may optimize a path through the code that is hardly ever taken, by moving the `dmb` on the much travelled code path to a possibly slower position, only to remove the `dmb` in the path not often taken). Adding this optimization pass does therefore require more consideration.

As a proof of concept I implemented a simple version of a pass for Optimization C: `ARMMoveBarriersPastBranchesPass`[13]. For a basic block it looks whether it ends in a `dmb` and some of the successor blocks start in a `dmb`,

[11] Technically this means placing it in the next basic block, or even creating a new basic block.

[12] `https://github.com/reinhrst/llvm/commit/c4f073`

[13] `https://github.com/reinhrst/llvm/commit/fcf796`

Listing 4: Loop of the `writer` function of the seqlock algorithm, left the sequentially consistent version, right the acquire/release version, in the middle the version with the double `dmb`s removed by hand (Optimization C)

```
.LBB0_1:              .LBB0_1:              .LBB0_1:
  dmb  ish              dmb  ish              dmb  ish
  sub  r4, r1, #1       sub  r4, t1, #1       sub  r4, r1, #1
  str  r4, [r2]         str  r4, [r2]         str  r4, [r2]
  add  r0, r0, #1       add  r0, r0, #1       add  r0, r0, #1
  dmb  ish              dmb  ish              dmb  ish
  cmp  r0, r3           cmp  r0, r3
  dmb  ish
  str  r0, [r12]        str  r0, [r12]        str  r0, [r12]
  dmb  ish              dmb  ish              dmb  ish
  dmb  ish
  str  r0, [lr]         str  r0, [lr]         str  r0, [lr]
  dmb  ish              dmb  ish              dmb  ish
  dmb  ish
  str  r1, [r2]         str  r1, [r2]         str  r1, [r2]
  add  r1, r1, #2       add  r1, r1, #2       add  r1, r1, #2
  dmb  ish
                                              cmp  r0, r3
  bne  .LBB0_1          bne  .LBB0_1          bne  .LBB0_1
                        dmb  ish
```

again after moving. If this is the case it will consider removing the `dmb` at the end of the current block, and adding it to any successor block not staring in the `dmb`. It will only do so if the successor block has only one predecessor, hence the block it moves to will at most be executed as many times as the `dmb` we removed.

I feel confident that the `ARMRemoveDoubleBarriersPass` can be sent upstream for inclusion in LLVM. The `ARMMoveBarriersPastBranchesPass` requires more consideration, because of the doubts described above. Testing should be done to see that it indeed results in a speedup in the majority of the cases, and perhaps a more sophisticated method should be used to rearange the `dmb`s.

With these passes in place, LLVM compiles the test seqlock algorithm to the hand optimized version shown in Optimization 3. This means that the sequentially consistent version runs 40% faster than without the optimizations, and now has the same performance as the acquire/release version, thus taking the burden off the programmer to have to reason about memory models in this particular case.

References

[1] H.-J. Boehm, "Threads cannot be implemented as a library," in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 261–268.

[2] D. Vyukov, "Benign data races: what could possibly go wrong?" 2013. [Online]. Available: http://software.intel.com/en-us/blogs/2013/01/06/benign-data-races-what-could-possibly-go-wrong

[3] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *Computers, IEEE Transactions on*, vol. 100, no. 9, pp. 690–691, 1979.

[4] L. Maranget, S. Sarkar, and P. Sewell, "A tutorial introduction to the ARM and POWER relaxed memory models," 2012.

[5] H.-J. Boehm and S. V. Adve, "Foundations of the C++ concurrency memory model," in *ACM SIGPLAN Notices*, vol. 43, no. 6. ACM, 2008, pp. 68–78.

[6] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, "Mathematizing C++ concurrency," in *ACM SIGPLAN Notices*, vol. 46, no. 1. ACM, 2011, pp. 55–66.

[7] W. Deacon, "From weak to weedy, effective use of memory barriers in the ARM Linux kernel," 2013.

[8] H.-J. Boehm, "Can seqlocks get along with programming language memory models?" in *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*. ACM, 2012, pp. 12–20.

[9] V. Vafeiadis and F. Z. Nardelli, "Verifying fence elimination optimisations," in *Static Analysis*. Springer, 2011, pp. 146–162.

Appendix A
Explanation of the machine code

This appendix explains how to read the machine code used in this report.

Throughout this report we focus on the loop of the `writer` function. In C this are 4 store statements, with some flow control to make the loop.

```
    for (i=1; i<=MAXI; i++) {
        store(lock, ++local_lock);
        store(x1, i);
        store(x2, i);
        store(lock, ++local_lock);
    }
```

The machine code for the sequentially consistent memory model looks thus. Because the syntax of might look unfamiliar, I added the C-like equivalent between brackets.

```
   .LBB0_1:
(2)  dmb  ish          (barrier)
(2)  sub  r4, r1, #1   (r4=r1-1)
(2)  str  r4, [r2]     (*r2=r4)
(1)  add  r0, r0, #1   (r0=r0+1)
(2)  dmb  ish          (barrier)
(1)  cmp  r0, r3       (r0 == r3?)
(3)  dmb  ish          (barrier)
(3)  str  r0, [r12]    (*r12=r0)
(3)  dmb  ish          (barrier)
(4)  dmb  ish          (barrier)
(4)  str  r0, [lr]     (*lr=r0)
(4)  dmb  ish          (barrier)
(5)  dmb  ish          (barrier)
(5)  str  r1, [r2]     (*r2=r1)
(5)  add  r1, r1, #2   (r1=r1+2)
(5)  dmb  ish          (barrier)
(1)  bne  .LBB0_1      (goto .LBB0_1 if not equal)
```

All "variables" in this code are registers, many of which were filled before. They do not relate one-to-one to the C variables, but some do. `r0` is `i`, `r2`, `r12` and `lr` contain the *addresses* of `lock`, `x1` and `x2`. `r3` contains the value 999,999,999, the highest value in the loop. The comparison to see if this is the last loop iteration is made in line (7), even though this is only acted upon by the `bne` in the last line.

In front of each line of machine code I specified to which line C code it relates. This makes it easy to see that the (sequentially consistent) stores are indeed translated to a `dmb`, followed by a `str`, followed by another `dmb`, as is defined in the mapping[14].

APPENDIX B
INTERACTION BETWEEN THREADS

Seqlock is an algorithm that prevents write starvation. Since there is no lock on the writer thread (in the one-writer-version), the writer should always be able to keep writing at the same speed. The price you pay is that if there is a lot of writing, a reader may have to try many times before it manages to read some data correctly.

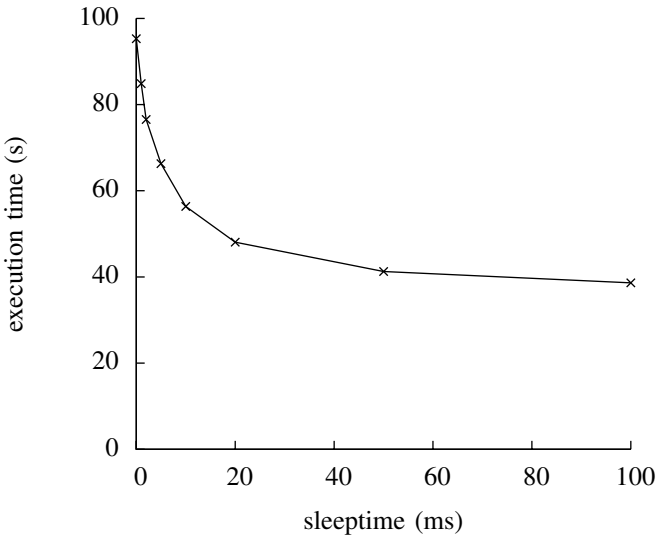

Fig. 3: Seqlock execution time for different sleep times in the reader thread

According to the description above, the writing thread should not in any way be influenced by what the reading thread does (as long as both threads run on their own core). This however turns out not to be true on the ARM architecture. In the tests in the body of the report, there was a 100ms sleep between reads. As figure 3 shows, a smaller sleep results in a slower overall execution. This is not because the CPU is busier; it does not matter if we change the sleep command into a busy sleep loop.

I did not research where this difference in performance comes from. One possible reason is that the reader generated more traffic on the memory bus, thereby slowing down the writer. An alternative explanation is that somehow the writer has to "help" in handling the `dmb`s that the reader calls (for instance requiring the writer to signal that it does not have writes for synchronization). Additional research is needed here, but it is something a programmer of highly optimized code will have to take into account.

Above also may explain why there is a small but clear difference in performance between our Optimization C code and the acquire/release code we see in figure 2 on page 5, even though the machine code is almost the same. If, because of a small difference in the order of the instructions, the reader is slightly more likely to have to retry the read, this leads to the reader processing more `dmb`s, thereby slowing the writer down. In this light one might consider seqlock to be unsuitable for these benchmarking tests. I believe that the algorithm is appropriate, as long as one is aware of this behaviour, since actual (non-benchmarking) code might exhibit similar behaviour.

[14] http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html