Disclaimer.

Since the explanation of the problem of the contest seems very vague to me, I took some time and explain my thought and problem approach. In case some readers are not familiar with the concept.

This is not an academic paper by any means, this should not be cited as peer-reviewed article, because it is not. I am not a compiler engineer, never was and probably never will be. Following information is just the result of messing with LLVM for two weekends.

Problem introduction.

In ancient times when computers were made of stone with primitive tools by cavepeople CPUs ran operations within concrete order: CPU fetches an instruction, fetches operands if they are available, pass the instruction with operands to execution unit, execute the instruction and write result of the instruction to register file.

There is a problem introduced by condition within the cycle - *if* operands are available. If they are not CPU stalls until it is possible to fetch those.

One of the solutions to stall problem was introduction of out-of-order CPU pipelines and speculative execution[1].

The key idea of speculative execution is CPU calculate other instructions ahead of time and, whenever the time of their execution has come, pretend they were executed just now and pull results from under the carpet in order introduced to machine in first place.

In general, the out-of-order execution is more broad than that. Some of basic operations (including memory operations such as store and load) could be reordered by

[1] Obviously more famous not for the execution performance boost, but for “Spectre” and “Meltdown” vulnerabilities. Almost T.J. Kaczynski reference in «Digit Reversal Without Apology» by Lara Pudwell: «Better known for his other work».

CPU in order to improve performance. This approach introduces us a memory ordering concept.

Depending on underlying hardware (and toolchain on that matter) we can expect certain memory ordering approaches from the system. From strictest to less strict they go as follows: sequential consistency, total store ordering, weak memory model with data dependency and notoriously weak memory model[2]. If you are not familiar, please go read a link in a footnote, I will wait here.

There are plenty of servers that currently runs with ARM CPUs with a weak memory model. This may cause problems in multithreaded environment. Let us start with classical example.

CPU1	CPU2
x = 1	y = 1
a = y	b = x

Suppose we have a multithreaded program with two global variables: a and b, initially set to 0 (a = 0, b = 0). Each CPU puts a value of 1 to variables x and y.

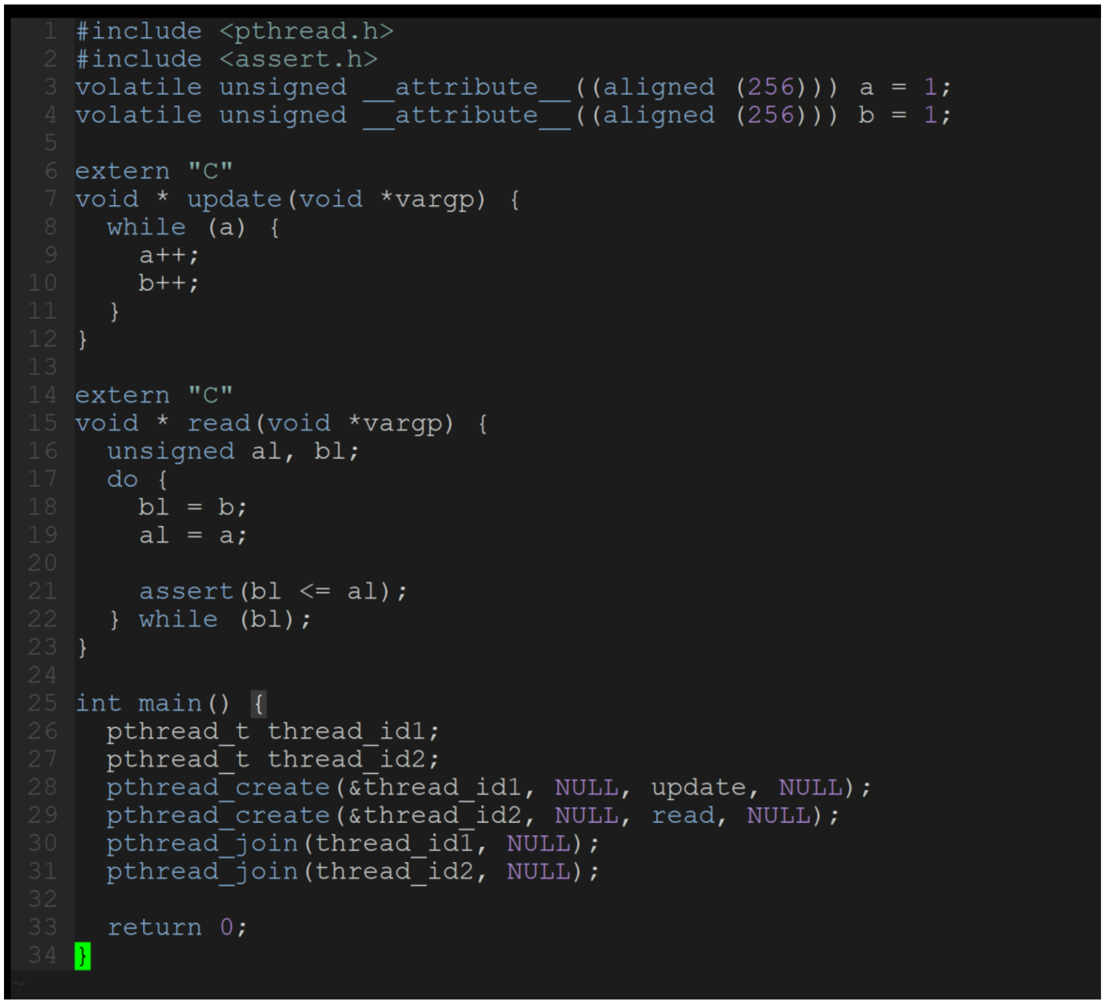
No matter what CPU executes first – CPU1 or CPU2 we, as a sane person, expect to have a == 1 OR b == 1 OR both if we got lucky. We can have some execution races, but the assigning x and y value of one should go first, right?

Not really.

Due to weak memory model CPU can reorder instructions within execution of the single-threaded domain if the reordering does not change the execution flow. This means CPU1 can reorder instructions for its thread, the CPU2 can do the same and as a result programmer (or and user) end up with two variables a == b == 0.

[2] For further reading refer to: https://preshing.com/20120930/weak-vs-strong-memory-models/. Also, read about Alpha AXP memory model, this is hilarious.

Since that example seems quite synthetic, please, let me introduce real-life example of such program[3][4] (see Figure 1).



```
 1 #include <pthread.h>
 2 #include <assert.h>
 3 volatile unsigned __attribute__((aligned (256))) a = 1;
 4 volatile unsigned __attribute__((aligned (256))) b = 1;
 5
 6 extern "C"
 7 void * update(void *vargp) {
 8   while (a) {
 9     a++;
10     b++;
11   }
12 }
13
14 extern "C"
15 void * read(void *vargp) {
16   unsigned al, bl;
17   do {
18     bl = b;
19     al = a;
20
21     assert(bl <= al);
22   } while (bl);
23 }
24
25 int main() {
26   pthread_t thread_id1;
27   pthread_t thread_id2;
28   pthread_create(&thread_id1, NULL, update, NULL);
29   pthread_create(&thread_id2, NULL, read, NULL);
30   pthread_join(thread_id1, NULL);
31   pthread_join(thread_id2, NULL);
32
33   return 0;
34 }
```

Figure 1. Multithreaded example for relaxed/strict MM

Here we have program with two separate threads, one to update the values and other to read them. We expect code to be executed in strict order, so the assert on line 21 will never hit. If you compile and run it on some Intel CPU it will do so and run until you pull the plug without any problems. That happens because Intel utilize TSO

[3] This is slightly modified program, originally written by Viktor Mustlya for Konstantin Vladimirov's Intel C++ course. All props to them.

[4] In order to have some consistency within paper all code will be in images, but I will provide link to repository separately, with instructions how to compile.

memory model. Pretty much any ARM CPU will cause assertion due to reordering of stores and loads.

Problem statement.

The problem to solve is not quite clear described, so I decided to think a bit and rephrase initial statement as below.

I think many companies have a huge codebase that written with some concurrency safety in mind. The code runs well, but they are not expecting change of paradigm with running programs on weak memory model microprocessors. It is probably not feasible to rewrite it due to cost/resources issues. Can we do something without rewriting the multimillion lines-of-code projects?

Problem approach.

Probably. In order to make programs run in certain order, CPU manufacturers introduced an instruction type – fence (or barrier). Fence marks the place in program that splits execution flow to before and after and guaranties that all instructions before fence will be executed prior to that fence and instructions after fence – will be executed after it. The CPU can reorder instructions before fence, can reorder them after, but fence marks a border for out-of-order execution that CPU reordering should consider.

If we modify the initial example as follows:

CPU1	CPU2
x = 1	y = 1
fence	fence
a = y	b = x

we ensure that instruction “x = 1” will ran strictly before “a = y”, same applies to “y = 1” and “b = x”.

ARM architecture has different fence instructions:

- DSB or Data synchronization barrier, completes when all instructions before this instruction complete;
- DMB or Data memory barrier, ensures that all explicit memory accesses before the DMB instruction complete before any explicit memory accesses after the DMB instruction start;
- ISB or Instruction Synchronization Barrier, flushes the pipeline in the processor, so that all instructions following the ISB are fetched from cache or memory, after the ISB has been completed.

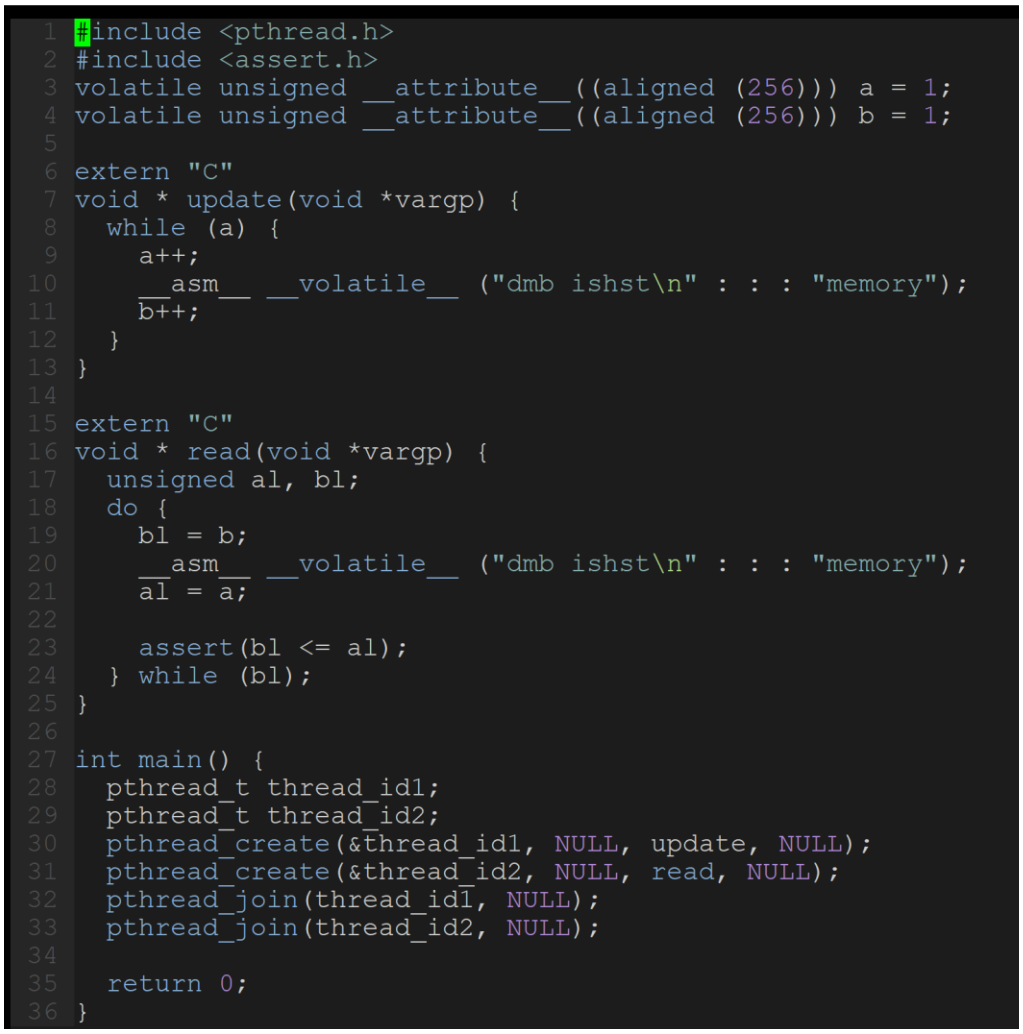
In a real program we can place fences by hand, using __asm__ or using some locks and atomics. We can use fence instruction to make the execution flow of the program stricter (see Figure 2). After adding fence, the assert would not be hit.

I investigated for a while and unfortunately could not find any non-academical solution to putting fences in code. Obviously, people who trying to fix concurrency in weak memory models are smart but 99 times out of 100 you cannot just pick their solution from the shelf and introduce it to the development process. Almost any tool I could find works on small snippets of Assembly code and strange input formats[5]. Almost none of them were updated for eternity.

Since I have a computer security background I though of major security instrumentalization, such as stack canaries and Meltdown/Spectre fixes and decided to use same approach to the instrumentalization of memory instructions at the compilation stage.

We can automatically put fences around load and store instructions and thus fix some of the issues of weak memory model. For the cost of performance, of course.

[5] Also, probably a sacrifice of the first-born if you are not carefully reading modified MIT license



```
#include <pthread.h>
#include <assert.h>
volatile unsigned __attribute__((aligned (256))) a = 1;
volatile unsigned __attribute__((aligned (256))) b = 1;

extern "C"
void * update(void *vargp) {
  while (a) {
    a++;
    __asm__ __volatile__ ("dmb ishst\n" : : : "memory");
    b++;
  }
}

extern "C"
void * read(void *vargp) {
  unsigned al, bl;
  do {
    bl = b;
    __asm__ __volatile__ ("dmb ishst\n" : : : "memory");
    al = a;

    assert(bl <= al);
  } while (bl);
}

int main() {
  pthread_t thread_id1;
  pthread_t thread_id2;
  pthread_create(&thread_id1, NULL, update, NULL);
  pthread_create(&thread_id2, NULL, read, NULL);
  pthread_join(thread_id1, NULL);
  pthread_join(thread_id2, NULL);

  return 0;
}
```

Figure 2. Same example, but with fences

Instrumentalization module, actual submission.

Modern compilers, such as LLVM have modular structure. LLVM compiler has front ends for different languages, back ends to architecture-dependent code and modules that works on intermediate representation level (LLVM IR). Front end parses parses and lexes the source code generating an intermediate language representation and the second layer converts the intermediate representation to actual assembly machine code optimized for different processor architectures.

LLVM compiler introduces passes – modules that can make certain changes to IR. Passes are usually used for optimizations, instrumentalization and IR modification in general.

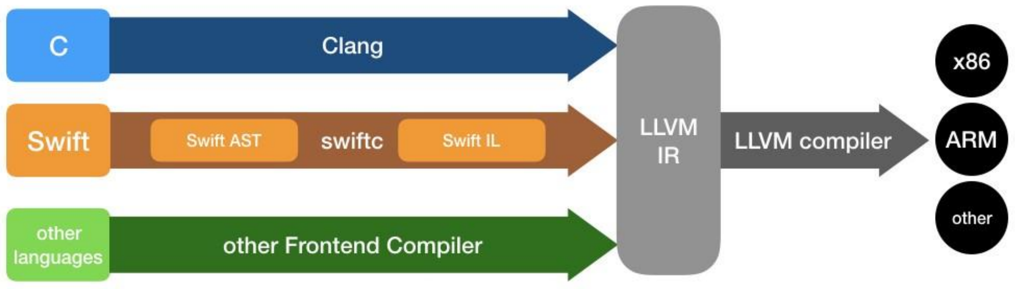


Figure 3. LLVM Frontend-Backend Compiler Architecture[6]

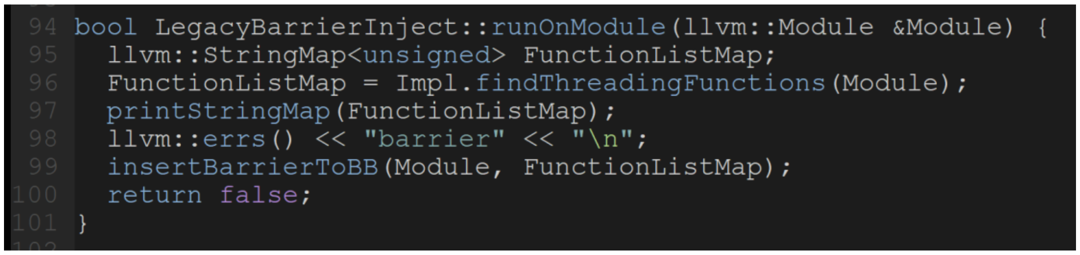
As a submission I introduce the LLVM-module that does two things:

- It inspects module and finds all functions that being run concurrently. For the sake of prototyping we only search for pthread functions;
- Instrumenting all load and store instructions on spoken functions.

I will walk through all module to explain how it works. Let us look at the callback inside the LLVM pass (see Figure 4).

It creates a StringMap structure FunctionListMap. This structure saves all functions that being run with pthread-create function.

[6] Source of image and LLVM explanations: https://medium.com/@JMangia/swift-c-llvm-compiler-optimization-842012568bb7

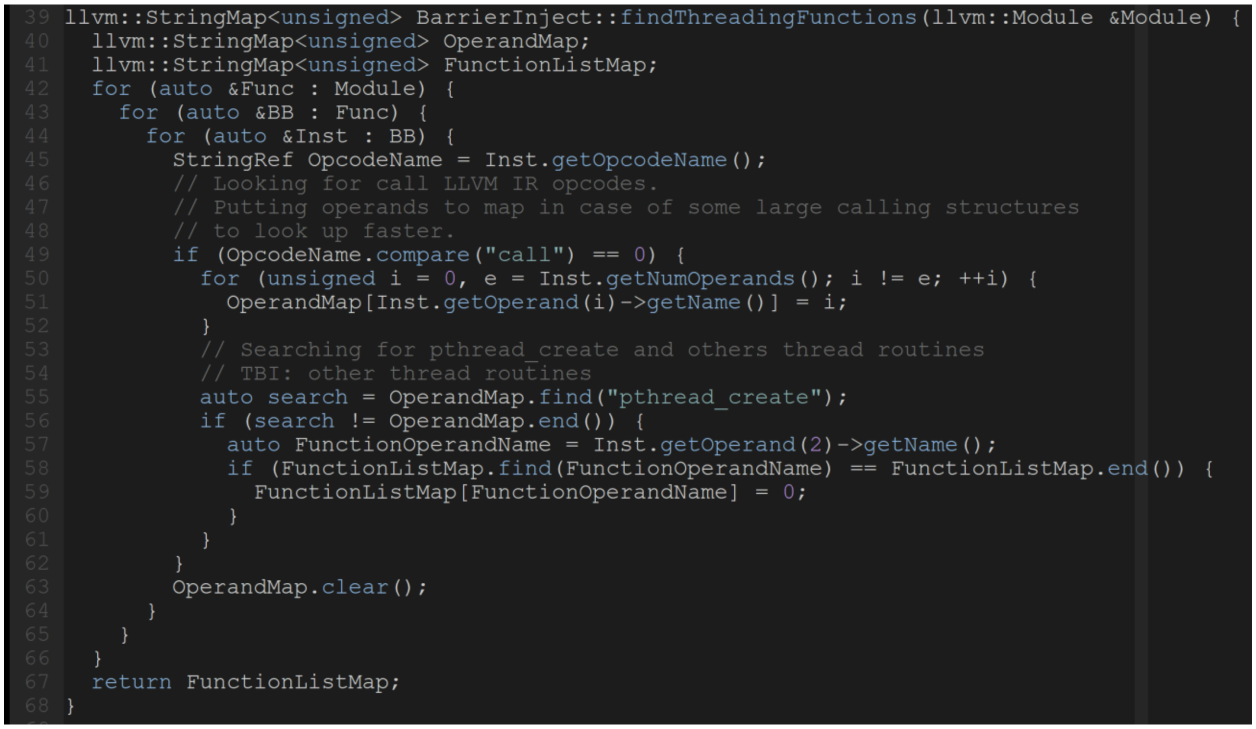


```
bool LegacyBarrierInject::runOnModule(llvm::Module &Module) {
  llvm::StringMap<unsigned> FunctionListMap;
  FunctionListMap = Impl.findThreadingFunctions(Module);
  printStringMap(FunctionListMap);
  llvm::errs() << "barrier" << "\n";
  insertBarrierToBB(Module, FunctionListMap);
  return false;
}
```

Figure 4. runOnModule callback

The map with these instructions being passed to next fence inject routine – insertBarrierToBB.

Source code for the findThreadingFunction function introduced on the figure (see Figure 5).

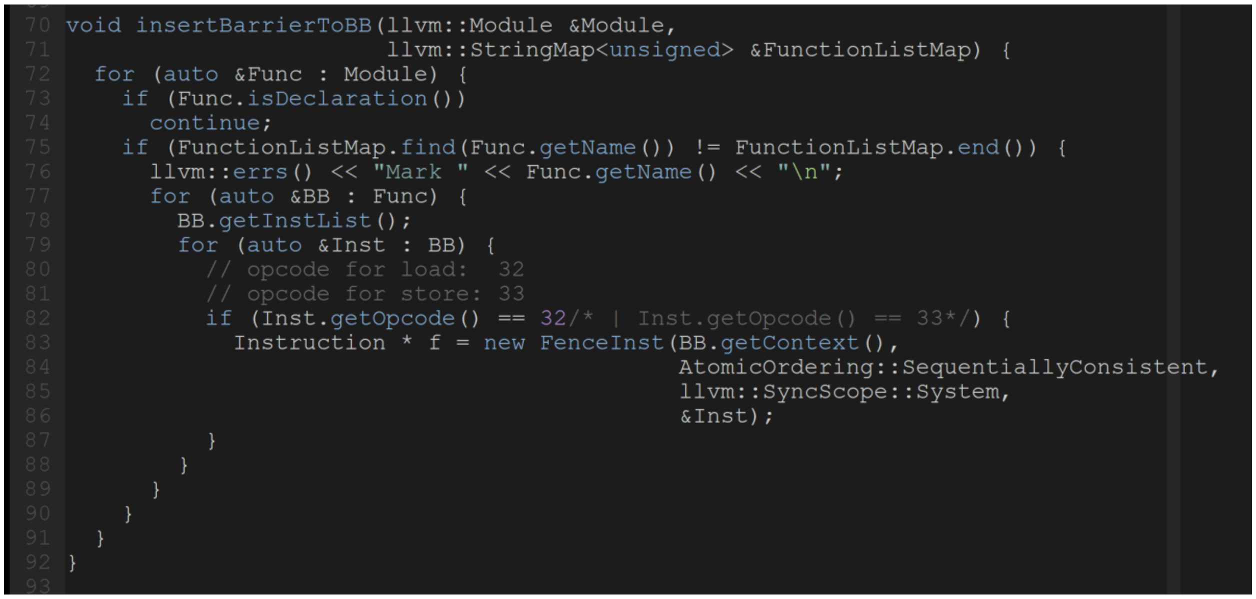


```
llvm::StringMap<unsigned> BarrierInject::findThreadingFunctions(llvm::Module &Module) {
  llvm::StringMap<unsigned> OperandMap;
  llvm::StringMap<unsigned> FunctionListMap;
  for (auto &Func : Module) {
    for (auto &BB : Func) {
      for (auto &Inst : BB) {
        StringRef OpcodeName = Inst.getOpcodeName();
        // Looking for call LLVM IR opcodes.
        // Putting operands to map in case of some large calling structures
        // to look up faster.
        if (OpcodeName.compare("call") == 0) {
          for (unsigned i = 0, e = Inst.getNumOperands(); i != e; ++i) {
            OperandMap[Inst.getOperand(i)->getName()] = i;
          }
          // Searching for pthread_create and others thread routines
          // TBI: other thread routines
          auto search = OperandMap.find("pthread_create");
          if (search != OperandMap.end()) {
            auto FunctionOperandName = Inst.getOperand(2)->getName();
            if (FunctionListMap.find(FunctionOperandName) == FunctionListMap.end()) {
              FunctionListMap[FunctionOperandName] = 0;
            }
          }
        }
        OperandMap.clear();
      }
    }
  }
  return FunctionListMap;
}
```

Figure 5. findThreadingFunction

This module iterates through function and opcodes of module. We are looking for opcode "call" and acquiring its operands. If one of the operands is "pthread_create" we decide to add the function threaded to return StringMapStructure. This will work for pretty much any threading routines except esoteric ones. Why we do so? Because we should not instrumentalize code that runs on a single thread. We will put a lot of fences and achieve nothing since single-threaded programs should run correctly within the domain.

The function insertBarrierToBB is self-explanatory. We are looking for store and load opcodes (store is commented out on screenshot for test purposes, so we are instrumenting only loads) and if we found one, we add a fence instruction right before it.



```
void insertBarrierToBB(llvm::Module &Module,
                       llvm::StringMap<unsigned> &FunctionListMap) {
  for (auto &Func : Module) {
    if (Func.isDeclaration())
      continue;
    if (FunctionListMap.find(Func.getName()) != FunctionListMap.end()) {
      llvm::errs() << "Mark " << Func.getName() << "\n";
      for (auto &BB : Func) {
        BB.getInstList();
        for (auto &Inst : BB) {
          // opcode for load:  32
          // opcode for store: 33
          if (Inst.getOpcode() == 32/* | Inst.getOpcode() == 33*/) {
            Instruction * f = new FenceInst(BB.getContext(),
                                            AtomicOrdering::SequentiallyConsistent,
                                            llvm::SyncScope::System,
                                            &Inst);

          }
        }
      }
    }
  }
}
```

Figure 6. insertBarrierToBB function

After running this LLVM pass on the module we will get LLVM IR with instrumentalized routines.

How it can be used.

1) Apply patches obtained from my repository (see https://github.com/outofhere/Research/tree/master/2020/weak_memory_model_llmv_pass) to your freshly llvm-10 repository and compile it;
2) Get yourself a nice failing program (see Figure 7 and Figure 8) and compile it to bitcode for ARM:

```
$LLVM_DIR/bin/clang-10 -S -emit-llvm --target=aarch64-arm-none-eabi threadingrace.cc -o threadingrace.ll
```

3) Make sure we don’t have fence instructions in bitcode yet (see Figure 9);

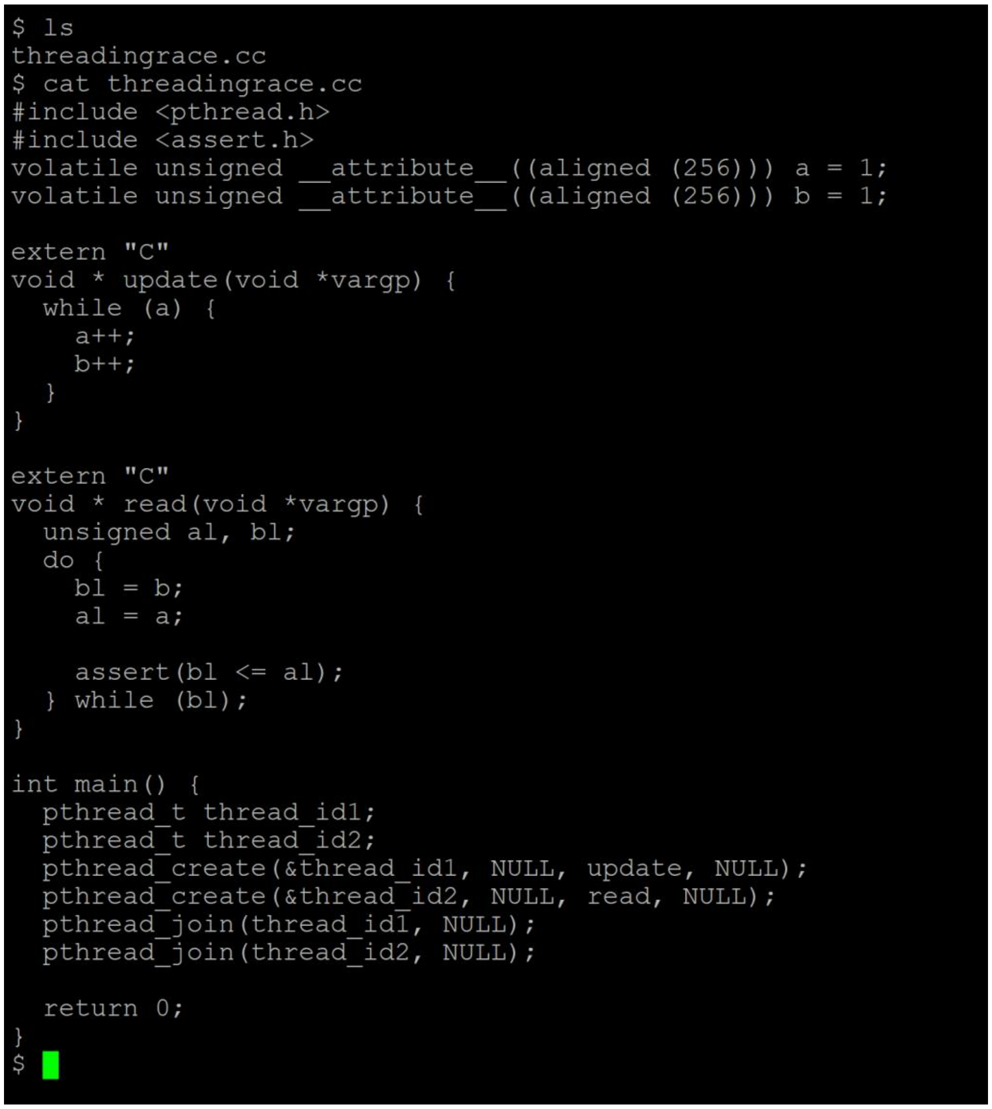
4) Run custom llvm-pass as follows (see Figure 10):

```
$LLVM_DIR/bin/opt -load LLVMBarrierInject.so -
barrier-inject threadingrace.ll >
threadingrace_out.bc
```

5) Compile bitcode to *.S file (see Figure 11):

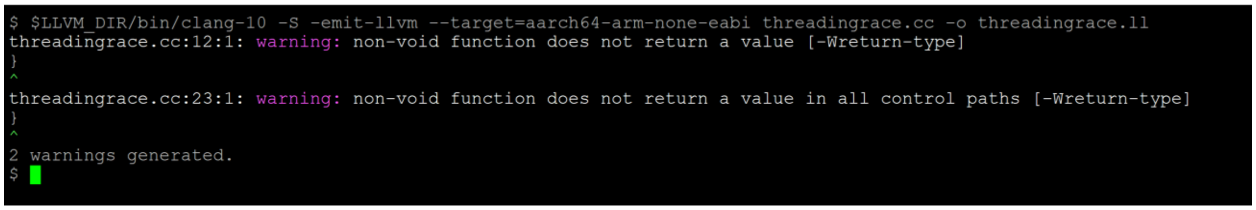
```
$LLVM_DIR/bin/llc threadingrace_out.bc
```

6) At this point we already have an assembly code, so we can ensure that there are fences put (see Figure 11), but in order to do everything safe and sound let’s compile it to the end with aarch64 compiler (see Figure 13).



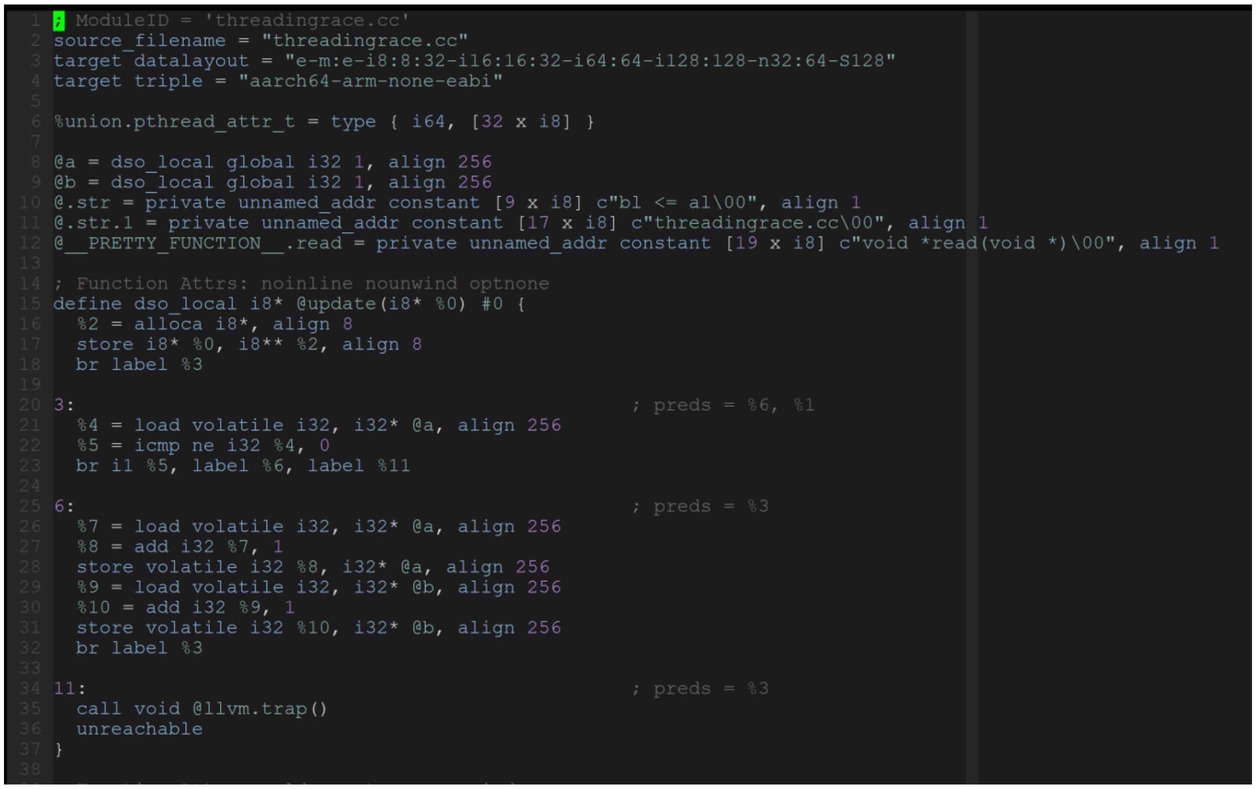
```
$ ls
threadingrace.cc
$ cat threadingrace.cc
#include <pthread.h>
#include <assert.h>
volatile unsigned __attribute__((aligned (256))) a = 1;
volatile unsigned __attribute__((aligned (256))) b = 1;

extern "C"
void * update(void *vargp) {
  while (a) {
    a++;
    b++;
  }
}

extern "C"
void * read(void *vargp) {
  unsigned al, bl;
  do {
    bl = b;
    al = a;

    assert(bl <= al);
  } while (bl);
}

int main() {
  pthread_t thread_id1;
  pthread_t thread_id2;
  pthread_create(&thread_id1, NULL, update, NULL);
  pthread_create(&thread_id2, NULL, read, NULL);
  pthread_join(thread_id1, NULL);
  pthread_join(thread_id2, NULL);

  return 0;
}
$
```

Figure 7. Code of the program without fences



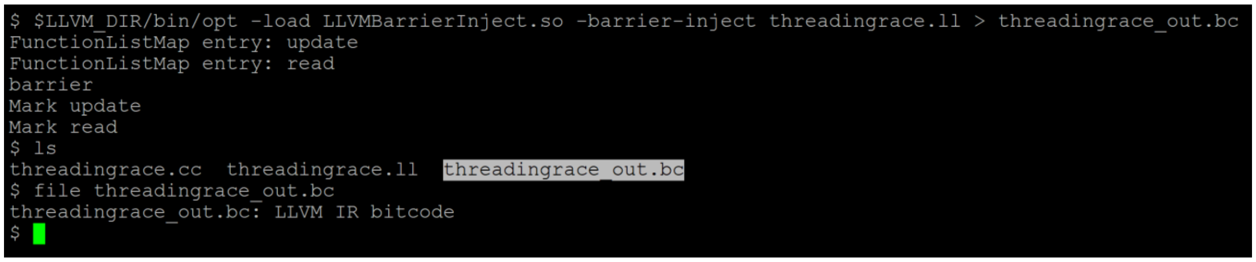
```
$ $LLVM_DIR/bin/clang-10 -S -emit-llvm --target=aarch64-arm-none-eabi threadingrace.cc -o threadingrace.ll
threadingrace.cc:12:1: warning: non-void function does not return a value [-Wreturn-type]
}
^
threadingrace.cc:23:1: warning: non-void function does not return a value in all control paths [-Wreturn-type]
}
^
2 warnings generated.
$ 
```

Figure 8. Compilation process



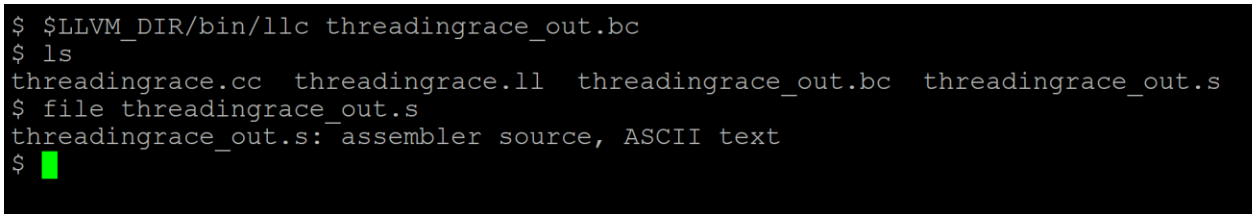
```
; ModuleID = 'threadingrace.cc'
source_filename = "threadingrace.cc"
target datalayout = "e-m:e-i8:8:32-i16:16:32-i64:64-i128:128-n32:64-S128"
target triple = "aarch64-arm-none-eabi"

%union.pthread_attr_t = type { i64, [32 x i8] }

@a = dso_local global i32 1, align 256
@b = dso_local global i32 1, align 256
@.str = private unnamed_addr constant [9 x i8] c"bl <= al\00", align 1
@.str.1 = private unnamed_addr constant [17 x i8] c"threadingrace.cc\00", align 1
@__PRETTY_FUNCTION__.read = private unnamed_addr constant [19 x i8] c"void *read(void *)\00", align 1

; Function Attrs: noinline nounwind optnone
define dso_local i8* @update(i8* %0) #0 {
  %2 = alloca i8*, align 8
  store i8* %0, i8** %2, align 8
  br label %3

3:                                                ; preds = %6, %1
  %4 = load volatile i32, i32* @a, align 256
  %5 = icmp ne i32 %4, 0
  br i1 %5, label %6, label %11

6:                                                ; preds = %3
  %7 = load volatile i32, i32* @a, align 256
  %8 = add i32 %7, 1
  store volatile i32 %8, i32* @a, align 256
  %9 = load volatile i32, i32* @b, align 256
  %10 = add i32 %9, 1
  store volatile i32 %10, i32* @b, align 256
  br label %3

11:                                               ; preds = %3
  call void @llvm.trap()
  unreachable
}

```

Figure 9. LLVM IR without barriers



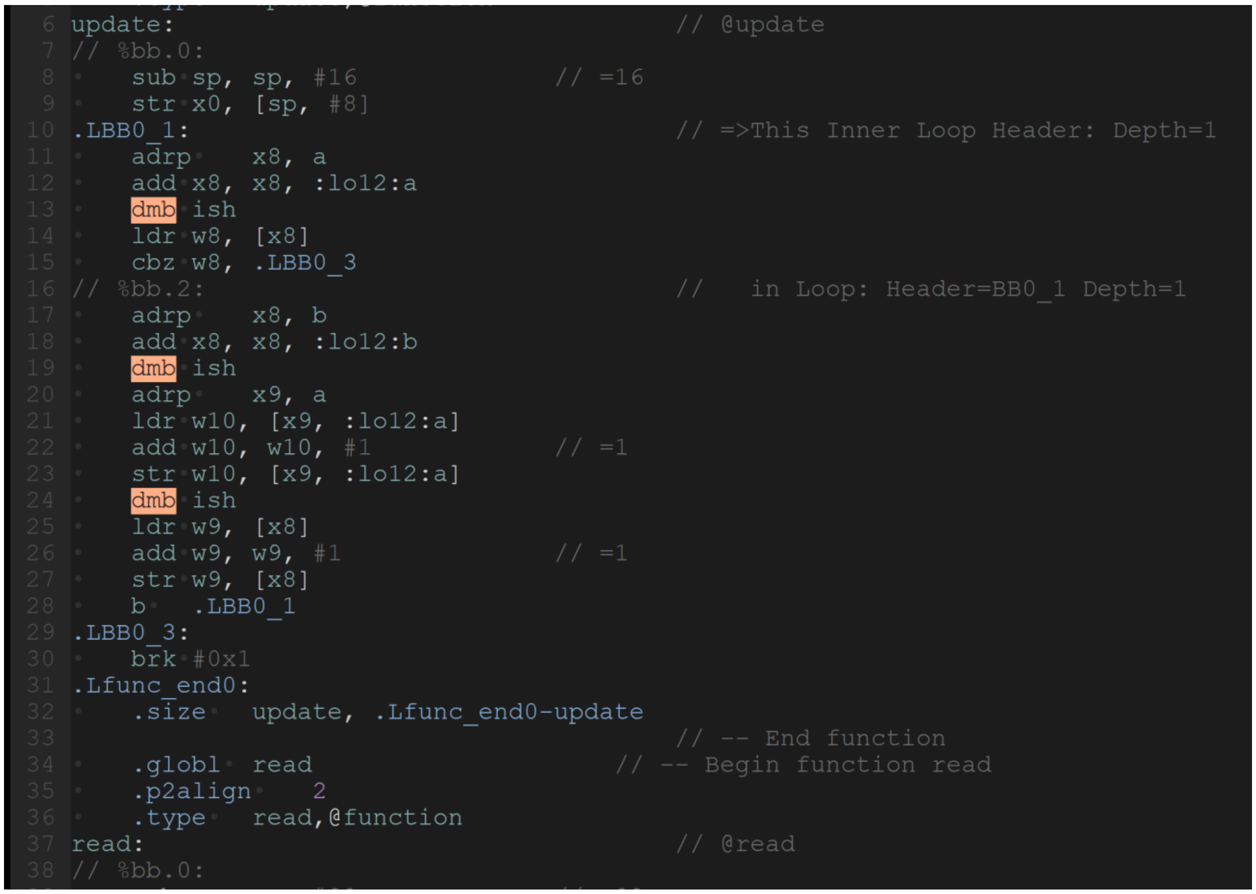
```
$ $LLVM_DIR/bin/opt -load LLVMBarrierInject.so -barrier-inject threadingrace.ll > threadingrace_out.bc
FunctionListMap entry: update
FunctionListMap entry: read
barrier
Mark update
Mark read
$ ls
threadingrace.cc  threadingrace.ll  threadingrace_out.bc
$ file threadingrace_out.bc
threadingrace_out.bc: LLVM IR bitcode
$ 
```

Figure 10. Running pass on the LLVM IR



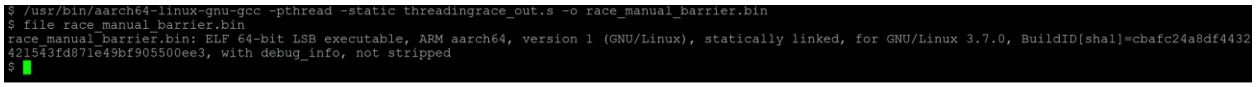
```
$ $LLVM_DIR/bin/llc threadingrace_out.bc
$ ls
threadingrace.cc  threadingrace.ll  threadingrace_out.bc  threadingrace_out.s
$ file threadingrace_out.s
threadingrace_out.s: assembler source, ASCII text
$ 
```

*Figure 11. Bitcode after pass to *.S*



```
update:                                         // @update
// %bb.0:
	sub sp, sp, #16                 // =16
	str x0, [sp, #8]
.LBB0_1:                                        // =>This Inner Loop Header: Depth=1
	adrp	x8, a
	add x8, x8, :lo12:a
	dmb ish
	ldr w8, [x8]
	cbz w8, .LBB0_3
// %bb.2:                                       //   in Loop: Header=BB0_1 Depth=1
	adrp	x8, b
	add x8, x8, :lo12:b
	dmb ish
	adrp	x9, a
	ldr w10, [x9, :lo12:a]
	add w10, w10, #1                // =1
	str w10, [x9, :lo12:a]
	dmb ish
	ldr w9, [x8]
	add w9, w9, #1                  // =1
	str w9, [x8]
	b	.LBB0_1
.LBB0_3:
	brk #0x1
.Lfunc_end0:
	.size	update, .Lfunc_end0-update
                                        // -- End function
	.globl	read                    // -- Begin function read
	.p2align	2
	.type	read,@function
read:                                           // @read
// %bb.0:
```

Figure 12. Expanded instructions before LDR instructions

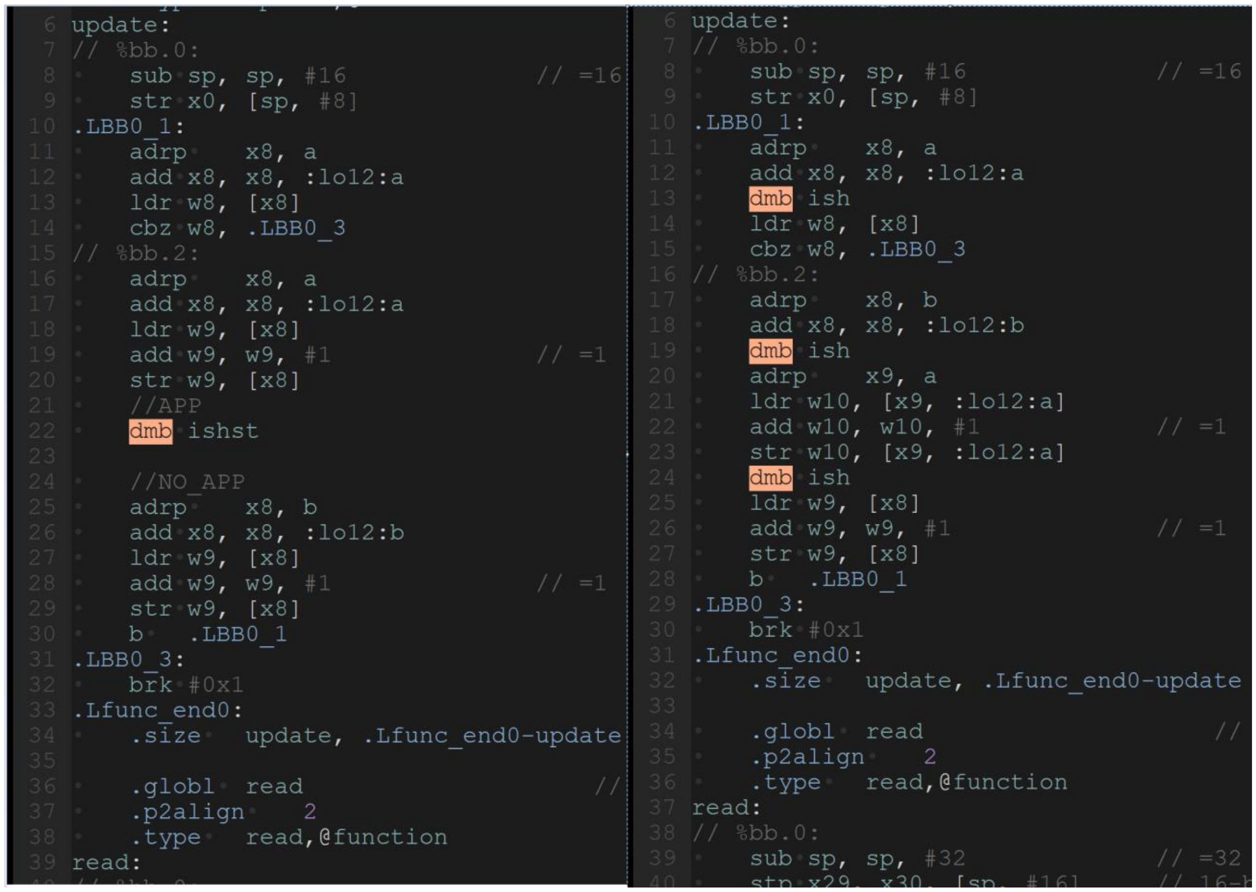


```
$ /usr/bin/aarch64-linux-gnu-gcc -pthread -static threadingrace_out.s -o race_manual_barrier.bin
$ file race_manual_barrier.bin
race_manual_barrier.bin: ELF 64-bit LSB executable, ARM aarch64, version 1 (GNU/Linux), statically linked, for GNU/Linux 3.7.0, BuildID[sha1]=cbafc24a8df4432
421543fd871e49bf905500ee3, with debug_info, not stripped
$ 
```

Figure 13. End of compilation and linking process

Wrap up.

The current implementation of fence insertion pass is quite crude and non-efficient. For reference, see manually put fence versus automated (see Figure 14).



```
update:
// %bb.0:
    sub sp, sp, #16             // =16
    str x0, [sp, #8]
.LBB0_1:
    adrp    x8, a
    add x8, x8, :lo12:a
    ldr w8, [x8]
    cbz w8, .LBB0_3
// %bb.2:
    adrp    x8, a
    add x8, x8, :lo12:a
    ldr w9, [x8]
    add w9, w9, #1              // =1
    str w9, [x8]
    //APP
    dmb ishst

    //NO_APP
    adrp    x8, b
    add x8, x8, :lo12:b
    ldr w9, [x8]
    add w9, w9, #1              // =1
    str w9, [x8]
    b   .LBB0_1
.LBB0_3:
    brk #0x1
.Lfunc_end0:
    .size   update, .Lfunc_end0-update

    .globl  read                    //
    .p2align    2
    .type   read,@function
read:
```

```
update:
// %bb.0:
    sub sp, sp, #16             // =16
    str x0, [sp, #8]
.LBB0_1:
    adrp    x8, a
    add x8, x8, :lo12:a
    dmb ish
    ldr w8, [x8]
    cbz w8, .LBB0_3
// %bb.2:
    adrp    x8, b
    add x8, x8, :lo12:b
    dmb ish
    adrp    x9, a
    ldr w10, [x9, :lo12:a]
    add w10, w10, #1            // =1
    str w10, [x9, :lo12:a]
    dmb ish
    ldr w9, [x8]
    add w9, w9, #1              // =1
    str w9, [x8]
    b   .LBB0_1
.LBB0_3:
    brk #0x1
.Lfunc_end0:
    .size   update, .Lfunc_end0-update

    .globl  read                //
    .p2align    2
    .type   read,@function
read:
// %bb.0:
    sub sp, sp, #32             // =32
```

Figure 14. Manual fence on the left, only loads pass on the right

Fence instructions may require hundreds of cycles which often we cannot afford on productive systems.

If we want instrumentalize all store and load instructions, we end up with 7 DMB instructions in total for a small snippet of code. If we put fences only before loads, we will have 3 DMBs for a same snippet and non-zero chance that we still have races. This still will work for the simple code I presented, but probably will not in large-scaled systems.

In order to full implement this approach we should utilize some of the academic paper and write a good analytical pass which decides if we require fence between memory instructions. This is feasible and was made for some of the academic papers I stumbled upon during research.

What happens if we instrumentalize a code already written with atomics? We can use some of the optimizations described in EuroLLVM’s presentation “Efficient code generation for weakly ordered architectures”: https://llvm.org/devmtg/2014-04/PDFs/Talks/Reinoud-EuroLLVM.pdf

It describes how we can get rid of some of repeating fence instructions and how to optimize fencing with loops.

I strongly believe that automatic instrumentalization is a way to go for this problem, however it is not feasible to write somewhat decent analytical pass in the period of two weeks with a team of one person. If you want to PM some of your thoughts on the issue – feel free to do so, I left an e-mail at the end of the paper.

24/05/2020

Igor Chervatyuk (ichervatyuk@gmail.com)