Detecting Address Dependencies Broken by the LLVM Backend in the Linux Kernel

Martin Fink
Martin.Fink@cit.tum.de

Abstract

This paper discusses the issue of dependency ordering in the context of weak memory models and the Linux kernel. The Linux Kernel Memory Model (LKMM) and C11 memory model differ in slight aspects, most notably in their implementation of memory-order-consume, which can cause compilers to break program assumptions by reordering instructions in ways that are legal in C11 but not in the LKMM.

To address this problem, ongoing work is proposing adding LLVM analysis passes to detect broken dependencies in LLVM's middle end [4]. In this work, we extend this approach by implementing an additional LLVM analysis pass for the aarch64 backend, which detects additional broken dependencies introduced by LLVM's backend optimization passes.

Our experimental results demonstrate that our approach can detect artificially broken dependencies in the Linux kernel, which can improve confidence in the correctness of the compiler and the Linux kernel itself.

1 Background

1.1 Memory Consistency Models

Memory Consistency Models (MCM) specify in which order operations can become visible to other processes in a single system. Generally, we differentiate between three different memory models that provide different guarantees.

Sequentially Consistent Ordering Reads and writes are executed in the exact order they are defined in. While interleaving of processes is allowed, reordering of any instructions is not. This Memory Model offers the strongest guarantees, but it is not found in modern architectures, as it leaves potential performance gains on the table. If two memory operations do not depend on each other, the CPU might be able to reorder one of the operations to a program point where it will not impact the behavior of the current thread.

Total Store Ordering While writes are still executed in the order they were defined in the program code, the processor is allowed to reorder reads. Most notably, this memory model is implemented in x86 CPUs.

Weak Memory Models In Weak Memory Models, reads and writes are allowed to be reordered. This type of memory model gives the weakest guarantees and is implemented in Arm and Risc-V chips. The programmer is expected to insert explicit memory barriers if instructions should not be ordered across a point in the program. Additionally, weak architectures are allowed to speculatively execute loads and stores before conditions on which the load depends have been resolved. There exists the notion of data dependencies [10], which imposes an ordering on these instructions if they depend on the outcome of another.

In fig. 1 we show two programs that allow for different behaviors depending on which memory model they are being run on. On CPUs implementing TSO such as x86, the code is unproblematic, since thread A would not be permitted to reorder the writes. On weaker CPUs such as ARM, this code is problematic without an explicit memory barrier.

```
WRITE(X, 42);
WRITE(lock, 0);
```

(a) Thread A

```
if (READ(lock)==0)
  READ(X);
```

(b) Thread B

Figure 1: Two threads in parallel where reordering of thread A might introduce unwanted behavior in thread B

1.2 C memory model

C11 [12] introduced threads, atomics, and a memory model with it. Functions such as `atomic_load`, `atomic_store` take an additional argument specifying the memory synchronization order. There are five possible memory synchronizations: (1) `memory_order_relaxed` No guarantees

made, no barriers inserted, (2) `memory_order_seq_cst` sequential consistency ordering; operations are executed in the order they are defined in the program code, (3) `memory_order_release` allows no reordering before the load operation using this ordering, (4) `memory_order_acquire` allows no reordering after the store operation using this ordering, and (5) `memory_order_consume`. Ordering (5) is almost the same as ordering (3), with the important distinction that no values may be reordered before the load operation that *depend on the value being loaded*. In today's production compilers, this ordering is strengthened to `memory_order_acquire`, as no efficient and correct implementation of `memory_order_consume` is known. Racy accesses to non-atomics are undefined behavior post C11 and should only be used for memory not shared between threads.

1.3 The Linux Kernel Memory Model

While similar to the C11 memory model (which it predates), the Linux Kernel Memory Model (LKMM) differs in some aspects. Some of the macros used to access shared memory (e.g. `rcu_dereference`) depend on `memory_order_consume` [7], which is not provided by any implementation of the C11 memory model today. This is one of the reasons that many declare the C11 memory model unfit for the Linux kernel [1].

The Linux Kernel Memory Model is defined in [2]. It abstracts into events, which are divided into three categories: [6]

1. Read events. These are implemented by macros such as `READ_ONCE` or `rcu_dereference`.
2. Write events. These are implemented by macros such as `WRITE_ONCE` or `atomic_set`.
3. Fence events. These are implemented by macros such as `rcu_read_lock` and correspond to memory barriers.

These macros provide some guarantees to the programmers, such as preventing load or store tearing or inserting memory barriers on weak CPUs that don't provide dependency ordering such as the DEC Alpha. Additionally, they prevent merging, removing, or duplicating memory accesses. Memory access to shared memory without the use of one of these macros is undefined behavior and may lead to bugs and unintended consequences.

Different events may be linked by the following relations:

Address dependencies The address of a read/write is dependent on the address read in an earlier part of the program. An example can be seen in fig. 2.

Control dependencies A read/write is conditionally executed, while the condition is dependent on a value read in an earlier part of the program. An example can be seen in fig. 3.

```
int a[20];
int i;

i = READ_ONCE(x);
READ_ONCE(a[i]);
```

Figure 2: An example of an address dependency. The address of the second `READ_ONCE` depends on the value read into `i`.

```
if (READ_ONCE(x))
    WRITE_ONCE(y, 1);
```

Figure 3: An example of a control dependency. The execution of the `WRITE_ONCE` macro depends on the value read from `x`.

Most CPUs the Linux kernel supports provide dependency ordering on single threads, with one exception being the DEC ALPHA. On this CPU, the macros mentioned above also insert memory barriers.

1.4 LLVM

LLVM is a set of compiler and toolchain technologies, designed around an intermediate representation (IR) [8, 5]. The IR serves as a portable high-level assembly language, which can be targeted by compiler frontends such as clang[1], rustc[2], swiftc[3], and others. The LLVM project includes everything that is needed to go from a C program to an executable and is designed in a highly modular fashion.

$$\mathbb{P}_{\mathsf{C}} \xrightarrow{\mathsf{clang}} \mathbb{P}_{\mathsf{IR}} \xrightarrow{\mathsf{LLVM.opt}} \mathbb{P}'_{\mathsf{IR}} \xrightarrow{\mathsf{LLVM.codegen}} \mathbb{P}_{\mathsf{target}} \qquad (1)$$

Figure 4: Workflow of clang and LLVM

In section 1.4 we see how a C program is compiled to IR, optimized by opt, and then transformed into machine code by codegen. Most optimizations are performed on the target-independent IR, but there are also some target-specific in the machine IR (MIR), which is part of LLVM's codegen.

1.4.1 Machine Intermediate Representation

Machine IR is LLVM's format that is used to generate code in its very last stage of the code generation pipeline [9]. It is a target-agnostic form that abstracts over actual machine instructions. Each `MachineInstr` consists of an opcode and a list of operands. The Machine IR may be in SSA form using virtual registers or in non-SSA form (post-register allocation)

[1] https://clang.llvm.org
[2] https://www.rust-lang.org
[3] https://swift.org

with physical registers (e.g. `X0`, `X1`) as operands. Additional operands include immediates, flags, and more.

LLVM provides some methods to query generic information about an instruction, e.g. `mayLoad`, `mayStore`, `isCall`, etc. If no such method is available for a desired behavior, this needs to be implemented manually with a condition over the opcode of an instruction. This is highly target-specific and non-portable, but currently, the only way to implement some specific optimizations/analyses.

1.5 Clang Built Linux

While the default compiler toolchain is gcc, there is also support for building the kernel with clang [3]. Most notably, the Android kernel is built with clang instead of gcc. Due to the extensible nature of LLVM, we chose clang/LLVM as our compiler toolchain to implement our analysis.

2 Motivation

In this guided research project we extend the project initially developed by Heidekrüger [4] by developing an analog checker in LLVM's code generation backend. Since the backend implements its target-specific optimizations, there exists a possibility that those optimizations break the assumptions of the LKMM and result in broken dependencies.

3 Design

The algorithm to detect broken dependencies is heavily inspired by the algorithm running in LLVM's middle end [4]. It is based on a breadth-first search. This is required to make sure that in control flow graphs like fig. 5, block `bb.2` is visited before `bb.3`, and both blocks `bb.2` *and* `bb.3` are visited before `bb.4`.

Dependency chains We informally define dependency chains as chains that are started by volatile loads, that contain all instructions that use the value loaded. Each use of the value is pre-dominated by the volatile load.

For a single volatile load, there might be multiple dependency chains, going through different paths in the control flow.

The algorithm in use is described in more detail in [4], so we will not repeat its details here. It consists of three passes:

1. **Annotation pass** Annotate volatile loads/stores. This pass runs first in the optimization pipeline.

2. **Verification pass** Verify dependencies have not been broken. This pass runs after all optimization passes in the middle end.

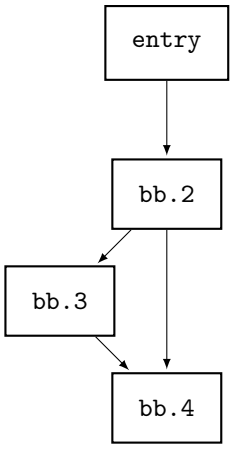

Figure 5: Example of the basic blocks in a function

3. **Backend verification pass** Verify dependencies have not been broken by the code generation pipeline. This pass runs last in the code generation phase, just before emitting the final assembly code. The main work of this thesis has been spent on this pass.

4 Implementation

Our implementation can be found on GitHub[4], where we plan on continuously improving our current algorithm.

Passes 1 and 2 run on LLVM's middle-end IR, while pass 3 runs on LLVM's MIR (Machine IR). Since pass 3 runs just before emitting the final code, it is run after physical register allocation has been performed and is therefore not in SSA form anymore.

We focus on the AArch64 architecture, which is the architecture that is used by ARMv8 and ARMv9. This architecture uses a weak memory model and is a prevalent user of the Linux kernel mainly through Android.

In the following sections, we will only describe sections of the algorithm that significantly differ from the implementation in the middle-end, which is already described in [4].

4.1 Register to value mapping

To keep track of which registers were defined by which instructions, we implemented a register ↔ instruction/argument mapping. In the following text, we will refer to instructions/arguments as *values*, even though they are not related to LLVM IR's `Values`. When entering a function, we map all argument registers the function uses to special argument placeholders. These mark that a value is passed to the function as an argument rather than being defined in the function itself.

[4] https://github.com/martin-fink/llvm-project/tree/dep-checker-backend

When visiting an instruction, we check which registers are defined by the instruction and map those to the instruction itself. When looking up the value stored in a register, the lookup can be done in $O(log(n))$ complexity (n being the number of registers stored in the map), since the mapping is stored in `std::map`. This is superior to an approach where for each value lookup the control flow graph needs to be walked backward until it is found, which would result in a time complexity of $O(n)$ (n being the number of preceding instructions).

Note that at any point in a program, more than one value may be stored in this register $\leftrightarrow$ value mapping, depending on which path the program took through the control flow graph. In the example fig. 6 we see that in block 3, `x0` may be defined by the instruction in block 1 or block 2, depending on which path was taken.

In our implementation, we handle multiple values as multiple dependency chains. Again, in example fig. 6, if `x1` was part of a dependency chain, the control flow graphs would result in two dependency chains: {`x1` $\rightarrow$ `mov x0, x1`} and {`x1` $\rightarrow$ `mov x0, x1` $\rightarrow$ `add x0, 1`}.

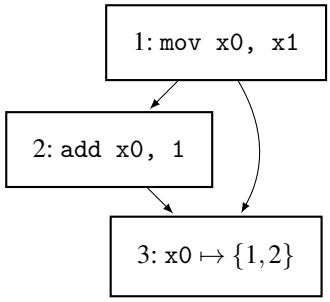

Figure 6: Example of a register possibly holding two different values, depending on the path taken in the control flow graph.

4.2 Metadata propagation

In the middle end, dependency chain information is attached as metadata nodes directly to the instructions. Since these metadata nodes don't exist in the backend anymore, we need to propagate them using another mechanism. Early attempts were made using debug information with explicit debug instructions. This was not suitable, as it had several drawbacks: Debug instructions could only be attached to registers, not other instructions themselves. Debug instructions were frequently lost or reordered and with them the critical information about dependency chains.

We chose to use the much more suited `pcsection` metadata, which was introduced in LLVM in mid-2022 [11]. While not developed for our use case, we were able to abuse them as metadata for the code generation backend. In this process, we also developed two bug fixes for LLVM regarding the implementation and handling of pcsection metadata, which was merged[5], but unfortunately later reverted due to some bugs found in their implementation.

Pcsection metadata is attached to instructions much like IR-level metadata. We copy all LKMM-related metadata generated by the annotation pass into pcsection metadata at the end of the IR verification pass. The instruction selectors and backend optimizers try to keep all pcsection metadata, but there exist some cases where it is lost. This is something we can currently not do anything about but wait for fixes in LLVM itself.

4.3 Instruction handling

Since handling each instruction individually is simply not feasible, we differentiate between the following instruction classes:

4.3.1 Load/store instructions

For load and store instructions we need to check if any metadata annotations are attached to them, and if yes, keep track of dependency chain beginnings or endings. We also need to check if they store or load potential dependency chain values, or overwrite existing dependency chain values. In these cases, we try to minimize false positives and be overly cautious and discard any overwritten dependency chains. We detect if instructions belong in this class through the `mayLoad` and `mayStore` functions. This also means that conditional loads and stores are handled as non-conditional at the moment. This is a possible improvement in the future.

4.3.2 Call instructions

For call instructions, we try to start our interprocedural analysis with a fixed depth of 4 nested calls. For some functions (e.g. external functions, intrinsic functions, ...) it is not possible to analyze the called function. In this case, we again try to minimize false positives and discard potentially broken dependencies.

Additionally, we do not handle functions that pass arguments on the stack, as this would require a more extensible analysis using stack accesses, which we do not do at the moment. This could be implemented in the future, but as the kernel does not define many such functions, we chose to skip them for now.

4.3.3 Return instructions

Return instructions return dependency chains to the caller. This may be a new dependency chain started in the called function, or a continuation of a dependency chain of the caller, where the returned value depends on an argument passed to the called function, which is part of a dependency chain.

[5] https://reviews.llvm.org/D141048

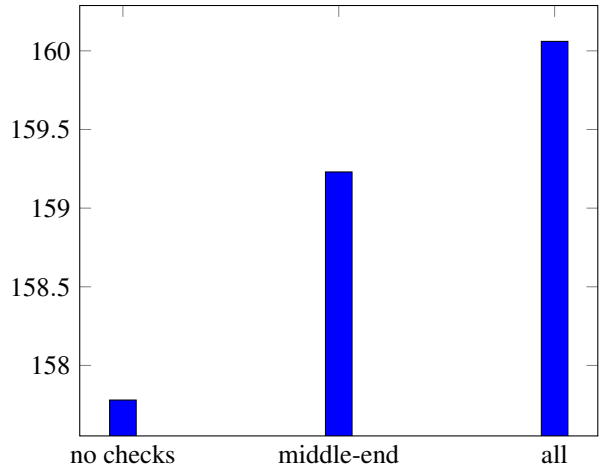

Figure 7: Runtime with no checks, only middle-end checks, and all checks

4.3.4 Inline assembly instructions

Since we cannot analyze inline assembly instructions, we discard all dependency chains that run through inline assembly instructions. This is again an overestimation of non-broken dependency chains, but we are not aware of a practical solution to solve this issue. Most inline assembly instructions in the kernel are jump tables e.g. switch-case statements.

4.3.5 Other instructions

For other instructions, we just add them to the dependency chains if a value that is part of a dependency chain runs through them.

5 Evaluation

To evaluate the dependency checker, we used our tool to compile the Linux kernel for AArch64, since our backend verification pass is written for this architecture and it is a prevalent weak memory model architecture. We ran the checker on Linux 6.2.0-rc2 on an allyesconfig.

To check for the correctness of our tool, we additionally added test cases to the kernel tree, which we artificially broke in the middle end to ensure those cases were caught by our tool. Additionally, we planned on manually checking every reported broken dependency in the kernel itself.

We ran our tests on an AMD EPYC 7713P 64-Core Processor with 512 GB of memory and NixOS 22.11.20230325.6facb7e.

5.1 Overheads

We ran a full Linux build with no checks, with only the middle-end passes, and with all three passes enabled.

As we can see in fig. 7, the checks have a small performance overhead, that can be tolerated in builds, as we expect this tool to be a monitoring tool, rather than being enabled for every build.

5.2 Results

Unfortunately, at the time of writing, our dependency checker did not find any dependencies broken by the backend. This was the expected result, as the optimizations in the backend are not as extensive and involved as those in the middle end.

6 Future work

In the future, we want to improve our implementation by tightening the over-estimations and possibly finding actual dependencies broken by the compiler backend. One possibility would be to further check inline assembly instructions or more accurately check the behaviour of individual instructions, which might be overlooked or handled too generally at the moment.

Additionally, we might look into handling control dependencies in addition to address dependencies.

References

[1] URL: https://gcc.gnu.org/legacy-ml/gcc/2012-02/msg00013.html (visited on 03/24/2023).

[2] J. Alglave, L. Maranget, P. E. McKenney, A. Parri, and A. Stern. “Frightening Small Children and Disconcerting Grown-Ups: Concurrency in the Linux Kernel”. In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’18. Williamsburg, VA, USA: Association for Computing Machinery, 2018, 405–418. ISBN: 9781450349116. DOI: 10.1145/3173162.3177156. URL: https://doi.org/10.1145/3173162.3177156.

[3] *Clang Built Linux*. URL: https://clangbuiltlinux.github.io (visited on 03/24/2023).

[4] P. Heidekrüger. “Dependency Ordering in the Linux Kernel”. Bachelor’s Thesis. 2021. URL: https://pbhdk.com/files/ba-thesis.pdf.

[5] C. Lattner and V. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*. Palo Alto, California, Mar. 2004.

[6] *Linux Kernel Memory Barriers*. URL: https://www.kernel.org/doc/Documentation/memory-barriers.txt (visited on 03/24/2023).

[7] *Linux Kernel Memory Model*. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0124r4.html (visited on 03/24/2023).

[8] LLVM. *LLVM Project*. URL: https://llvm.org (visited on 03/24/2023).

[9] *Machine code description*. URL: https://llvm.org/docs/CodeGenerator.html#machine-code-representation (visited on 03/28/2023).

[10] L. Maranget, S. Sarkar, and P. Sewell. *A Tutorial Introduction to the ARM and POWER Relaxed Memory Models*. URL: https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf (visited on 03/28/2023).

[11] *PC-Keyed Metadata at Runtime*. URL: https://discourse.llvm.org/t/rfc-pc-keyed-metadata-at-runtime/64191 (visited on 03/28/2023).

[12] *Programming languages — C – The C11 standard*. URL: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf (visited on 03/28/2023).