UDK 004

Static Memory Consistency Constraints Checking

Andrianov P.S. (Ivannikov Institute for System Programming of RAS)

Mutilin V.S. (Ivannikov Institute for System Programming of RAS)

Memory model describes the memory consistency requirements in a multithreading system. Compiler optimizations may violate the consistency requirements due to bugs, and the program behavior will differ from the required one. The bugs in compiler optimizations, like incorrect instruction reordering, are very difficult to detect, because they may occur with a very low chance in real execution on a hardware.

There are different approaches of formal verification for memory consistency requirements, but the challenge is that the approaches are not scalable for industrial software. In the paper we present the MCC tool that was evaluated on the industrial virtual machine ARK VM and was able to find a real bug in a compiler optimization.

The MCC is a static tool, which allows to check all possible executions of a particular test, not relying on a hardware execution. The approach also includes test suite generation and specification of memory consistency properties.

Keywords: static analysis, memory model, compiler optimizations

1. Introduction

Nowadays processors have one or more layers of cache memory, which improves performance. However, their use also throws up new challenges. *Hardware memory model* defines necessary and sufficient conditions to guarantee that memory writes by other processors will be visible to the current processor, and that the current processor's writes will be visible to other processors. There exists a strong memory model, which guarantees that all processors always see exactly the same values for any given memory location. An opposite case is a weaker memory model.

There is different software, which can be executed over different hardware architectures. So, languages, like C++ or Java, specify its own memory models, which describe software behaviour independently from hardware. Java Memory Model (JMM) [1] provides strong guarantees, for example, if Java program does not have data races the programmer may assume sequentially consistent behavior.

ARK VM[1] is a general virtual machine, which is a part of a unified operating system Har-

[1] https://gitee.com/openharmony-sig/arkcompiler_runtime_core.git

Shared variable: *var.x*; local variables: *sum*, *r*

```
...                          ...
                             r = var.x
loop:                        loop:
  wait()                       wait()
  sum += var.x                 sum += r
...                          ...
```

Before optimization | After optimization

Fig. 1. Example of a program in pseudocode before and after optimization

monyOS [2]. Like the other virtual machines ARK VM supports running multithreaded programs. Such programs consist of threads concurrently executing VM instructions, e.g. writing to or reading from the memory. Some instructions have a special memory consistency semantics providing guarantees for a programmer when the other threads will see memory updates. For example, for Java there are strong guarantees that threads will see the results of writing to and reading from volatile variables in a sequentially consistent order. This memory consistency guarantees are specified in JMM. ARK VM has its own memory model (MM).

ARK VM should satisfy the requirements of MM, i.e. both ARK interpreter and ARK compiler. In our method we focused on the latter, i.e. verifying that the ARK compiler fulfills memory consistency requirements. More specifically we are interested in correctness of optimization transformations performed in Just-in-Time (JIT)/ Ahead-of-Time (AoT) of the ARK VM compiler. The method checks that ARK compiler optimizations generate code in target hardware instruction set architecture (HW ISA) which does not break memory consistency requirements of the input program.

The main contribution is an approach to detect memory consistency violations in the industrial software. The approach is more general and may be applied to other VMs or compilers.

2. Motivating example

Consider the following example (in pseudocode) in Fig. 1.

Here in the loop there are two actions: *wait* and access to a shared memory *var.x*. A compiler optimization transforms the code and extracts the read of the shared variable from the loop to the local variable *r*. The transformation modifies the program logic. Imagine, for example, a second thread, which updates *var.x* and then wakes the initial thread. Then the

[2] https://www.harmonyos.com/en/

code after optimization will be able to read shared *var.x* before wake, but it is not possible before optimization.

The task of proving that the left program is not equivalent to the right, is very complicated, especially due to the presence of a loop. Also, dynamic tools likely may not reproduce the problem, because it requires both a specific interleaving sequence between threads and specific cache updates in a hardware. Thus, we have a task to develop an approach to detect such kind of bugs.

3. Method Overview

The MCC approach breaks into three major parts:

- MCC-props - specifications of consistency properties;
- MCC-testgen - a test generator (OTK tool [2]);
- MCC-core - a core verification engine.

The main workflow is presented in Fig. 2. One has to specify constraints, or invariants, which should be hold on the all program executions. MCC-testgen provides a test suite for a specified property. When the test is compiled, the compiler optimizations are triggered. MCC-core verifies, that the specified property holds on each compiled test (in native code).

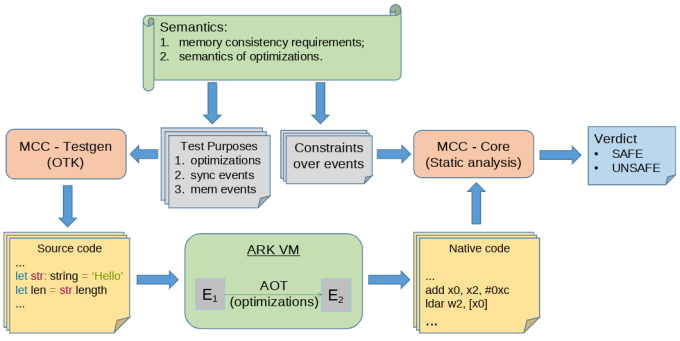

Fig. 2. MCC approach overview.

4. MCC properties

As it was already written, in the approach we have two representations of program: before compiler optimizations and after it. The first one is in the form of source code with the corresponding VM MM. The second one is in the form of native code with corresponding HW MM.

On each representation we can define a set of possible executions including sequence of

events defined by the program order and the relation between written and read values, i.e. which writes are seen by which reads. In general, we want to check that any execution in the native code should be possible in the source code.

MM defines a set of synchronization events which significantly restricts the relation between reads and writes. In many cases MM forbids to transfer reads and writes through synchronization events. In our work we rely on this fact and specify our properties in terms of the sequence of events along the execution.

In general, we formulate our properties in terms of sequence of events with modalities referring to time, similar to Linear Temporal Logic, but for the sake of effectiveness we introduced a simple language in the form of source code model comments presented later. The language is used to specify MM properties of ARK VM, namely *ordering* and *existence* specifications.

4.1. Ordering specification

There are different kinds of events in program: synchronization events (*sync*), non-synchronization events (*unsync*) and others. Synchronization events used for synchronization between threads, they provide some guarantees for user, for example, access to Java volatile variable or acquire of a lock. Thus, synchronization events cannot be reordered. Non-synchronization events are accesses to the shared data, they depends from synchronization events, but may not depend to each other. Thus, non-synchronization events can be reordered with each other, but cannot be reordered with synchronization events.

So, this is ordering specification:

- synchronization events cannot be reordered;
- non-synchronization events cannot be reordered with synchronization events.

As it was already written, operations with volatile variables are an example of synchronization events in JMM. In ARK VM there are intrinsic calls. The intrinsics have different flags, related to optimizations. And a specific set of flags means, that the intrinsic call cannot be optimized and can be expressed as a synchronization event. Reads and writes of a memory cell are non-synchronization events. For ARK VM they are expressed as load/store into memory. Thus, we have a set of non-synchronization events, each of them can not be reordered through any synchronization event.

An example of specification is presented in Fig. 3. It contains a call of *wait* - a *sync* event and a read of shared data - an *unsync* event.

```
...
loop:
  // MODEL: sync
  wait()
  // MODEL: unsync
  sum += var.x
...
```

Fig. 3. Example of a program with a specification inside

So, intrinsics with the flags are synchronization event, and can not be optimized. In the opposite case, it is possible to optimize and even remove the call.

4.2. Existence specification

One more generic property is that synchronization event cannot be totally removed by optimization. Actually, this is very strong condition and dead code optimization can remove the synchronization events. However, the check is rather important and in our experiments we found a bug with missing flag using this property.

5. MCC test generator

MCC-testgen is built upon a method for test generation using extended ISA model (OTK) [2]. We extend this method by supporting *model of situations* with data types related to MCC-props. We base on the *model of situations* which are being developed for generating tests covering logic of optimizations in ARK compiler. We extend them with situations related to MCC-props. For example, MCC-testgen *model of situations* introduces synchronization data types which are combined with general models. Thus, MCC-testgen generates tests for covering both general logic of optimization and a memory consistency property.

Currently, MCC-testgen provides *model of situations* for five optimizations: loop-invariant code motion (LICM), LICM for conditions (LICMCond), loop peeling (LP), redundant loop elimination (RLE), loop unrolling (LU). It means, that MCC-testgen generates tests specifically to trigger the corresponding optimizations. Then, it combines the test with memory actions.

Every test is a sequence of actions, which trigger the optimization. For example, it may have loops with different number of iterations, depth and so on. Between the actions, related to optimization, there are actions, related to memory model events.

Every memory action consists from three parts: pre-action, target action and post-action. Target action may be a write to a local register, a write to a shared object or an empty action. Pre-action and post-action can contain sync events, read/write to a shared object or an empty action. There may be several memory action triples in a single test.

The testgen randomly iterates different kinds of memory actions and generate a specified number of tests. For each test generated with the MCC-testgen the MCC-core verifies that ARK compiler optimizations satisfy memory consistency requirements stated in MCC-props.

6. MCC core

MCC-core is a verifier of MCC properties on a VM program. It may be an any program, but we used tests, provided by MCC-testgen. Actually, MCC needs not only an HW assembly program with a specification, which operations are sync and unsync events. The relation is extracted from a compiler dump with debug information.

The MCC-props specification is written inside a VM program using model comments. We just need to mark synchronization and non-synchronization events inside source code of a test. Then MCC-core checks that every specification constraint in terms of VM instructions is fulfilled in the HW program. For every model event in terms of VM instructions, MCC-core finds its HW assembler representation. And then it checks that the specified relation is the same. For example, assembler instructions for event A are before assembler instructions for event B.

MCC-core result is a verdict saying, whether it has detected a scenario of execution of HW variant which does not satisfy a specification property.

MCC-core takes a VM program (or directory) and configuration as input. Internally, it performs the following steps:

- *Model extraction* – extracts a model of events: synchronization and non-synchronization with relation to the origin source code.
- *Preparation* – obtains an assembly dump from the compiler.
- *Assembler parser* – obtains a relation of assembler instructions to the origin source code, that is performed using the compiler dump information.
- *Constraint checker* – checks, that every synchronization event is not reordered or removed in the assembler code.

Model extraction is performed just by model comments: if a line is marked as *sync*, it is considered as a synchronization event. And *unsync* means non-synchronization event. There

is also a possibility to specify constraints directly, for example, $A > B$, meaning the event A must be after B. In general case the model comments should be set manually. However, as we use a test generator, it performs all the work. So, the model comments are a part of OTK test templates.

Preparation stage takes a source code, produces a bytecode file and then runs compiler to get a compiler dump in Intermediate Representation (IR). A compiler dump file contains different information, but we use it to extract relation from IR assembler operations to origin source lines.

To extract the relation, we search pattern:

$$< IRoperation > ...(< filename >:< sourceline >)$$

However, not all IR operations has the information, thus it may be not complete. As source lines are marked with *sync* and *unsync* events, using the relation, we may obtain the same information about the assembler operations.

The constraint checker is implemented as a model checker based on Configurable Program Analysis (CPA) [3, 4]. We implemented *Sync CPA* which checks MCC ordering and existence properties. Its limitation still is false alarms due to dead code.

7. Discussion on method completeness

MCC approach performs static checks, and allows to consider all executions (see soundness of CPA approach). So, for a particular test and a particular property the MCC approach is able to be sound. For dynamic methods executing HW assembly on the processor the most tricky part is reproducing bugs, because they have a low probability and require rare cache coherence protocol states to occur.

The sources of incompleteness come from the techniques used in the components.

First, since we are basing on MCC-testgen and rely on the concrete set of generated tests. The tests reach some coverage of *model of situations* and we can measure progress related to the consistency model. However, we are dealing with tests, hence 100% coverage does not give 100% guarantee of covering all situations in the implementation.

Second, the set MCC properties elaborated from a memory model may not be complete. Here we rely on our expert knowledge.

8. Evaluation

To evaluate the MCC tool we used a machine with Ubuntu 22.04, Intel Core i5-6600K CPU @ 3.50GHz × 4 and 32 Gb RAM. We generated 100 tests for the LICM optimization and then checked MCC properties with MCC core. We run the tests on the origin ARK VM and on the ARK VM with bug introduced. The bug is related to incorrect intrinsic flag, which potentially may enable a compiler optimization in forbidden cases. The results are presented in Table 1.

Stage	Wall Time	CPU Time	Passed	Failed
Test generation	8 m 14 s	21 m 31 s	-	-
Normal run				
MCC-core	45 s	45 s	100	0
A bug introduced				
MCC-core	42 s	43 s	86	14

Table 1

Evaluation

Current version of the test suite for one LICM optimization achieves about 42% of line coverage in *compiler* directory. Anyway, we are mostly interested in coverage of target code, which works with memory events. Usually the target code is represented as conditions checking the properties of instructions, i.e. whether it is a sync event or not. It not a big number in line of code, but important for memory consistency. We manually inspected the collected coverage and found that we covered such conditions.

With the MCC tool there was found a real bug. It has been existed for a long time in optimization of intrinsic calls[3]. Some operations can be reordered with the call, which may lead to unexpected behaviours.

9. Related work

For checking JMM requirements there exists benchmark sets, like *JCStress*[4], which targets on concurrency bugs, including memory model violations. The benchmark set allows to reproduce MM bugs indeed, however, the chance is very low. For example, an introduced bug in volatile processing was fixed after several months[5], and test failure rate was about 0,0001%.

[3]https://gitee.com/openharmony-sig/arkcompiler_runtime_core/pulls/1389

[4]https://github.com/openjdk/jcstress

[5]https://gitee.com/openharmony-sig/arkcompiler_runtime_core/pulls/993

One more approach for memory consistency verification is formal prove with Alloy [5]. The approach requires to describe memory models of language and hardware, and also a mapping from one to another (a model of compiler). Then, it is possible to prove with Alloy model checker, if there is no forbidden execution found after "compilation". The main problem of the approach is performance, as with all formal methods. Authors used a rather powerful machine with four 16-core 2.1 GHz AMD Opteron processors and 128 GB of RAM. However, proving that compilation from C11 to x86 takes about 3 hours for an execution with 5 synchronization events. Need to say, that comparison of memory models (without "compilation" task) takes less time, especially, if a counterexample was found.

DoItYourself tool [6] allows to generate litmus tests for a specific hardware memory model. Power and x86 are supported. Memory model specification is written on Coq, and then the tool generates the test with a potential forbidden execution. If the forbidden execution is observed on a real test run, it means, that the hardware does not fulfill its specification. The approach suits for checking hardware memory model in case we have a complete memory model specification. However, it does not consider compilation level and, moreover, compiler optimizations. Thus, it solves another task.

10. Conclusion

We presented an approach for practical detection of memory consistency violations. It was applied to an industrial virtual machine and found a complicated bug in compiler optimization.

References

1. J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. (2013) Java memory model, §17.4. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se19/html/jls-17.html
2. S. Zelenov and S. Zelenova, "Model-based testing of optimizing compilers," in *Testing of Software and Communicating Systems*, A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 365–377.
3. D. Beyer, T. A. Henzinger, and G. Théoduloz, "Configurable software verification: concretizing the convergence of model checking and program analysis," in *Proceedings of CAV*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 504–518.
4. D. Beyer and M. Keremoglu, "CPAchecker: A tool for configurable software verification," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Springer Berlin

Heidelberg, 2011, vol. 6806, pp. 184–190.

5. J. Wickerson, M. Batty, T. Sorensen, and G. Constantinides, "Automatically comparing memory consistency models," 01 2017, pp. 190–204.
6. J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Fences in weak memory models," in *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 258–272.