Nidhugg – Manual
Version 0.2

Nidhugg is a bug-finding tool which targets bugs caused by concurrency and relaxed memory consistency in concurrent programs. It works on the level of LLVM internal representation, which means that it can be used for programs written in languages such as C or C++.

Currently Nidhugg supports the SC, TSO, PSO, POWER and ARM (partial) memory models. Target programs should use pthreads for concurrency, and each thread should be deterministic when run in isolation.

Contents

1 Stateless Model Checking with DPOR

Stateless model checking systematically tests a program by running a test case under many different thread schedules. This is done systematically, in such a way that at least one execution is explored for each equivalence class.

A test case must satisfy the following two conditions

Finiteness The test case must be finite in the following sense: There must be a bound $n \in \mathbb{N}$ such that all executions of the test case terminate within n execution steps. (The user does not need to know the actual bound value.) If there is no such bound, you can still use Nidhugg to analyze the program, provided that you allow the tool to impose an artificial bound (see switches `--unroll` and `--max-search-depth`).

Determinism The test case must be deterministic in the following sense: In a given state, executing a given execution step must always take the system to the same new state. This means that the test case may not e.g. check the time, or perform file I/O. If this condition is not satisfied, Nidhugg *may* be able to detect the nondeterminism and report it as an error with an attached error trace (currently only supported under SC/TSO/PSO). However, there is no guarantee for this, and a crash is a likely outcome.

2 Supported Memory Models

Nidhugg supports the memory models SC, TSO, PSO, POWER and ARM.

The SC semantics are the classical interleaving semantics described in [1]. The TSO and PSO memory models, used on the x86 and SPARC architectures, are described in [2]. For POWER and ARM, we use the semantics described in [1].

For SC, TSO and PSO, we use the analysis technique described in [4]. The technique is precise, and guarantees to explore all behaviors.

For POWER, we use the analysis technique described in [1]. The technique is precise, and guarantees to explore all behaviors.

For ARM, we use an adaptation of the technique in [1]. The technique is an *under-approximation* of ARM, and does *not* guarantee to explore all behaviors.

References

[1] Lamport, Leslie. "How to make a multiprocessor computer that correctly executes multiprocess programs." Computers, IEEE Transactions on 100.9 (1979): 690-691.

[2] The SPARC architecture manual. Englewood Cliffs, NJ 07632: PTR Prentice Hall, 1994.

[3] Alglave, Jade, Luc Maranget, and Michael Tautschnig. "Herding cats: Modelling, simulation, testing, and data mining for weak memory." ACM Transactions on Programming Languages and Systems (TOPLAS) 36.2 (2014): 7.

[4] Abdulla, Parosh Aziz, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, Konstantinos Sagonas. "Stateless model checking for TSO and PSO." Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2015. 353–367.

[5] Abdulla, Parosh Aziz, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson. "Stateless model checking for POWER." Pending peer review.

3 Running Nidhugg

The typical workflow of analyzing a test case with Nidhugg is as follows:

1. Compile the source code into LLVM assembly using a compiler such as e.g. clang, clang++, llvm-gcc, etc.

2. Optionally use `nidhugg` to transform the assembly into new assembly that can be analyzed more efficiently.

3. Use `nidhugg` to analyze the LLVM assembly file.

These steps are detailed in Section 3.1. For convenience, Nidhugg provides a script `nidhuggc` which automates all the steps, for the case when the test case is contained in a single C file. Section 3.2 describes how to use `nidhuggc`.

3.1 Using Nidhugg Directly

Compilation Since Nidhugg works on LLVM internal representation, source code in high-level languages such as e.g. C must be compiled before it can be analyzed. This can be done with some compilers based on LLVM. C programs are conveniently compiled with the clang compiler as follows:

```
$ clang CFLAGS -emit-llvm -S -o FILE.ll FILE.c
```

Here *CFLAGS* can be any switches you would normally give to your compiler when compiling *FILE*`.c`. Notice that if optimization switches such as e.g. `-O2` are given to clang, then the corresponding optimizations are performed on the code *before* analysis. This can be beneficial for analysis time consumption, but may change the behavior of the program, e.g. in cases where shared variables are not properly marked as `volatile`.

Make certain that the compiler supports the version of LLVM against which Nidhugg is built. For clang this is easiest done by making certain that the LLVM version is the same as the clang version.

Multi-File Compilation To compile a test case which consists of multiple C files, the procedure is similar to how you would normally compile the code: Use e.g. clang to compile each C file into an object file, but add switches `-emit-llvm -S` to generate the object file as LLVM assembly. Then link the object files using llvm-link.

```
$ clang -c CFLAGS -emit-llvm -S -o FILE_0.ll FILE_0.c
⋮
$ clang -c CFLAGS -emit-llvm -S -o FILE_n.ll FILE_n.c
$ llvm-link -o FILE.ll FILE_0.ll ··· FILE_n.ll
```

Transformation Before analyzing a test case with Nidhugg, the test case code can be automatically rewritten in various ways, e.g. to improve analysis performance or to put a bound on an infinite test case to make it finite. This is done with a call to nidhugg as follows:

```
$ nidhugg TFLAGS --transform=OUTFILE.ll INFILE.ll
```

Here we transform the code given as LLVM assembly in *INFILE*`.ll`, and write the resulting code to the file *OUTFILE*`.ll`. The switches given in *TFLAGS* specify which particular transformations should be performed. See Section 3.1.2 for details about the available transformations.

Analysis A finite and deterministic test case in the form of LLVM assembly may be analyzed by Nidhugg using a command like the following:

```
$ nidhugg FLAGS {--sc,--tso,--pso,--power,--arm} FILE.ll \
    [ -- PROGRAM ARGUMENTS ]
```

Additional switches are detailed in Section 3.1.1. Nidhugg will systematically run executions of the test case, covering all execution equivalence classes, under the given memory model. Then it terminates, either giving an error trace or stating that no errors were found.

```
// test.c
#include <pthread.h>
#include <assert.h>

volatile int x = 0, y = 0, c = 0;

void *thr1(void *arg){
  y = 1;
  // __asm__ volatile ("mfence" ::: "memory");
  if(!x){
    c = 1;
    assert(c == 1);
  }
  return NULL;
}

int main(int argc, char *argv[]){
  pthread_t t;
  pthread_create(&t,NULL,thr1,NULL);
  x = 1;
  // __asm__ volatile ("mfence" ::: "memory");
  if(!y){
    c = 0;
    assert(c == 0);
  }
  pthread_join(t,NULL);
  return 0;
}
```

Figure 1: Small Dekker test in C.

Typical Usage For a small example of typical usage of Nidhugg, consider the C code listed in Figure 1. We can analyze it under SC with the following sequence of commands:

```
$ clang -o test.ll -S -emit-llvm test.c
$ nidhugg -sc test.ll
```

Nidhugg will here tell us that no error was detected (the assert statements are never violated). If we rerun the same test case under TSO, Nidhugg will instead give us an error trace describing how the assert statements may be violated under TSO. See Section 3.3 for details about how to interpret the error trace.

In this case we performed no transformation. In the common case that a test case contains (unbounded) loops, one may instead want to use commands such as the following:

```
$ clang -o test.ll -S -emit-llvm test.c
$ nidhugg --unroll=10 --transform=test.trans.ll test.ll
$ nidhugg -sc test.trans.ll
```

3.1.1 Analysis Switches

`--sc/--tso/--pso/--power/--arm` Analyze the test case under the SC, TSO, PSO, POWER or ARM memory model respectively.

`--source/--optimal/--observers/--rf` Use the Source DPOR, Optimal-DPOR, Optimal-DPOR with Observers, or Optimal Reads-From-centric SMC algorithm for analysis, respectively. Optimal DPOR guarantees no redundant (sleep set blocked) executions, at the cost of some runtime overhead and a larger worst-case memory complexity. Observers additionally uses the observers optimisation; not considering two memory writes to the same address to be racing unless the result is read. Reads-from-centric SMC semantically strengthens Observers, in the sense that it always needs at most as many traces as Observers. However, RF-SMC is a significantly different algorithm. Rather than focusing on the ordering of racing, or otherwise dependent, operations, RF-SMC only concerns itself with which read operations read which write operations. Due to optimality of such an approach requiring solving NP-hard decision problems, in general, RF-SMC relies on a heuristic for performance. If the heuristic fails, it falls back to a slower decision procedure and performance might suffer, but we have not been able to observe this happening in practice. **Optimal-DPOR, Optimal-DPOR with Observers, and Optimal Reads-From-centric SMC algorithms are supported only for the SC memory model.**

`--check-robustness` Perform a robustness analysis. In addition to considering the usual safety criteria, also check robustness, and report an error if a non-robust execution is found. **Supported only for SC, TSO, PSO.**

`--extfun-no-race=FUN` Assume that the external function `FUN`, when called as blackbox, does not participate in any races. May be given several times. See Section 4.3.1. **Supported only for SC, TSO, PSO.**

`--malloc-may-fail` By default, when the test case calls `malloc`, Nidhugg will assume that the call succeeds, allocate memory and provide the resulting memory pointer as return value for the call. But, if `--malloc-may-fail` is given to Nidhugg, then the case when `malloc` (or `calloc`) fails and returns NULL is also analyzed. **Supported only for SC, TSO, PSO.**

`--no-check-mutex-init` If this switch is given, the operations lock, trylock, unlock, destroy, cond_wait are allowed to operate on mutex variables without the mutex being previously initialized by an explicit call to `pthread_mutex_init`. This is useful because it allows analysis of programs using static mutex initialization instead of explicit calls to `pthread_mutex_init`.

`--print-progress` Continually print the number of hitherto analyzed executions to the terminal while running the analysis.

`--print-progress-estimate` Same as `--print-progress`, but also estimate the total number of executions of the test case, and correspondingly print a progress percentage together with the execution count.

`--version` Print the Nidhugg version and exit.

PROGRAM ARGUMENTS, if given, will be sent as arguments (argv) to the test case.

3.1.2 Transformation Switches

`--transform=`*OUTFILE* Don't analyze the input test case. Instead run program transformations on it, and output the resulting code as LLVM assembly to the file *OUTFILE*. This switch should always be used for program transformation, and should be combined with switches that specify specific transformations.

`--no-spin-assume` Disable the `spin-assume` transformation. By default, the transformation will detect spin loops in the code, and replace them by `assume` statements. This transformation maintains satisfaction or dissatisfaction of the safety properties in the test case. However, this transformation does not in general maintain robustness violations.

`--unroll=`*N* Unroll all loops *N* times, making the program loop free. A thread that attempts to execute a basic block from a loop more than *N* times in one go will be blocked. This is useful to make a finite test case out of an infinite one.

Notice that this unrolling works on loops in the LLVM internal representation after compilation and possibly optimization. These loops do not necessarily correspond one-to-one with the loops in the high-level language.

3.1.3 Exit Status

Exit status for Nidhugg.

- 0: Successful termination. If Nidhugg was run in analysis mode, the analysis completed without finding any errors.
- 42: Error detected. Analysis completed after finding an error in the analysed program.
- Other: Nidhugg failed to complete its task for some reason.

3.2 Using Nidhuggc for Single-File C Programs

In order to simplify the case where a test case consists of only a single C file, Nidhugg provides a script `nidhuggc` which automates the process described in Section 3.1. The script `nidhuggc` can be used as follows:

```
$ nidhuggc CFLAGS -- {--sc,--tso,--pso,--power,--arm} \
    AFLAGS TFLAGS FILE.c [ -- PROGRAM ARGUMENTS ]
```

Here *CFLAGS* are switches that will be given to the C compiler, *TFLAGS* are Nidhugg transformation switches, and *AFLAGS* are Nidhugg analysis switches. Furthermore, *PROGRAM ARGUMENTS*, if given, will be sent as arguments to the program under test. One common way to use `nidhuggc` is this call:

```
$ nidhuggc -O2 -- --tso --unroll=10 FILE.c
```

3.2.1 Switches

`--version` Print the Nidhugg version and exit.

`--c` Interpret the input file as C code regardless of its file name extension.

`--cxx` Interpret the input file as C++ code regardless of its file name extension.

`--clang=`*PATH* Specify the path to clang.

`--clangxx=`*PATH* Specify the path to clang++.

`--nidhugg=`*PATH* Specify the path to nidhugg.

3.3 Understanding Error Traces

In order to understand Nidhugg error traces, it is necessary to understand the way Nidhugg identifies threads. Each thread is associated with a thread ID (tid). The tid of the initial thread is `<0>`, the threads that are created by `<0>` are named, in order, `<0.0>`, `<0.1>`, `<0.2>` etc. The threads created by `<0.1>` are named `<0.1.0>`, `<0.1.1>`, etc.

Since the operational semantics of the different memory models differ significantly, the error traces are also different under different models.

3.3.1 Error Traces under TSO and PSO

In addition to actual program threads, Nidhugg also reasons about *auxiliary threads*. These are hypothetical threads that carry out non-deterministic events such as updates from a store buffer to memory. The auxiliary threads belonging to a certain real thread <i_0.····.i_n> are named <i_0.····.i_n/0>, <i_0.····.i_n/1>, etc. For example, in the case TSO, the auxiliary thread `<0/0>` takes care of the memory updates from the store buffer of the initial thread `<0>`.

Now we can consider the example error trace listed in Figure 2. It was produced by running the following command on the test case given in Figure 1:

```
$ nidhuggc -O2 -- --tso test.c
```

Each line corresponds to an event, which is either the execution of an instruction or a memory update. Each event is identified by a pair (*tid*,*i*), where *tid* is the thread ID of the executing thread, and *i* is the per-thread index of that event. Notice that some events (e.g. `(<0>,2)`) are missing in the trace. The missing events are events that are local, and do not affect any other thread. Each event is annotated with the line of C code that produced that instruction. Notice that one line of C code may correspond to several instructions.

We point out some main features of the error trace in Figure 2. First we see that the load of `y` in event `(<0>,4)` precedes the update to `y` by the other thread in event `(<0.0/0>,1)`. Hence thread `<0>` sees the initial value 0 for `y`. Similarly the load `(<0.0>,2)` precedes the update `(<0/0>,1)`, and so the other thread sees the value 0 for `x`. At the end of the trace, we see that thread `<0>` updates `c` with the value 0, just before thread `0.0` executes `assert(c == 1)`, and so the assertion fails.

```
Error detected:
 (<0>,1) test.c:16: int main(int argc, char *argv[]){
     (<0.0>,1) test.c:6: void *thr1(void *arg){
 (<0>,4) test.c:21: if(!y){
       (<0.0/0>,1) UPDATE test.c:7: y = 1;
 (<0>,5) test.c:21: if(!y){
 (<0>,8) test.c:23: assert(c == 0);
 (<0>,11) test.c:25: pthread_join(t,NULL);
     (<0.0>,2) test.c:9: if(!x){
   (<0/0>,1) UPDATE test.c:19: x = 1;
     (<0.0>,3) test.c:9: if(!x){
       (<0.0/0>,2) UPDATE test.c:10: c = 1;
   (<0/0>,2) UPDATE test.c:22: c = 0;
     (<0.0>,6) test.c:11: assert(c == 1);
              Error: Assertion violation at (<0.0>,9): (c == 1)
```

Figure 2: A TSO error trace for the program shown in Figure 1.

```
Error detected:
(<0>): Entering function main
(<0>,1): test.c:16 pthread_create(&t,NULL,thr1,NULL);
         store 0x0100000000000000 to [0x1ac39b0]
         store 0x01 to [0x1acd740]
  (<0.0>): Entering function thr1
(<0>,2): test.c:17 x = 1;
         store 0x01000000 to [0x1ad3110]
(<0>,3): test.c:19 if(!y){
         load 0x00000000 from [0x1ac0020] source: (initial value)
(<0>,4): test.c:20 c = 0;
         store 0x00000000 to [0x1acd680]
(<0>,5): test.c:21 assert(c == 0);
         load 0x00000000 from [0x1acd680] source: (<0>,4)
(<0>,6): test.c:23 pthread_join(t,NULL);
         load 0x0100000000000000 from [0x1ac39b0] source: (<0>,1)
  (<0.0>,1):
             load 0x01 from [0x1acd740] source: (<0>,1)
  (<0.0>,2): test.c:6 y = 1;
             store 0x01000000 to [0x1ac0020]
  (<0.0>,3): test.c:8 if(!x){
             load 0x00000000 from [0x1ad3110] source:
             (initial value)
  (<0.0>,4): test.c:9 c = 1;
             store 0x01000000 to [0x1acd680]
  (<0.0>,5): test.c:10 assert(c == 1);
             load 0x00000000 from [0x1acd680] source: (<0>,4)
  Error: Assertion failure at test.c:10: c == 1
```

Figure 3: A POWER error trace for the program shown in Figure 1.

3.3.2 Error Traces under POWER and ARM

In the operational semantics of POWER and ARM, there are no auxiliary threads or auxiliary events (as is the case under TSO and PSO). Instead, each memory access is equipped with a parameter at the time when it is committed. The parameter choice determines the behavior of the event and its relation (memory ordering) to other events. For stores, the parameter determines how the store is coherence-ordered with other stores to the same memory location. For loads, the parameter determines the source of the read value, i.e., the store which provided the observed value.

Now we can consider the example error trace given in Figure 3. It was produced by running the following command on the test case given in Figure 1.

```
$ nidhuggc -O2 -- --power test.c
```

Events represent the execution of a memory access. They are identified as a pair `(`*tid*`,`*i*`)` where *tid* is the ID of the executing thread, and i is the index in program order of the executed memory access instruction. Local instructions are not visible in the error trace. For each event, the error trace contains the corresponding line of source code, and an explanation of the behavior of the event when it was committed.

For a store event, the explanation has the form `store` v `to [`a`]` where v is the written value, and a is the memory address where it was stored. Notice that the value and address are dependent on the architecture of the machine running Nidhugg. For example, for event `(<0>,2)` in Figure 3, the value 1 appears as `0x01000000` because the test was run on a little endian machine. The parameter (coherence order position) of the store event is not printed.

For a load event, the explanation has the form `load` v `from [`a`] source:` s where v is the observed value that was read from memory address a. The parameter of the load event is given as the source s. The source is the store event which provided the observed value. In the general case, a load may have multiple sources, in case the load reads multiple bytes written by different stores.

A special case which is visible in Figure 3 is the event `(<0>,1)`. This event, corresponding to the call to `pthread_create`, performs two stores: The thread ID of the new thread is written to the output variable `t` here located at `[0x1ac39b0]`. Furthermore, a byte-sized store is performed to `[0x1acd740]`. The latter store is used to simulate communication between threads during thread creation and thread joining.

The error trace also contains lines specifying when the execution of each thread enters and exits functions. Notice that the entering and exiting of functions is related to when instructions contained in those functions are *fetched*. This may occur far earlier than the fetched instructions are committed. Therefore it is not uncommon or unexpected that the exit from a function appears earlier in the error trace than the committing of its contained instructions.

For more details on the representation of (error) traces under POWER and ARM, see [1].

References

[1] Abdulla, Parosh Aziz, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson. "Stateless model checking for POWER." Pending peer re-

view.

4 Compatibility: Pthreads, stdlib, etc.

4.1 Pthreads

The following pthread functions are currently supported by Nidhugg:

`pthread_create`

`pthread_join`

`pthread_exit`

`pthread_mutex_init`

`pthread_mutex_lock`

`pthread_mutex_unlock`

`pthread_mutex_destroy`

The following are only supported under SC, TSO and PSO.

`pthread_self`

`pthread_mutex_trylock`

`pthread_cond_init`

`pthread_cond_signal`

`pthread_cond_broadcast`

`pthread_cond_wait`

`pthread_cond_destroy`

4.2 The Standard C library

Nidhugg has limited support for functions in the standard C library. The functions listed below have custom support. Other functions in the standard library can, under SC, TSO and PSO, be called as blackboxes. See Section 4.3.

Functions with custom support:

`malloc`

`calloc`

`free`

`atexit` (**Supported only for SC, TSO, PSO.**)

`__assert_fail` (Called by the `assert` macro in assert.h.)

4.3 External Functions (SC, TSO, PSO only)

While running an execution, if Nidhugg encounters an unknown external function (e.g. some function from the standard C library), it will handle it by printing a warning and making *an actual call* to the function. This will usually work fine for well-behaved functions such as e.g. the ones in `string.h`. But may cause undefined behavior in other cases, such as e.g. when calling functions that deal with signal handling or file I/O.

Warning
Notice that the external functions are actually run on your system. If the analyzed code does undesirable things to the system, those things will really be executed.

4.3.1 External Functions and Races

Since external functions are called as blackboxes, there is no way for Nidhugg to know which memory locations will be accessed by the functions, which mutexes will be locked or unlocked, etc. In order to provide a complete under-approximation, Nidhugg therefore assumes that external functions conflict with *all* other accesses to memory, mutexes, etc. This may cause Nidhugg to explore a huge number of superfluous computations, which differ only in the order between unrelated or irrelevant external function calls. To avoid this problem, the user may use the switch `--extfun-no-race` to specify external functions which should be assumed to not participate in races.

For example, consider the program given in Figure 4. It has two threads, each calling `printf` once. There are no (real) data races. Figure 5 shows analysis of the program using `nidhuggc`. Here the calls to `printf` are assumed to race both with each other, and with the loads of `t0` and `t1` in the main thread. Nidhugg explores computations corresponding to all the different ways to resolve the races. In total 16 computations are explored. One single computation would suffice. In Figure 6 we show how to use the switch `--extfun-no-race` to fix the problem, and correctly explore only one computation.

4.4 Verifier Functions

Nidhugg also provides a number of functions that can be convenient for modelling and verification. Declare the functions in your C code before use, making sure to use the same signatures as described here.

`void __VERIFIER_assume(int b)` This function checks the value of `b`. If `b` = 0, the execution blocks indefinitely, otherwise the call does nothing.

`int __VERIFIER_nondet_int()` Returns a non-deterministic integer value. The function gives support for a very limited non-determinism. Each call to `__VERIFIER_nondet_int` will return a value which is unpredictable before the analysis. But a given call to `__VERIFIER_nondet_int` by a given thread will always return the same value in all executions that are explored. **Supported only for SC, TSO, PSO.**

The value returned by `__VERIFIER_nondet_int` can be forced by specifying `--verifier-nondet-int=`*value*.

```
// testprintf.c

#include <pthread.h>
#include <stdio.h>

void *p(void *arg){
  printf("foo\n");
  return NULL;
}

int main(int argc, char *argv[]){
  pthread_t t0, t1;
  pthread_create(&t0,NULL,p,NULL);
  pthread_create(&t1,NULL,p,NULL);
  pthread_join(t0,NULL);
  pthread_join(t1,NULL);
  return 0;
}
```

Figure 4: A small race free program.

`unsigned int __VERIFIER_nondet_uint()` Same as `__VERIFIER_nondet_int`. **Supported only for SC, TSO, PSO.**

`__VERIFIER_atomic_*` Any function, defined by the user, whose name starts with `__VERIFIER_atomic_` will execute atomically. This also means that it will execute under sequential consistency, and act as a full memory fence.

Atomic functions may not contain control flow when using Optimal-DPOR (`--optimal`), and additionally not writes to multiple memory locations when using Observers (`--observers`), or the analysis may not be sound. Prefer using built-in atomic functions provided by the compiler. **Supported only for SC, TSO, PSO.**

4.5 Fences and Intrinsics

The following are currently supported memory fences:

`__asm__ volatile ("" ::: "memory")` Compiler fence. Supported. Notice that a compiler fence may affect how the compiler produces the LLVM assembly. Nidhugg, on the other hand, will ignore the compiler fence since it has no effect on the hardware level.

`__asm__ volatile ("mfence" ::: "memory")` Full memory fence. Supported under SC (with no effect), TSO and PSO.

`__asm__ volatile ("sync" ::: "memory")` POWER sync fence. See [1].

`__asm__ volatile ("lwsync" ::: "memory")` POWER lwsync fence. See [1].

`__asm__ volatile ("eieio" ::: "memory")` POWER eieio fence. See [1].

```
$ nidhuggc --sc testprintf.c
* Nidhuggc: $ clang -o /tmp/tmp4ndgl1_l/tmpjs60nht_.ll -S -emit-llvm -g testprintf.c
* Nidhuggc: $ /usr/bin/nidhugg --sc /tmp/tmp4ndgl1_l/tmpjs60nht_.ll
WARNING: Calling unknown external function printf as blackbox.
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
Trace count: 16 (also 0 sleepset blocked)
No errors were detected.
Total wall-clock time: 0.08 s
```

Figure 5: Analysis of the program in Figure 4. Here `printf` is assumed to race with any memory access. This causes a wasteful exploration of 16 different computations.

```
$ nidhuggc --sc --extfun-no-race=printf testprintf.c
* Nidhuggc: $ clang -o /tmp/tmpsqqgz4fz/tmpabpqpy9z.ll -S -emit-llvm -g testprintf.c
* Nidhuggc: $ /usr/bin/nidhugg --sc --extfun-no-race=printf /tmp/tmpsqqgz4fz/tmpabpqpy9z.ll
WARNING: Calling unknown external function printf as blackbox.
foo
foo
Trace count: 1 (also 0 sleepset blocked)
No errors were detected.
Total wall-clock time: 0.07 s
```

Figure 6: Analysis of the program in Figure 4 using the switch `--extfun-no-race`. As intended, only one computation is explored.

`__asm__ volatile ("isync" ::: "memory")` POWER isync fence. See [1].

`__asm__ volatile ("dmb" ::: "memory")` ARM dmb fence. See [1].

`__asm__ volatile ("dsb" ::: "memory")` ARM dsb fence. See [1].

`__asm__ volatile ("isb" ::: "memory")` ARM isb fence. See [1].

`__sync_synchronize()` Full memory fence. **Supported only for SC, TSO, PSO.**

`__atomic_thread_fence(__ATOMIC_SEQ_CST)` Full memory fence. **Supported only for SC, TSO, PSO.**

Built-in atomic functions provided by the compiler are typically supported under SC, TSO and PSO, but only for the model `__ATOMIC_SEQ_CST`. This allows use of e.g. compare and exchange (`__atomic_compare_exchange_n`) or atomic increase (`__atomic_add_fetch`).

References

[1] Alglave, Jade, Luc Maranget, and Michael Tautschnig. "Herding cats: Modelling, simulation, testing, and data mining for weak memory." ACM Transactions on Programming Languages and Systems (TOPLAS) 36.2 (2014): 7.