Check Quick Start

1 Getting Started

1.1 Downloads

The tools in the tutorial are distributed to work within VirtualBox. If you don't already have Virtual Box installed, please download VirtualBox from this link, and double-click the file to begin installation:

`https://www.virtualbox.org`

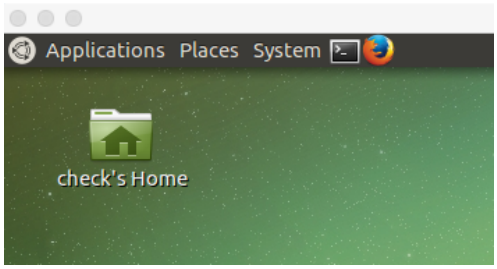

Figure 1: Closeup of icon for starting a terminal window.

Then, download our tutorial VM package from the link below, and double-click on the downloaded .ova file, and then follow the dialog box (e.g. click "import") to import it into Virtual Box:

`http://check.cs.princeton.edu/tutorial_vm/Check_Tools_VM.ova`

Once imported, a VirtualBox manager window should open up. Double-click the VM's entry in the sidebar of VirtualBox to start it. Once the instance is started, click on the small obscure terminal icon in the upper-left corner (just to the right of 'System', as shown in Figure 3) to open a terminal window. The remainder of the tutorial will involve navigating files and typing commands into the terminal window.

1.2 What's in here?

In this distribution we have included:

- Fillable PipeCheck example SC microarchitecture (`~/pipecheck_tutorial/uarches/SC_fillable.uarch`)
- Completed PipeCheck example SC microarchitecture (`~/pipecheck_tutorial/uarches/solutions/SC.uarch`)
- Fillable PipeCheck example for TSO microarchitecture. (`~/pipecheck_tutorial/uarches/TSO_fillable.uarch`)
- Completed PipeCheck example for TSO microarchitecture (`~/pipecheck_tutorial/uarches/solutions/TSO.uarch`)
- TriCheck example for TSO microarchitecture
- Default compiler mappings file from C11 to that TSO microarchitecture

Predicate	Returns True if...
DataFromInitialStateAtPA r	load r reads from the initial state of the litmus test
DataFromFinalStateAtPA w	store w writes a value equal to the final state of the litmus test (if a final condition is specified)
HasDependency rel i1 i2	the litmus test specifies a relation rel from microop i1 to microop i2
IsAnyRead i	microop i is a read
IsAnyWrite i	microop i is a write
IsAnyFence f	microop f is a fence
AccessType type i	microop i is of access type "type"
FenceType type f	microop f has fence type "type"
OnCore n i	microop i is on core n, where n is an integer identifier
ProgramOrder i1 i2	microop i1 is before microop i2 in the program order
SameCore i1 i2	microops i1 and i2 are on the same core
SameData i1 i2	microops i1 and i2 read and/or write the same data value
SameMicroop i1 i	microops i1 and i2 are the same microop
SamePhysicalAddress i1 i2	microops i1 and i2 access the same physical address
SameVirtualAddress i1 i2	microops i1 and i2 access the same virtual address
StageName n "name"	define stage with integer identifier n and a reference name

Table 1: Summary of μSpec predicate constructs (apart from node/edge-related predicates) for building axioms and macros.

- Template-based litmus test generator for TriCheck-style C-to-ISA generation
- More than 50 litmus tests in the .test format (`~/pipecheck_tutorial/tests/SC_tests` and `~/pipecheck_tutorial/tests/TSO_tests`)

1.3 High-Level Directory Structure

From a terminal window, the PipeCheck portion of the tutorial can be reached by typing:

```
cd /home/check/pipecheck_tutorial/
```

Likewise, the TriCheck portion of the tutorial can be reached by typing:

```
cd ~/TriCheck
```

2 Glossary of μSpec predicates and primitives

2.1 Predicates

Table 1 summarizes the key predicate terminology used for most μSpec axioms and macros.

2.2 Primitives

Table 2 similarly summarizes the other μSpec primitives used to construct axioms and macros in a microarchitecture model.

3 How to write a uarch with μSpec

There are three major components of a μSpec microarchitecture definition:

- Stage identifier definitions
- Macros (optional)
- Axioms

Primitive	Description
pA => pB	predicate pA logically implies predicate pB
pA /\pB	logical conjunction of predicates pA and pB
pA \/ pB	logical disjunction of predicates pA and pB
~p	logical negation of predicate p
% <comment>	uspec comment
c	predefined to be the core ID
exists microop “var”, <exists body>	existential quantification
forall microop “var”, <forall body>	universal quantification
NodeExists ((i1, stage1)))	the μhb node representing microop i1 in stage1 is present in the μhb graph
NodesExist [(i1, stage1); (i2, stage2); ...]	the μhb nodes representing microop i1 in stage 1, microop i2 in stage 2, etc. are present in the μhb graph.
EdgeExists ((i1, stage1), (i2, stage2))	a happens-before edge exists between microop i1 in stage1 and microop i2 in stage2
EdgesExist [((i1, stage1), (i2, stage2)); ((i3, stage3), (i4, stage4)); ...]	happens-before edges exist between microop i1 in stage1 and microop i2 in stage2, as well as between microop i3 in stage3 and microop i4 in stage4, etc.
AddEdge ((i1, stage1), (i2, stage2), ”label12”)	adds an edge starting at the node representing microop i1 in stage1 and ending at the node representing microop i2 in stage2. Label is required, but may be empty (e.g., ””).
AddEdges [((i1, stage1), (i2, stage2), ”label12”); ((i3, stage3), (i4, stage4), ”label34”); ...]	Similar to AddEdge, but used to add multiple edges with one primitive.
Axiom “name”: <axiom body>.	defines an axiom with the specified reference name, where the axiom body is comprised of predicates, quantifiers, connectives, and/or macros.
CoreOf i	returns the Core ID associated with microop i
DefineMacro “name”: <macro body>.	defines a macro with the specified reference name that can be instantiated as part of an axiom. The macro body is comprised of predicates, quantifiers, connectives, and/or macros.
ExpandMacro name	expands macro with specified reference name

Table 2: Summary of other μSpec primitives for building axioms and macros.

3.1 Stage identifier definitions

Stage identifier definitions must come first in the microarchitecture, and define the various events that the microarchitectural axioms operate on. Each stage identifier must be associated with a unique number used by the solver backend. A stage identifier has the form:

```
StageName <unique number> "<name of stage>".
```

So for instance, to define Fetch, Execute, and Writeback stages for a microarchitecture, one could do:

```
StageName 0 "Fetch".
StageName 1 "Execute".
StageName 2 "Writeback".
```

3.2 Macro Definitions

Macros can be used to define μSpec fragments that can be included as part of an axiom. They are useful for frequently used μSpec fragments as well as to break up large axioms into smaller components in the file. The syntax of a macro definition is as follows:

```
DefineMacro "<macro name>":
<uSpec fragment>.
```

A macro expansion substitutes the text of the macro when called:

```
ExpandMacro <macro name>
```

For example, `SC_fillable.uarch` breaks up its `Read_Values` axiom into three macros (`BeforeAllWrites`, `No_SameAddrWrites_Btwn_Src_And_Read`, and `Before_Or_After_Every_SameAddrWrite`). The definition of the `BeforeAllWrites` macro is:

```
DefineMacro "BeforeAllWrites":
  DataFromInitialStateAtPA i /\
  forall microop "w", (
    (IsAnyWrite w /\ SamePhysicalAddress w i /\ ~SameMicroop i w) =>
    AddEdge ((i, Execute), (w, (0, MemoryHierarchy)), "fr", "red")).
```

and the three macros are invoked in the `Read_Values` axiom as shown below:

```
Axiom "Read_Values":
forall microops "i",
IsAnyRead i =>
(
      ExpandMacro BeforeAllWrites
      \/
      (
        ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read
        /\
        ExpandMacro Before_Or_After_Every_SameAddrWrite
      )
).
```

3.3 Axiom Definitions

Axiom syntax is very similar to macro syntax, but an axiom definition must specify a complete axiom rather than just a μSpec fragment. So for instance, all variables used by the axiom must be defined in it (unlike in the case of macros above). The μSpec axiom syntax is:

```
Axiom "<axiom name>":
<uSpec statement>.
```

4 How to write a litmus test

The Check framework uses a custom litmus test format that by convention uses the `.test` extension. This format allows users to specify the common components of litmus tests (like instructions, data values, and final conditions), but also allows users to specify information related to address translation and virtual memory, such as both the virtual and physical addresses for an instruction. The format of a litmus test is:

"Alternative"
<instructions>
<relationships>
<final conditions> (optional)
["Alternative"
<instructions>
<relationships>
<final conditions> (optional)]*
<outcome>

Each `Alternative` above specifies an architecture-level execution of the litmus test that must be examined. A given `Alternative` specifies values for all loads and stores in the program, as well as any final

conditions that apply. At the end of the test, the `outcome` specifies whether the test should be allowed or forbidden by the architecture specification. Note that this means that a separate `.test` file may be required for each architecture-level memory model being tested, as tests may be allowed by one model but forbidden by others. The syntax of the individual components of a litmus test are explained in further detail below.

4.1 <instructions>

<globalID> <coreID> <threadID> <intraInstID> <opcode>
<globalID>: a unique ID for every Instruction
<coreID>: the core on which the thread is running
<threadID>: unique thread ID per core (always 0 except in COATCheck)
<intraInstID>: the intra-instruction ID (always 0 except in COATCheck)
Example: 0 0 0 0 (Read Acquire RMW (VA 0 0) (PA 0 0) (Data 0))

4.1.1 <opcode>

"(*Read*|*Write*|*Fence* <**flags**> <**va**> <**pa**> <data>)"

Read|*Write*|*Fence*: obvious
<flags>: zero or more string identifiers (e.g., "RMW", "Acquire", etc.)
<va>: "(VA <vtag> <vindex>)"
    <vtag>: virtual address tag (integer)
    <vindex>: virtual address index (always 0 except in COATCheck)
<pa>: "(PA <ptag> <pindex>)"
    <ptag>: physical address tag (integer) (should match vtag except in COATCheck)
    <pindex>: physical address index (always 0 except in COATCheck)

4.2 <relationship>

"Relationship <name> <globalID0> 0 -> <globalID1> 0"

Example: "Relationship po 0 0 -> 1 0"

4.3 <outcome>:

Permitted|*Forbidden*|*Required*|*Unobserved*

4.4 Example litmus test (sb.test for TSO)

Alternative
0 0 0 0 (Write (VA 0 0) (PA 0 0) (Data 1))
1 0 0 0 (Read (VA 1 0) (PA 1 0) (Data 0))
2 1 0 0 (Write (VA 1 0) (PA 1 0) (Data 1))
3 1 0 0 (Read (VA 0 0) (PA 0 0) (Data 0))
Relationship po 0 0 -> 1 0
Relationship po 2 0 -> 3 0
Permitted

Note: the program order (po) relationship is required. This informs Check of the order of instructions on each core. The po relation must be transitively expanded in the litmus test *.test file. The Check predicate, ProgramOrder, relies on this relationship being specified. Other relationships that can be specified (for use by the Check primitive HasDependency) are: addr, data, ctrl, and ctrlisync.

5 How to run

5.1 Running a single test with `check`

To run a test through a microarchitecture that is in the `uarches` directory of the `pipecheck_tutorial` repository:

```
check -i <input test file> -m <name of uarch file>
```

This will generate `<input test file>.pdf` in your current directory.

So for example, to run `mp` on `SC_fillable.uarch`:

```
check -i ~/pipecheck_tutorial/tests/SC_tests/mp.test -m SC_fillable.uarch
```

will generate `mp.pdf` in your current directory.

If you want to run a test through a microarchitecture that is in a directory other than `uarches` (including subdirectories of `uarches`), use the `-d` flag:

```
check -i <input test file> -m <name of uarch file> -d <directory of uarch file>
```

So for example, to run the solution `SC.uarch` on `mp`, you can run the following:

```
check -i ~/pipecheck_tutorial/tests/SC_tests/mp.test -m SC_fillable.uarch
-d ~/pipecheck_tutorial/uarches/solutions/
```

The `check` script has other options that allow you to control input and output, including generating only GraphViz files rather than converting to PDF, increased verbosity of output, and different output directories. To see a full list, run `check` with the `-h` option (`check -h`).

5.2 Running a suite of tests with `run_tests`

To run a suite of litmus tests through a microarchitecture in the `uarches` directory of the repository:

```
run_tests -t <test file directory> -m <name of uarch file>
```

This will run all the tests in `<test file directory>` on the uarch provided, and put the output GraphViz files in `/pipecheck_tutorial/out/`. At the end, it will output statistics on

- how many tests were run
- how many were correct/too strict/buggy

To run a suite through a microarchitecture in a directory other than `uarches`, use the `-d` flag:

```
run_tests -t <test file directory> -m <name of uarch file> -d <directory of uarch file>
```

The `run_tests` script has other options that allow you to control input and output, including increased verbosity of output and choice of output directory. To see a full list, run `run_tests` with the `-h` option (`run_tests -h`).

5.3 Generating PDFs from GraphViz files with `gen_graph`

To convert a GraphViz file into an actual PDF of the graphs in it, use the `gen_graph` script:

```
gen_graph -i <input GraphViz file>
```

This will generate a PDF in your current directory with the same name as that of the input file. So for example,

```
gen_graph -i mp.test.gv
```

will generate `mp.test.pdf` in your current directory, and delete `mp.test.gv`.

5.4 Full stack verification with TriCheck

NOTE: The HLL memory model we provide is the C11 herd model from prior work [1]. Before running TriCheck, define the TRICHECK_HOME environment variable:

```
export TRICHECK_HOME=/home/check/TriCheck
```

Paths to relevant TriCheck files:

```
Litmus test templates: $TRICHECK_HOME/tests/templates
C11 herd model: $TRICHECK_HOME/util/herd
Compiler mappings: $TRICHECK_HOME/util/compile.txt
Litmus test generator: $TRICHECK_HOME/util/release-generate-tests.py
TriCheck:  $TRICHECK_HOME/util/release-run-all.py
TriCheck output parser: $TRICHECK_HOME/util/release-parse-results.py
```

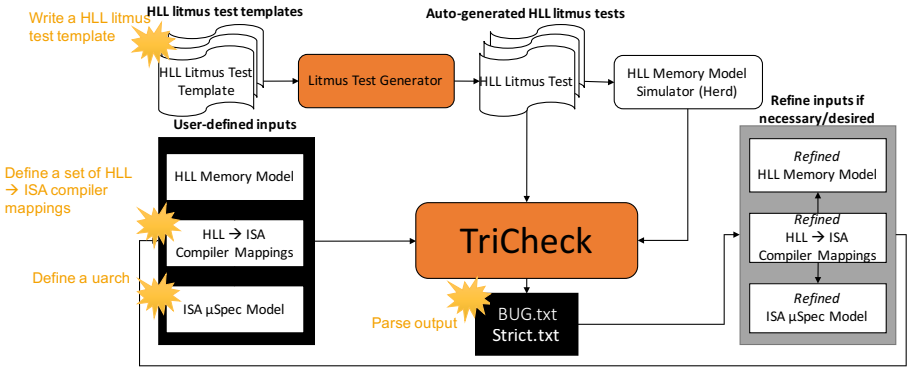

Figure 2: Overview of TriCheck inputs and flow described below

5.4.1 How to write a C11 herd litmus test template

Store Buffering (SB)	
P0	P1
W x ← 1	W y ← 1
R y → 0	R x → 0

Figure 3: ISA pseudo-code for Store Buffering (SB) litmus test.

```
cd $TRICHECK_HOME/tests/templates
```

Sample C11 herd litmus test template, sb.litmus:

```
C <TEST>

{
[x] = 0;
[y] = 0;
}

P0 (atomic_int* y, atomic_int* x) {
```

```
  atomic_store_explicit(x,1,memory_order_<ORDER_STORE>);
  int r0 = atomic_load_explicit(y,memory_order_<ORDER_LOAD>);
}

P1 (atomic_int* y, atomic_int* x) {
  atomic_store_explicit(y,1,memory_order_<ORDER_STORE>);
  int r1 = atomic_load_explicit(x,memory_order_<ORDER_LOAD>);
}
exists (0:r0=0 /\ 1:r1=0)
```

5.4.2 How to write TriCheck compiler mappings

Edit compile.txt in util:

```
cd $TRICHECK_HOME/util
vim compile.txt
```

C11 HLL primitives in compile.txt will be mapped to regular ISA **Read** and **Write** operations surrounded by any number of semicolon-delimited prefix and suffix ISA **Fence** instructions with the **AccessTypes** specified in compile.txt.

5.4.3 How to run TriCheck

To run TriCheck and generate tests that use **all** combinations of C11 HLL memory ordering primitives, and to generate tests that use **only** compiler mappings (i.e., no ISA primitives for C11 atomic operations):

```
cd $TRICHECK_HOME/util
./release-generate-tests.py --all --fences
./release-run-all.py --pipecheck=/home/check/pipecheck_tutorial/src/pipecheck
```

By default, TriCheck will generate and run tests for all templates in $TRICHECK_HOME/tests/templates and all uarches in $TRICHECK_HOME/uarches. To see other options:

```
cd $TRICHECK_HOME/util
./release-generate-tests.py --help
./release-run-all.py --help
```

5.4.4 How to parse TriCheck output

To create files that list overly-constrained results (Strict.txt) and under-constrained results (BUG.txt) for each uarch:

```
cd $TRICHECK_HOME/util
./release-parse-results.py
```

Files will be located in: $TRICHECK_HOME/util/results/<uarch>

References

[1] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. In *43rd Annual Symposium on Principles of Programming Languages (POPL)*, 2016.