Automated Analysis of Weak Memory Models

Artem Yushkovskiy[1,2]

MSc Candidate

Supervisors: **Assoc. Prof. Keijo Heljanko**[1]
Docent Igor I. Komarov[2]

[1]Department of Computer Science,
School of Science,
Aalto University (Espoo, Finland)

[2]Faculty of Information Security
and Computer Technologies,
ITMO University (Saint Petersburg, Russia)

Espoo, Saint Petersburg, 2018

Problem statement (Цель работы)

To rework the proof-of-concept memory model-aware analysis tool `Porthos` [2] by:

- extending the C-like input language,
- revising its architecture and
- re-implementing the tool in order to enhance performance, extensibility, reliability and maintainability

Task specification (Задачи работы)

- Study the general framework for memory model-aware analysis of concurrent programs [1];
- Review the existing tools for memory model-aware analysis;
- Examine the existing architecture of `Porthos`, its strengths and weaknesses;
- Design a new architecture for `PorthosC` that allow to extend the input language to the (large subset of) C language, be robust, transparent, efficient and extensible.

Verification of concurrent software

Example: Write-write reordering (compiler relaxations)

{ x=0; y=0; }	
P	Q
p_0 : $x \leftarrow 1$	q_0 : $y \leftarrow 1$
p_1 : $r_p \leftarrow y$	q_1 : $r_q \leftarrow x$
exists $(r_p = 0 \wedge r_q = 0)$	

Verification of concurrent software

Example: Write-write reordering (compiler relaxations)

{ x=0; y=0; }	
P	Q
$p_0 : x \leftarrow 1$	$q_0 : y \leftarrow 1$
$p_1 : r_p \leftarrow y$	$q_1 : r_q \leftarrow x$
exists $(r_p = 0 \wedge r_q = 0)$	

Sequential Consistency

p_0, p_1, q_0, q_1	$(0; 1)$
q_0, q_1, p_0, p_1	$(1; 0)$

Verification of concurrent software

Example: Write-write reordering (compiler relaxations)

{ x=0; y=0; }	
P	Q
$p_0 : x \leftarrow 1$	$q_0 : y \leftarrow 1$
$p_1 : r_p \leftarrow y$	$q_1 : r_q \leftarrow x$
exists $(r_p = 0 \wedge r_q = 0)$	

Sequential
Consistency

p_0, p_1, q_0, q_1	$(0; 1)$
q_0, q_1, p_0, p_1	$(1; 0)$
p_0, q_0, p_1, q_1	$(1; 1)$
p_0, q_0, q_1, p_1	$(1; 1)$
q_0, p_0, p_1, q_1	$(1; 1)$
q_0, p_0, q_1, p_1	$(1; 1)$

Verification of concurrent software

Example: Write-write reordering (compiler relaxations)

{ x=0; y=0; }	
P	Q
$p_0 : x \leftarrow 1$	$q_0 : y \leftarrow 1$
$p_1 : r_p \leftarrow y$	$q_1 : r_q \leftarrow x$
exists $(r_p = 0 \land r_q = 0)$	

Sequential Consistency

p_0, p_1, q_0, q_1	$(0;1)$
q_0, q_1, p_0, p_1	$(1;0)$
p_0, q_0, p_1, q_1	$(1;1)$
p_0, q_0, q_1, p_1	$(1;1)$
q_0, p_0, p_1, q_1	$(1;1)$
q_0, p_0, q_1, p_1	$(1;1)$

Total Store Order

$\underline{p_1}, \underline{p_0}, q_0, q_1$	$(0;1)$	$p_0, p_1, \underline{q_1}, \underline{q_0}$	$(0;1)$	$\underline{p_1}, \underline{p_0}, \underline{q_1}, \underline{q_0}$	$(0;1)$
$q_0, q_1, \underline{p_1}, \underline{p_0}$	$(1;0)$	$\underline{q_1}, \underline{q_0}, p_0, p_1$	$(1;0)$	$\underline{q_1}, \underline{q_0}, \underline{p_1}, \underline{p_0}$	$(1;0)$
$\underline{p_1}, q_0, \underline{p_0}, q_1$	$(0;1)$	$p_0, \underline{q_1}, p_1, \underline{q_0}$	$(0;1)$	$\underline{p_1}, \underline{q_1}, \underline{p_0}, \underline{q_0}$	$(0;0)$
$\underline{p_1}, q_0, q_1, \underline{p_0}$	$(0;1)$	$p_0, \underline{q_1}, \underline{q_0}, p_1$	$(1;1)$	$\underline{p_1}, \underline{q_1}, \underline{q_0}, \underline{p_0}$	$(0;0)$
$q_0, \underline{p_1}, \underline{p_0}, q_1$	$(1;1)$	$\underline{q_1}, p_0, p_1, \underline{q_0}$	$(0;0)$	$\underline{q_1}, \underline{p_1}, \underline{p_0}, \underline{q_0}$	$(0;0)$
$q_0, \underline{p_1}, q_1, \underline{p_0}$	$(1;0)$	$\underline{q_1}, p_0, \underline{q_0}, p_1$	$(1;0)$	$\underline{q_1}, \underline{p_1}, \underline{q_0}, \underline{p_0}$	$(0;0)$

Verification of concurrent software

Example: Write-write reordering (compiler relaxations)

{ x=0; y=0; }	
P	Q
p_0 : $x \leftarrow 1$	q_0 : $y \leftarrow 1$
p_1 : $r_p \leftarrow y$	q_1 : $r_q \leftarrow x$
exists $(r_p = 0 \land r_q = 0)$	

Sequential Consistency

p_0, p_1, q_0, q_1	$(0;1)$
q_0, q_1, p_0, p_1	$(1;0)$
p_0, q_0, p_1, q_1	$(1;1)$
p_0, q_0, q_1, p_1	$(1;1)$
q_0, p_0, p_1, q_1	$(1;1)$
q_0, p_0, q_1, p_1	$(1;1)$

Total Store Order

$\underline{p_1}, \underline{p_0}, q_0, q_1$	$(0;1)$	$p_0, p_1, \underline{q_1}, \underline{q_0}$	$(0;1)$	$\underline{p_1}, \underline{p_0}, \underline{q_1}, \underline{q_0}$	$(0;1)$
$q_0, q_1, \underline{p_1}, \underline{p_0}$	$(1;0)$	$\underline{q_1}, \underline{q_0}, p_0, p_1$	$(1;0)$	$\underline{q_1}, \underline{q_0}, \underline{p_1}, \underline{p_0}$	$(1;0)$
$\underline{p_1}, q_0, \underline{p_0}, q_1$	$(0;1)$	$p_0, \underline{q_1}, p_1, \underline{q_0}$	$(0;1)$	$\underline{p_1}, \underline{q_1}, \underline{p_0}, \underline{q_0}$	$(0;0)$
$\underline{p_1}, q_0, q_1, \underline{p_0}$	$(0;1)$	$p_0, \underline{q_1}, \underline{q_0}, p_1$	$(1;1)$	$\underline{p_1}, \underline{q_1}, \underline{q_0}, \underline{p_0}$	$(0;0)$
$q_0, \underline{p_1}, \underline{p_0}, q_1$	$(1;1)$	$\underline{q_1}, p_0, p_1, \underline{q_0}$	$(0;0)$	$\underline{q_1}, \underline{p_1}, \underline{p_0}, \underline{q_0}$	$(0;0)$
$q_0, \underline{p_1}, q_1, \underline{p_0}$	$(1;0)$	$\underline{q_1}, p_0, \underline{q_0}, p_1$	$(1;0)$	$\underline{q_1}, \underline{p_1}, \underline{q_0}, \underline{p_0}$	$(0;0)$

Verification of concurrent software

Example: Store buffering (hardware relaxations)

{ x=0; y=0; }	
P	Q
p_0 : $x \leftarrow 1$	q_0 : $y \leftarrow 1$
p_1 : $r_p \leftarrow y$	q_1 : $r_q \leftarrow x$
exists ($r_p = 0 \wedge r_q = 0$)	

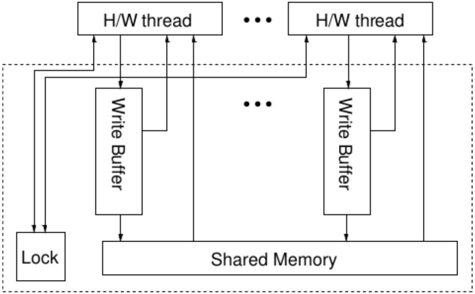


Figure: An x86-TSO abstract machine [3]

The weak memory model

Axiomatic semantics: The definition

- **Event** $\in \mathbb{E}$, a low-level primitive operation:
  - *memory event* $\in \mathbb{M} = \mathbb{R} \cup \mathbb{W}$: access to a local/shared memory,
  - *computational event* $\in \mathbb{C}$: computation over local memory, and
  - *barrier event* $\in \mathbb{B}$: synchronisation fences;
- **Relation** $\subseteq \mathbb{E} \times \mathbb{E}$:
  - *basic relations*:
    - *program-order* relation po $\subseteq \mathbb{E} \times \mathbb{E}$: (control-flow),
    - *read-from* relation `rf` $\subseteq \mathbb{W} \times \mathbb{R}$: (data-flow), and
    - *coherence-order* relation `co` $\subseteq \mathbb{W} \times \mathbb{W}$: (data-flow);
  - *derived relations*:
    - *union* `r1 | r2`,
    - *sequence* `r1 ; r2`,
    - *transitive closure* `r+`,
    - $\cdots$;
- **Assertion** over relations or sets of events:
  - *acyclicity*, *irreflexivity* or *emptiness*

The weak memory model

Testing candidate executions

{ x=0; y=0; }	
P	Q
p_0 : $x \leftarrow 1$	q_0 : $y \leftarrow 1$
p_1 : $r_p \leftarrow y$	q_1 : $r_q \leftarrow x$
exists $(r_p = 0 \wedge r_q = 0)$	

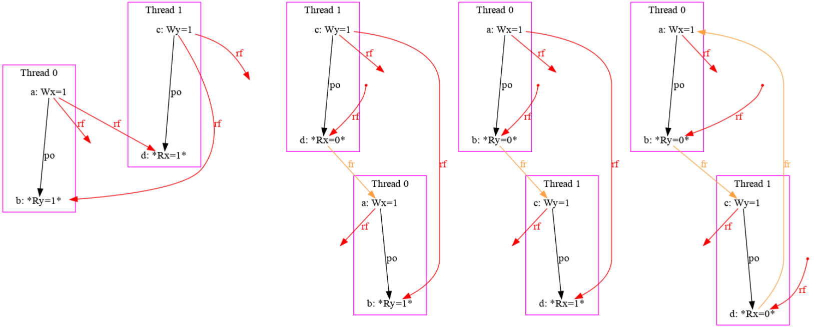


Figure: The four candidate executions allowed under x86-TSO

The weak memory model

Testing candidate executions

{ x=0; y=0; }	
P	Q
p_0 : $x \leftarrow 1$	q_0 : $y \leftarrow 1$
p_1 : $r_p \leftarrow y$	q_1 : $r_q \leftarrow x$
exists $(r_p = 0 \land r_q = 0)$	

SC model:

...

```
fr = (rf^-1; co)
acyclic(fr ∪ po)
```

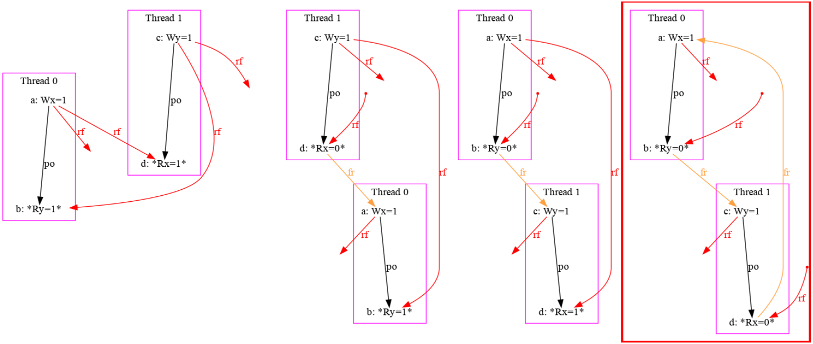


Figure: The four candidate executions allowed under x86-TSO

Tools for memory model-aware analysis

- `diy` tool suite:
  - `diy`, `diycross` and `diyone`, litmus tests generators,
  - `litmus`, a litmus test concrete executor, and
  - `herd`, a weak memory model simulator;
- the stateless model checkers (`CHESS`, `Nidhugg`);
- the tool for automated synthesis of the synchronisation primitives `musketeer`;
- the instrumenting compiler `goto-cc` which is a part of `CBMC` model checker;
- the tool `Porthos` for analysing the portability of the C programs;
- and others.

Portability analysis

The `Porthos` tool

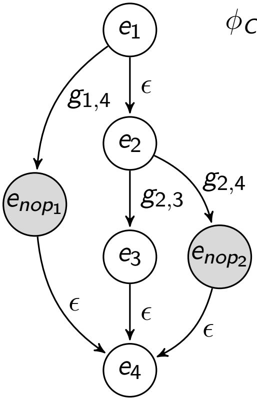
- Let the function $cons_{\mathcal{M}}(P)$ calculate the set of executions of program P consistent under the memory model $\mathcal{M}$.

Definition (Portability [2])

Let $\mathcal{M}_{\mathcal{S}}$, $\mathcal{M}_{\mathcal{T}}$ be two weak memory models. The program P is portable from $\mathcal{M}_{\mathcal{S}}$ to $\mathcal{M}_{\mathcal{T}}$ if $cons_{\mathcal{M}_{\mathcal{T}}}(P) \subseteq cons_{\mathcal{M}_{\mathcal{S}}}(P)$

- Portability as an SMT-based bounded reachability problem: $\phi = \phi_{CF} \wedge \phi_{DF} \wedge \phi_{\mathcal{M}_{\mathcal{T}}} \wedge \phi_{\neg\mathcal{M}_{\mathcal{S}}}$
- $\mathtt{SAT}(\phi) \implies$ the portability bug

Encoding for the control-flow: An example



$$
\begin{aligned}
\phi_{CF} = & [\mathbf{x}(e_2) \Rightarrow \mathbf{x}(e_1)] \\
& \wedge [\mathbf{x}(e_3) \Rightarrow \mathbf{x}(e_2)] \\
& \wedge [\mathbf{x}(e_{nop_1}) \Rightarrow \mathbf{x}(e_1)] \\
& \wedge [\mathbf{x}(e_{nop_2}) \Rightarrow \mathbf{x}(e_2)] \\
& \wedge [\mathbf{x}(e_4) \Rightarrow (\mathbf{x}(e_{nop_1}) \vee \mathbf{x}(e_3) \vee \mathbf{x}(e_{nop_2}))] \\
& \wedge [\mathbf{x}(e_{nop_1}) \wedge \mathbf{x}(e_1) \Rightarrow g_{1,4}] \\
& \wedge [(\mathbf{x}(e_3) \wedge \mathbf{x}(e_2)) \Rightarrow g_{2,3}] \\
& \wedge [(\mathbf{x}(e_{nop_2}) \wedge \mathbf{x}(e_2)) \Rightarrow g_{2,4}] \\
& \wedge \neg[\mathbf{x}(e_2) \wedge \mathbf{x}(e_{nop_1})] \\
& \wedge \neg[\mathbf{x}(e_3) \wedge \mathbf{x}(e_{nop_2})]
\end{aligned}
$$

Figure: Example of encoding for the control-flow of the event-flow graph

Encoding for the data-flow

- SSA-indices are computed as following:
  - any access to a shared variable (both read and write) increments its SSA-index;
  - only writes to a local variable increment its SSA-index (reads preserve indices);
  - no access to a constant variable or computed (evaluated) expression changes their SSA-index.

The data-flow of an event is encoded as following:

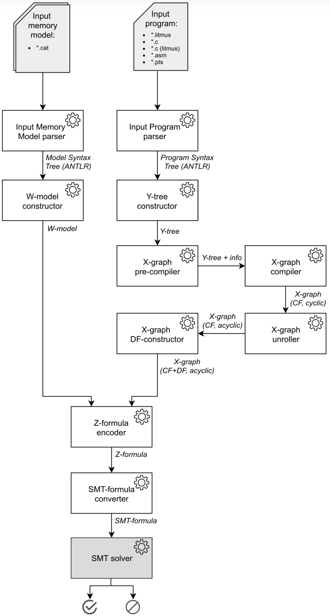
$$
\begin{aligned}
\phi_{DF_{e=\mathtt{load}(r\leftarrow l)}} &= [\mathsf{x}(e) \Rightarrow (r_{i+1} = l_{i+1})] \\
\phi_{DF_{e=\mathtt{store}(l\leftarrow r)}} &= [\mathsf{x}(e) \Rightarrow (l_{i+1} = r_i)] \\
\phi_{DF_{e=\mathtt{eval}(\cdot)}} &= [\mathsf{x}(e) \Rightarrow \mathbf{v}(e)]
\end{aligned}
\tag{1}
$$

The input language

The input language parser used by `Porthos` suffered from several disadvantages:

- it contained the parser code inlined directly into the grammar (hardly maintainable);
- the semantics of operations and kinds of variables (global or shared) were determined syntactically (4 different types of assignment:'=', ':=', '<-' and '<:-', each for different kinds of arguments);
- restricted syntax for expressions.
- In contrast, `PorthosC` uses the full C language grammar of proposed in the C11 standard [**jtc2011sc22**] and the visitor that converts the ANTLR grammar to the AST (`Y-tree`).

Architecture



The X-graph internal representation

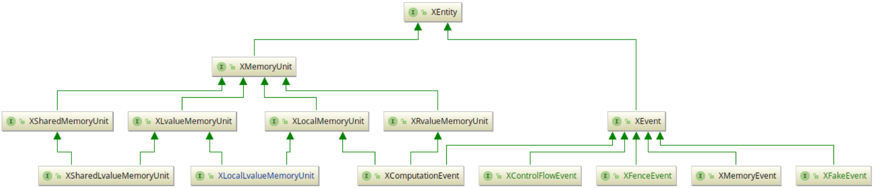


Figure: The inheritance tree of main X-graph interfaces

The X-graph compiler

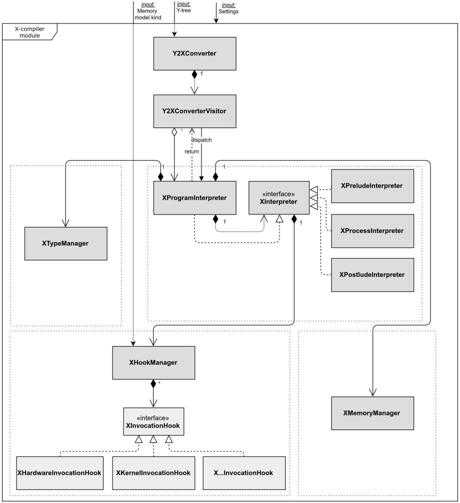


Figure: Main components of the X-compilation processing unit

X-graph unrolling

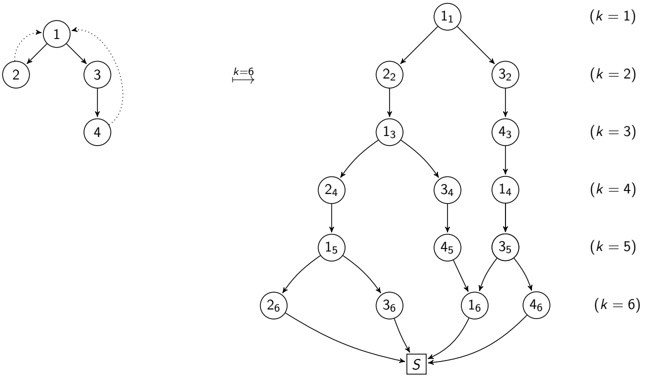


Figure: Example of the flow graph unrolling up to bound $k = 6$

Evaluation

Much better.
[to be done]

Summary

- The general framework for memory model-aware analysis was implemented in `PorthosC`;
- The input language has been extended;
- The old architecture of `Porthos` has been analysed and considered while designing the new architecture for `PorthosC`;
- to be done: more

Bibliography I

- Jade Alglave. "A shared memory poetics". In: *La Thèse de doctorat, L'université Paris Denis Diderot* (2010).
- Hernán Ponce de León et al. "Portability Analysis for Weak Memory Models. PORTHOS: One Tool for all Models". In: *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*. 2017, pp. 299–320. DOI: 10.1007/978-3-319-66706-5_15. URL: https://doi.org/10.1007/978-3-319-66706-5_15.
- Peter Sewell et al. "x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors". In: *Communications of the ACM* 53.7 (2010), pp. 89–97.