Safe Asynchronous Multicore Memory Operations

Matko Botincan, Mike Dodds, Alastair F. Donaldson, Matthew J. Parkinson
MOTIVATION

- Asynchronous memory operations are **efficient**
- Programs using them are **prone to bugs** (data races, memory safety)
- It is difficult to detect such bugs due to **nondeterminism**
- \( \rightarrow \) Need for formal verification techniques!
MAIN CONTRIBUTIONS

- Extension of separation logic for asynchronous memory operations
- Automation of proof technique
- Prototype implementation for a C-like language
Asynchronous memory operations
OPERATIONS

- **get**(x, y, s, t): copy a block of s bytes starting at host address y to local address x using tag t
- **put**(x, y, s, t): copy a block of s bytes starting at local address x to host address y using tag t
- **wait**(t): wait for all operations associated with t to terminate
A **get** issued by \( c_i \) can race with:

- A regular read or write by \( c_i \)
- A **get** or **put** by \( c_i \)
- A regular write access by the host core
- A **put** by \( c_j \) (where \( i \) might be equal to \( j \))

... given that the respective operations access overlapping memory regions
Implementation in separation logic with permissions
AVOIDING DATA RACES USING PERMISSIONS

- **Permission**: Real number \( p \in (0, 1] \)
- \( p = 1 \): Write permission
- \( p \in (0, 1) \): Read permission
- Permissions can be split
- After issuing a `get` or `put`, the thread loses the respective permissions until `wait` is called

\[ \Rightarrow \text{This guarantees absence of data races!} \]
PERMISSIONS - EXAMPLE

\textbf{get}(a, d, 3, t_1)

\{a: 1, b: 1, c: 1, d: 1, e: 1\}

\textbf{get}(b, d, 3, t_1)

\{a: 1, b: 1, c: 1, d: 1, e: 1\}

\textbf{put}(c, e, 3, t_1)

\{a: 0, b: 1, c: 1, d: 1/2, e: 1\}

\textbf{wait}(t_1)

\{a: 0, b: 0, c: 1, d: 1/4, e: 1\}

\textbf{get}(b, d, 3, t_1)

\{a: 0, b: 0, c: 1, d: 1/4, e: 1\}

\textbf{get}(a, d, 3, t_1)

\{a: 0, b: 0, c: 1/2, d: 1/4, e: 0\}

\textbf{wait}(t_1)

\{a: 1, b: 1, c: 1/2, d: 1/4, e: 0\}

\textbf{put}(c, e, 3, t_1)

\{a: 0, b: 0, c: 1/2, d: 1/4, e: 0\}

\textbf{wait}(t_1)

\{a: 1, b: 1, c: 1, d: 1, e: 1\}
Separation logic = Hoare logic + **separating conjunction** (‘∗’)

- Hoare triple: {P} C {Q}
- P₁ ∗ P₂: Heap (memory) can be divided into two disjoint parts such that one satisfies P₁ and the other satisfies P₂

\[
\text{FRAME} \quad \frac{\{P\} \ C \ \{Q\}}{\{P \ast F\} \ C \ \{Q \ast F\}}
\]
EXAMPLE: SPECIFICATION OF get

\{ \text{arr}_l(x, s, 1, xs) \ast \text{arr}_h(y, s, p, ys) \ast \text{pend}(t, \emptyset) \}\n
get(x, y, s, t)

\{ \text{pend}(t, \{y_h, x_l, s, p, ys\} \cup \emptyset) \}

- Thread needs read access \textbf{y} and write access to \textbf{x}
- After issuing \textbf{get}, the thread loses the permissions (needs to call \text{wait})
- Thread might still read from \textbf{y} if it has an additional permission
Permissions can be split arbitrarily

The respective permissions are temporarily lost after a `get` (or `put`) is issued (until `wait`)
Automation and Implementation
AUTOMATION AND IMPLEMENTATION

- **asyncStar**: tool built upon coreStar
Issues with Automation

- Permissions as fractions in $(0, 1]$ -> Represent them with binary trees
- Symbolic execution alone often does not converge -> Combine with abstract interpretation
- Calls to SMT solver expensive -> Only call the solver if necessary
### EVALUATION (ALL TIMINGS IN SECONDS)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Correct</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Symbolic states</td>
<td>Total time</td>
<td>%AI</td>
<td>%SMT</td>
</tr>
<tr>
<td>particle-sim</td>
<td>564</td>
<td>331</td>
<td>&lt; 1</td>
<td>98</td>
</tr>
<tr>
<td>1-buffer</td>
<td>67</td>
<td>13</td>
<td>&lt; 1</td>
<td>89</td>
</tr>
<tr>
<td>1-buffer-IO</td>
<td>80</td>
<td>31</td>
<td>&lt; 1</td>
<td>94</td>
</tr>
<tr>
<td>2-buffer</td>
<td>259</td>
<td>1268</td>
<td>&lt; 1</td>
<td>&gt; 99</td>
</tr>
<tr>
<td>2-buffer-IO</td>
<td>286</td>
<td>1871</td>
<td>&lt; 1</td>
<td>&gt; 99</td>
</tr>
<tr>
<td>3-buffer</td>
<td>412</td>
<td>7681</td>
<td>&lt; 1</td>
<td>&gt; 99</td>
</tr>
<tr>
<td>3-buffer-IO</td>
<td>443</td>
<td>8416</td>
<td>&lt; 1</td>
<td>&gt; 99</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Buggy</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Symbolic states</td>
<td>Total time</td>
<td>%AI</td>
<td>%SMT</td>
</tr>
<tr>
<td>particle-sim</td>
<td>58</td>
<td>27</td>
<td>2</td>
<td>97</td>
</tr>
<tr>
<td>1-buffer</td>
<td>32</td>
<td>7</td>
<td>&lt; 1</td>
<td>92</td>
</tr>
<tr>
<td>1-buffer-IO</td>
<td>36</td>
<td>16</td>
<td>&lt; 1</td>
<td>96</td>
</tr>
<tr>
<td>2-buffer</td>
<td>82</td>
<td>318</td>
<td>&lt; 1</td>
<td>&gt; 99</td>
</tr>
<tr>
<td>2-buffer-IO</td>
<td>88</td>
<td>389</td>
<td>&lt; 1</td>
<td>&gt; 99</td>
</tr>
<tr>
<td>3-buffer</td>
<td>113</td>
<td>618</td>
<td>&lt; 1</td>
<td>&gt; 99</td>
</tr>
<tr>
<td>3-buffer-IO</td>
<td>121</td>
<td>663</td>
<td>&lt; 1</td>
<td>&gt; 99</td>
</tr>
</tbody>
</table>
OPEN QUESTIONS/ISSUES

- The system is sound but not necessarily complete
- Proof of soundness?
- Evaluation: Only tested removing one wait (not a problem given that system is sound)
CONCLUSION

- Being able to automatically prove race-freeness of a program is a huge benefit.
- The presented prototype achieves this for a C-like language.
- Could provide the basis for more advanced tools and applied in other domains.