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**

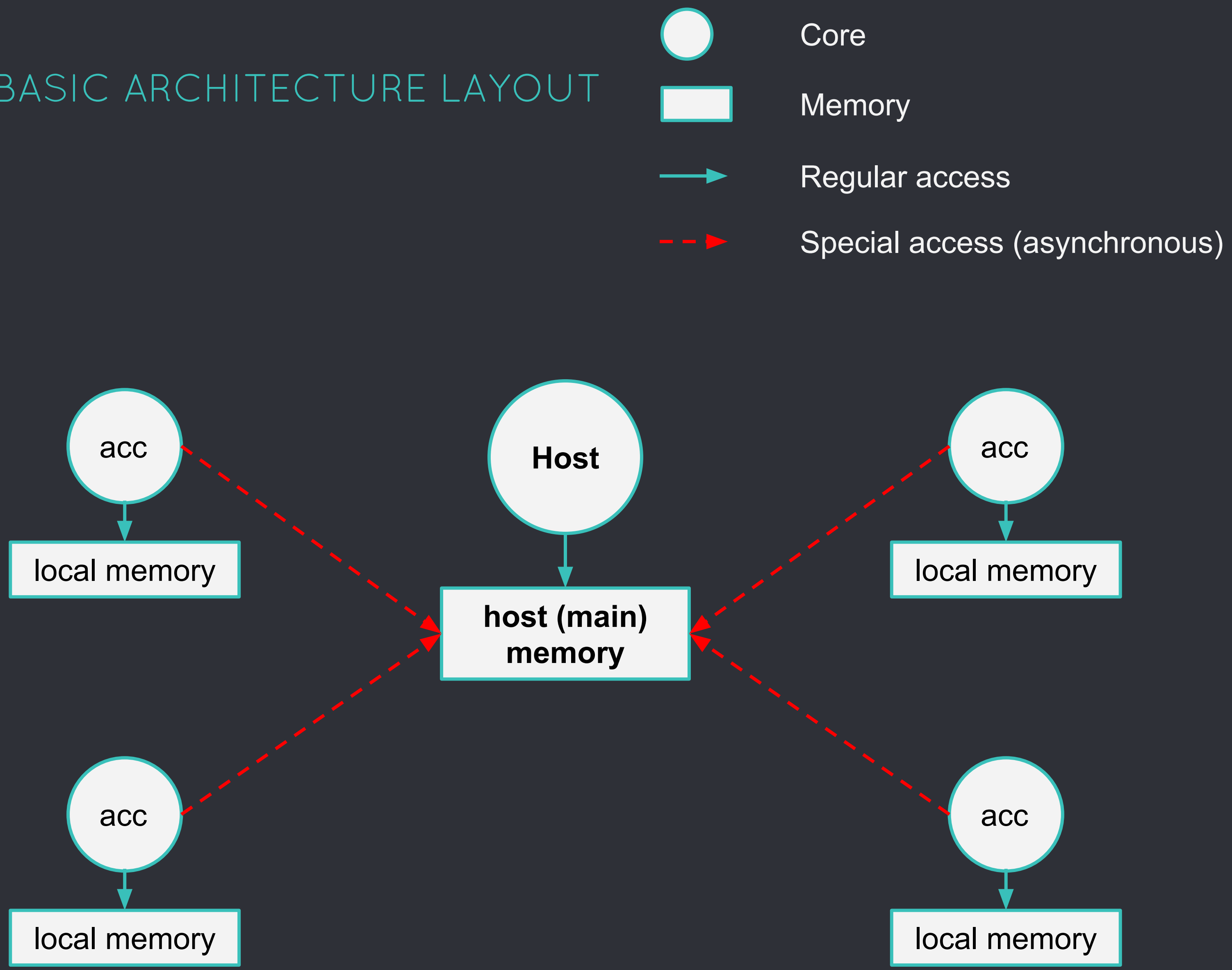
- **-> Need for formal verification techniques!**

MAIN CONTRIBUTIONS

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

Asynchronous memory operations

BASIC ARCHITECTURE LAYOUT



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

DATA RACES

A **get** issued by $\mathbf{c}_i$ can race with:

- A regular read or write by $\mathbf{c}_i$
- A **get** or **put** by $\mathbf{c}_i$
- A regular write access by the host core
- A **put** by $\mathbf{c}_j$ (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**∈(0, 1]
- **p** = 1: Write permission
- **p**∈(0, 1): Read permission
- Permissions can be split
- After issuing a **get** or **put**, the thread loses the respective permissions until **wait** is called

-> This guarantees absence of data races!

PERMISSIONS - EXAMPLE

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

get$(a, d, 3, t_1)$

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

get$(b, d, 3, t_1)$

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

put$(c, e, 3, t_1)$

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

wait(t_1)

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

SEPARATION LOGIC RECAP

- Separation logic = Hoare logic + **separating conjunction** (‘*’)
- Hoare triple: {P} C {Q}
- $P_1 * P_2$: Heap (memory) can be divided into two disjoint parts such that one satisfies P_1 and the other satisfies P_2

$$\textsc{Frame}\ \frac{\{P\}\ C\ \{Q\}}{\{P * F\}\ C\ \{Q * F\}}$$

EXAMPLE: SPECIFICATION OF **get**

$$\{\mathrm{arr}_l(x, s, 1, xs) * \mathrm{arr}_h(y, s, p, ys) * \mathrm{pend}(t, \mathcal{O})\}$$
$$\quad \mathbf{get}(x, y, s, t)$$
$$\{\mathrm{pend}(t, \{\langle y_h, x_l, s, p, ys\rangle \cup \mathcal{O}\}$$

- Thread needs read access **y** and write access to **x**
- After issuing **get**, the thread loses the permissions (needs to call wait)
- Thread might still read from **y** if it has an additional permission

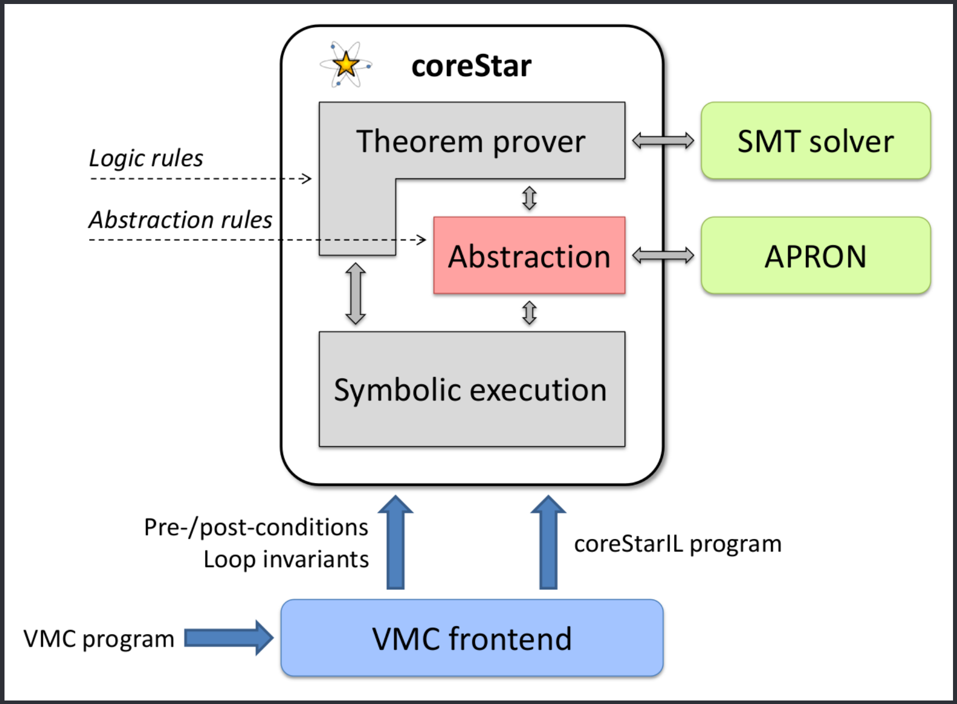
EXAMPLE PROOF OUTLINE (FROM PAPER)

$$\{\mathsf{arr}_\ell(x, s, 1, xs) * \mathsf{arr}_\ell(z, s, 1, zs) * \mathsf{arr}_h(y, s, \tfrac{1}{2}, ys) * \mathsf{pend}(t, \emptyset)\}$$
$$\quad \mathbf{get}(x, y, s, t);$$
$$\{\mathsf{arr}_\ell(z, s, 1, zs) * \mathsf{arr}_h(y, s, \tfrac{1}{4}, ys) * \mathsf{pend}\left(t, \{\langle y_h, x_\ell, s, \tfrac{1}{4}, ys\rangle\}\right)\}$$
$$\quad \mathbf{get}(z, y, s, t);$$
$$\{\mathsf{pend}\left(t, \{\langle y_h, x_\ell, s, \tfrac{1}{4}, ys\rangle, \langle y_h, z_\ell, s, \tfrac{1}{4}, ys\rangle\}\right)\}$$

- Permissions can be split arbitrarily
- The respective permissions are temporarily lost after a **get** (or **put**) is issued (until **wait**)

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)

Benchmark	Correct			
	Symbolic states	Total time	%AI	%SMT
particle-sim	564	331	< 1	98
1-buffer	67	13	< 1	89
1-buffer-IO	80	31	< 1	94
2-buffer	259	1268	< 1	> 99
2-buffer-IO	286	1871	< 1	> 99
3-buffer	412	7681	< 1	> 99
3-buffer-IO	443	8416	< 1	> 99

Benchmark	Buggy			
	Symbolic states	Total time	%AI	%SMT
particle-sim	58	27	2	97
1-buffer	32	7	< 1	92
1-buffer-IO	36	16	< 1	96
2-buffer	82	318	< 1	> 99
2-buffer-IO	88	389	< 1	> 99
3-buffer	113	618	< 1	> 99
3-buffer-IO	121	663	< 1	> 99

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