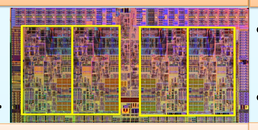
Using Escape Analysis in Dynamic Data Race Detection

Emma Harrington `15
Williams College
eh3@williams.edu

Multithreaded Programs

- Multithreaded programs can utilize multiple cores or multiple processors.
- But at the cost of greater programming complexity.



Race Conditions

Some resources should never be used simultaneously. Consider bathrooms.

Multithreaded programming can get awkward. Thread interference occurs when two threads access a shared resource without proper coordination.

Example: Suppose you have an account accessible from two ATMs. You have an idea. While your sister withdrawals $100 from one ATM, you withdrawal $100 from another. Will your scheme work?

Good Interleaving		Bad Interleaving	
Thread 1	Thread 2	Thread 1	Thread 2
`t1 = bal`		`t1 = bal`	
`bal = t1 - 100`			`t2 = bal`
	`t2 = bal`	`bal = t1 - 100`	
	`bal = t2 - 100`		`bal = t2 - 100`

Synchronization

Locks prevent interference.

Programmers use mutual exclusion locks to protect shared resources in multithreaded environments.

If your bank makes the ATMs acquire a lock before touching your balance, your scam will always fail.

Thread 1	Thread 2
`acquire lock`	
`t1 = bal`	
`bal = t1 + 100`	
`release lock`	
	`acquire lock`
	`t2 = bal`
	`bal = t2 – 100`
	`release lock`

Race Checkers

It's easy to forget a lock.

- Race checkers can find these mistakes.
- They must check every access to every variable to ensure proper synchronization.

Thread 1	Thread 2
`t1 = bal`	
	`acquire lock`
	`t2 = bal`
`bal = t1 + 100`	
	`bal = t2 – 100`
	`release lock`

The writes to the balance are unordered!

Improving Race Checker Performance

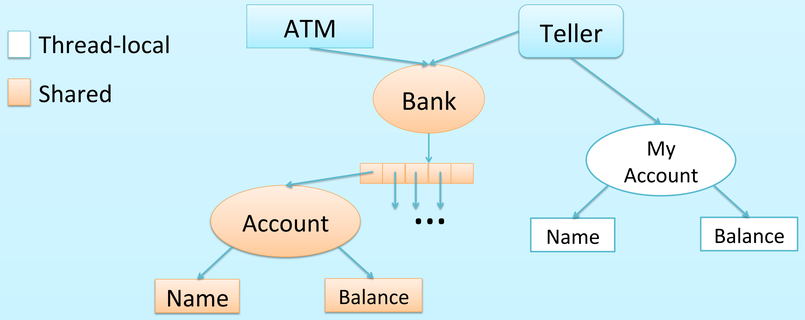
- When there are many resources and accesses, checking them can be expensive.
- Thread-local resources can be ignored by race checkers.

Dynamic Escape Analysis

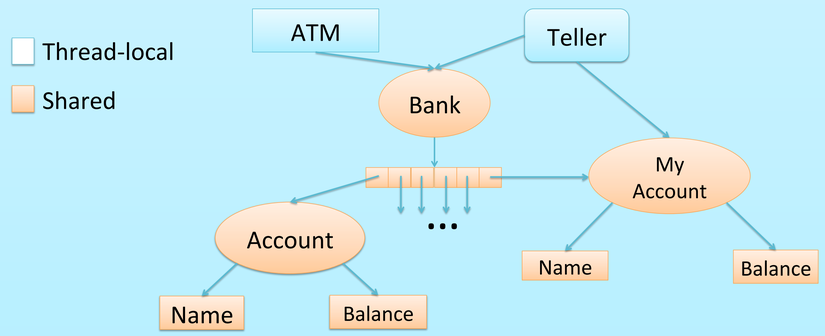
Basic Rules

- Objects are either shared or local.
  - Local objects are accessible from one thread.
  - Shared objects are accessible from multiple threads.
- Storing a reference to a thread local object in a shared object makes it and all objects reachable from it shared.

When my account is only accessible by the teller, there is no need to check accesses on my balance.



If the teller now writes our account to the bank, our balance can be reached by the ATM and the Teller so must be checked.

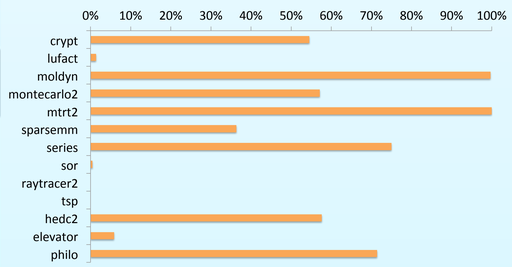


Implementation Details

- Build on top of the FastTrack race detector for Java [1].
- Evaluated on 13 benchmarks, including Java Grande Benchmarks [4].
- Static thread-local analysis (below) from IBM's WALA framework [2]. This analysis identifies classes, where every instance is only accessible from its creating thread, at compile time.

Results

How common are local accesses?



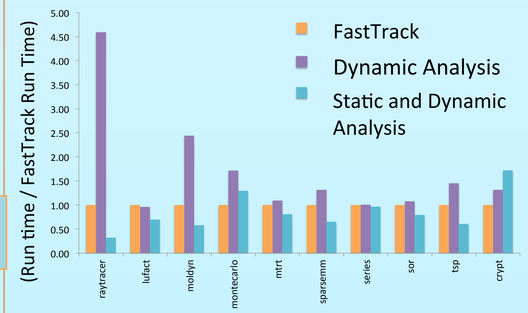
About half of the objects and 40% of the accesses are local and can be safely ignored by dynamic race detectors.

Does eliminating checks on local accesses speed up race detection?

On most of the benchmarks, identifying thread-local objects took longer than simply checking accesses on them.

Can adding a static analysis improve performance?

Some classes are inherently thread-local. If we identify these classes at compile time, we can ignore their instances at runtime.



The static escape analysis made my dynamic one more effective.

Seven out of the 13 benchmarks ran faster with the dynamic escape analysis tool applied on top of the static filter with average gains of 8%.

Future Work

- Use extra cores or the GPU to reduce the overhead by offloading the heap traversal required by dynamic escape analysis.
- Build into the garbage collector, which already efficiently traverses the heap.

References

[1] Flanagan, C., and Freund, S.N. FastTrack: Efficient and precise dynamic race detection. Commun. ACM 53, 11 (2010).
[2] T.J. Watson Libraries for Analysis (WALA), 2012.
[3] The Java Grande Multi-threaded Benchmarks.

Acknowledgements

Thanks to Professor Stephen N. Freund. This work was supported by NSF Grants 1116825 and 1421051.