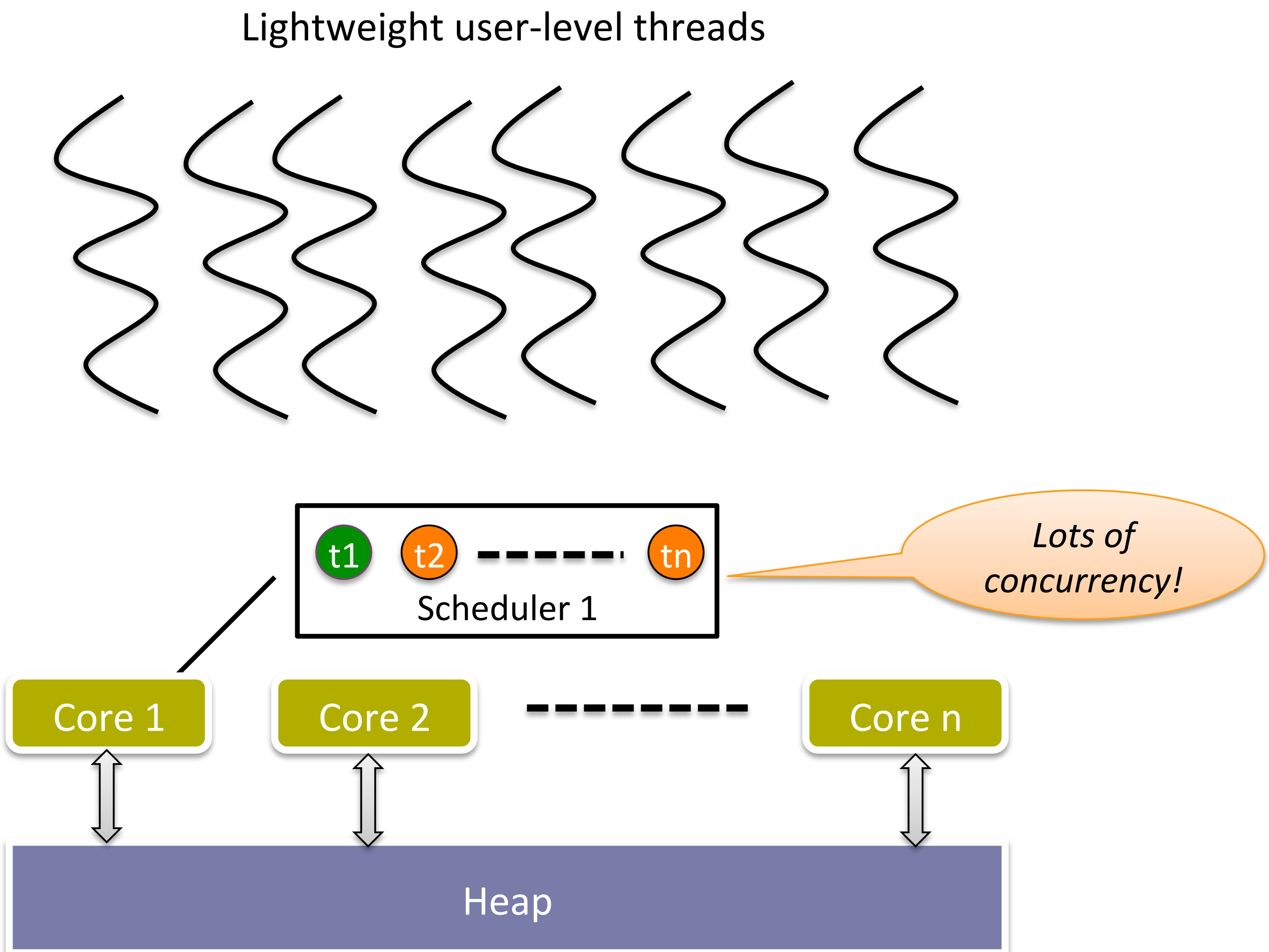
Eliminating Read Barriers through Procrastination and Cleanliness

KC Sivaramakrishnan
Lukasz Ziarek
Suresh Jagannathan

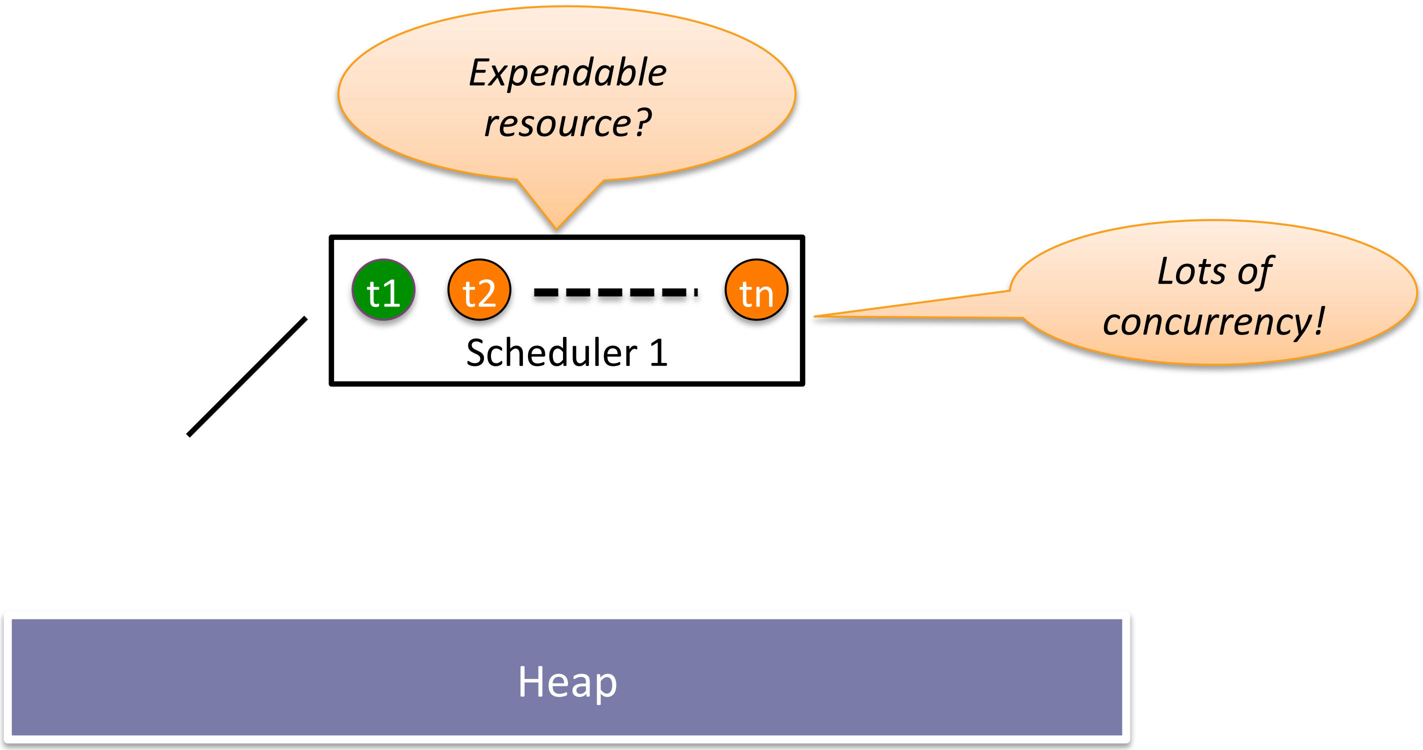
PURDUE UNIVERSITY

Big Picture



Lightweight user-level threads
t1
t2
tn
Scheduler 1
Lots of concurrency!
Core 1
Core 2
Core n
Heap

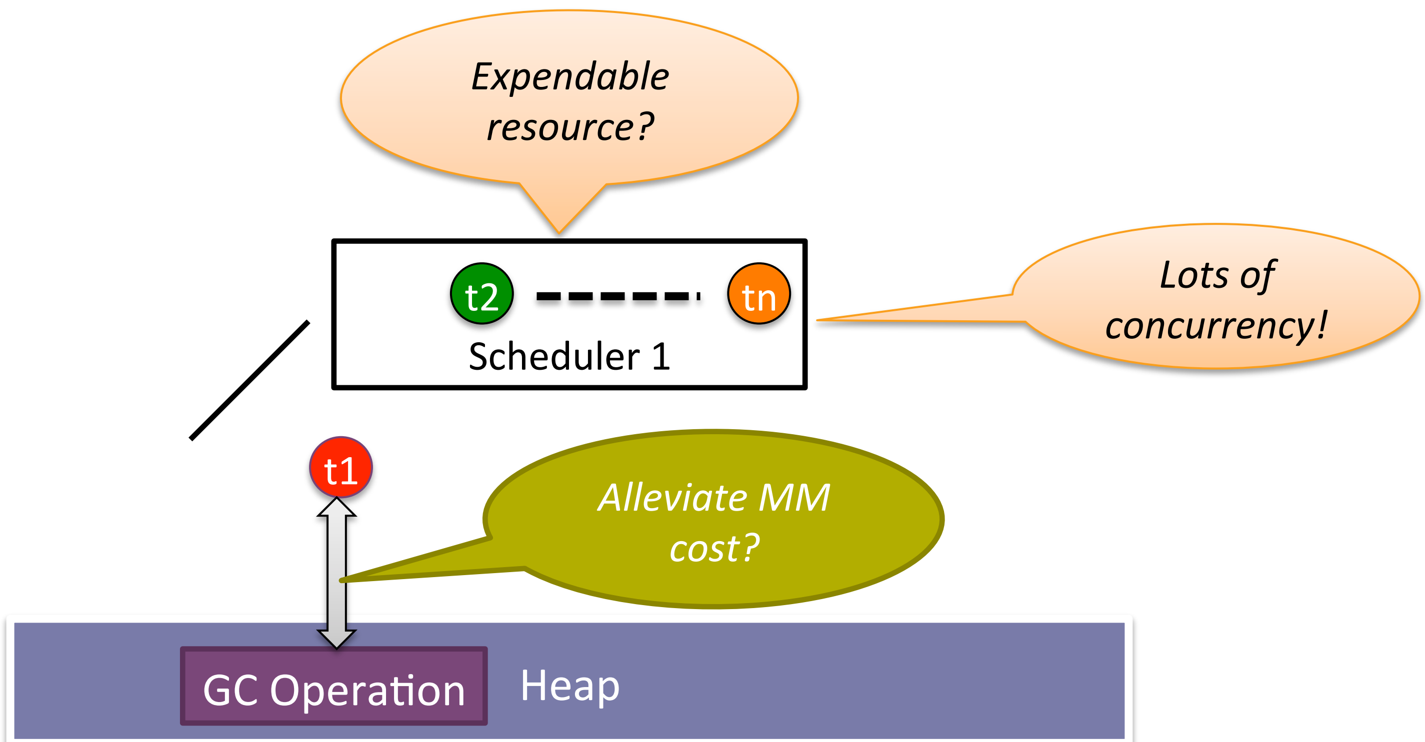
Big Picture



Expendable resource?
t1
t2
tn
Scheduler 1
Lots of concurrency!
Heap

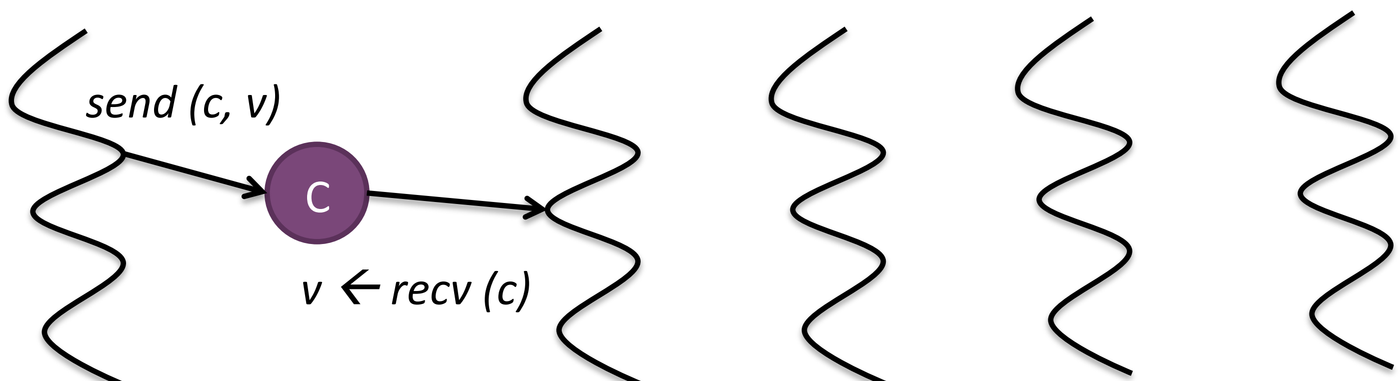
Big Picture

Exploit program concurrency
to
eliminate read barriers from thread-local collectors



MultiMLton

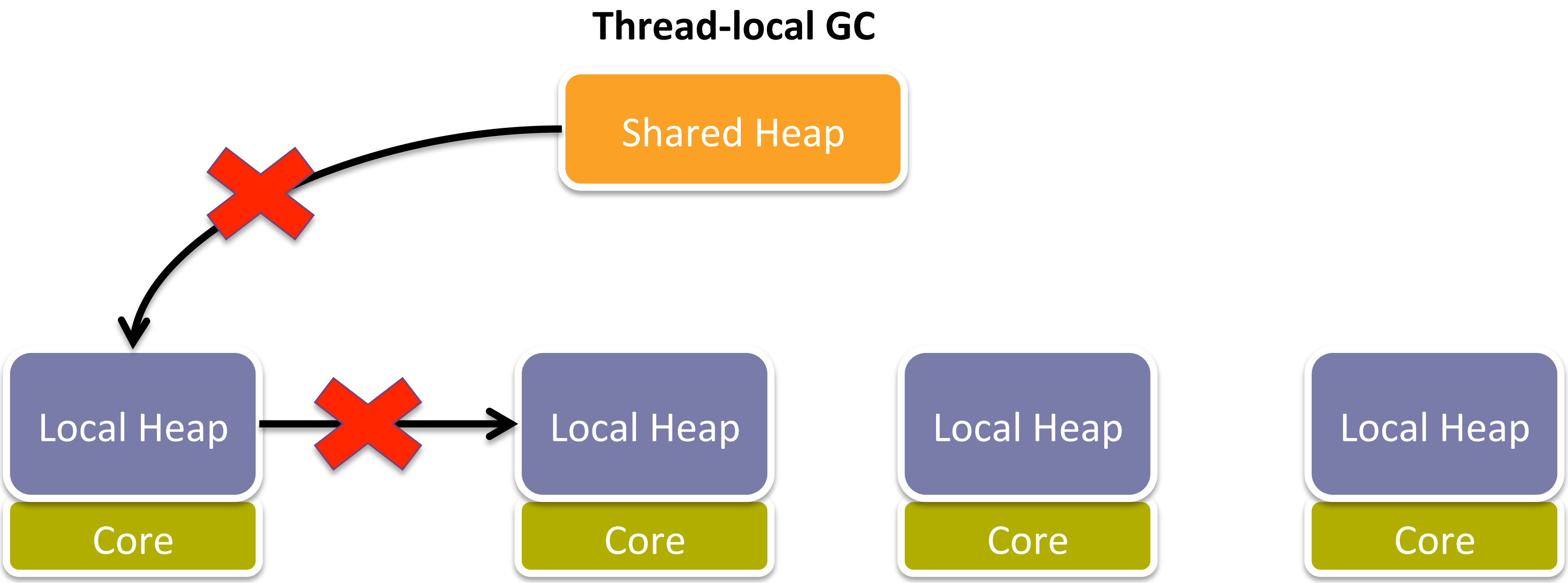
- Goals
  - Safety, Scalability, ready for future manycore processors
- Parallel extension of MLton – a whole-program, optimizing SML compiler
- Parallel extension of Concurrent ML
  - *Lots of Concurrency!*
  - Interact by sending messages over first-class channels



MultiMLton GC: Considerations

- Standard ML – functional PL with side-effects
  - Most objects are small and ephemeral
    - Independent generational GC
  - # Mutations << # Reads
    - Keep cost of reads to be low
- Minimize NUMA effects
- Run on non-cache coherent HW

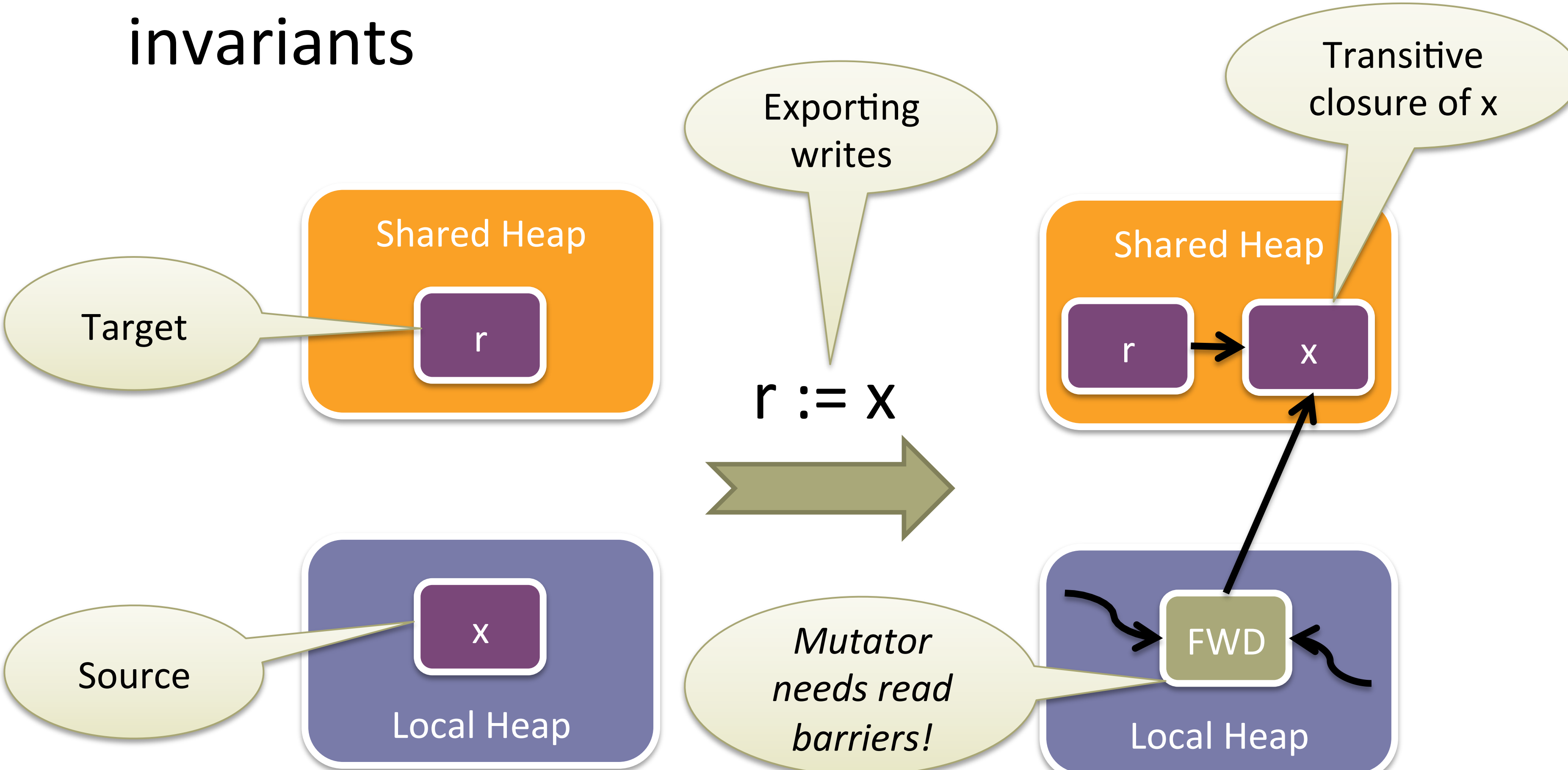
MultiMLton GC: Design



- NUMA Awareness
- Circumvent cache-coherence issues

Invariant Preservation

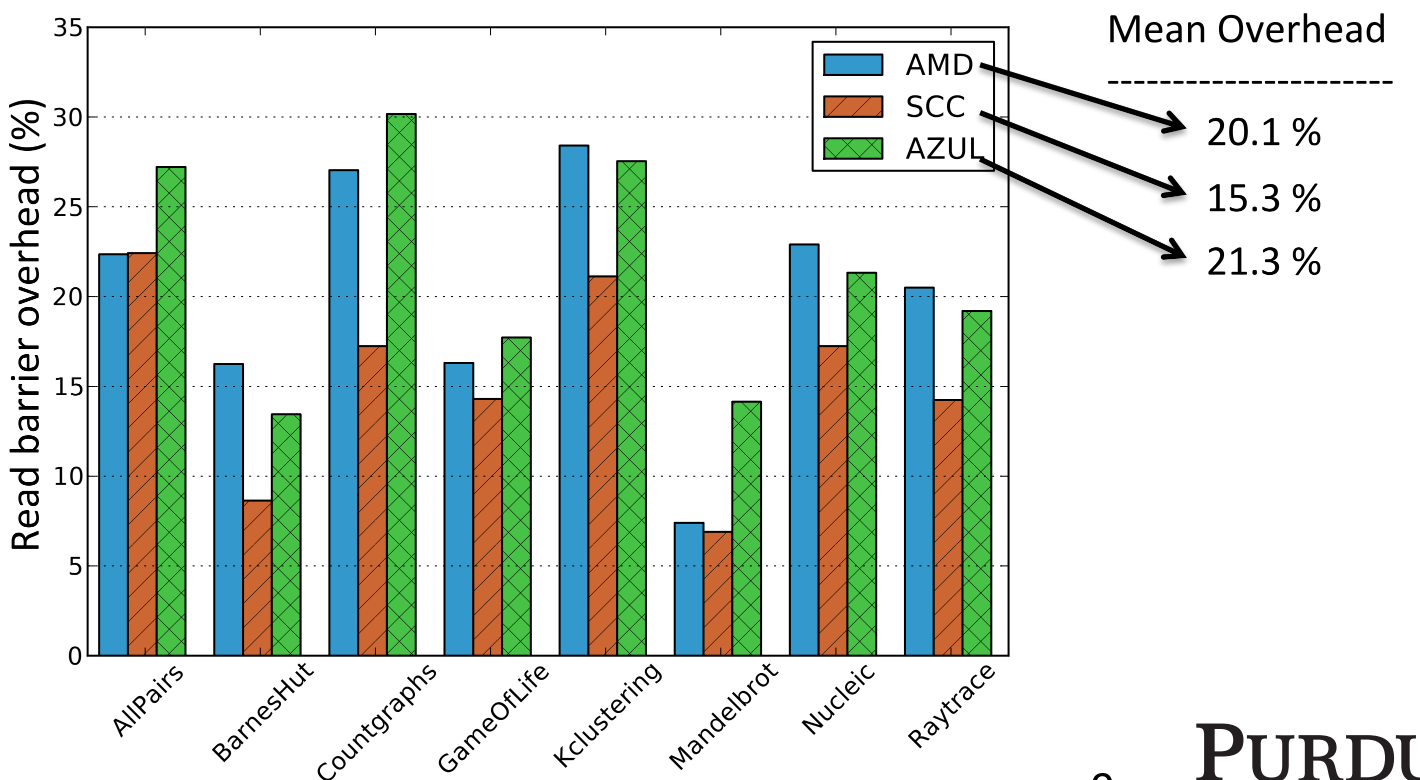
- Read and write barriers for preserving invariants



PURDUE UNIVERSITY

Challenge

- Object reads are pervasive
  - RB overhead ∝ cost (RB) * frequency (RB)
- Read barrier optimization
  - Stacks and Registers never point to forwarded objects



PURDUE
UNIVERSITY

Mutator and Forwarded Objects

$$\frac{\text{\# Encountered forwarded objects}}{\text{\# RB invocations}} < 0.00001$$

Eliminate read barriers altogether

PURDUE UNIVERSITY

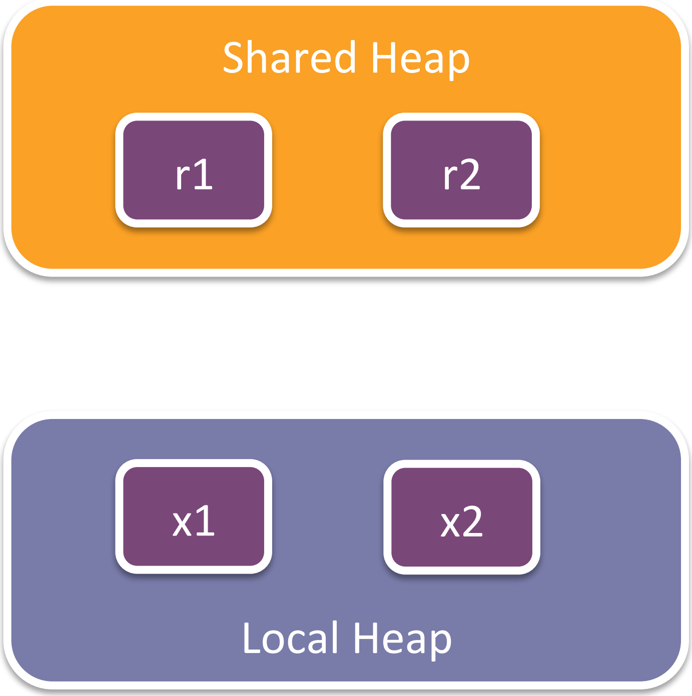
RB Elimination

- Visibility Invariant
  - Mutator does not encounter forwarded objects
- Observation
  - No forwarded objects created ⇒ visibility invariant ⇒ No read barriers
- Exploit concurrency → ***Procrastination!***

PURDUE UNIVERSITY

Procrastination

T1	T2
→ r1 := x1	r2 := x2

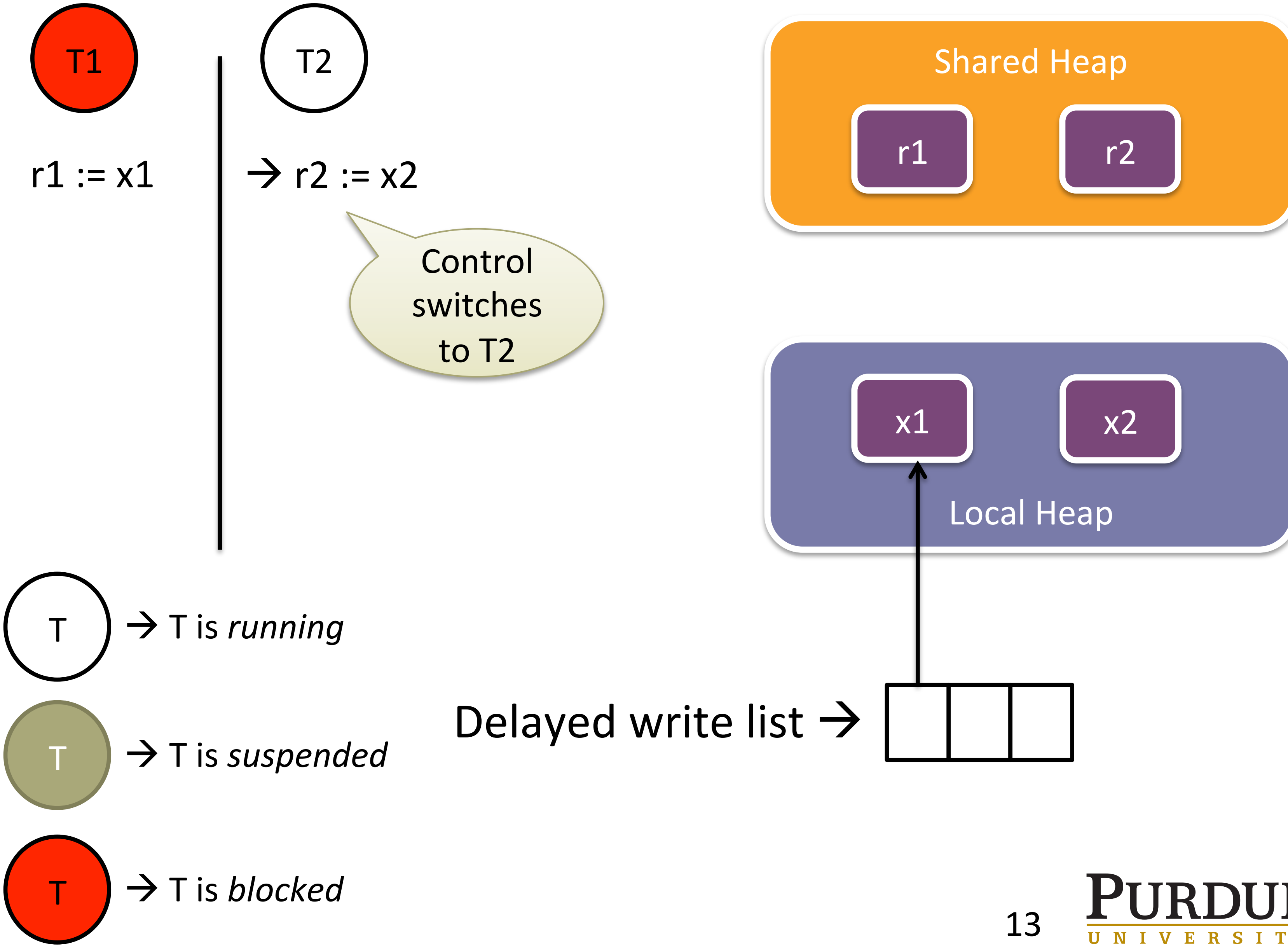


T → T is *running*

T → T is *suspended*

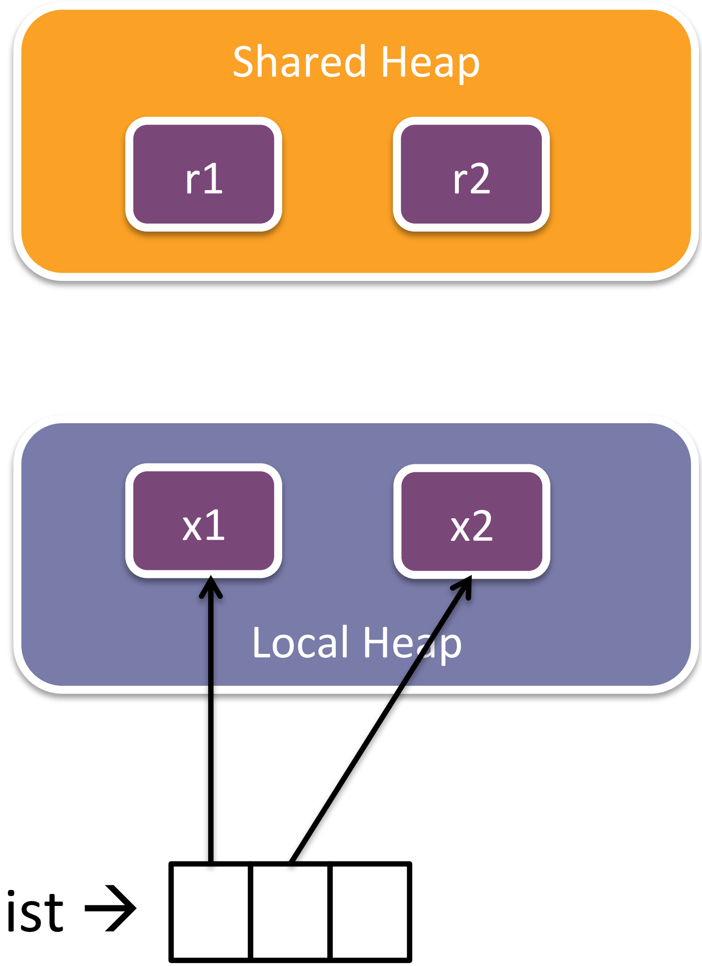
T → T is *blocked*

Procrastination



Procrastination

T1 | T2
r1 := x1 | r2 := x2



Delayed write list →

T → T is *running*

T → T is *suspended*

T → T is *blocked*

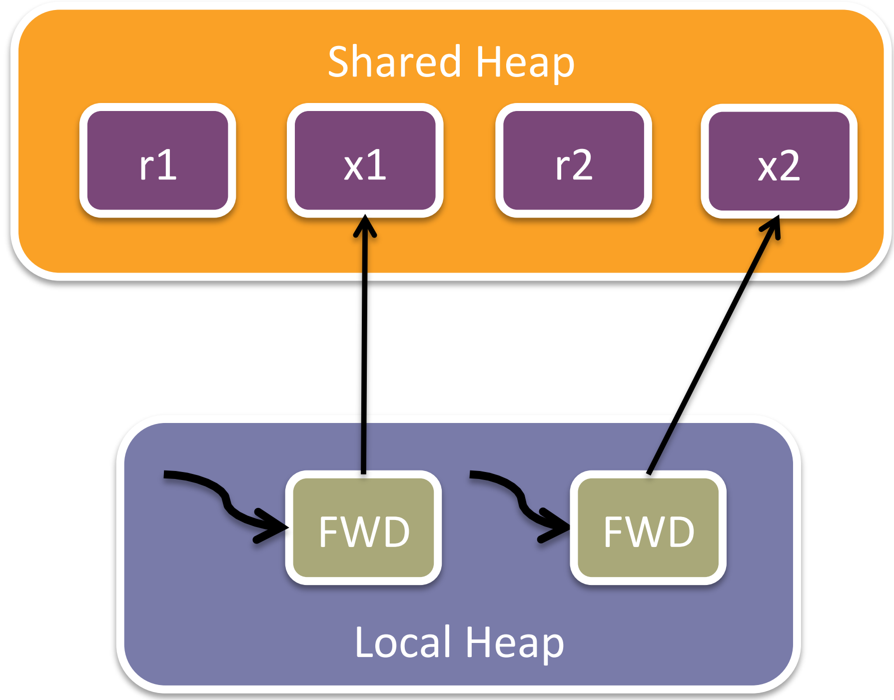
Procrastination

T1

r1 := x1

T2

r2 := x2



T → T is *running*

T → T is *suspended*

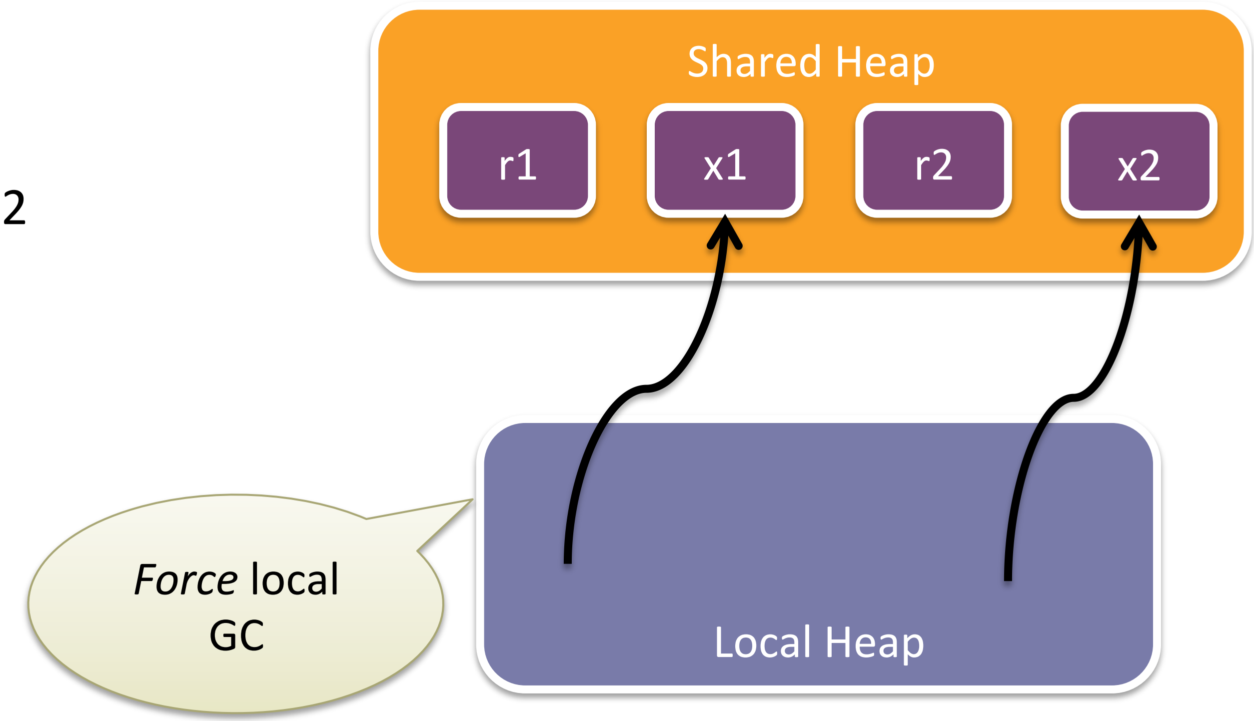
T → T is *blocked*

Delayed write list →

Procrastination

T1 | T2

→ r1 := x1 | r2 := x2



Delayed write list →

T → T is *running*

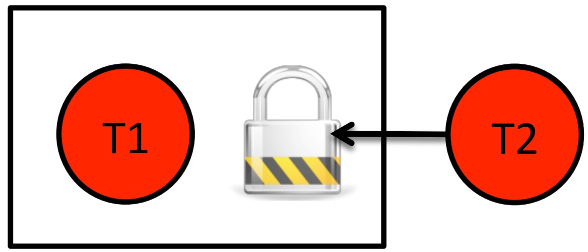
T → T is *suspended*

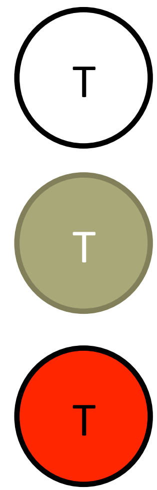
T → T is *blocked*

PURDUE UNIVERSITY

Correctness

- Does Procrastination introduce deadlocks?
  - Threads can be procrastinated while holding a lock!





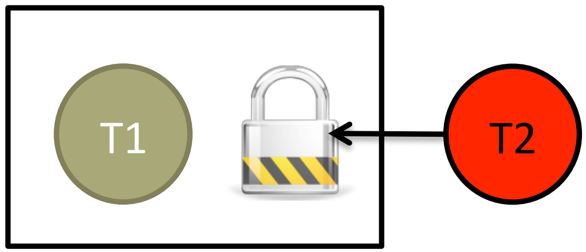
→ T is *running*

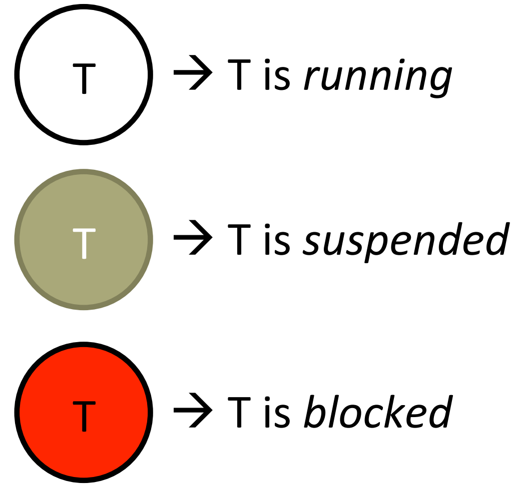
→ T is *suspended*

→ T is *blocked*

Correctness

- Does Procrastination introduce deadlocks?
  – Threads can be procrastinated while holding a lock!

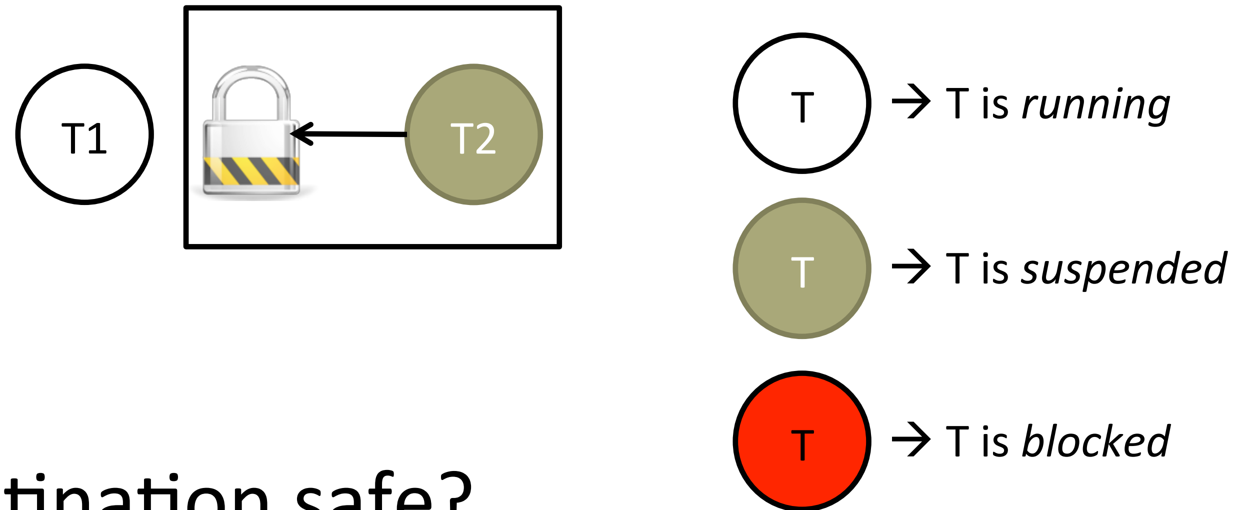




- Is Procrastination safe?
  – Yes. Forcing a local GC unblocks the threads.
  – No deadlocks or livelocks!

Correctness

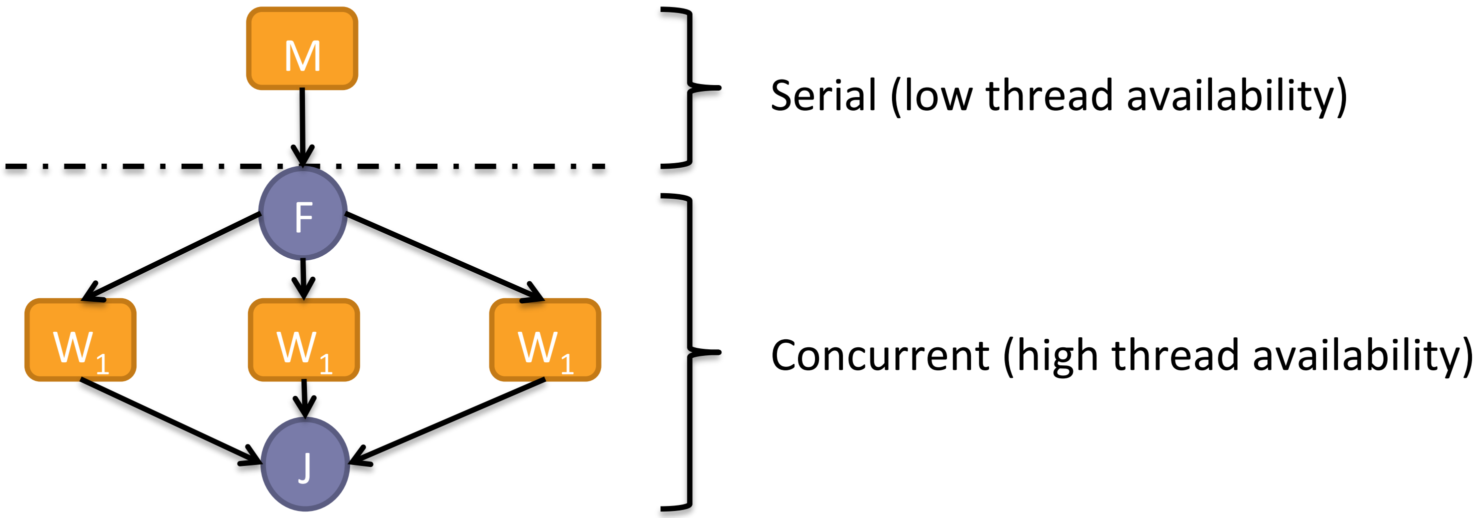
- Does Procrastination introduce deadlocks?
  - Threads can be procrastinated while holding a lock!



- Is Procrastination safe?
  - Yes. Forcing a local GC unblocks the threads.
  - No deadlocks or livelocks!

Is Procrastination alone enough?

- Efficacy (Procrastination) ∝ # Available runnable threads



- With Procrastination, **half** of local major GCs were *forced*

Eager exporting writes while preserving visibility invariant

PURDUE UNIVERSITY

Cleanliness

- A clean object closure can be lifted to the shared heap without breaking the visibility invariant

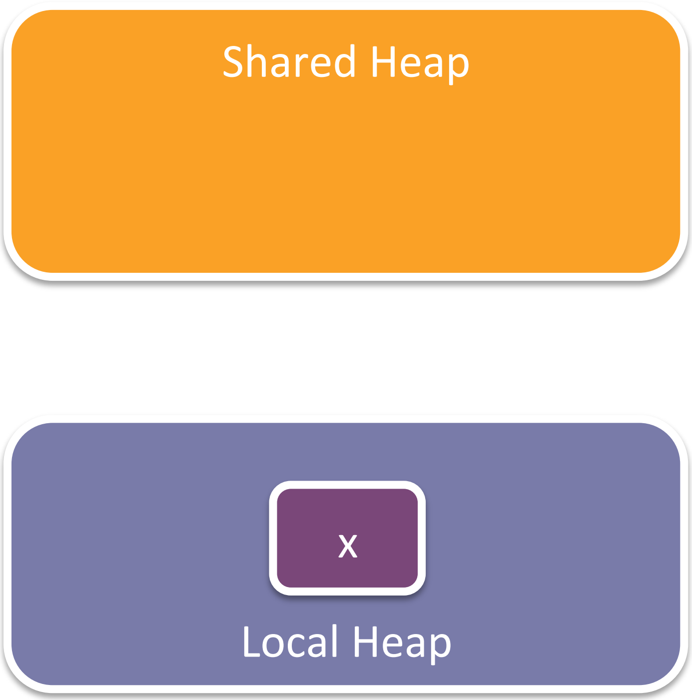
r := x

inSharedHeap (r)

inLocalHeap (x)
&&
isClean (x)

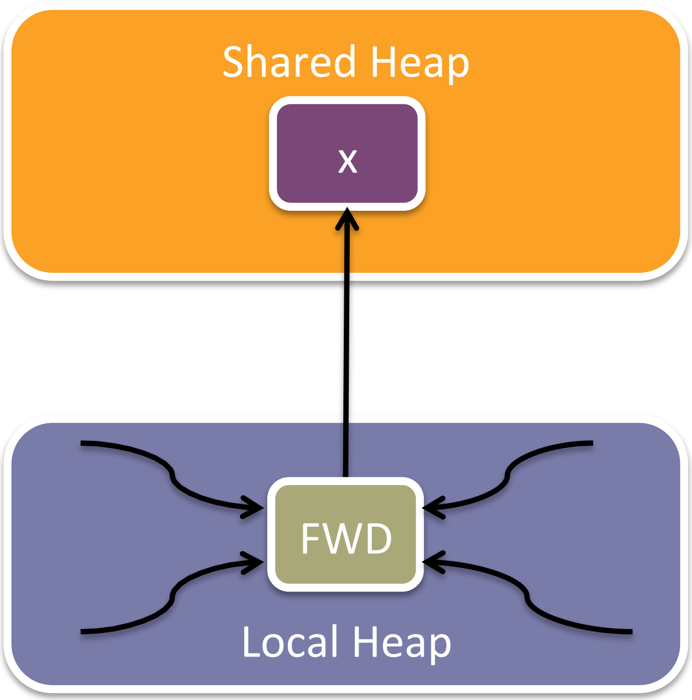
Eager write (no Procrastination)

Cleanliness: Intuition



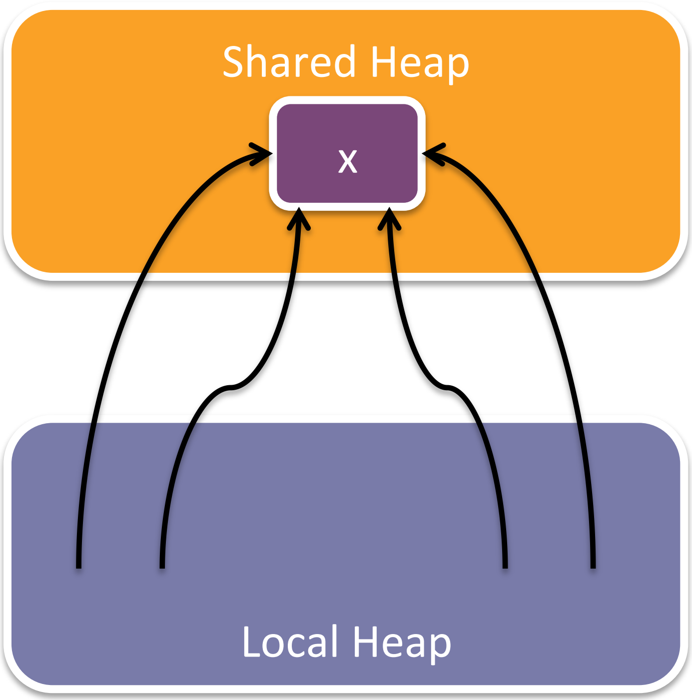
lift (x) to shared heap

Cleanliness: Intuition



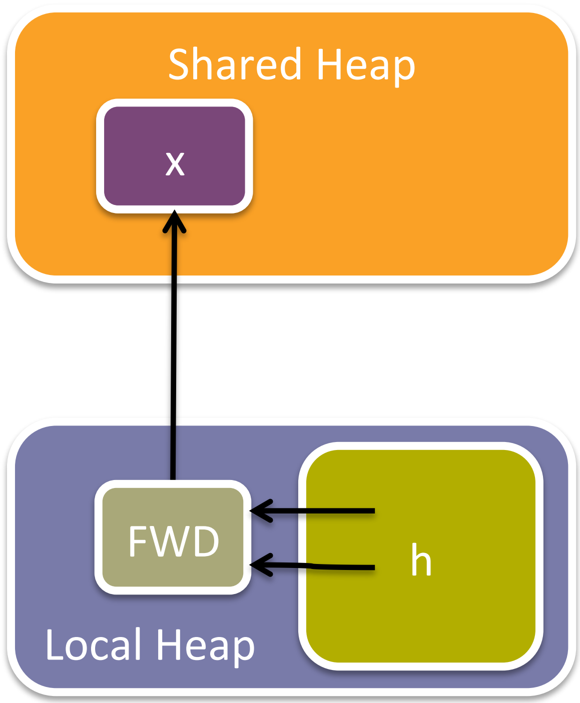
find all references to FWD

Cleanliness: Intuition



Need to scan the entire local heap

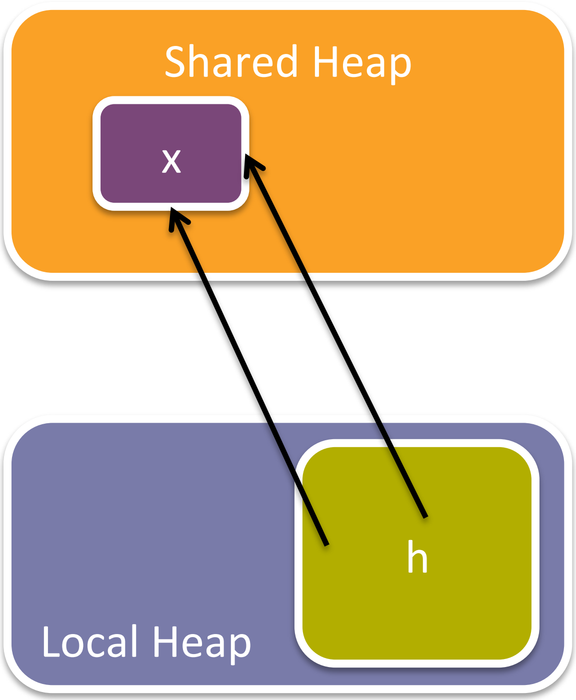
Cleanliness: Simpler question



sizeof (h) << sizeof (local heap)

Do all references originate from heap region *h?*

Cleanliness: Simpler question



sizeof (h) << sizeof (local heap)

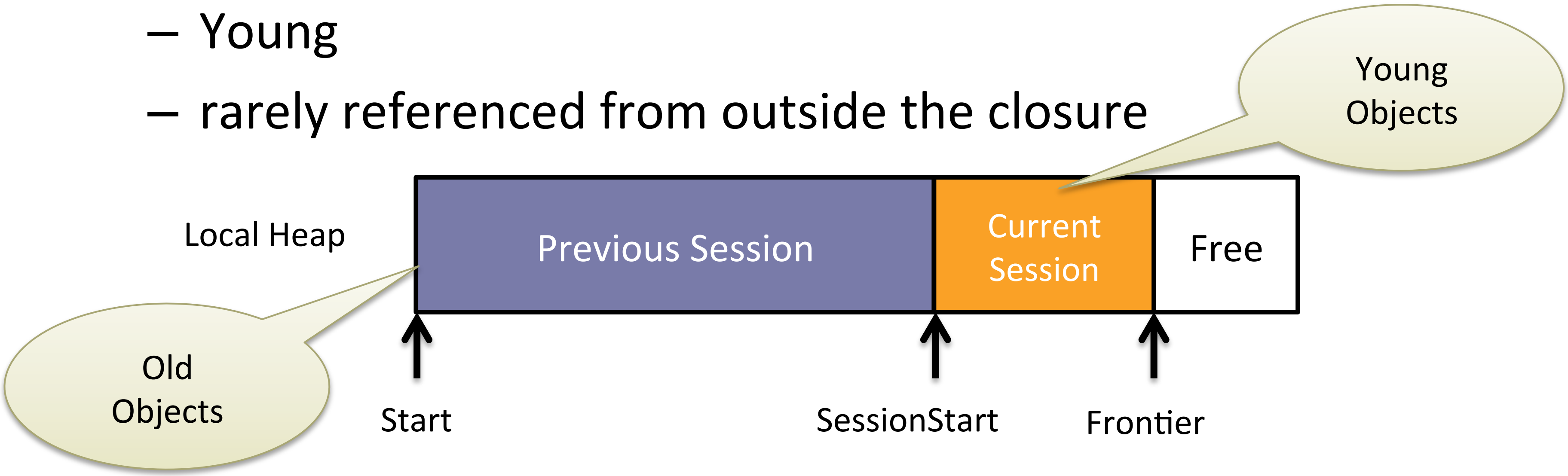
Only scan the heap region *h.*

Heap session!

PURDUE UNIVERSITY

Heap Sessions

- Source of an exporting write is often
  - Young
  - rarely referenced from outside the closure



- Current session closed & new session opened
  - After an exporting write, a user-level context switch, a local GC

PURDUE UNIVERSITY

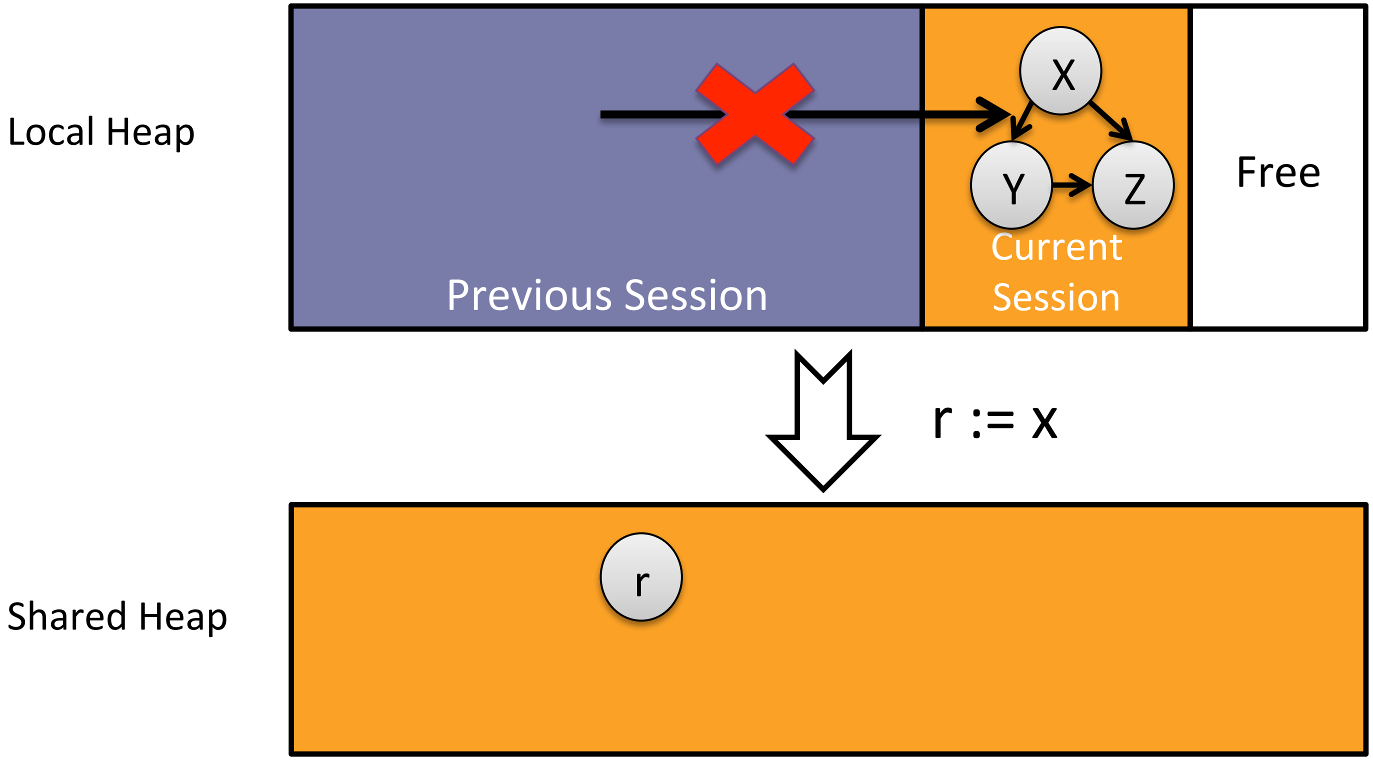
Heap Sessions

- Source of an exporting write is often
  - Young
  - rarely referenced from outside the closure

- Current session closed & new session opened
  - After an exporting write, a user-level context switch, a local GC
  - SessionStart is moved to Frontier
- Average current session size < **4KB**

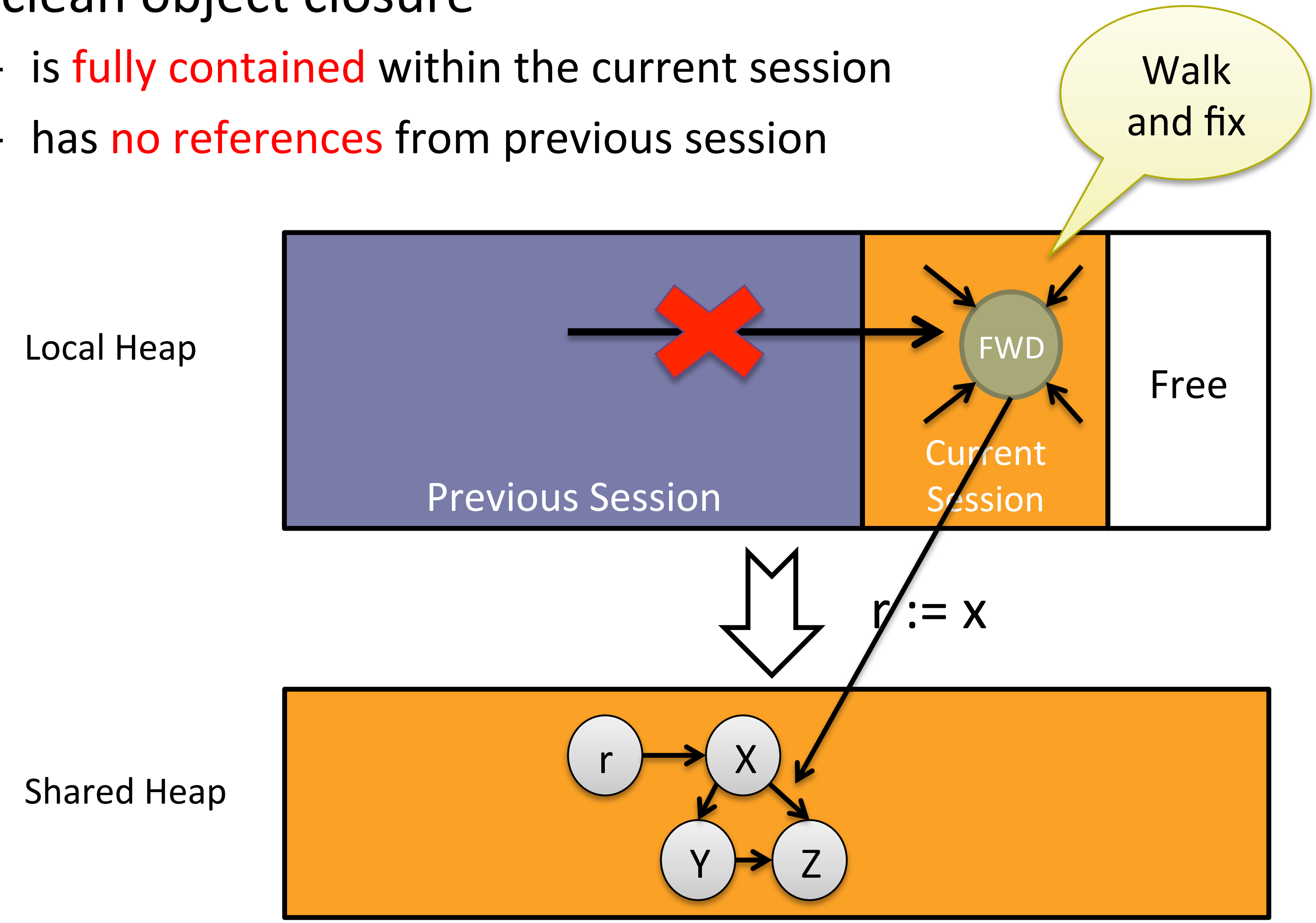
Cleanliness: Eager exporting writes

- A clean object closure
  - is fully contained within the current session
  - has no references from previous session



Cleanliness: Eager exporting writes

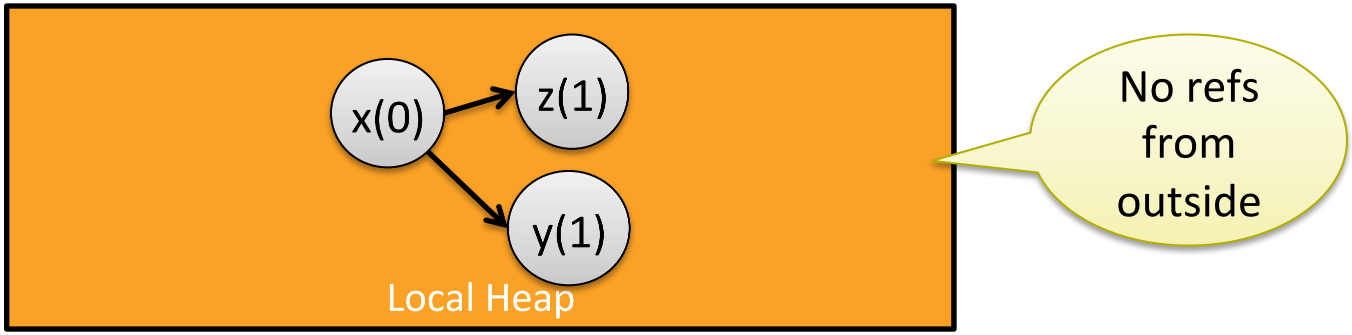
- A clean object closure
  - is fully contained within the current session
  - has no references from previous session



Avoid tracing current session?

- Many SML objects are tree-structured (List, Tree, etc,.)
  - Specialize for no pointers from outside the object closure
- $\forall x' \in$ transitive object closure (x),

  ref_count (x) = 0 && ref_count (x') = 1

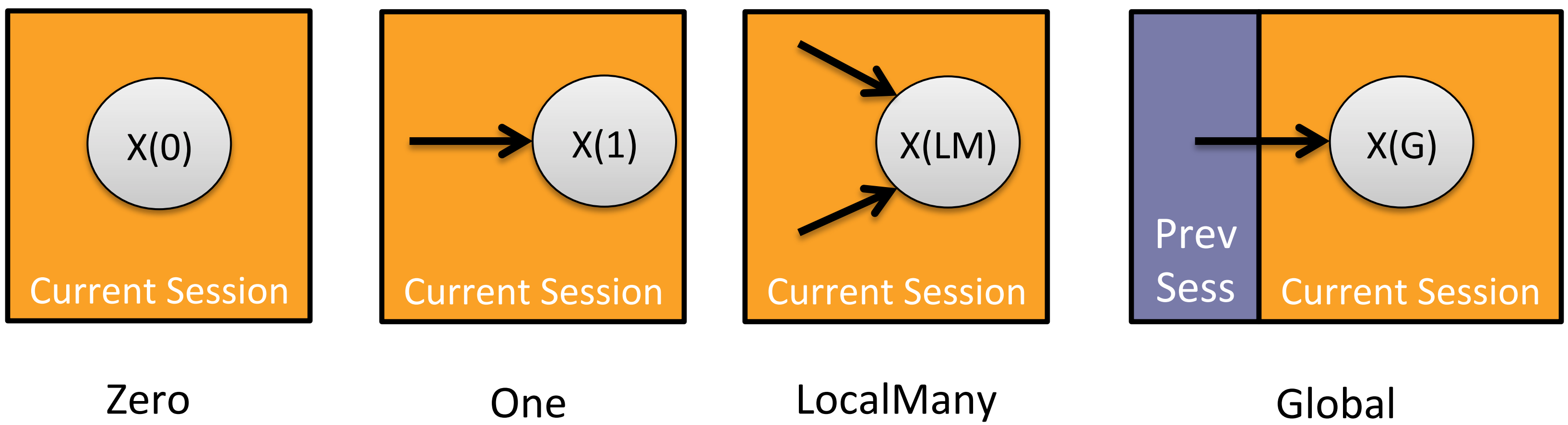


  - ref_count does not consider pointers from stack or registers

- Eager exporting write
  - No current session tracing needed!

Reference Count

- Purpose
  - Track pointers from previous session to current session
  - Identify tree-structured object



- Does not track pointers from stack and registers
  - Reference count only triggered during object initialization and mutation

Bringing it all together

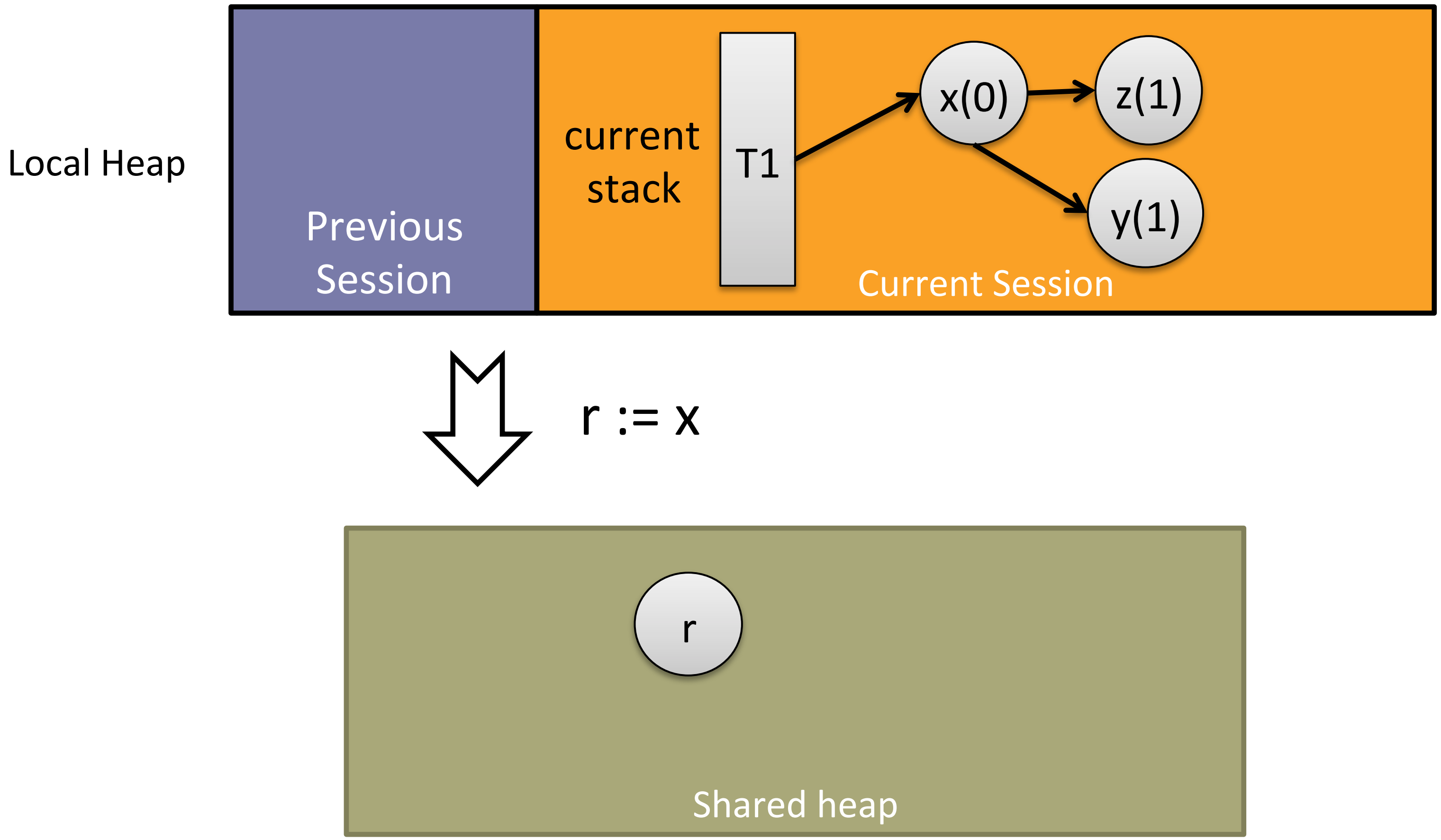
- $\forall x' \in$ transitive object closure (x),

  if **max (ref_count (x'))**

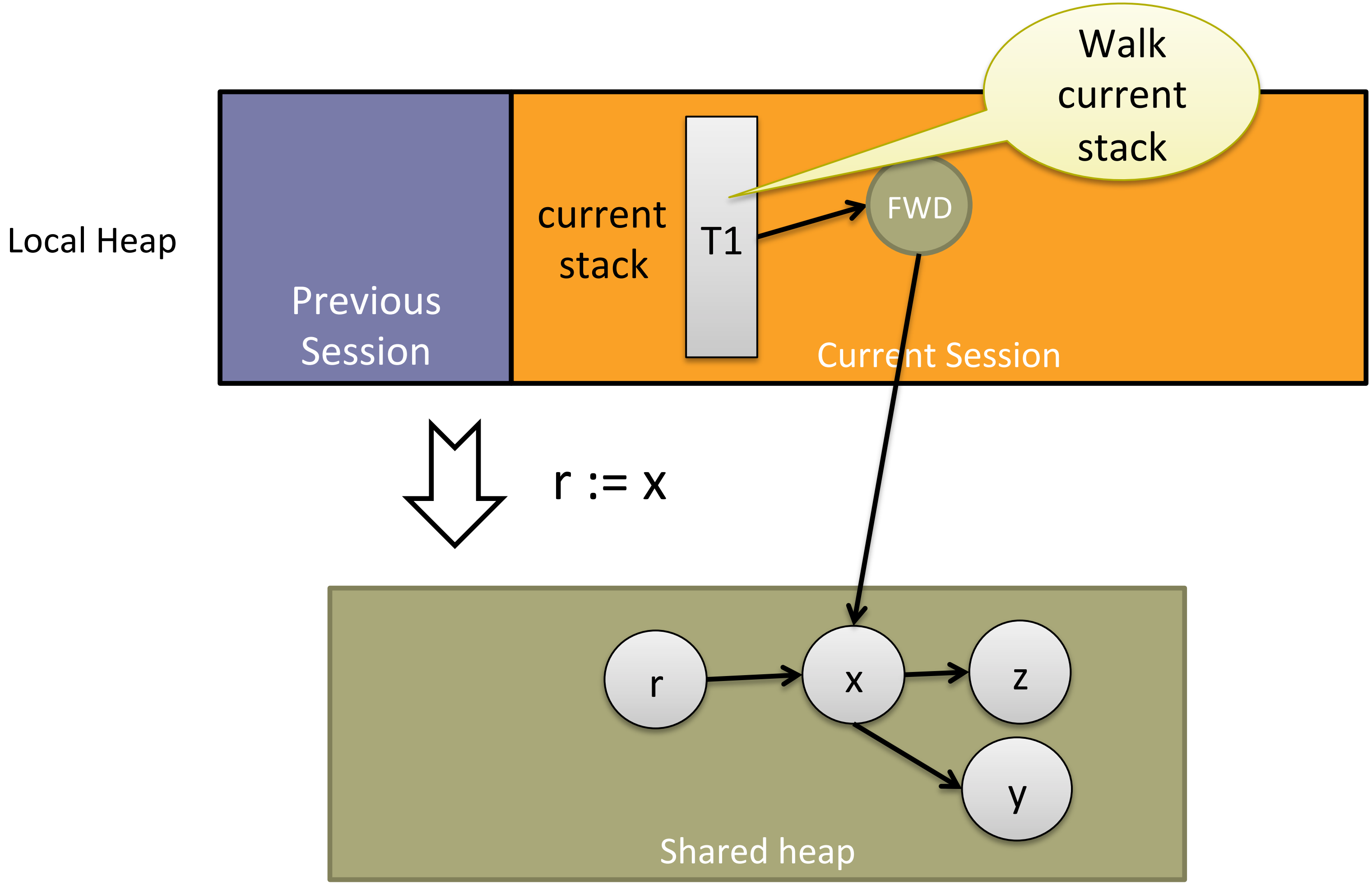
  - One & ref_count (x) = 0 ⇒ tree-structured (Clean) ⇒ *Session tracing not needed*
  - LocalMany ⇒ Clean ⇒ *Trace current session*
  - Global ⇒ 1+ pointer from previous session ⇒ *Procrastinate*

PURDUE UNIVERSITY

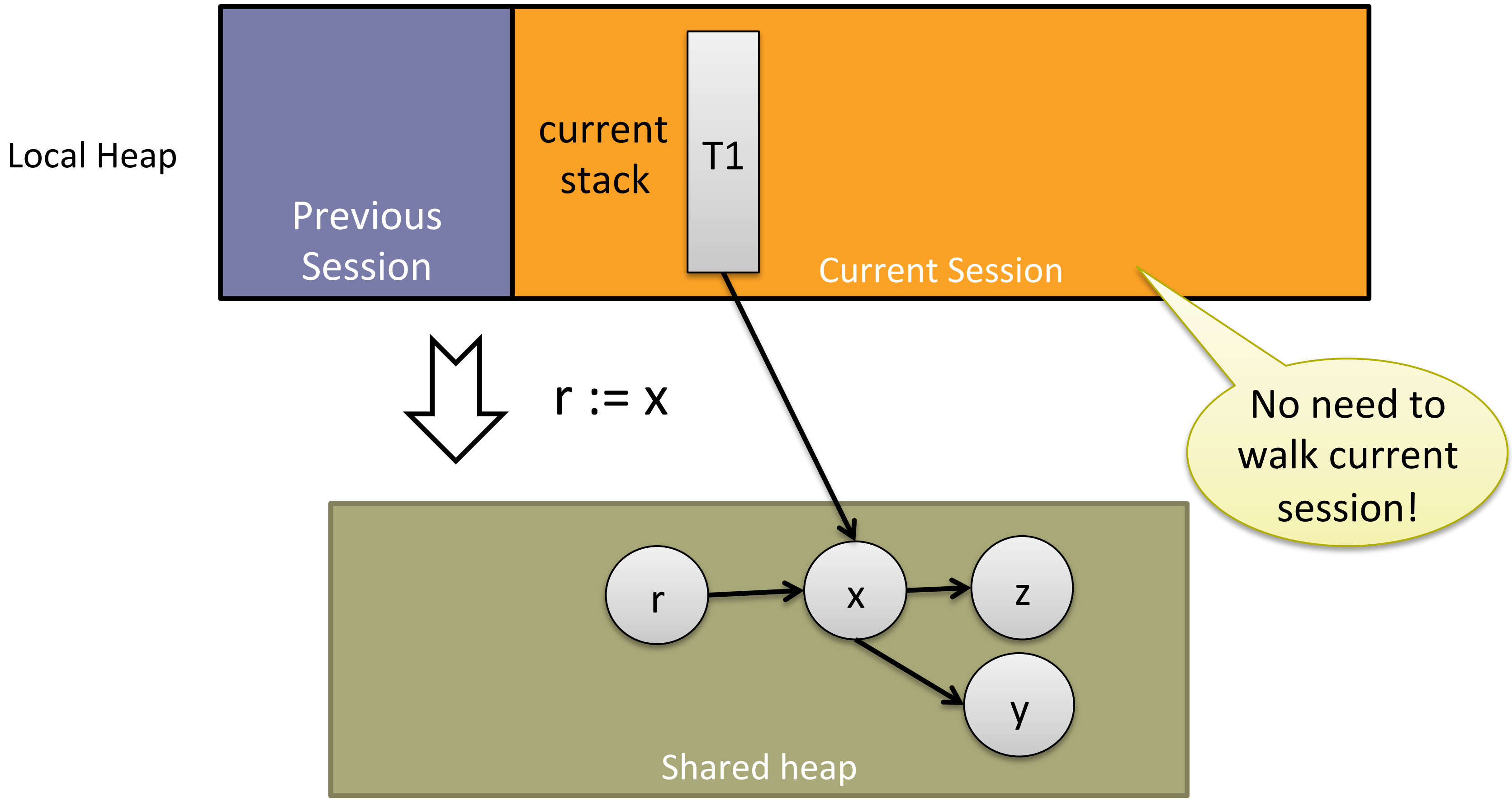
Example 1: Tree-structured Object



Example 1: Tree-structured Object

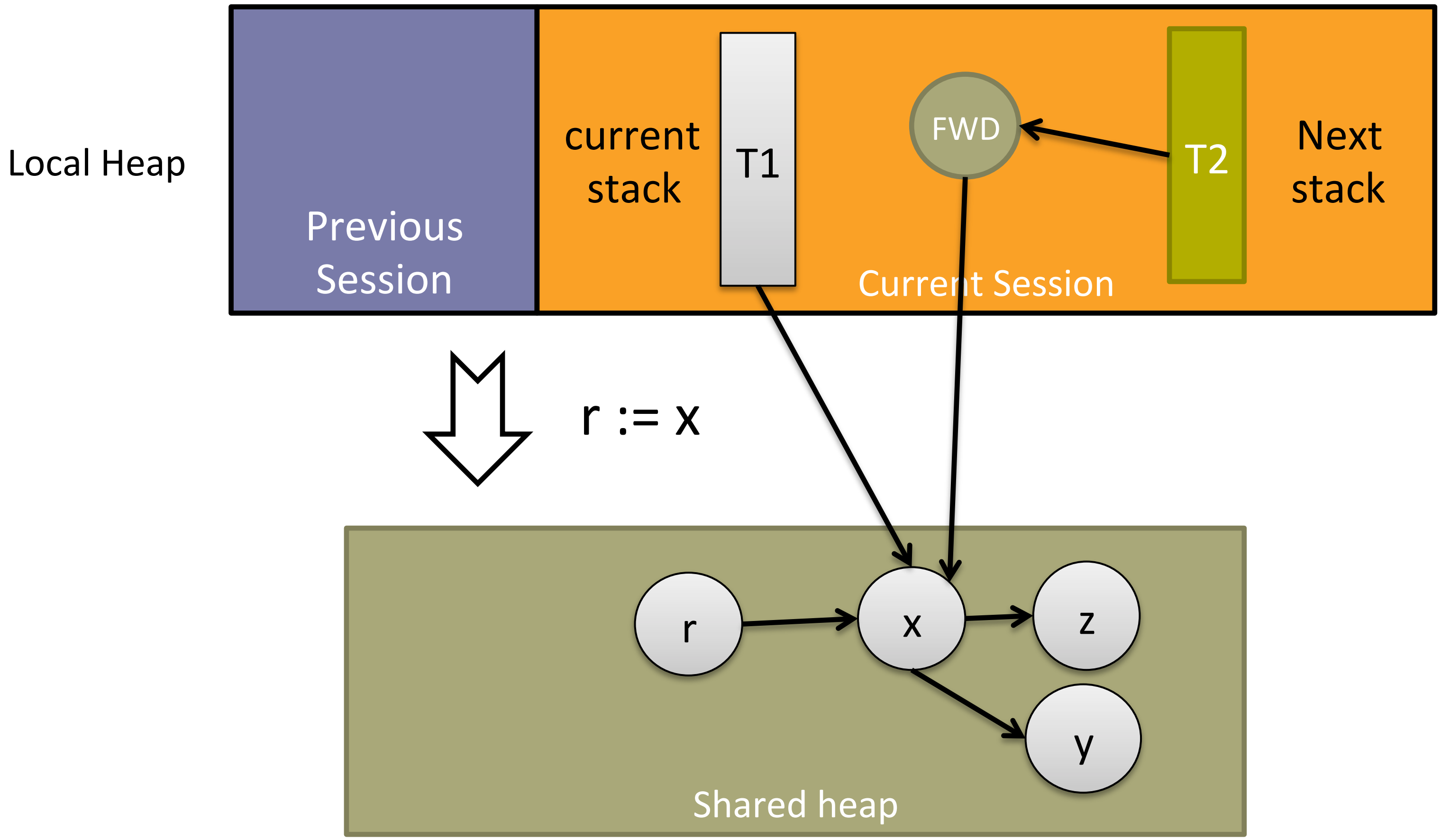


Example 1: Tree-structured Object

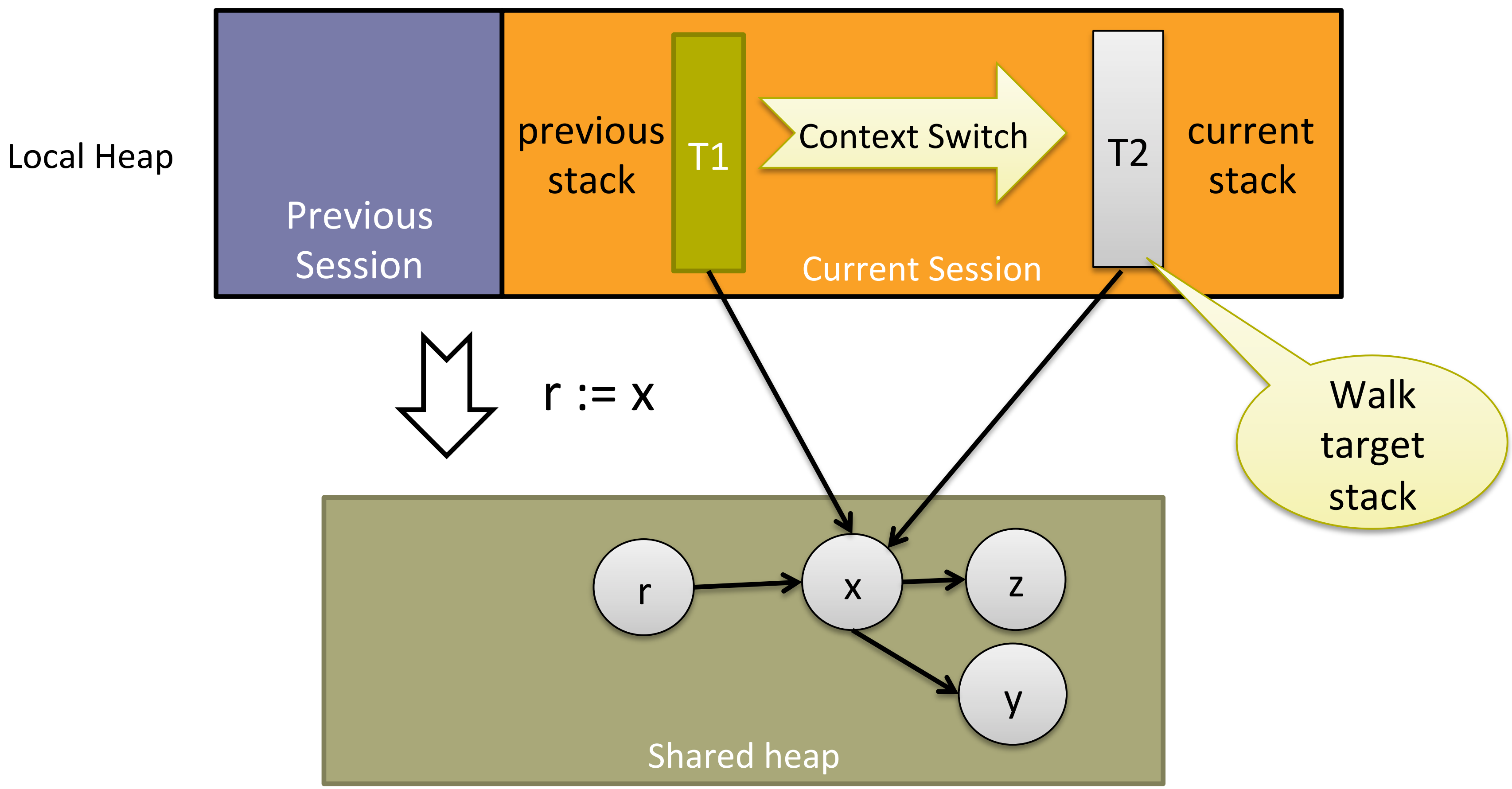


PURDUE UNIVERSITY

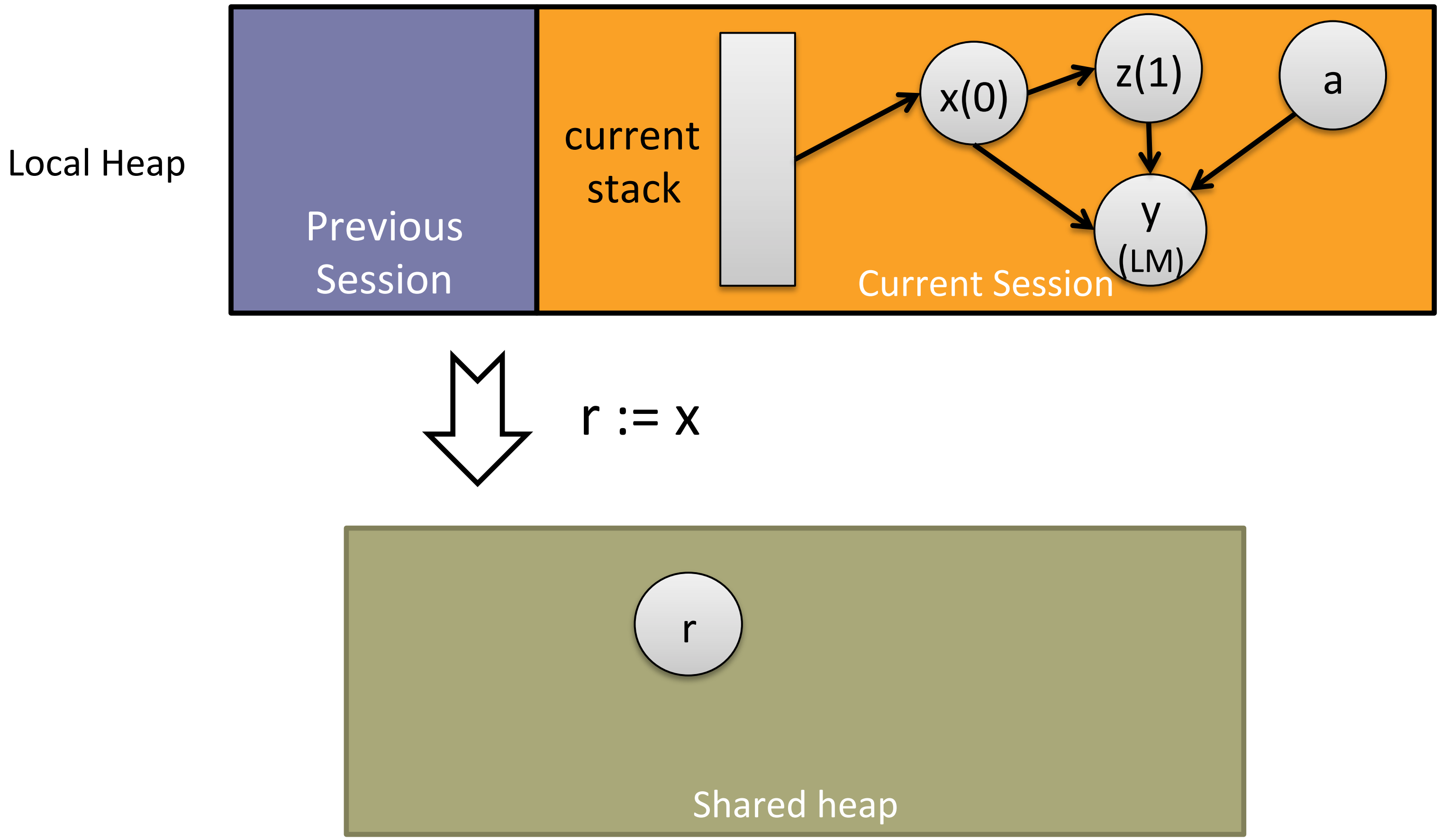
Example 1: Tree-structured Object



Example 1: Tree-structured Object

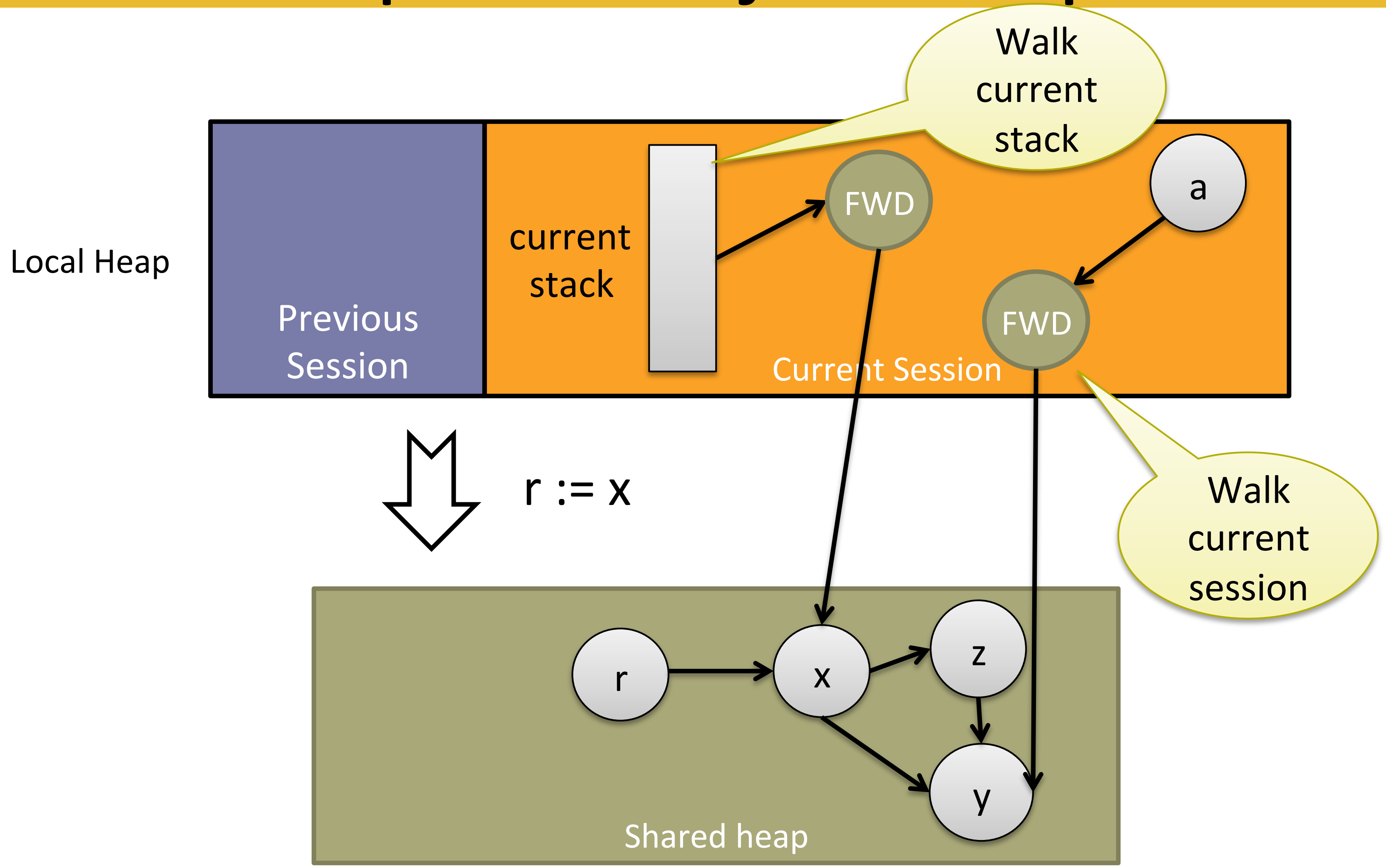


Example 2: Object Graph

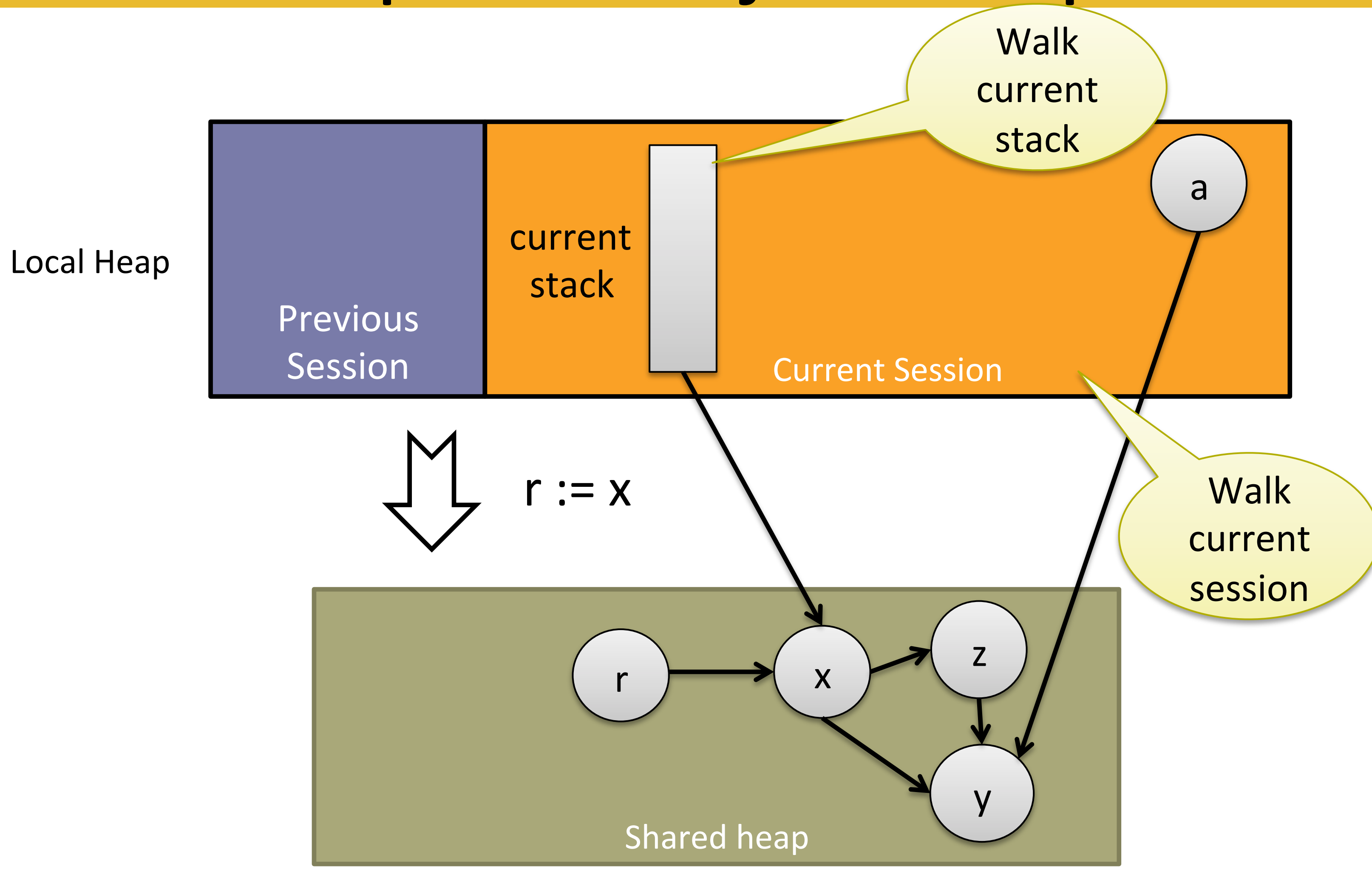


PURDUE UNIVERSITY

Example 2: Object Graph

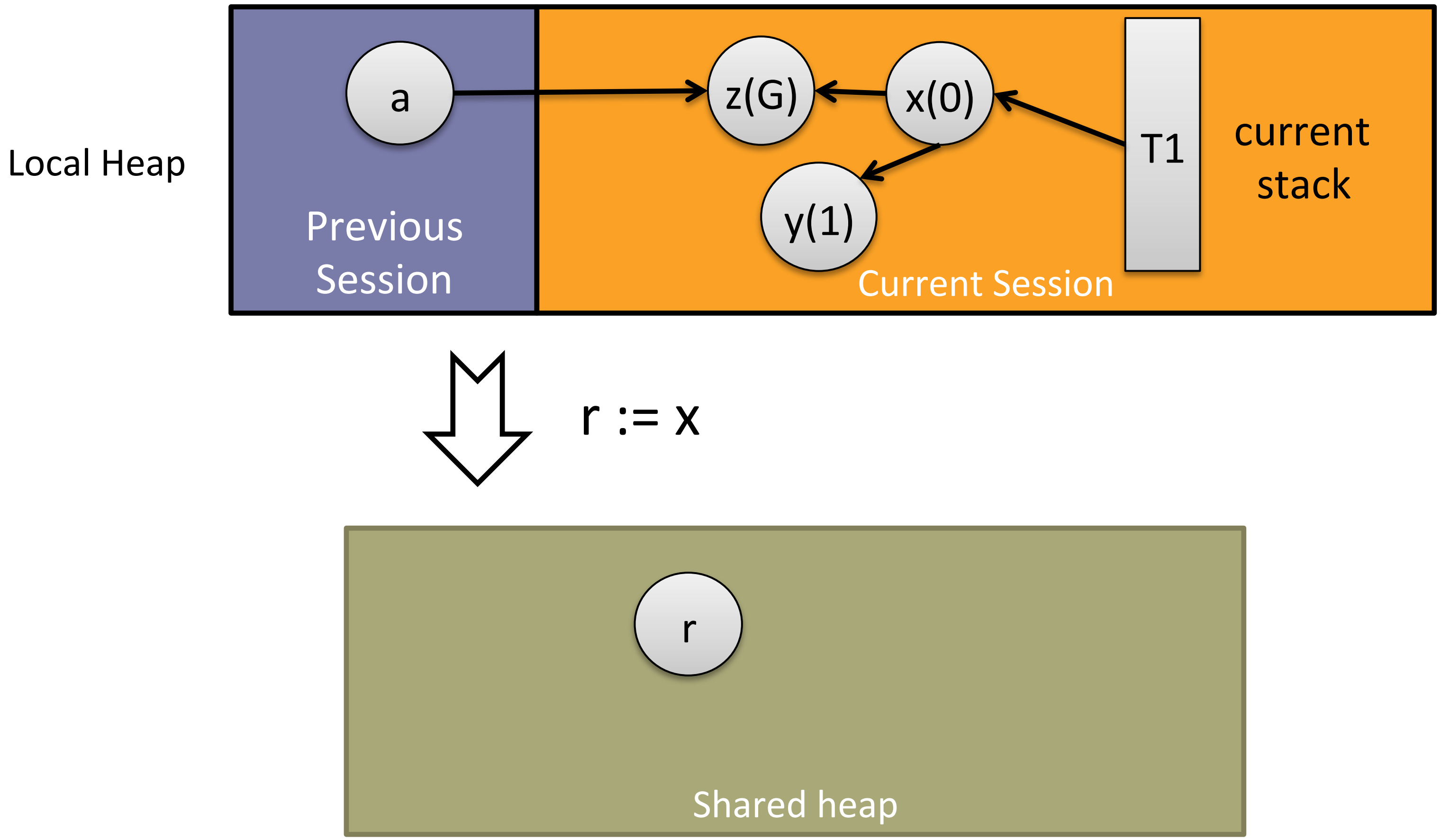


Example 2: Object Graph



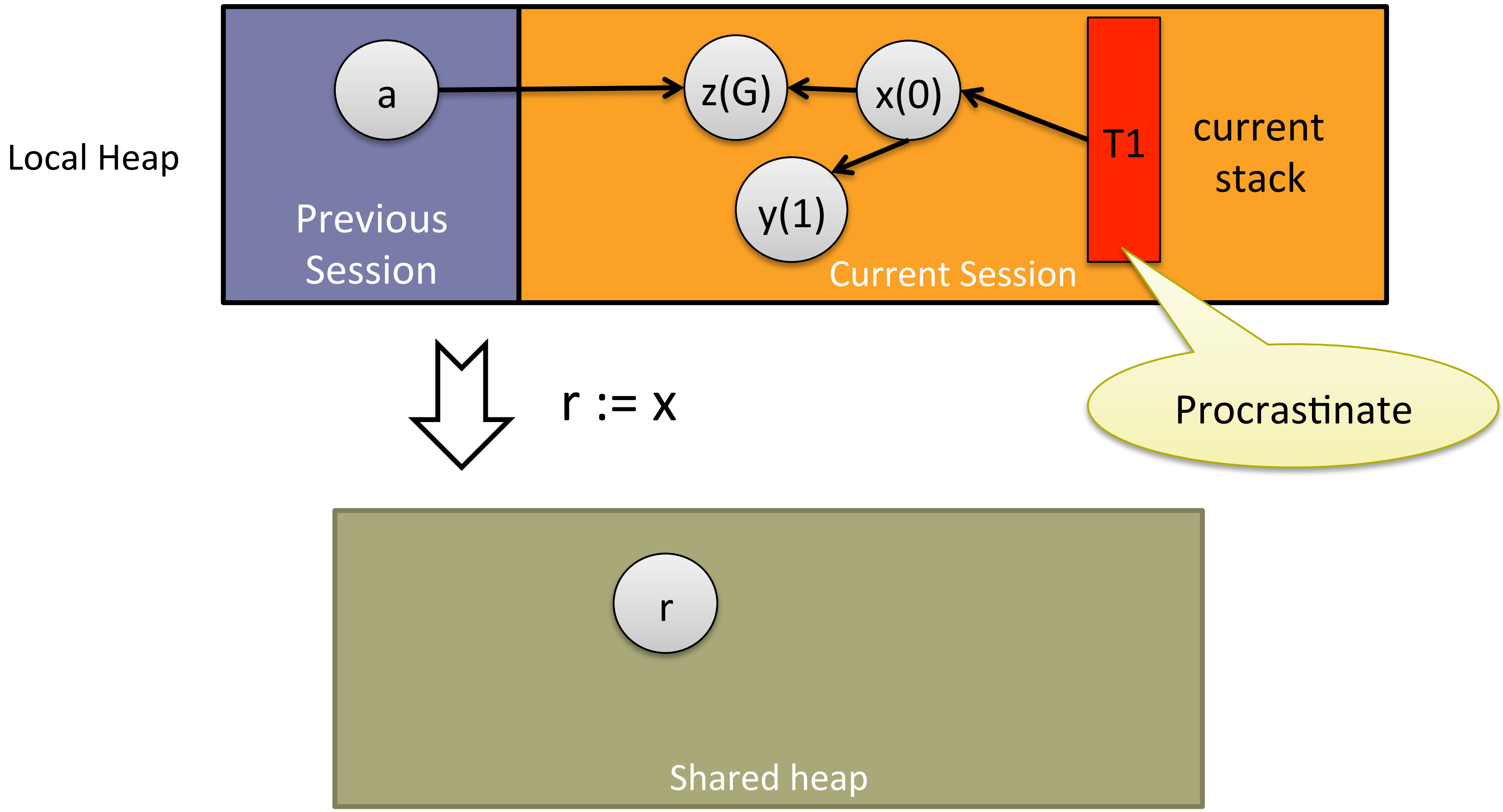
PURDUE UNIVERSITY

Example 3: Global Reference



PURDUE UNIVERSITY

Example 3: Global Reference



PURDUE UNIVERSITY

Immutable Objects

- Specialize exporting writes
- If immutable object in previous session
  - Copy to shared heap
    - Immutable objects in SML do not have *identity*
  - Original object unmodified
- Avoid space leaks
  - Treat large immutable objects as mutable

Cleanliness: Summary

- Cleanliness allows eager exporting writes while preserving visibility invariant
- With Procrastination + Cleanliness, **<1%** of local GCs were *forced*

Evaluation

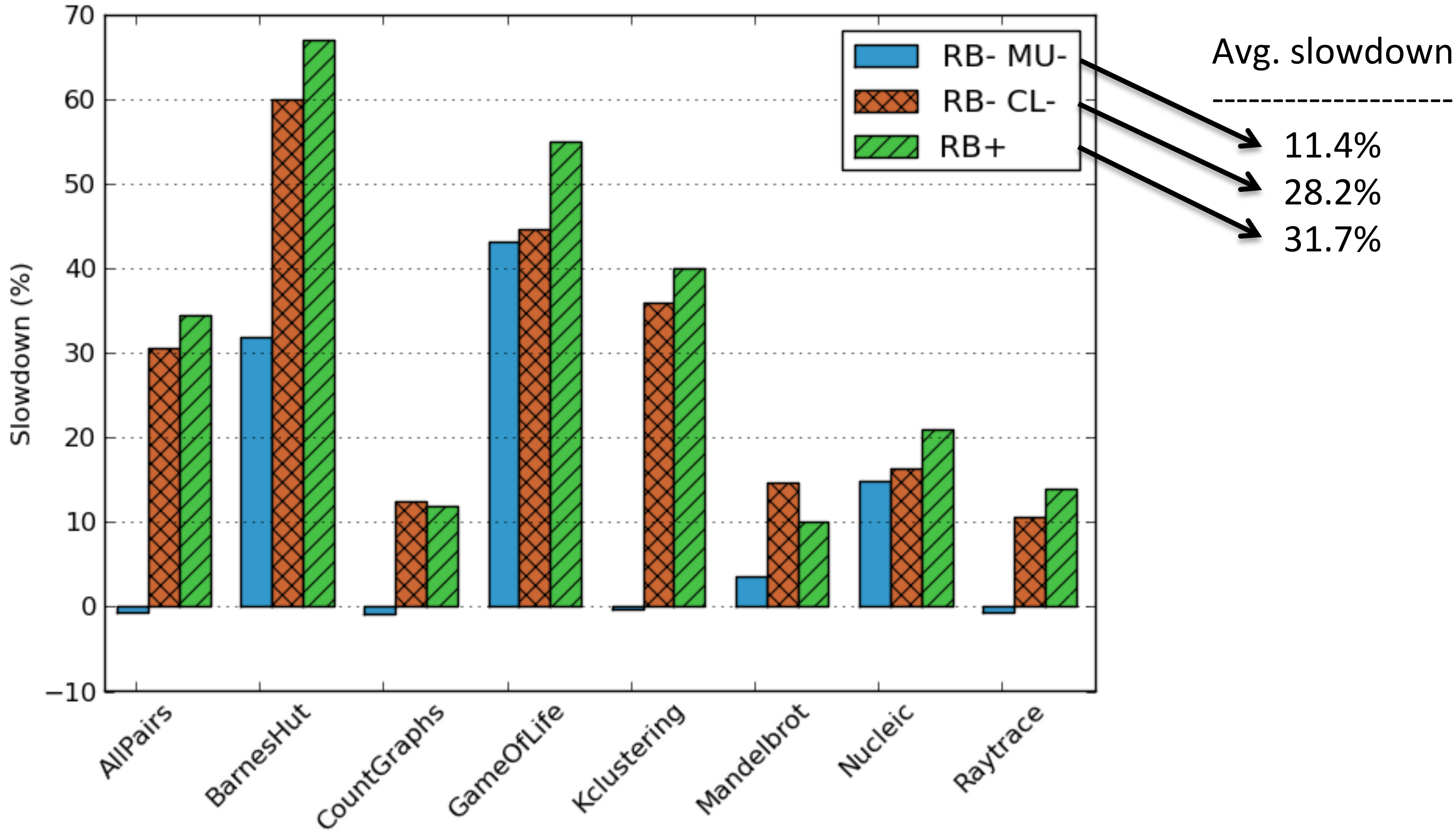
- Variants
  - **RB-** : TLC with Procrastination and Cleanliness
  - **RB+** : TLC with read barriers
- Sansom's dual-mode GC
  - Cheney's 2-space copying collection ←→ Jonker's sliding mark-compacting
  - Generational, 2 generations, No aging
- **Target Architectures**:
  - 16-core AMD Opteron server (NUMA)
  - 48-core Intel SCC (non-cache coherent)
  - 864-core Azul Vega3

Results

- **Speedup:** At 3X min heap size, RB- faster than RB+
  - AMD (16-cores) 32% (2X faster than STW collector)
  - SCC (48-cores) 20%
  - AZUL (864-cores) 30%
- **Concurrency**
  - During exporting write, 8 runnable user-level threads/core!

Cleanliness Impact

- **RB- MU-** : RB- GC ignoring mutability for Cleanliness
- **RB- CL-** : RB- GC ignoring Cleanliness (*Only Procrastination*)



Conclusion

- Eliminate the need for read barriers by preserving the visibility invariant
  - **Procrastination:** Exploit concurrency for delaying exporting writes
  - **Cleanliness:** Exploit generational property for eagerly perform exporting writes
- Additional niceties
  - Completely dynamic → Portable
  - Does not impose any restriction on the GC strategy

PURDUE UNIVERSITY

Questions?

MultiMLton

http://multimlton.cs.purdue.edu

Results

- On AMD, 16 Cores, 3X minimum heap size
- **Mutator time:**
  - STW GC spends the least amount of time in the mutator
    - No read/write barriers
  - Compared to STW GC, the mutator time of
    - RB- 18% more, RB+ 39% more
- **GC time:**
  - RB- spends the least amount time doing GC
  - RB- within 5% of RB+

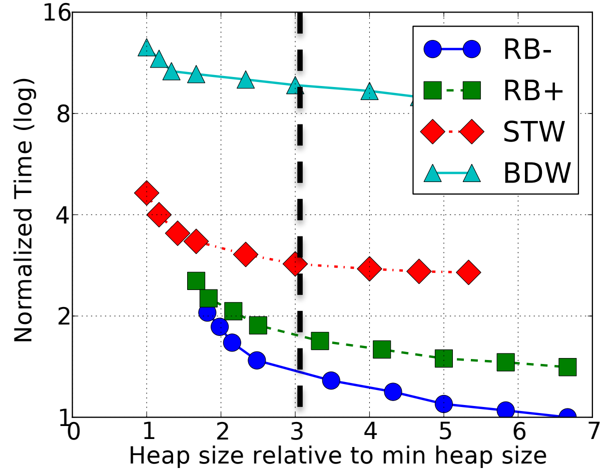
Performance on AMD (16-cores)

At 3X min heap size:

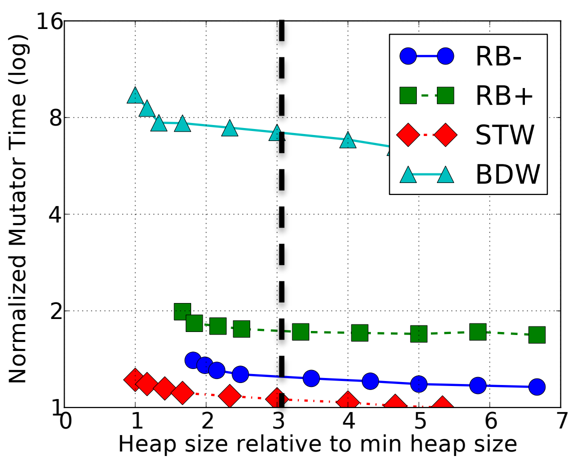
RB+ 32%

STW 106%

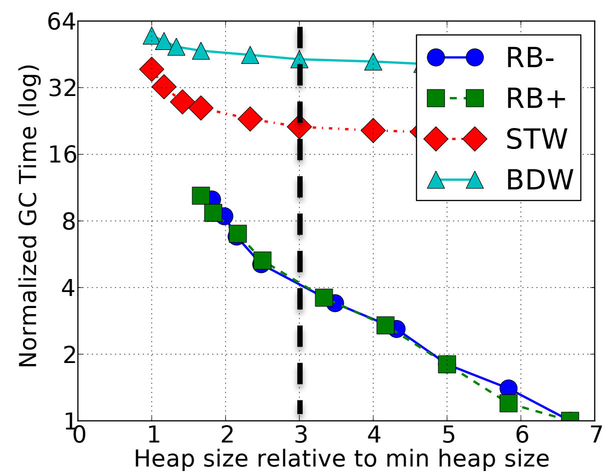
BDW 584%



(a) Total time

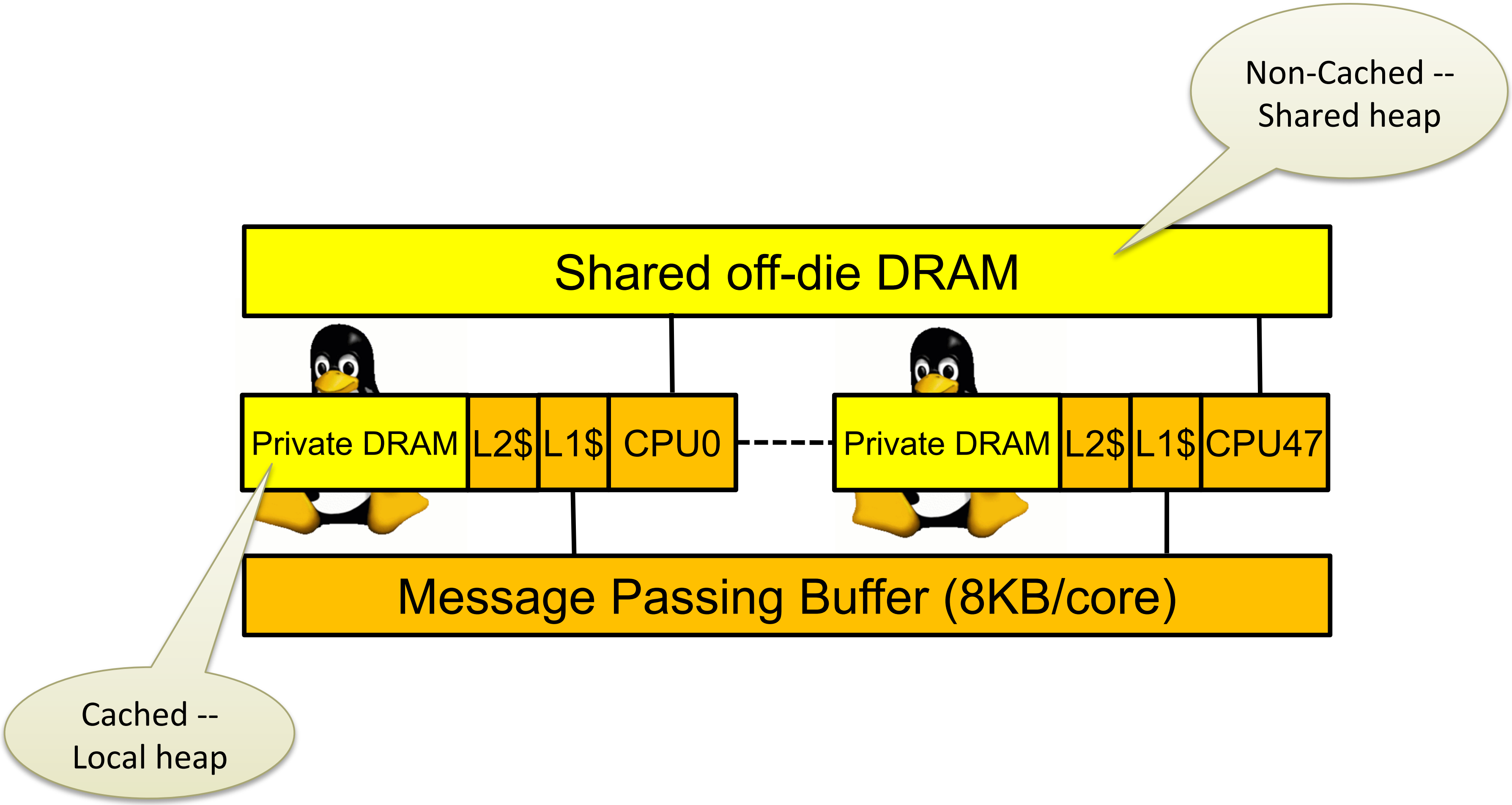


(b) Mutator time

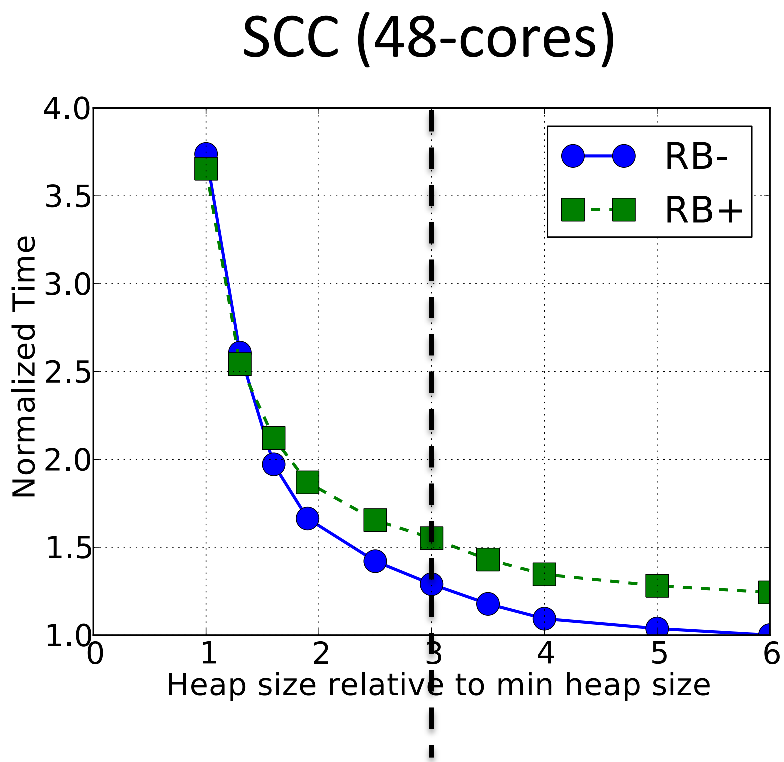


(c) GC time

MultiMLton - SCC implementation

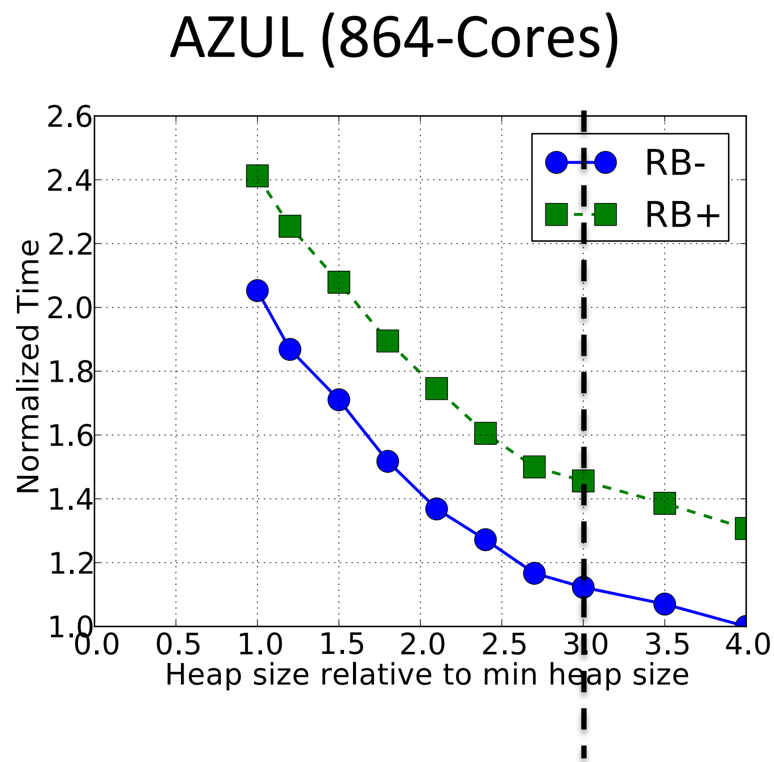


Total time: SCC and AZUL



20%

Non-cache coherent



30%

Scalable, cache-coherent

PURDUE UNIVERSITY

Cleanliness Impact (1)

Benchmark	AllPairs	BarnesHut	CountGraphs	GameOfLife	Kclustering	Mandelbrot	Nucleic	Raytrace
RB-	1831	46532	154	38621	25812	132	156	3523
RB- MU-	1831	4092312	192	735543	50323	209	433092	3743
RB- CL-	124232	67156821	50178	5867423	27023911	25491	912349	61198

Number of Preemptions on exporting writes

Benchmark	AllPairs	BarnesHut	CountGraphs	GameOfLife	Kclustering	Mandelbrot	Nucleic	Raytrace
RB-	0.08	0.17	0	3.54	0	1.43	0	1.72
RB- MU-	0.08	19.2	0.03	9.47	0.02	2.86	9.37	1.72
RB- CL-	38.55	100	0.18	99.75	21.64	86.22	19.3	24.86

Forced GCs as a % of total number of local major GCs

Benchmark Characteristics

Benchmark	Allocation Rate (MB/s)			Bytes Allocated (GB)				# Threads		
	AMD	SCC	AZUL	AMD	SCC	AZUL	% Sh	AMD	SCC	AZUL
AllPairs	817	53	1505	16	16	54	11	256	512	32768
Barneshut	772	70	1382	20	20	876	2	512	1024	32768
Countgraphs	2594	144	4475	24	24	1176	1	128	256	16384
GameOfLife	2445	127	4266	21	21	953	13	256	1024	8192
Kclustering	3643	108	8927	32	32	1265	3	256	1024	8192
Mandelbrot	349	43	669	2	2	32	8	128	512	8192
Nucleic	1430	87	4761	13	14	609	1	64	384	16384
Raytrace	809	54	2133	11	12	663	4	128	256	2048

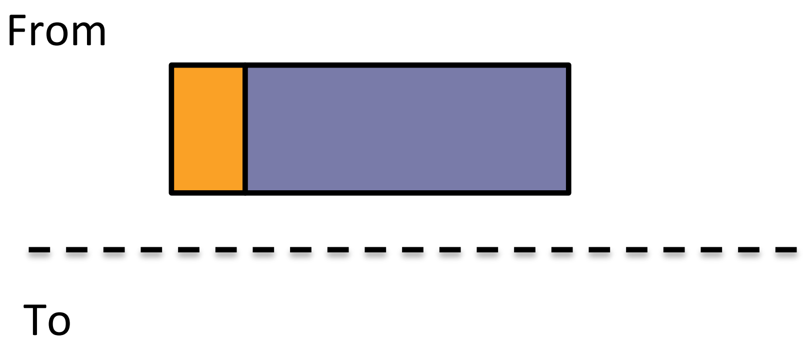
Session Impact

Benchmark	AllPairs	Barneshut	Countgraphs	GameOfLife	Kclustering	Mandelbrot	Nucleic	Raytrace
% LM clean	5.3	13.4	8.6	23.2	17.6	4.5	13.3	8.2
Avg. session size (bytes)	2908	1580	3612	1344	2318	8723	1264	1123

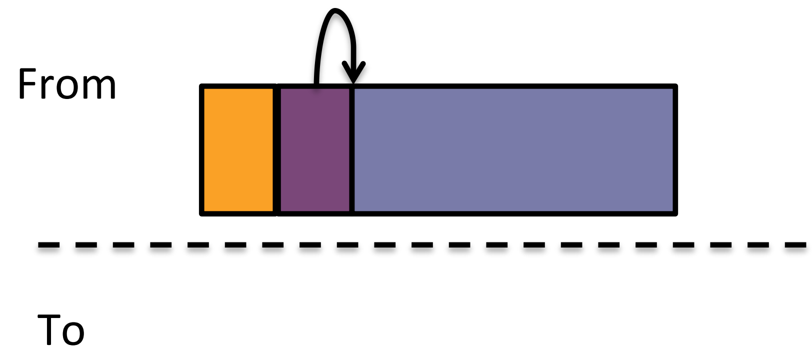
Figure 17: Impact of heap session: % LM clean represents the fraction of instances when a clean object closure has at least one object with `LOCAL_MANY` references.

Read Barrier

Conditional (Baker Style)

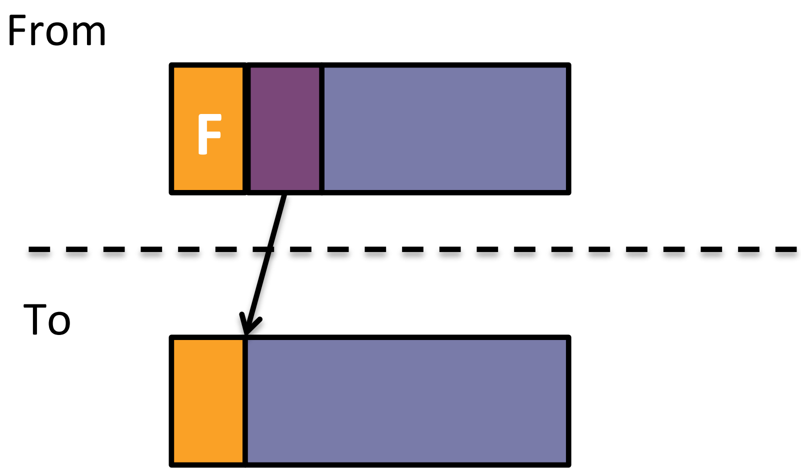


Unconditional (Brooks style)



Read Barrier

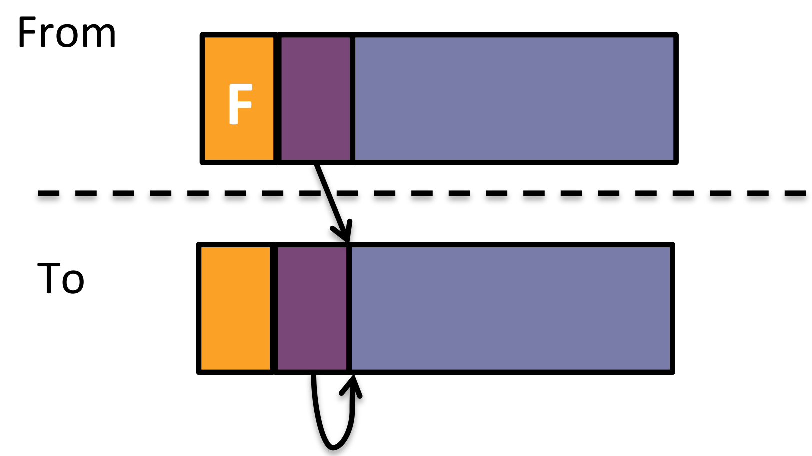
Conditional (Baker Style)




```
pointer readBarrier (pointer *p) {
  if (*(Header*)(p - HD_OFF) == F)
    return *(pointer*)p;
  return p;
}
```

Has Conditional Check

Unconditional (Brooks style)




```
pointer readBarrier (pointer *p) {
  return *(pointer*)(p - IND_OFF);
}
```

Needs extra header word

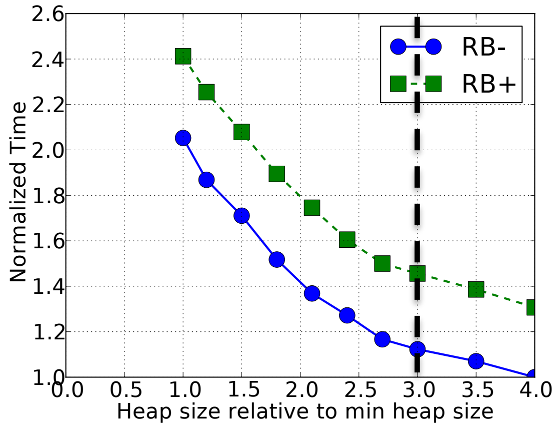
Read Barrier Optimizations

- Stacks and registers never point to forwarding pointers
- “Eager” read barriers (D.Bacon et al. POPL’93)
- Scan stack after exporting write
- Exporting write is a GC safe-point
- Reduces RB overhead by ~5%

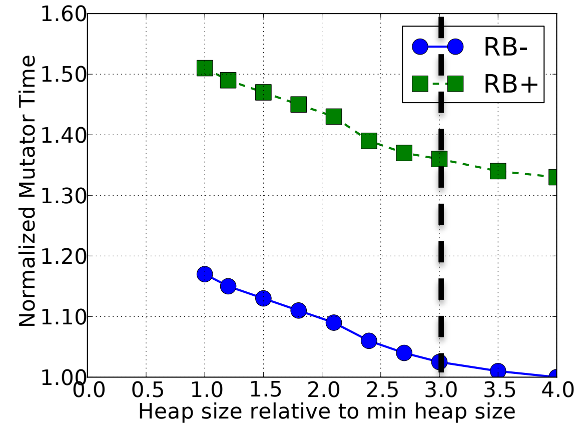
Performance on AZUL

At 3X min heap size:

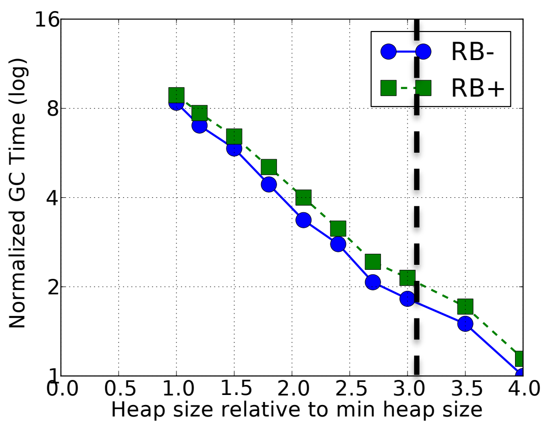
RB+ 30%



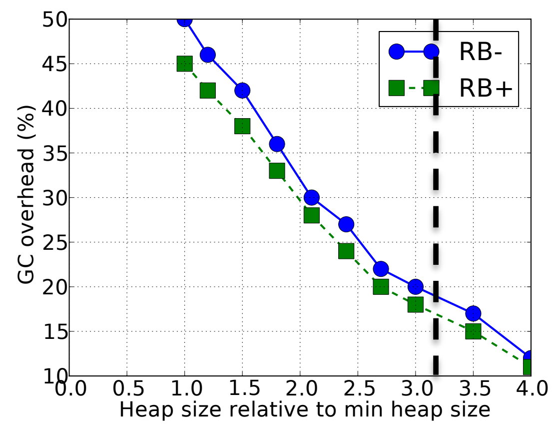
(a) Total time



(b) Mutator time



(c) Garbage collection time



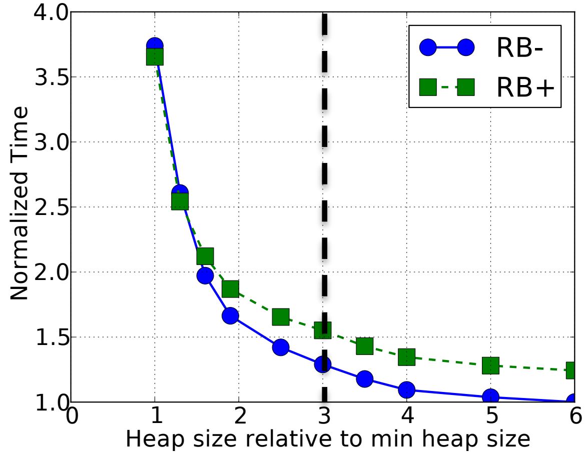
(d) Garbage collection overhead

PURDUE UNIVERSITY

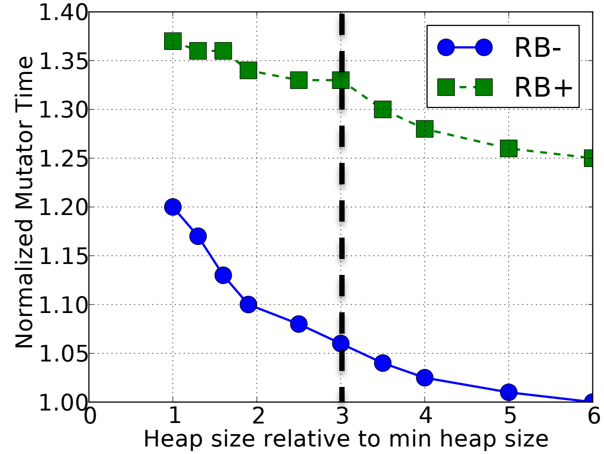
Performance on SCC

At 3X min heap size:

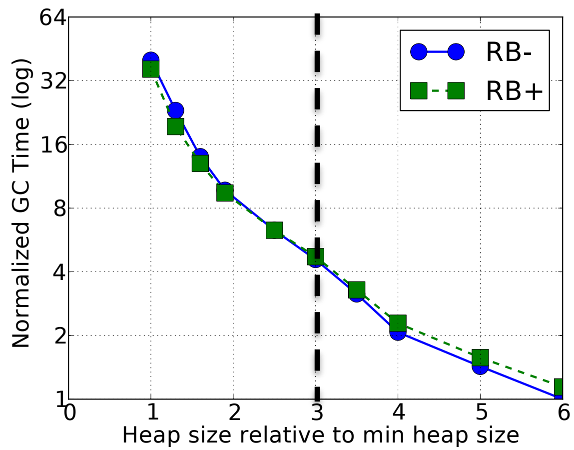
RB+ 20%



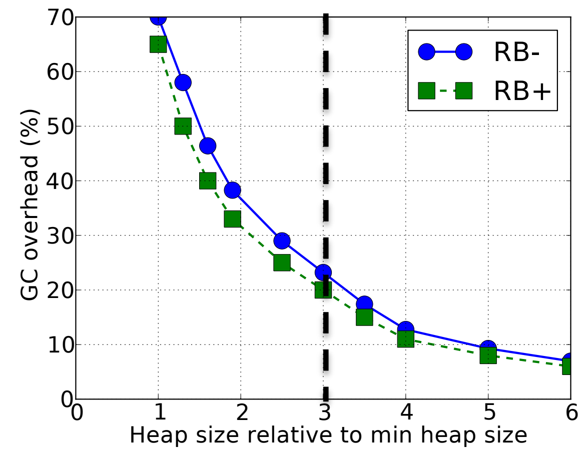
(a) Total time



(b) Mutator time

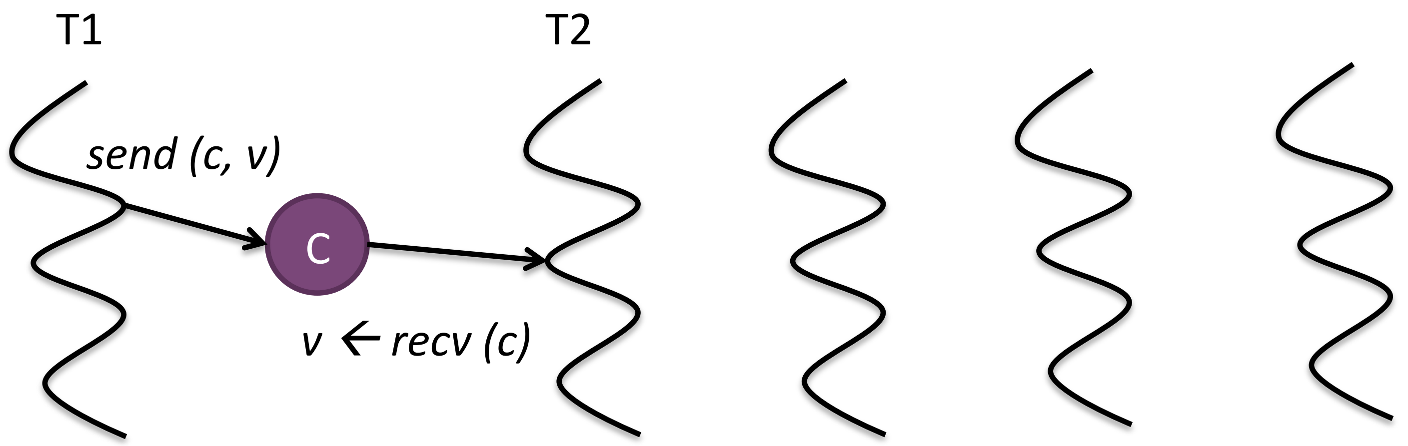


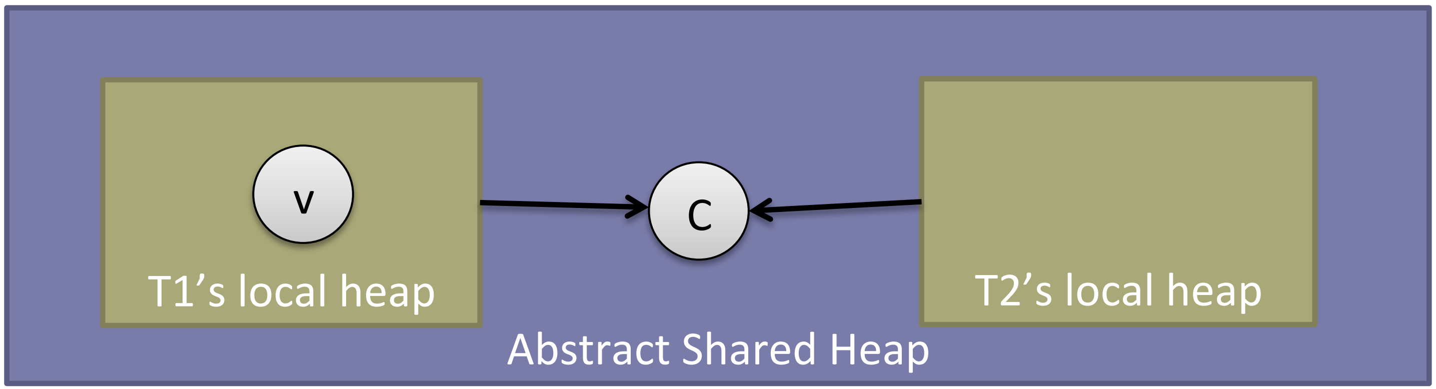
(c) Garbage collection time



(d) Garbage collection overhead

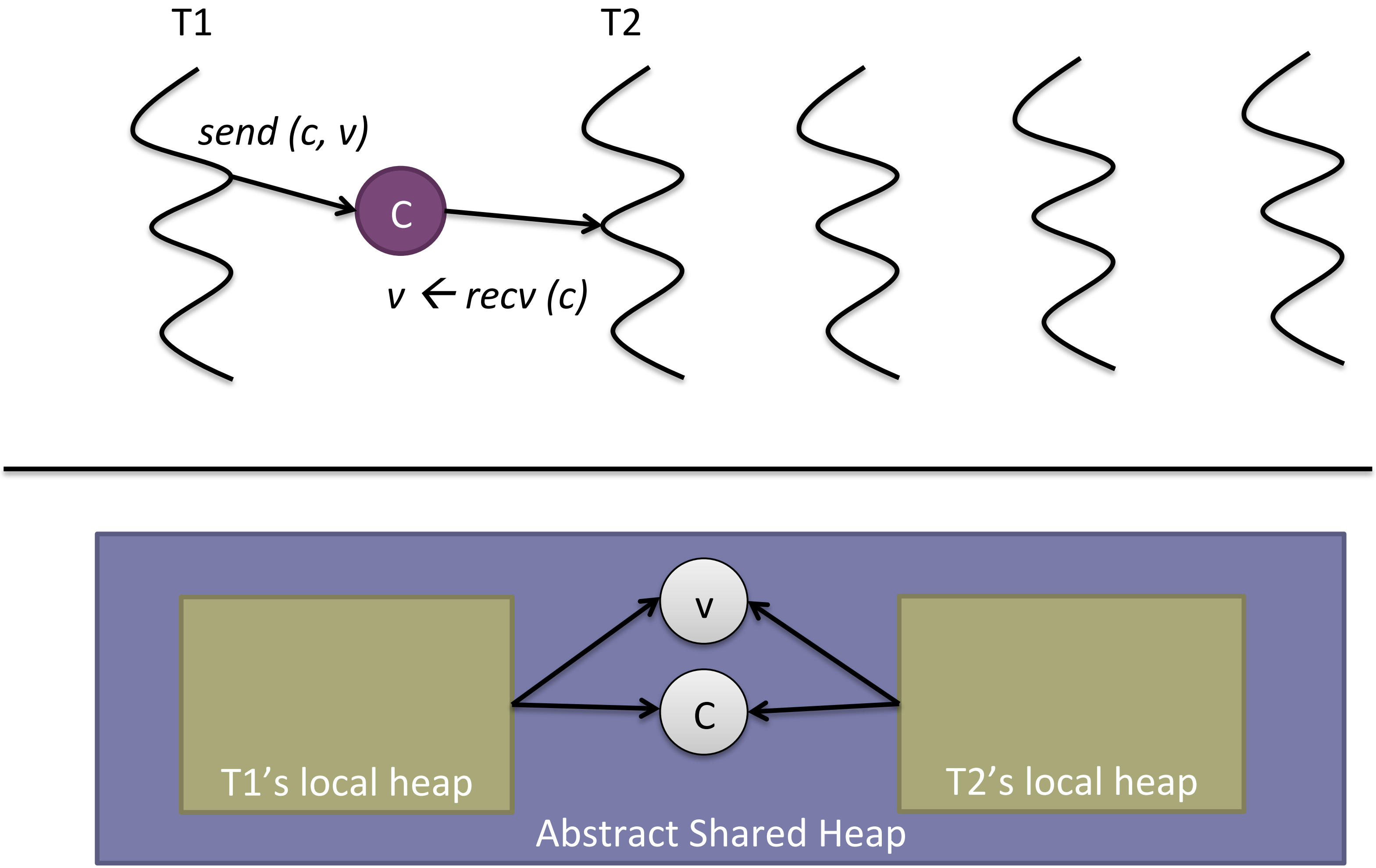
Under the hood





Before Communication

Under the hood



After Communication

PURDUE UNIVERSITY