Harvesting Memory-bound CPU Stall Cycles in Software with MSH

Zhihong Luo, Sam Son, and Sylvia Ratnasamy, *UC Berkeley*

Scott Shenker, *UC Berkeley & ICSI*

Presented by **Luofan Chen** and Jiyang Wang

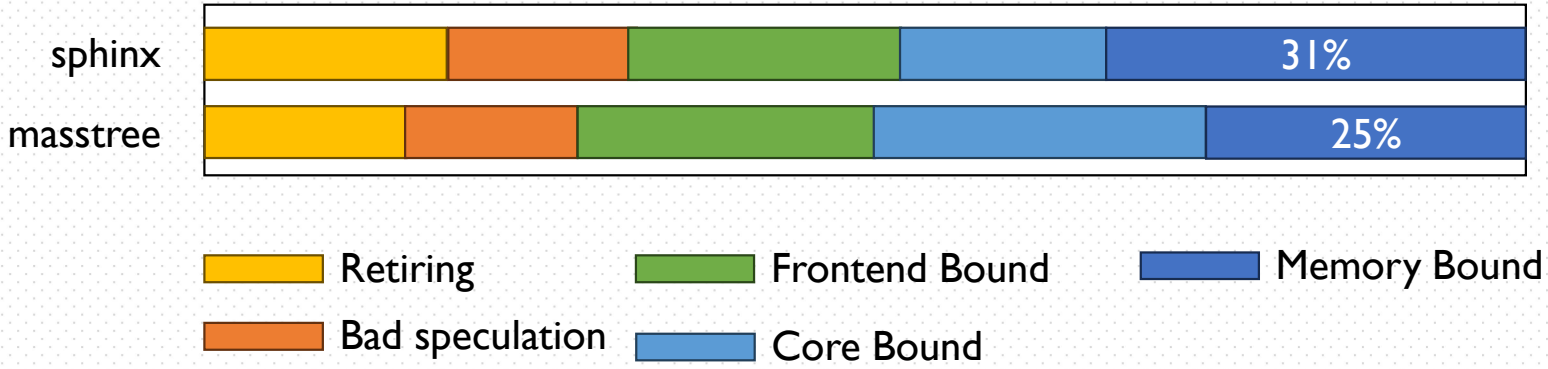
2024–10-22

Memory-bound stalls

- Cores waiting for memory access to finish
  - Example: Cache misses

Memory-bound stalls

- Cores waiting for memory access to finish
  - Example: Cache misses
- Memory-bound stalls happens *frequently* in datacenter workloads
  - A top-down analysis on sphinx and masstree



Reducing / Harvesting memory stalls

- **Prefetching: An async operation to fetch data to cache**
  - **Stream prefetchers & prefetch insertion via static analysis**
    - Hardware implemented in CPU
    - `-fprefetch-loop-arrays` by GCC

Prefetch next array element for later access

```
for (size_t i = 0; i < size; i++) {
    _mm_prefetch((char*)&arr[i + 8], _MM_HINT_T0);
    arr[i] *= 2.0;
}
```

Memory access in a ***stream*** manner

☹ **Have limited capability , unable to handle complex access patterns**

Reducing / Harvesting memory stalls

- **Prefetching: An async operation to fetch data to cache**
  - **Stream prefetchers & prefetch insertion via static analysis**
    - Hardware implemented in CPU
    - `-fprefetch-loop-arrays` by GCC

☹ **Have limited capability , unable to handle complex access patterns**

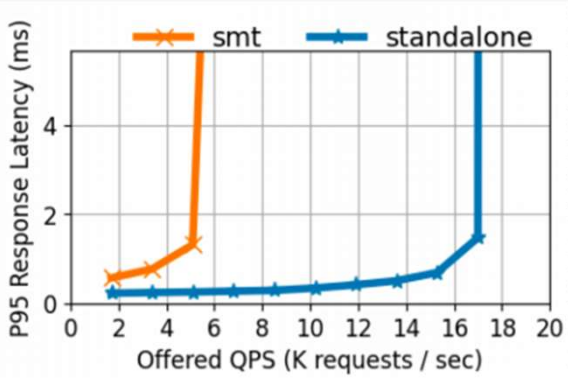
- **Runahead prefetchers**
  - Processor ***speculatively*** pre-process instructions during memory access stalls
  - Developers manually modify source code to ***emulate*** hardware behavior

☹ **Hardware complexity, source code modification…**

☹ **Both require prefetching end *sufficiently ahead* of load instruction**

Reducing / Harvesting memory stalls

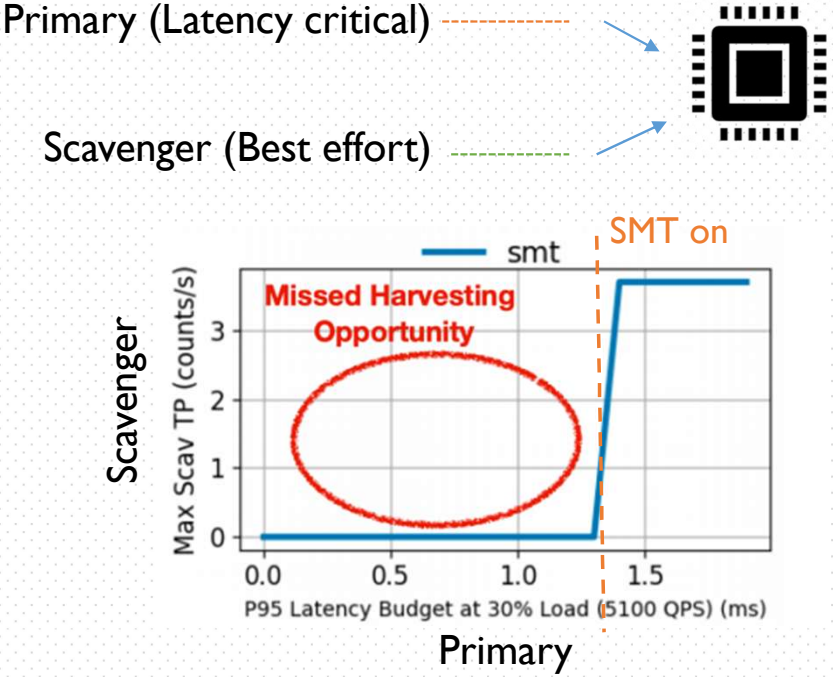
- Prefetching: An async operation to fetch data to cache
- **SMT (simultaneous multithreading)**
  - **Latency overhead**
    - **Can largely increase the primary latency**



Reducing / Harvesting memory stalls

- Prefetching: An async operation to fetch data to cache

- **SMT (simultaneous multithreading)**
  - **Latency overhead**
    - **Can largely increase the primary latency**
  - **Lack of configurability**
    - **Can only decide to turn it on/off**



Reducing / Harvesting memory stalls

- Prefetching: An async operation to fetch data to cache

- **SMT (simultaneous multithreading)**
  - **Latency overhead**
    - **Can largely increase the primary latency**
  - **Lack of configurability**
    - **Can only decide to turn it on/off**
  - **Incomplete harvesting**
    - **2-wide SMT do not have sufficient concurrency**
    - **When concurrent threads frequently incur stalls**

SMT leads to unsatisfactory harvesting performance

Are there better ways to harvest memory stalls using software?

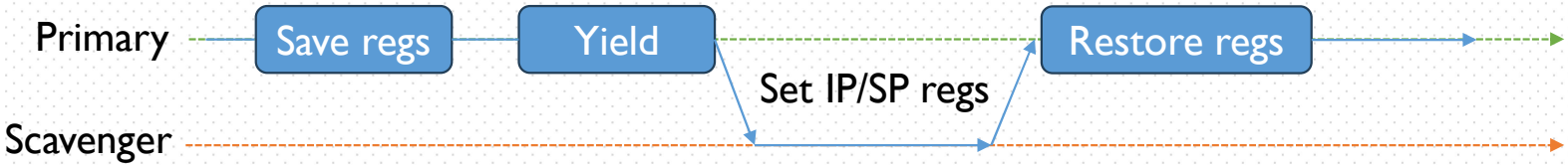
Opportunities

❑Profile-Guided Optimizations

❖**Intel's PEBS and LBR can profile memory stalls with negligible overhead**

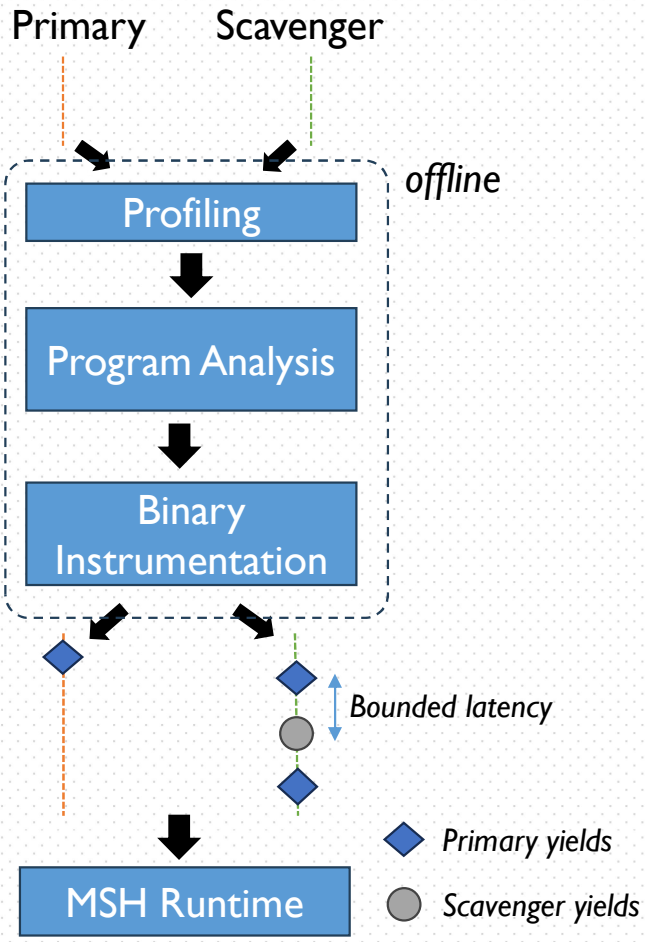
✓ **Transparently detecting memory stalls**

❑Light-weight coroutines



✓ **Lightweight, no system call, no change to VM mappings**

MSH overview



Merits of SMT

- **Transparent**
  - No rewriting efforts, applicable to any code
- **Efficient**
  - Efficiently utilize stall cycles for scavengers

- **Latency-aware**
  - Incur *minimal* latency overhead
  - Control over primary latency and scavenger throughput
- **Full harvesting**
  - Fully harvest stalls by interleaving scavenger executions

Overcome SMT's drawbacks

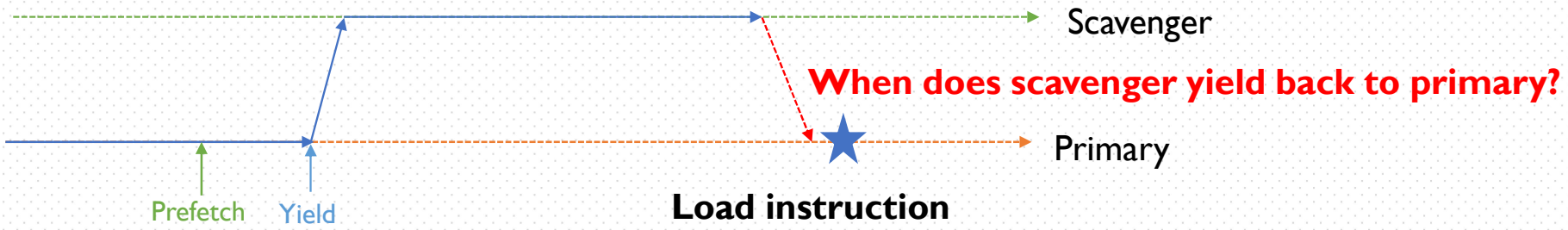
Yielding on primary

❑ **Identify possible yield locations**

❖ **load instructions with:**

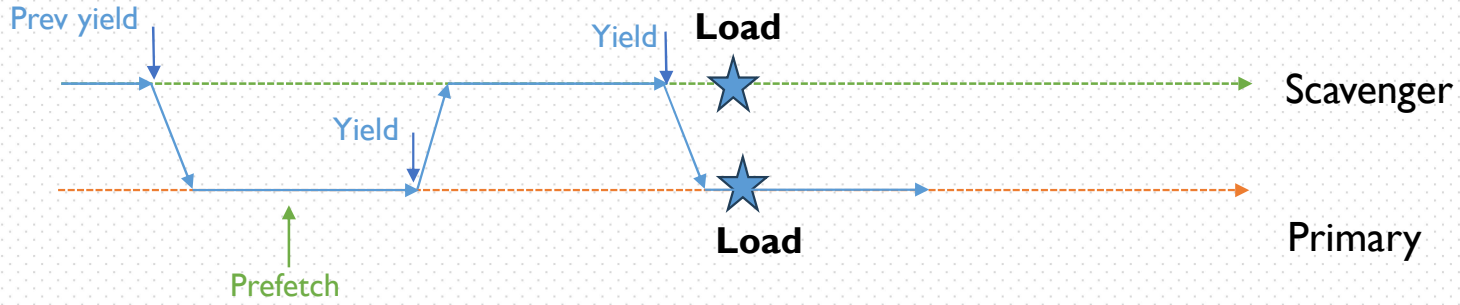
- ➢ **Significant portion of memory stalls** ➡ **Substantial stall cycles**
- ➢ **L3 cache miss likelihood** ➡ **Less impact of primary latency**

➢ Instrument with prefetch & yield ***before*** selected load instructions



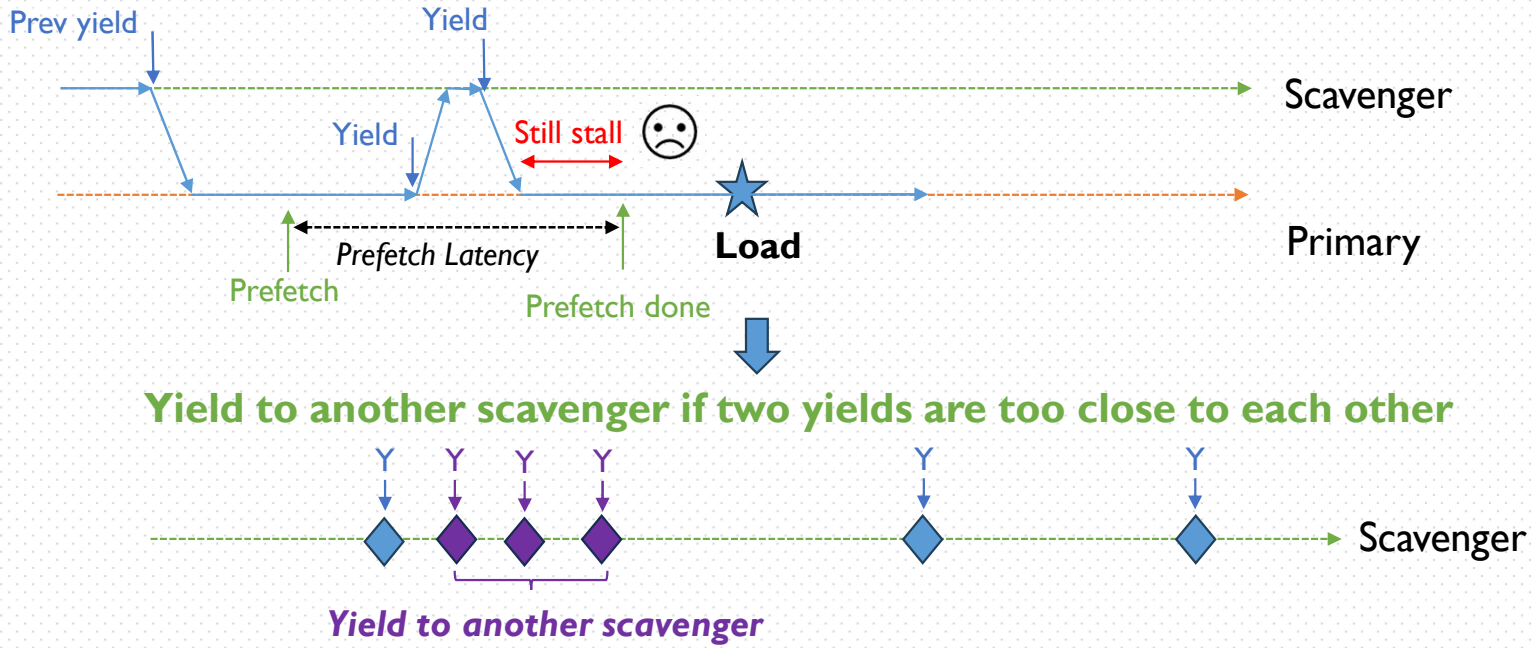
Yielding on scavenger

- **Primary yields: Identified the same way as primary**
  - ***Normally* go back to primary, to another scavenger if too close**



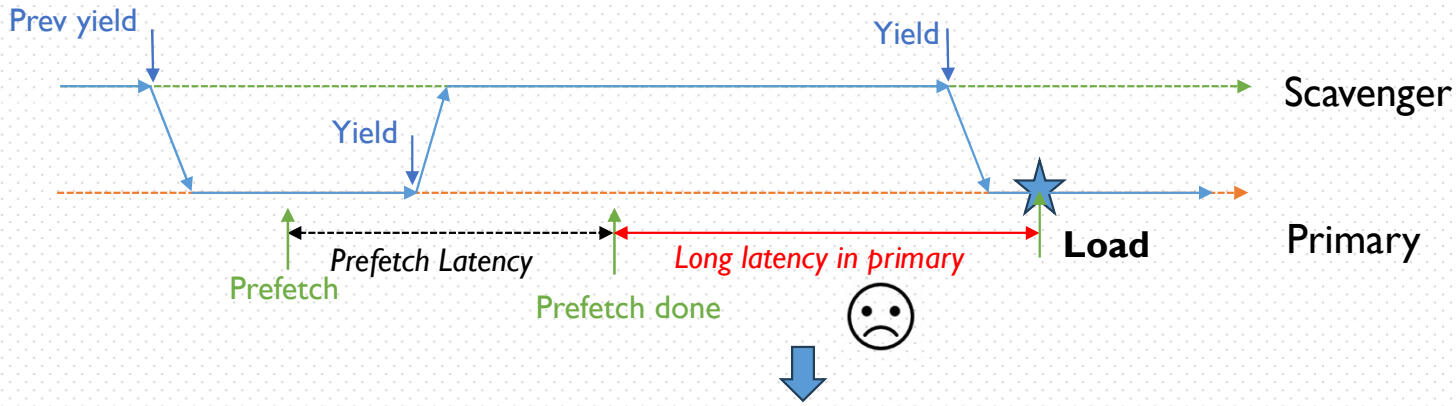
Yielding on scavenger

- **Primary yields: Identified the same way as primary**
  - ***Normally* go back to primary, to another scavenger if too close**



Yielding on scavenger

- **Primary yields: Identified the same way as primary**

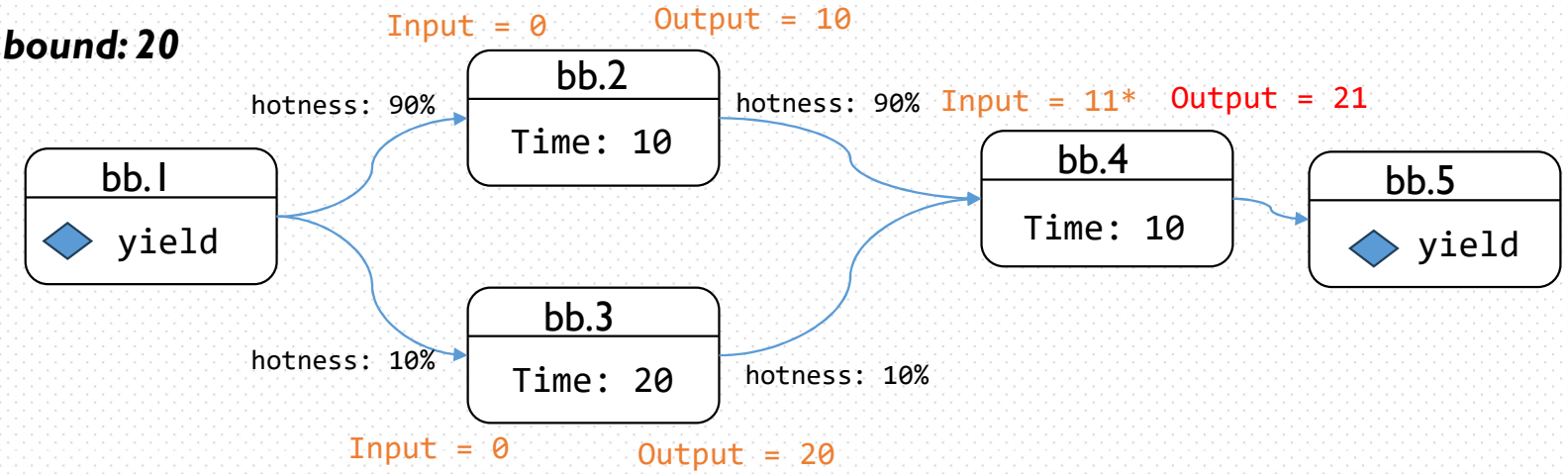


Insert addition yields (*Scavenger yields*) when two yields in scavenger are far away

Yielding on scavenger

- **Primary yields: Identified the same way as primary**
- **Scavenger yields: Used to bound the primary latency**
  - **Via data flow anaysis**

Latency bound: 20

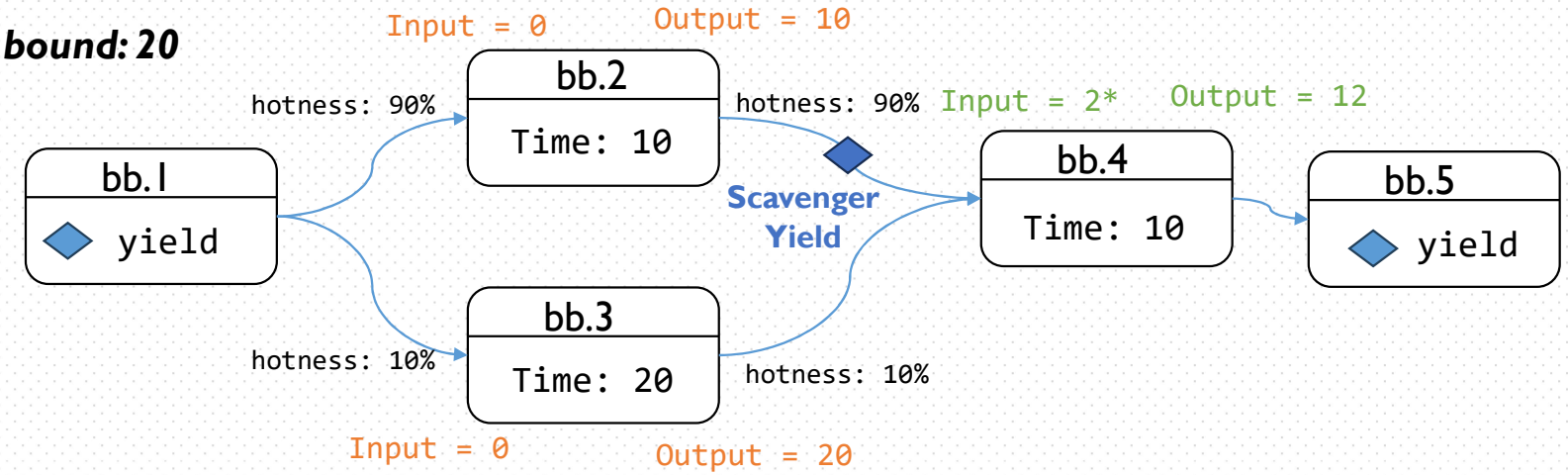


*: 10 * 90% + 20 * 10% = 11

Yielding on scavenger

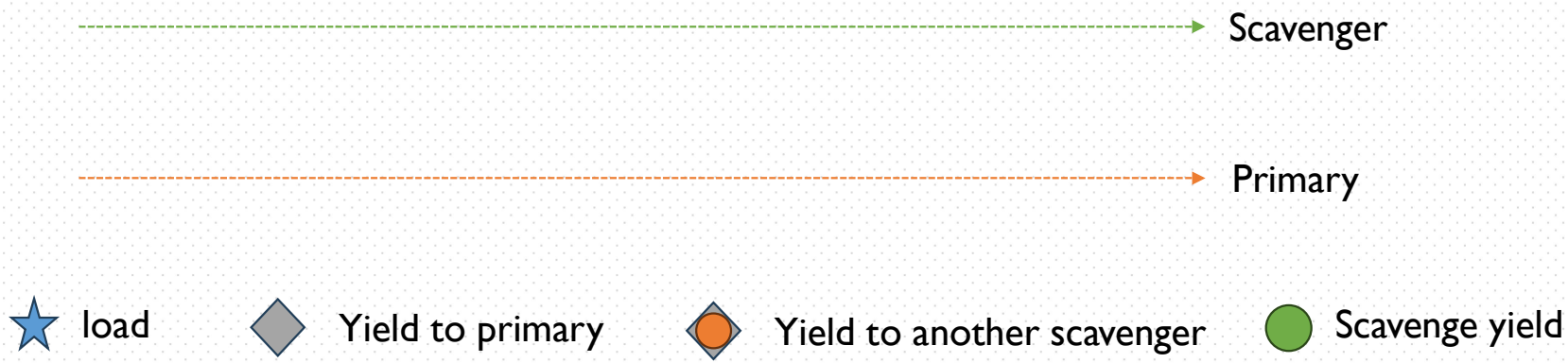
- Primary yields: identified the same way as primary
- Scavenger yields: Used to bound the primary latency
  - Via data flow anaysis

Latency bound: 20



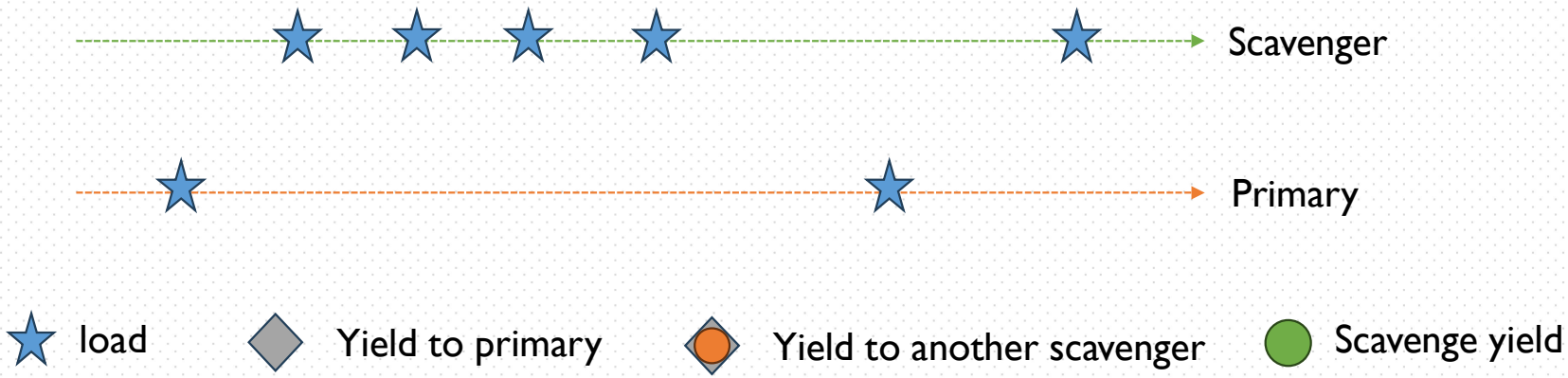
Summary: Yields

❑Instrumentation flow



Summary: Yields

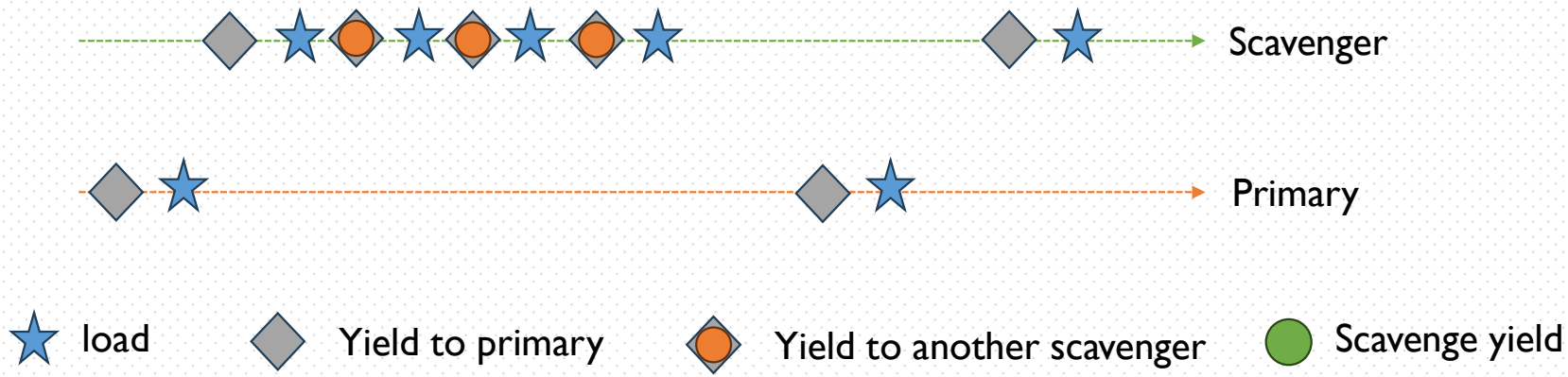
- **Instrumentation flow**
  - **Identify (primary) yields in primary and scavenger**



Summary: Yields

❑Instrumentation flow

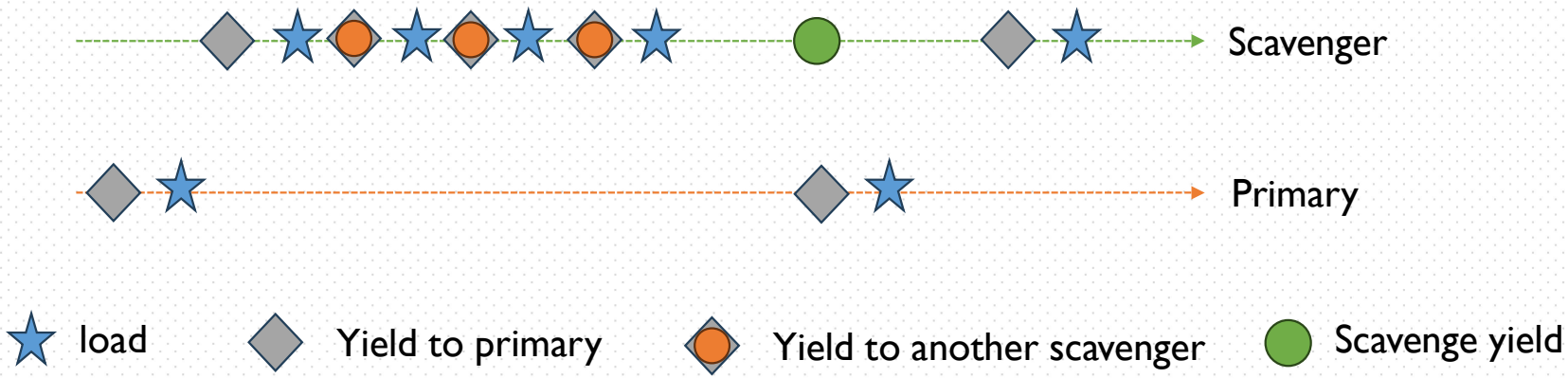
- ❖Identify (primary) yields in primary and scavenger
- ❖**Instrument primary yields in primary and scavenger**



Summary: Yields

❑Instrumentation flow

- ❖Identify (primary) yields in primary and scavenger
- ❖Instrument primary yields in primary and scavenger
- ❖**Perform control flow analysis and instrument scavenge yield**

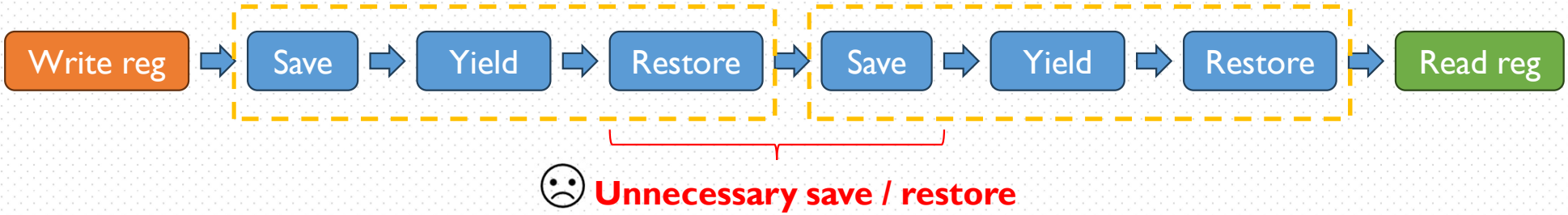


Optimizing yield cost

❑ $T_{yield} = T_{regsave/restore} + T_{control\_passing}$

Consumes most of the time

- Preserve only registers that are "live" at yield location
  - Through ***register liveness analysis***
- Exploit **per-loop** register saving/restoration

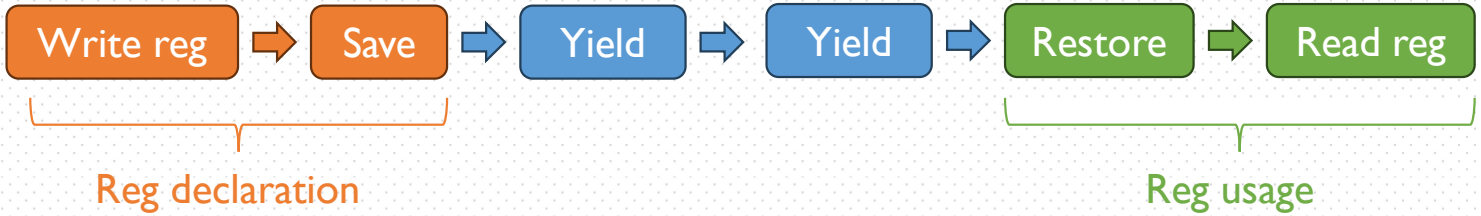


Optimizing yield cost

- $T_{yield} = T_{regsave/restore} + T_{control\_passing}$

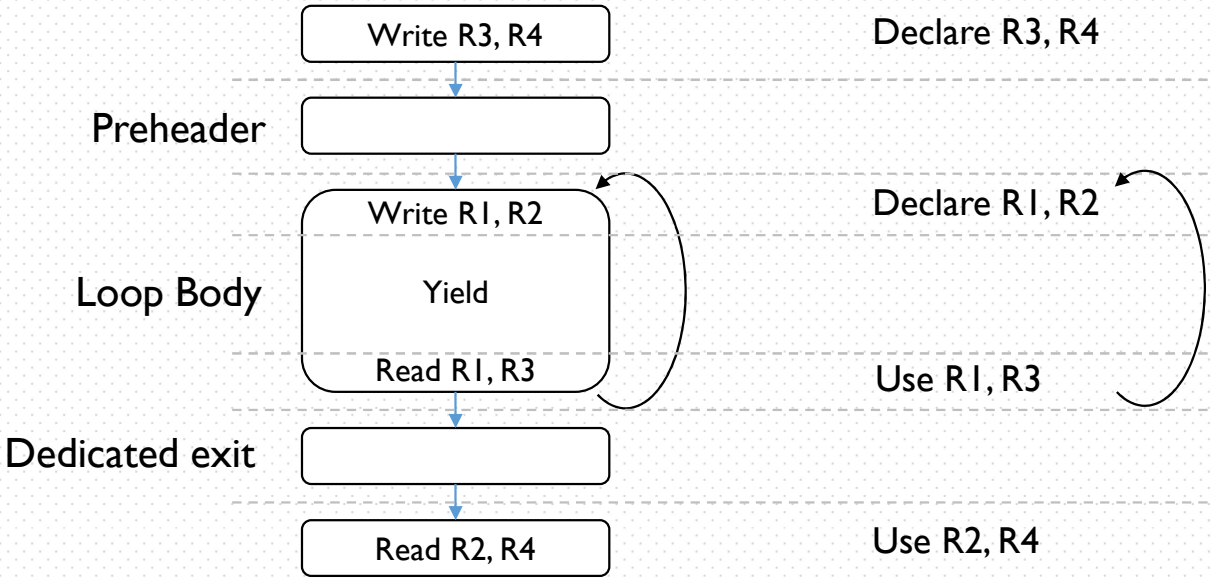
  Consumes most of the time

- Preserve only registers that are "live" at yield location
  - Through ***register liveness analysis***
- Exploit **per-loop** register saving/restoration
  - **General idea:** Save / Restore at declare / use site of the register



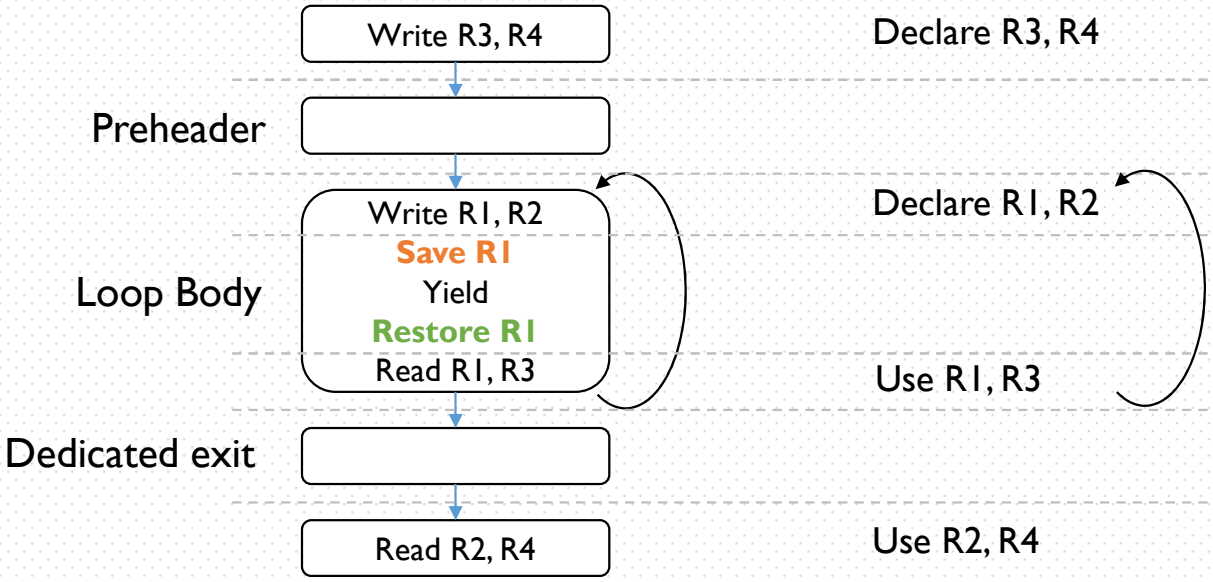
Optimizing yield cost

- Preserve only registers that are "live" at yield location
- Exploit **per-loop** register saving/restoration
  - **General idea:** Save / Restore at declare / use site of the register



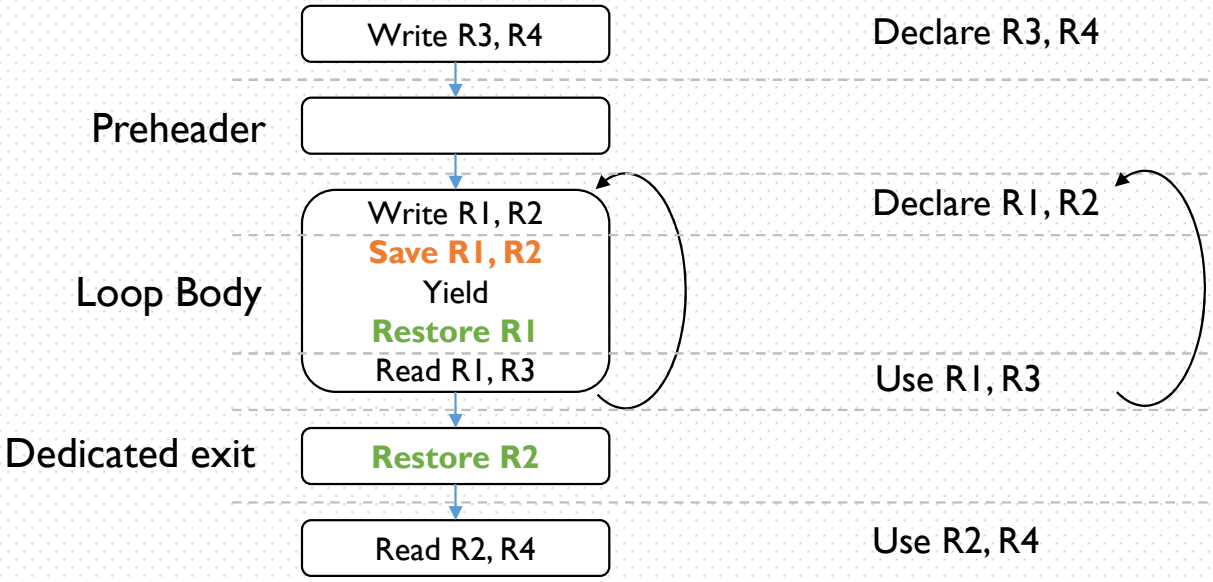
Optimizing yield cost

- Preserve only registers that are "live" at yield location
- Exploit **per-loop** register saving/restoration
  - **General idea:** Save / Restore at declare / use site of the register



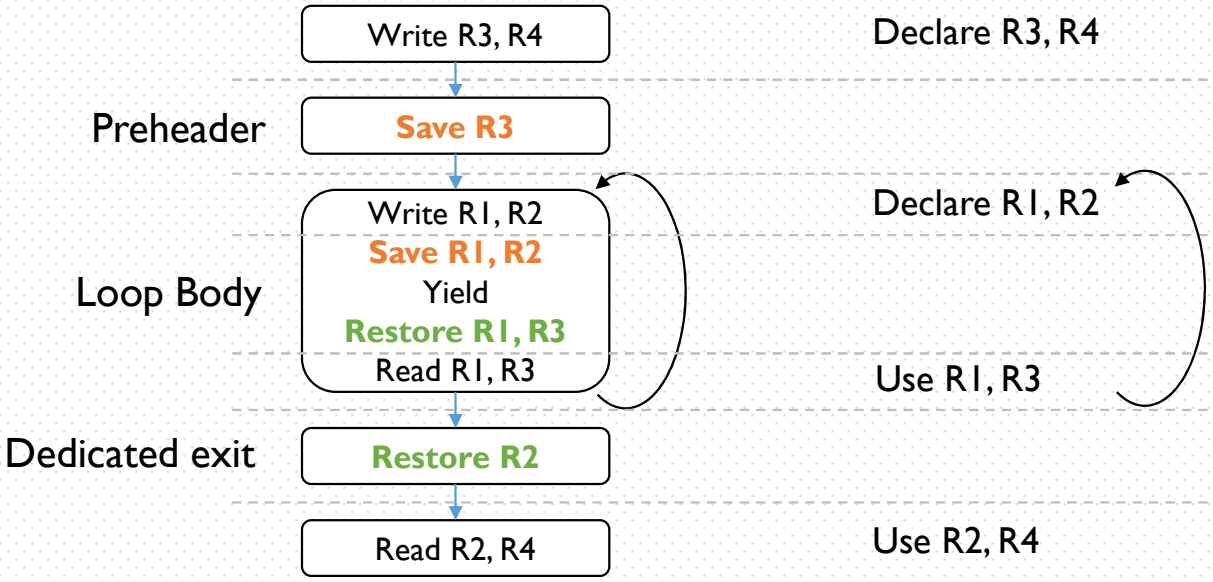
Optimizing yield cost

- Preserve only registers that are "live" at yield location
- Exploit **per-loop** register saving/restoration
  - **General idea:** Save / Restore at declare / use site of the register



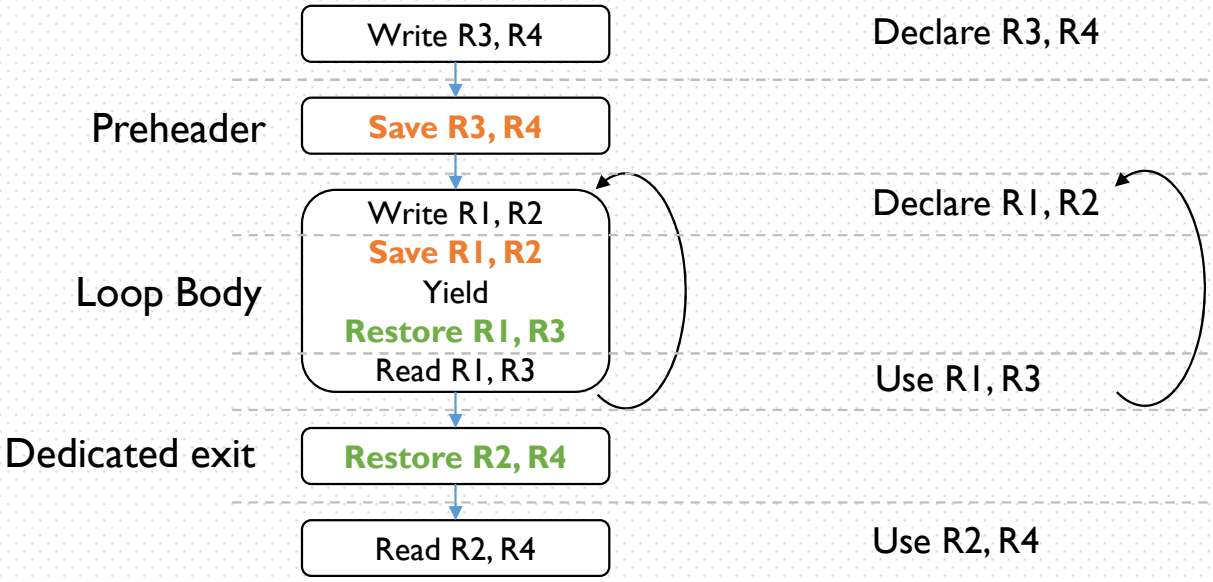
Optimizing yield cost

- Preserve only registers that are "live" at yield location
- Exploit **per-loop** register saving/restoration
  - **General idea:** Save / Restore at declare / use site of the register



Optimizing yield cost

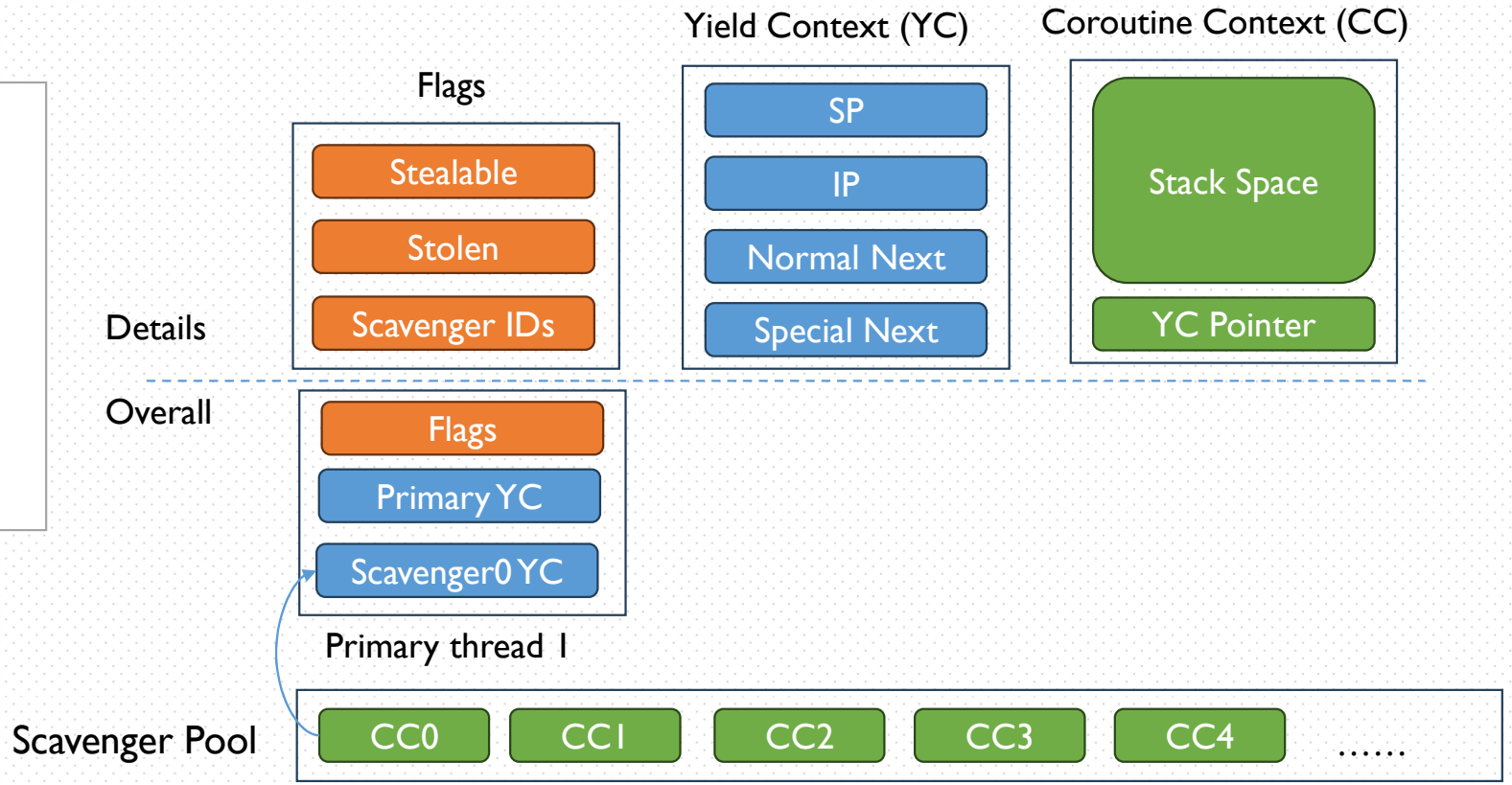
- Preserve only registers that are "live" at yield location
- Exploit **per-loop** register saving/restoration
  - **General idea:** Save / Restore at declare / use site of the register



MSH Runtime

❑Intercept thread state-changing function calls

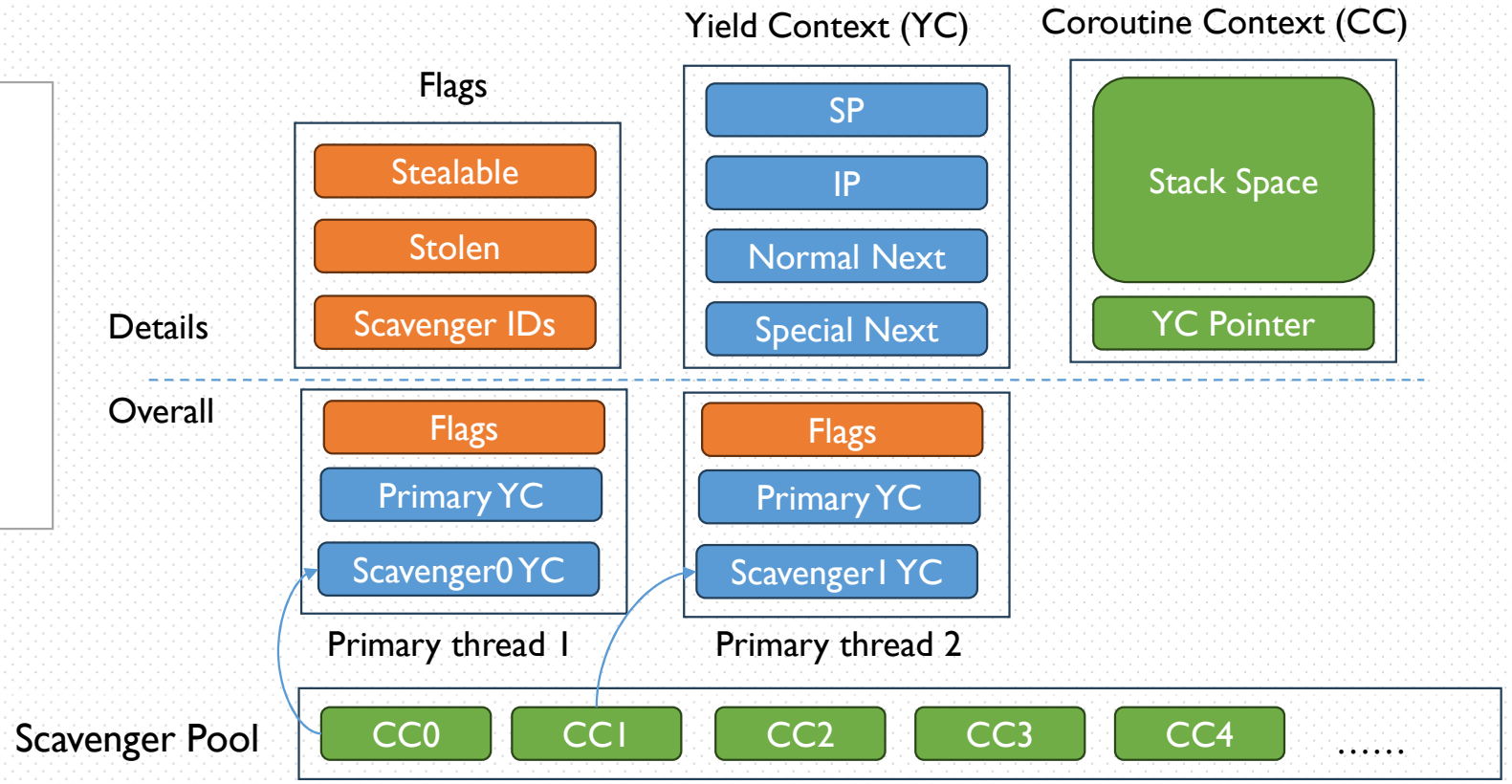
- ➢ Thread creation
- ➢ Thread blocking
- ➢ Thread destroy



MSH Runtime

❑Intercept thread state-changing function calls

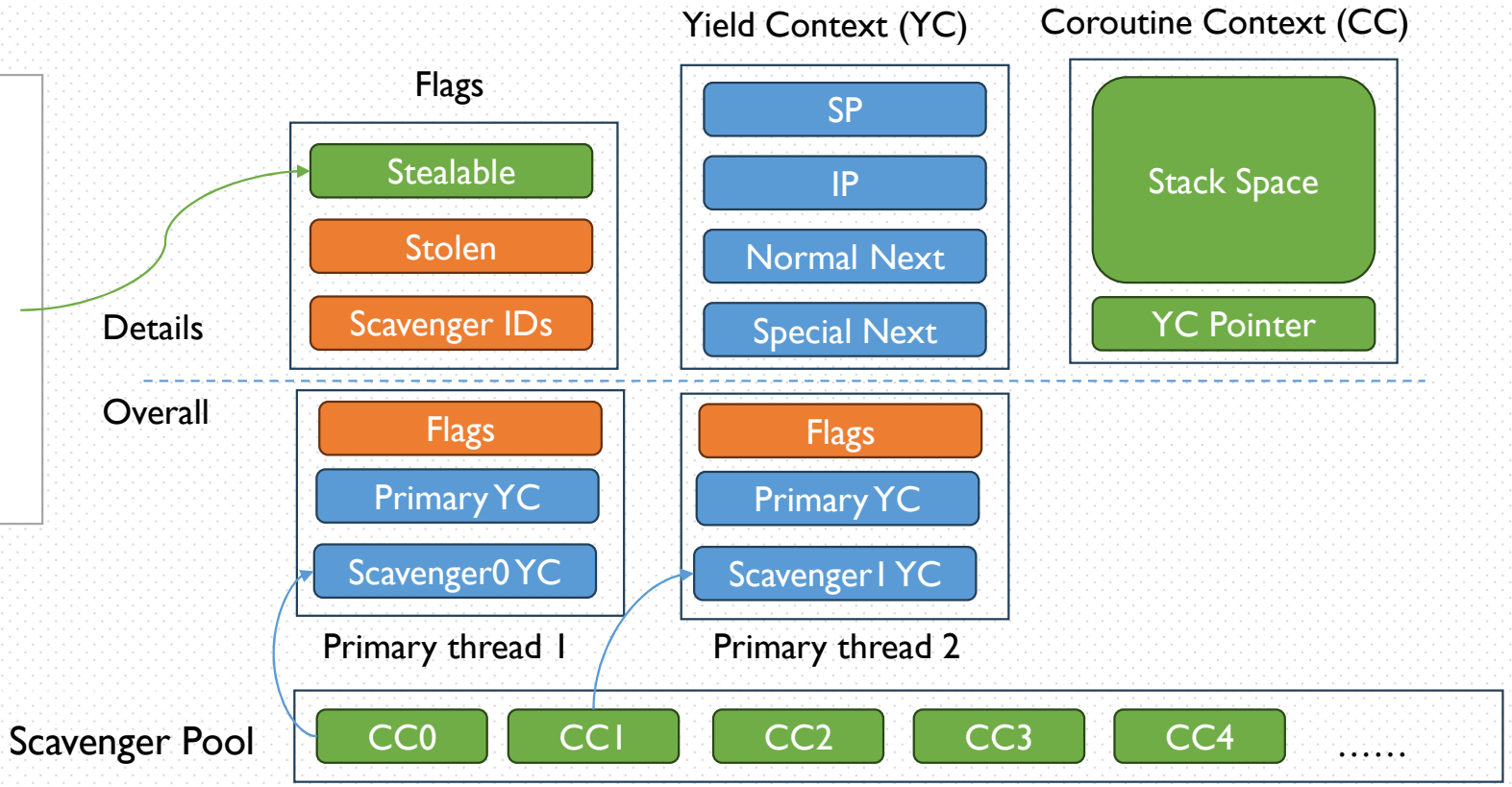
- Thread creation
  - Assign scavenger
- Thread blocking
- Thread destroy



MSH Runtime

❑Intercept thread state-changing function calls

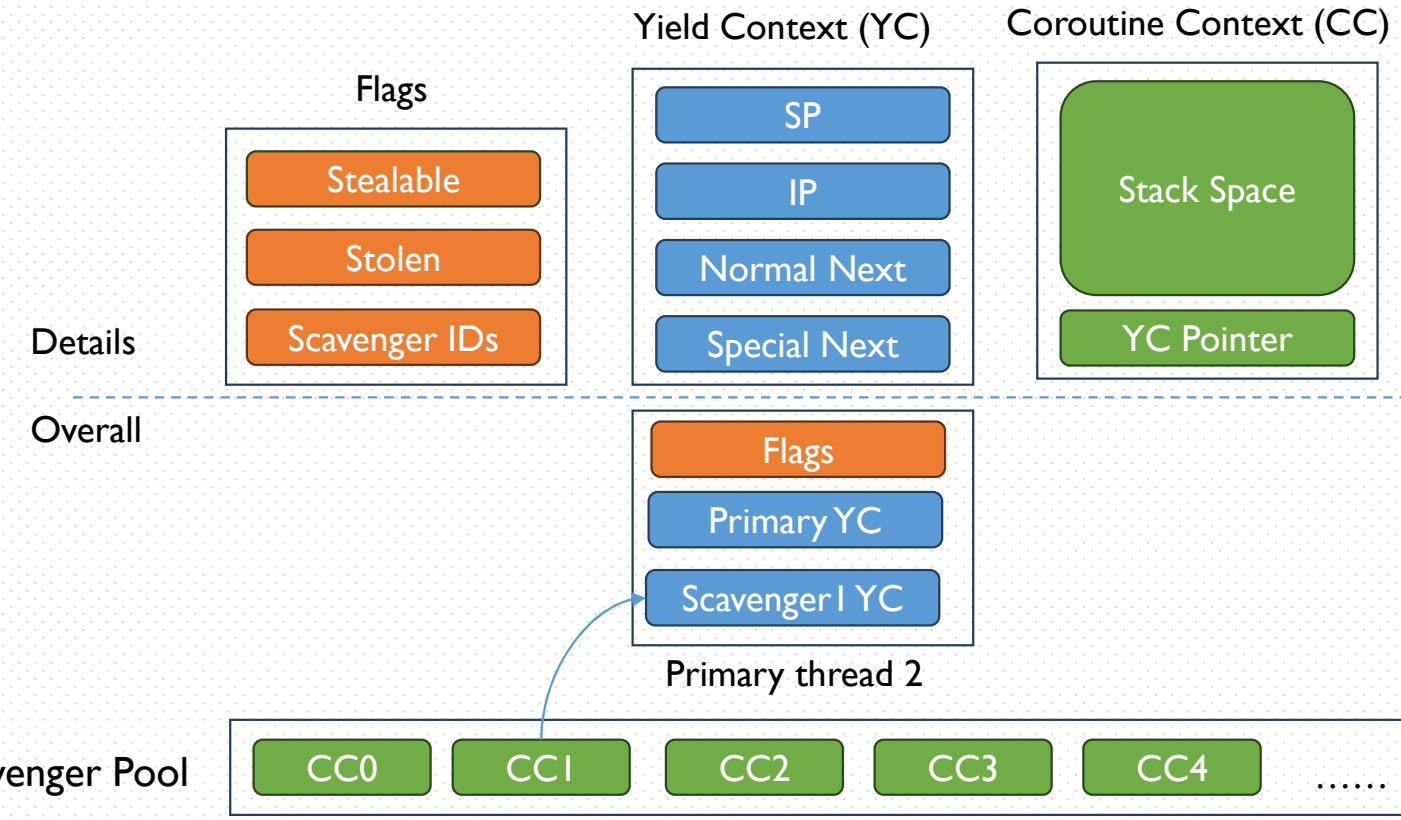
- Thread creation
  - Assign scavenger
- Thread blocking
  - Mark as stealable
- Thread destroy



MSH Runtime

❑Intercept thread state-changing function calls

- Thread creation
  - Assign scavenger
- Thread blocking
  - Mark as stealable
- Thread destroy
  - Free contexts



Implementation

- **Offline profiling**
  - **PEBS :** Load instructions causing cache misses
  - **LBR :** basic block execution count
  - **Bolt :** Register liveness, reaching definition analysis
- **MSH Runtime**
  - **LD_PRELOAD override pthread functions**

Setup: Testbed

- **Hardware**
  - **Dual-sockets 56-core Intel Xeon Platinum 8176 CPUs operating at 2.1GHz**
- **Metrics**
  - **Measure at primary workload with P95 latency**

Setup: Mechanisms

❑Classes of harvestable cycles

	KS (kernel scheduling)	SMT	MSH
Idle time	Y	Y	N
Memory stalls	N	Y	Y
Non-memory stalls	N	Y	N

❑Mechanisms

- ❖MSH only
- ❖MSH + KS
- ❖MSH + SMT/KS (use SMT when primary latency meets requirement, use KS otherwise)

Setup: Workload

❑Primaries

Workload	Detail	Configuration
Ptrchase* ❶❷	Random pointer chasing	8 thread , 16 MB array
Masstree ❶❷❸	In-memory key-value store	24 thread , Tailbench dataset
Sphinx ❶❷❸	Speech recognition system	6 thread , Tailbench dataset

❑Scavenger

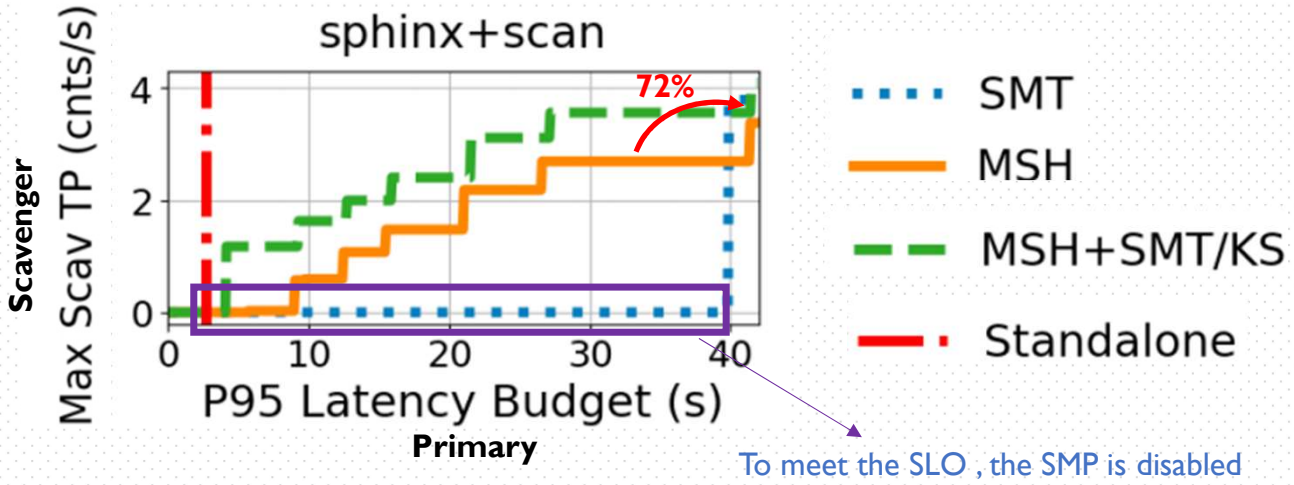
Workload	Detail	Configuration
Scan-creating* ◆	Scan the array and compute the sum	4 MB array
Ptrchase* ●		16 MB array
DFS ★	Graph analysis	CRONO benchmark
Connected component ★		

❶ Idle time ❷ Memory stall ❸ Non-memory stall ◆ High contention ● Frequent stall ★ Mixed

*: Synthetic workload to show MSH advantage

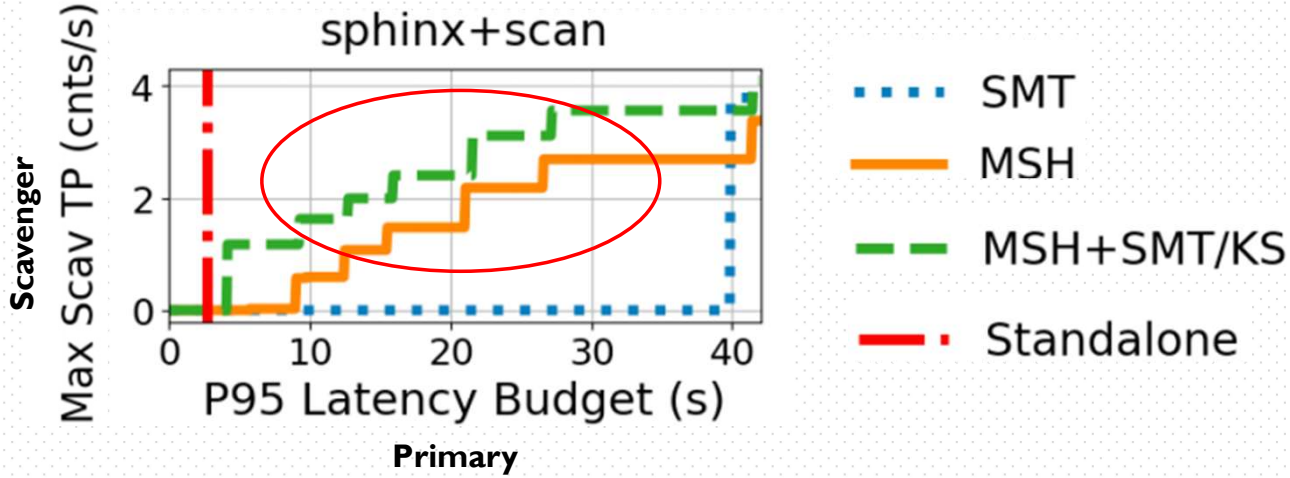
Evaluation: Compared with SMT

- MSH harvests up to **72%** scav. throughput of SMT with SMT disabled



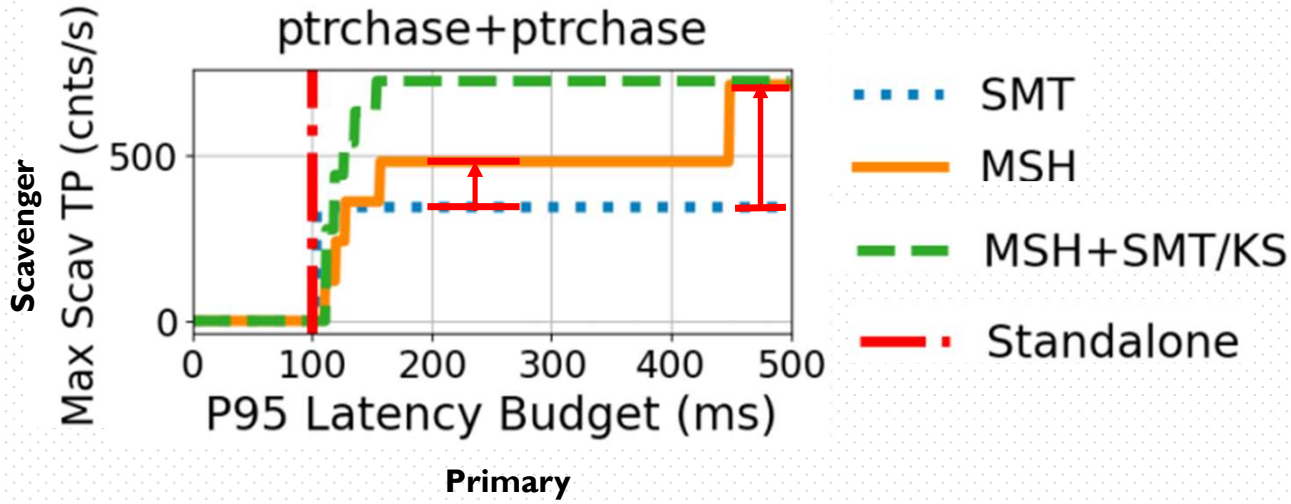
Evaluation: Compared with SMT

- MSH harvests up to 72% scav throughput of SMT with SMT disabled
- MSH can trade off primary latency between scavenger's throughput



Evaluation: Compared with SMT

- MSH harvests up to 72% scav throughput of SMT with SMT disabled
- MSH can trade off primary latency between scavenger's throughput
- MSH can fully harvest memory stalls when scav. stalls frequently



Evaluation: Compared with SMT

- MSH harvests up to 72% scav. throughput of SMT with SMT disabled
- MSH can trade off primary latency between scavenger's throughput
- MSH can fully harvest memory stalls when scav. stalls frequently

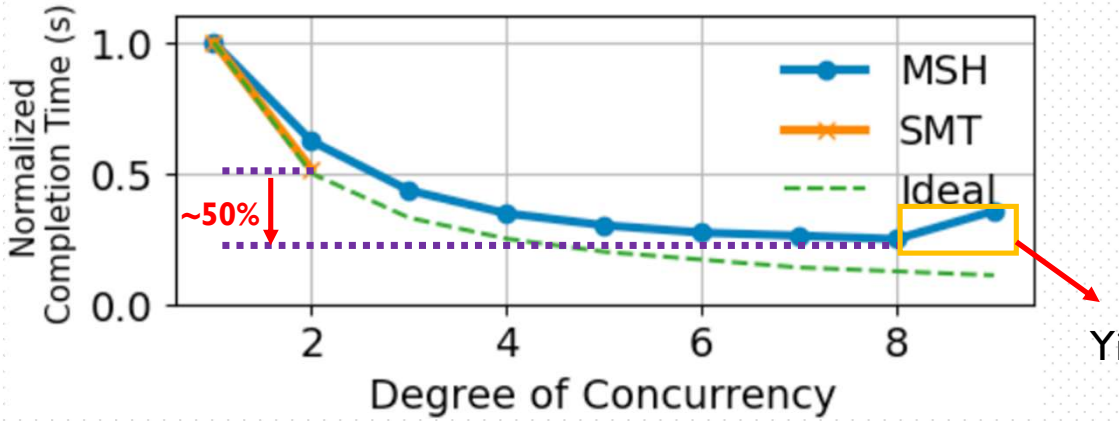
verify

	Workload	Thread count
SMT	Ptrchase (16 MB)	1, 2
MSH		1, 2, 4, 6, 8

Running multiple ptrchase threads, and measure the overall completion time

Evaluation: Compared with SMT

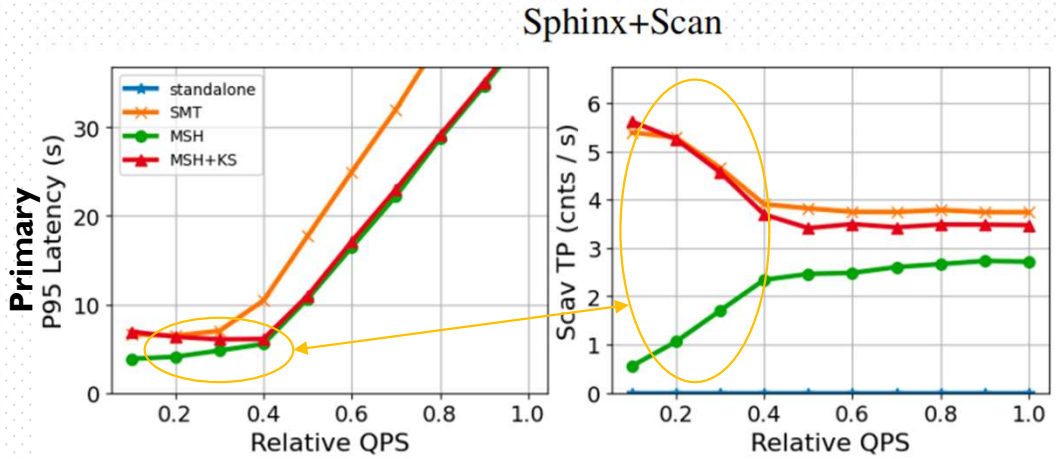
- MSH harvests up to 72% scav. throughput of SMT with SMT disabled
- MSH can trade off primary latency between scavenger's throughput
- MSH can fully harvest memory stalls when scav. stalls frequently



Yielding overhead larger than the benefits

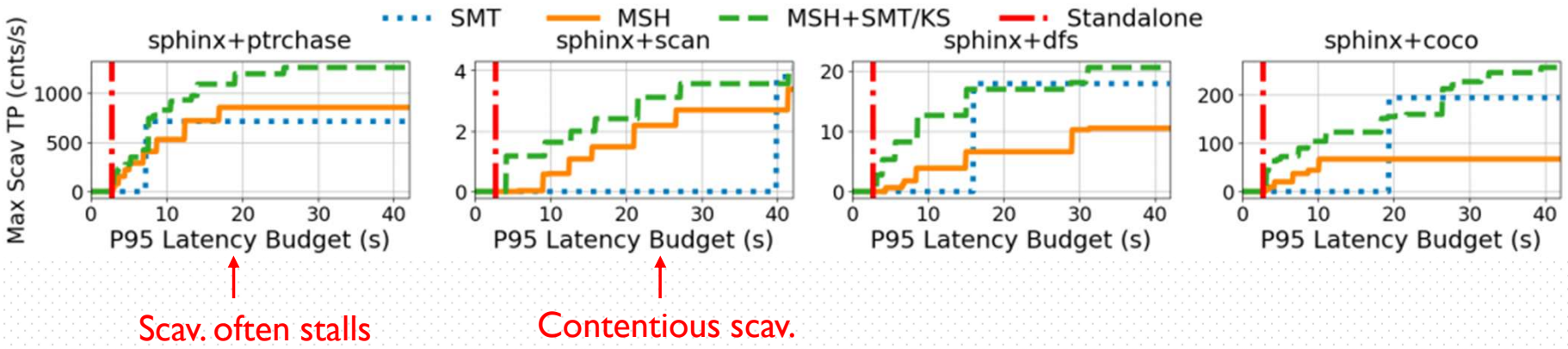
Evaluation: Compound mechanism

- **MSH + KS**
  - **Add small latency and get much higher throughput at low loads**



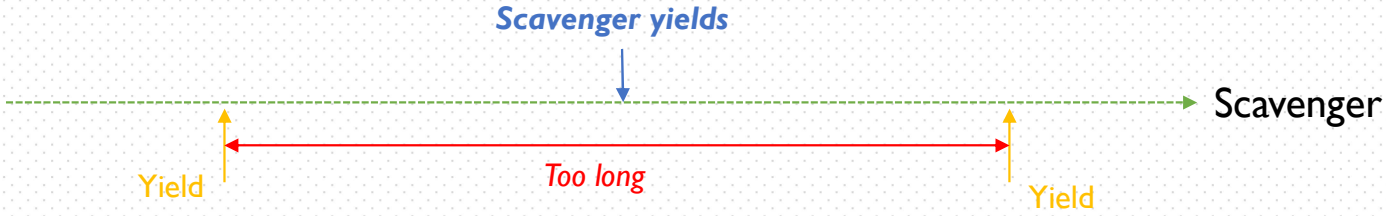
Evaluation: Compound mechanism

- MSH + KS
- **MSH + SMT/KS**
  - **Better than SMT-only under almost all latency SLOs**
    - **When scavengers often stall, SMT is on, harvesting all the stalls**
    - **For contentious scavengers, KS can also harvest idle time**



Evaluation: Breakdown

- MSH optimizations
  - Bounding yield overhead: select load instructions with:
    - Significant portion of memory stalls
    - L3 cache miss likelihood
  - Bounding inter-yield distances: Scavenger yields



  - Reducing yield cost

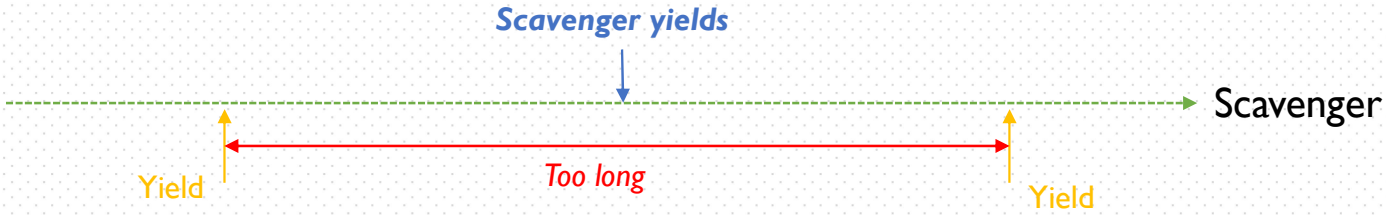
Evaluation: Breakdown

❑MSH optimizations

❖Bounding yield overhead: select load instructions with:

- ➢ Significant portion of memory stalls
- ➢ L3 cache miss likelihood

❖Bounding inter-yield distances: Scavenger yields



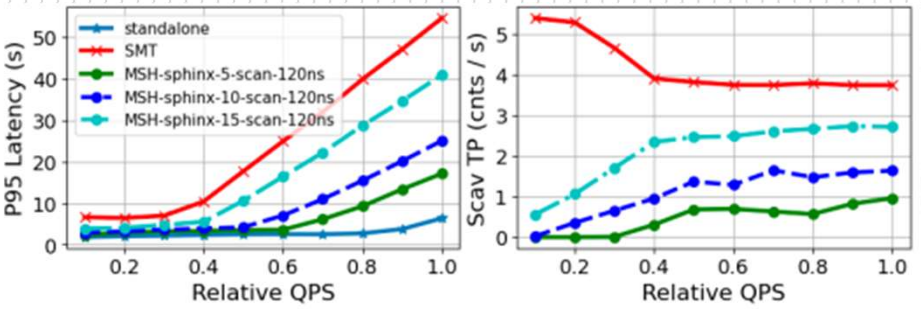
❖Reducing yield cost

Evaluation: Breakdown

- ❑Yield overhead bounding
- ❑Inter-yield distance

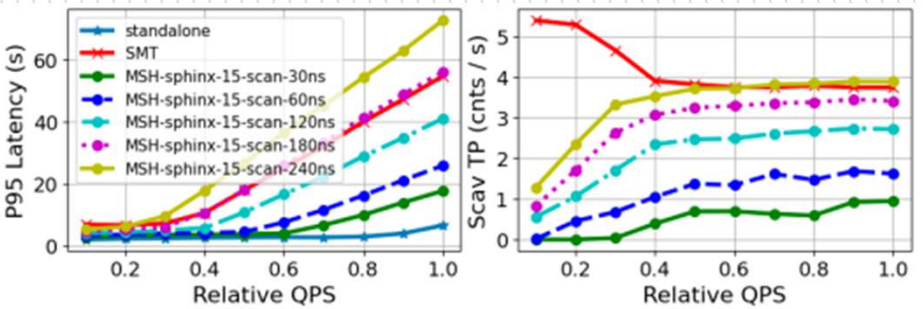
→ (Primary) Latency- (Scavenger) throughput trade-off

Latency bounding ↑ Latency ↑ Throughput



Aggregate yield overhead bound

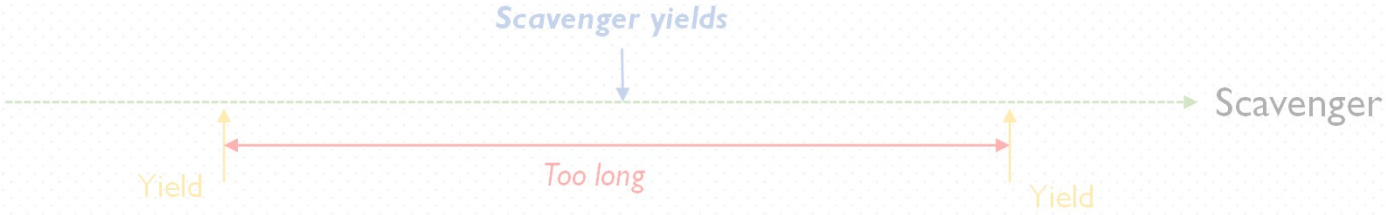
Inter-yield distance ↑ Latency ↑ Throughput



Scavenger inter-yield distance

Evaluation: Breakdown

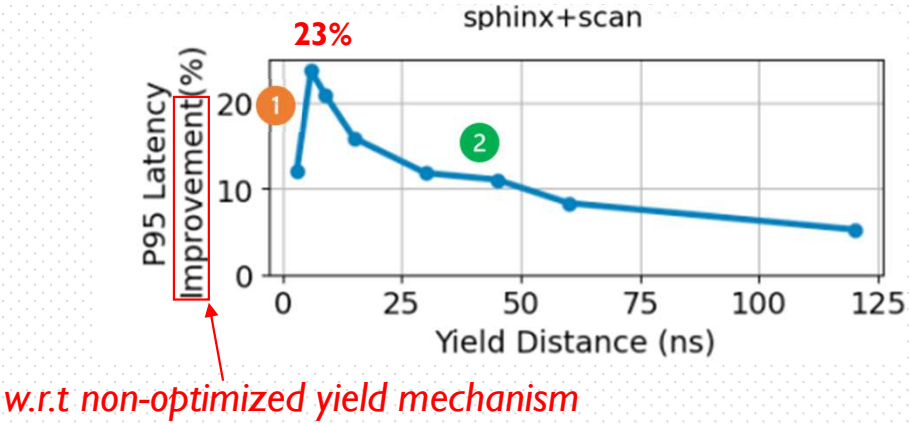
- MSH optimizations
  - Bounding yield overhead: select `load` instructions with:
    - Significant portion of memory stalls
    - L3 cache miss likelihood
  - Bounding inter-yield distances: Scavenger yields



  - **Reducing yield cost**

Evaluation: Breakdown

❑Measure primary's latency improvement for different inter-yield distance

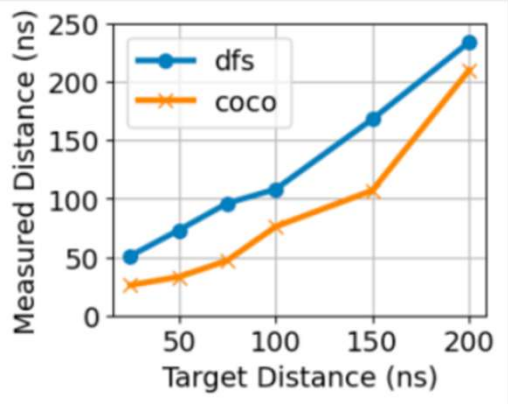


1. #Cache hit latency > #Duration of scavenger executing. Yield cost won't affect primary latency
2. Yield cost no longer plays the important part

Reducing yielding cost shows benefits in P95 latency of the primary

Evaluation: Effectiveness

❑Effectiveness of enforcing inter-yield distances



✓ MSH accurately enforces target inter-yield distances

❑Effectiveness of profiling

✓ Accurately capturing load inst. incurs minimal overhead

✓ Sample 100x more frequently slows down the application but does not affect profile results

Conclusion

- MSH effectively harvests memory-bound stalls through …
  - Profile-guided binary instrumentation
  - Light-weight yield mechanisms
  - Efficient run-time
- Pros:
  - The problem of memory-bound stall is interesting
  - The system is general and can be used together with other HW features
- Cons:
  - The evaluation scale seems relatively small
  - Some conclusions in the evaluation are not convincing
  - Is profile-based solution really a good idea?