OMR

Parallelism & Adaptive Garbage Collection Threading

Salman Rana

*(Special Thanks to **Aleksandar Micic**)*

OpenJ9

Overview

- Brief Overview of GC and OMR GC Technology
- Background: GC Parallelization
  - Throughput and Pause Time
- Need For Adaptive Threading
  - Parallelization Overhead
- Adaptive Threading: Core Idea
- GC Internals & Adaptive Threading Implementation
  - Dispatcher & Tasks
- Performance Results
- Future Work

Garbage Collection

"***garbage collection*** (***GC***) is a form of automatic memory management. The *garbage collector* attempts to reclaim memory which was allocated by the program, but is no longer referenced"

parallelism

- *Causes **unpredictable application pauses***
- *Runtime Costs*

....

Optimizations

Adaptive Threading

Garbage Collection

- From high (user) level it's compromise between:
  1) application throughput
  2) average/worst GC pause
  3) sometimes footprint (heap/native memory consumption)

- Internally, technology used may be significantly different....
  - Flat heap vs (fixed sized) regions
  - STW (stop the world) vs concurrent
  - First fit vs best fit allocation
  - Generational or not

OMR GC Technology

OMR

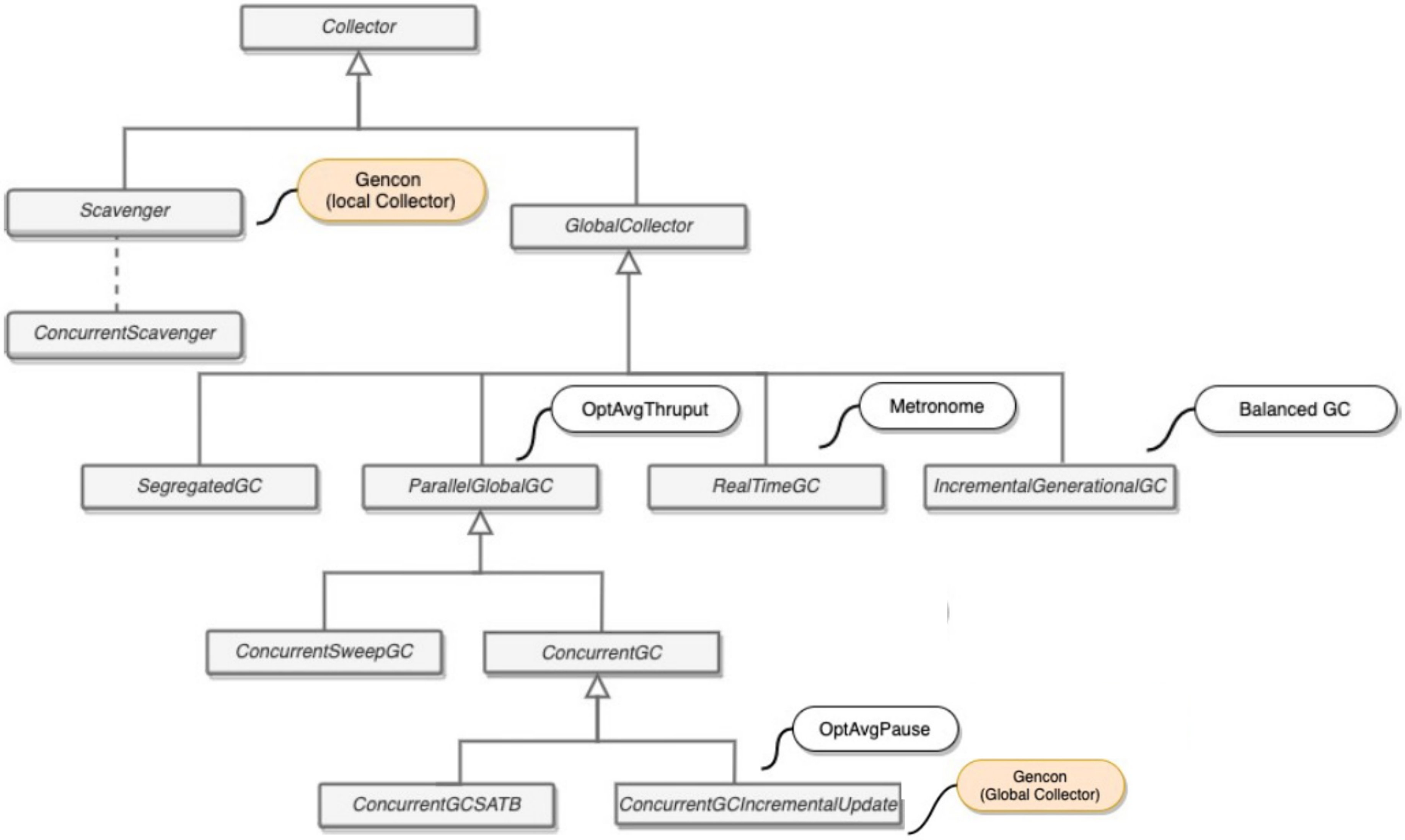
Policy	Core technology	Pros	Cons
optthruput	STW Mark and Sweep (and optionally Compact)	Very good ***throughput***, but still typically inferior to gencon (unless RS overhead is high)	High degree of heap ***fragmentation***, high ***pause times***
optavgpause	Concurrent Mark and Sweep (and optionally Compact)	***Lower pause times*** than optthruput	Slightly lower ***throughput*** than optthruput. Higher ***heap pressure*** due to floating garbage
gencon	- Generational (Tenure+Nursery) - Local Copying GC On Nursery - Concurrent Global Mark, Sweep (and optionally Compact)	Typically ***best throughput***, low average pauses ***[DEFAULT policy in OpenJ9]***	Tenure fragmentation may lead to ***global compact***

```
$ java -Xgcpolicy=[gencon,optthruput…] App
```

*Balanced & Realtime *(Implemented in OpenJ9 using OMR components)*

OpenJ9

Collector Internal High-Level View

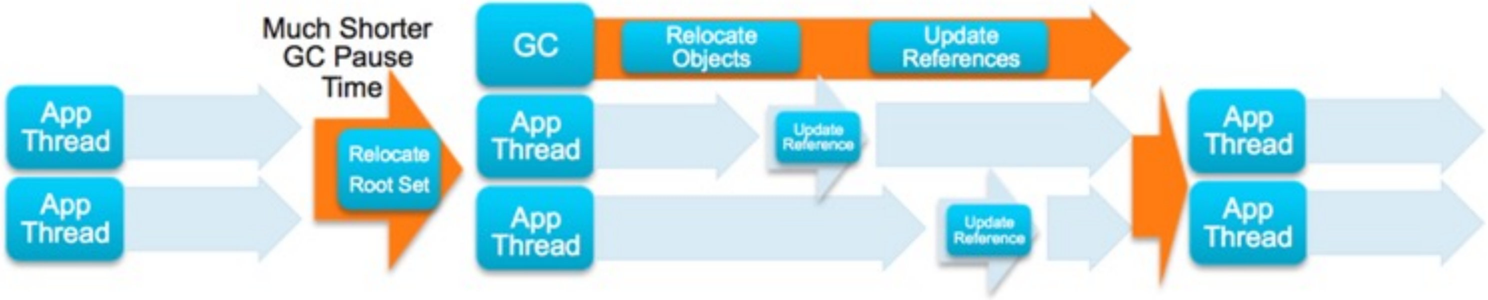


Garbage Collection

OMR

STW GC Cycle

Pause-Less (Concurrent) GC Cycle

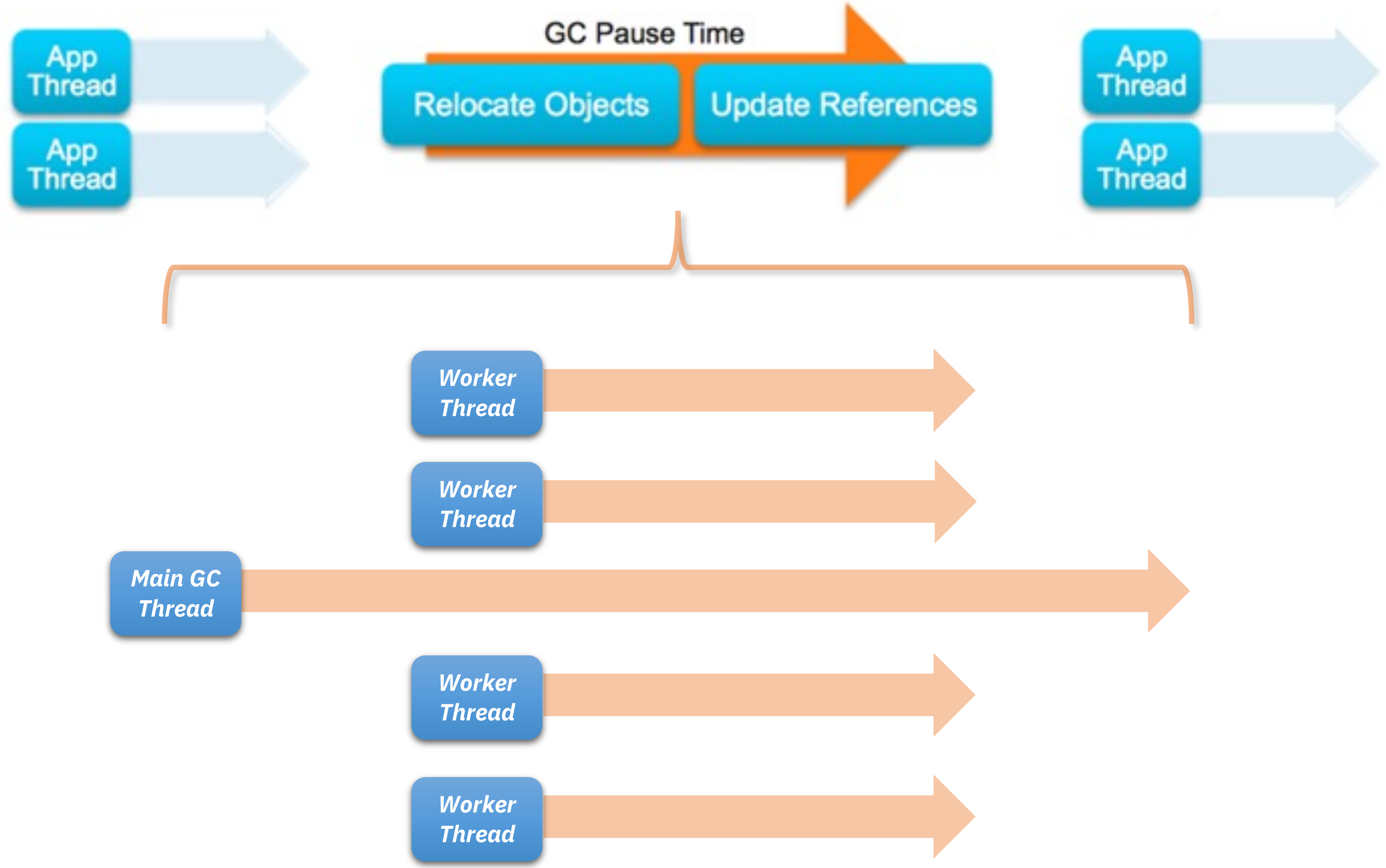


OpenJ9

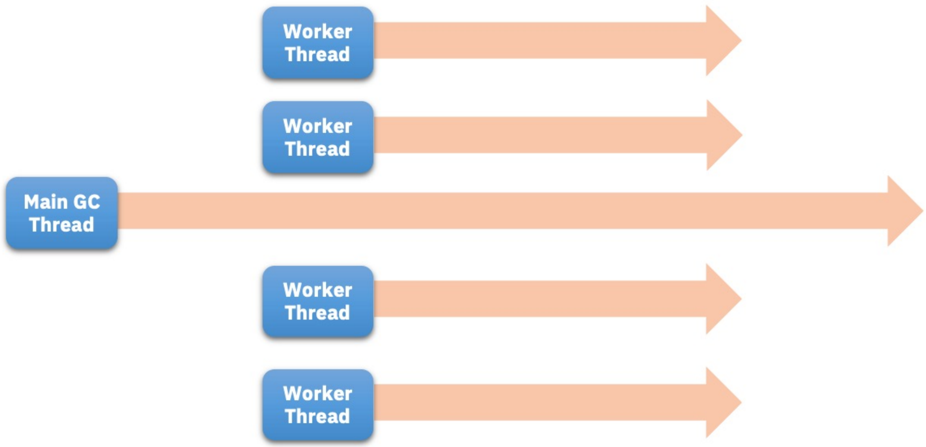
Parallelism

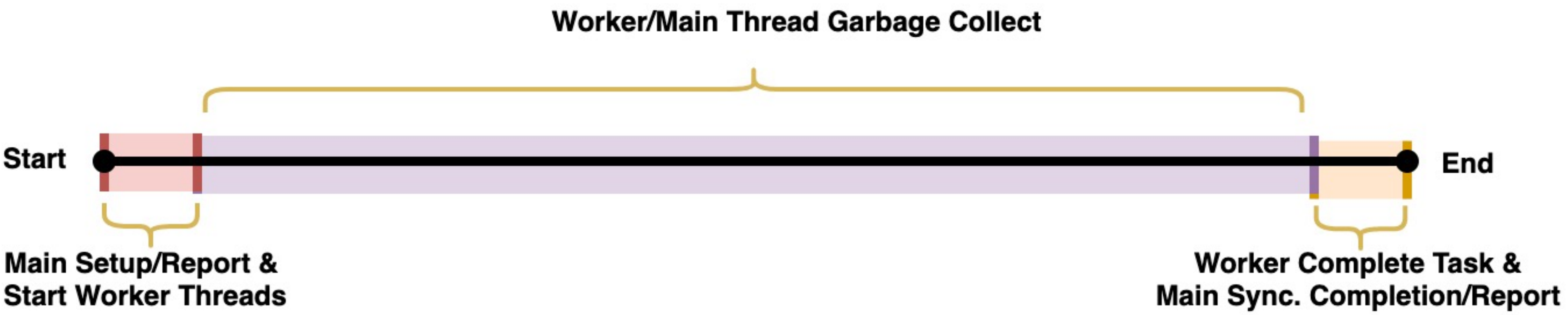
- Utilize available Resources
  - Multi-core processors
- All Collectors in major VMs
- Decrease pause time
- Tasks parallelized
  - GC operations completed in parallel by multiple worker/helper threads
  - e.g., object graph traversal by multiple threads
  - key in reducing GC cycle times
- Total GC Threads = # Hardware Threads
  - XX:ParallelGCThreads=X

GC Parallelism

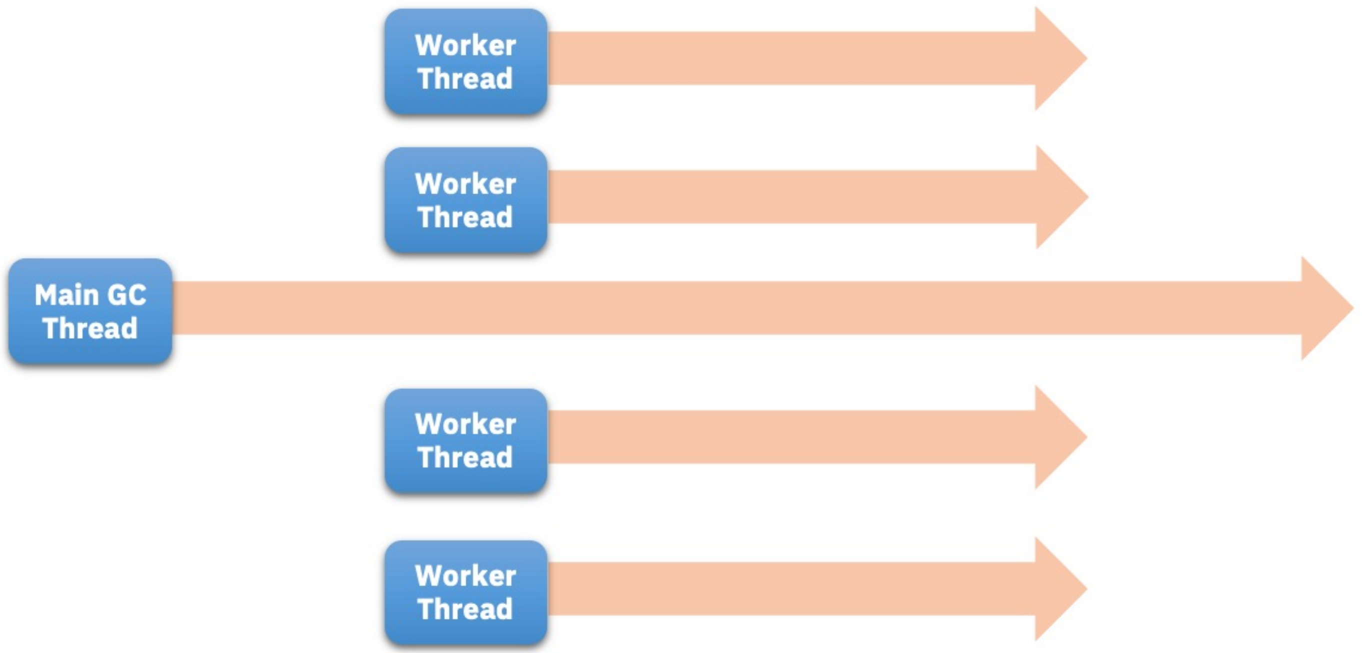


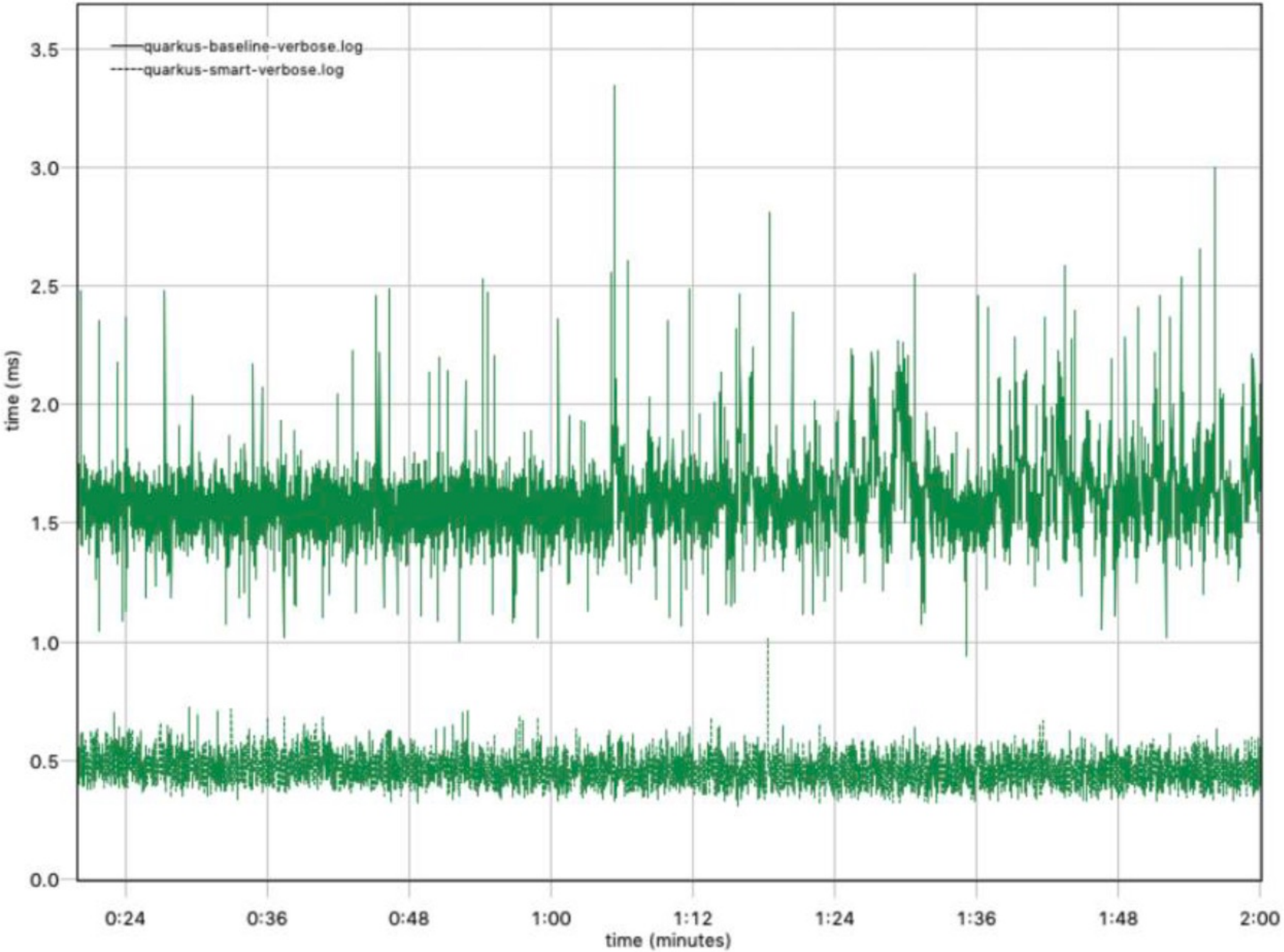
Parallelism





So What's the Issue?





48 Threads Utilized

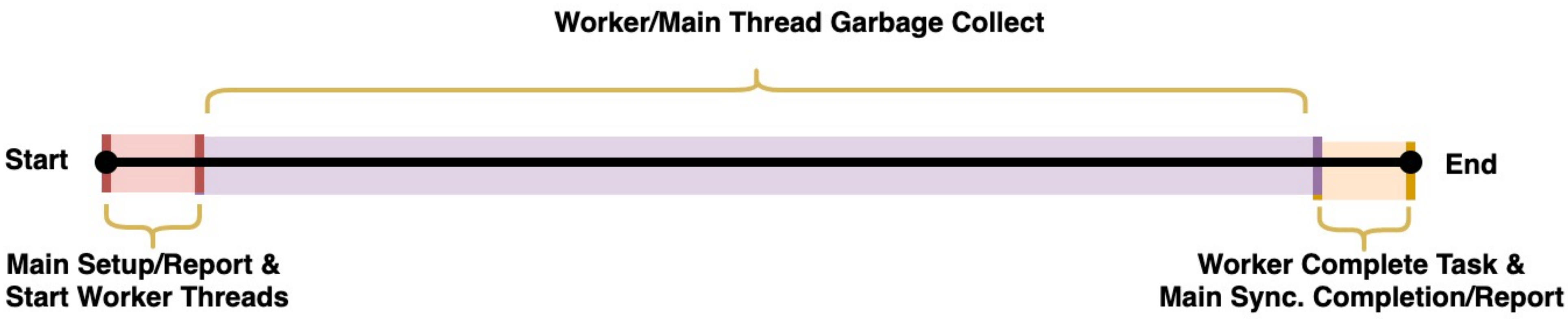
4 Threads Utilized

Parallelism

OMR

- Is there a cost?

- Additional requirements with multi-threading
  - Synchronize (critical sections and accessing global resources)
  - Manage threads (dispatch and suspend)



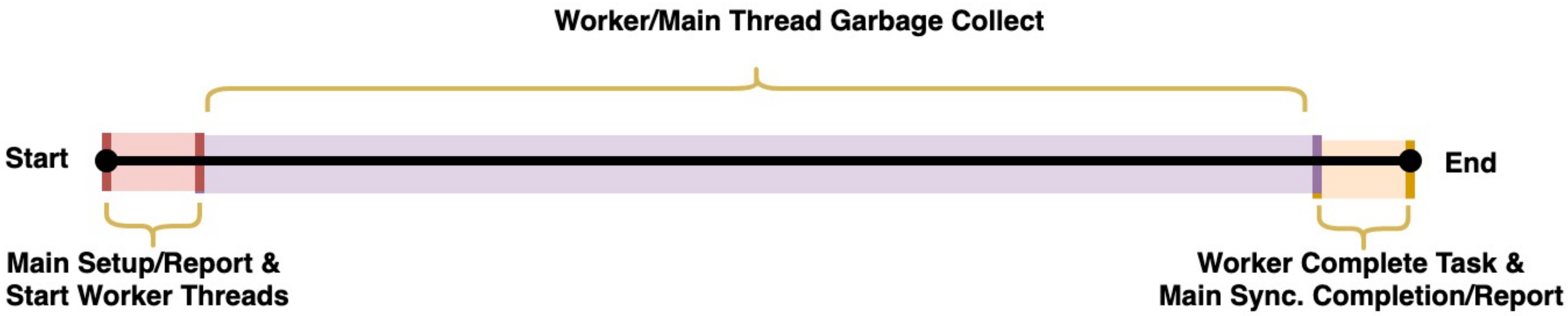
OpenJ9

Parallelism

- They may need to synchronize
  - E.g., Mark Map: one word (64bit) my contain bit for multiple objects. Different threads may be marking those objects and race on updating the word. Atomic operation (compare&swap) is used

  - GC threads frequently push/pop to/from Work Stack. Mutex is used

  - Notifying idle threads

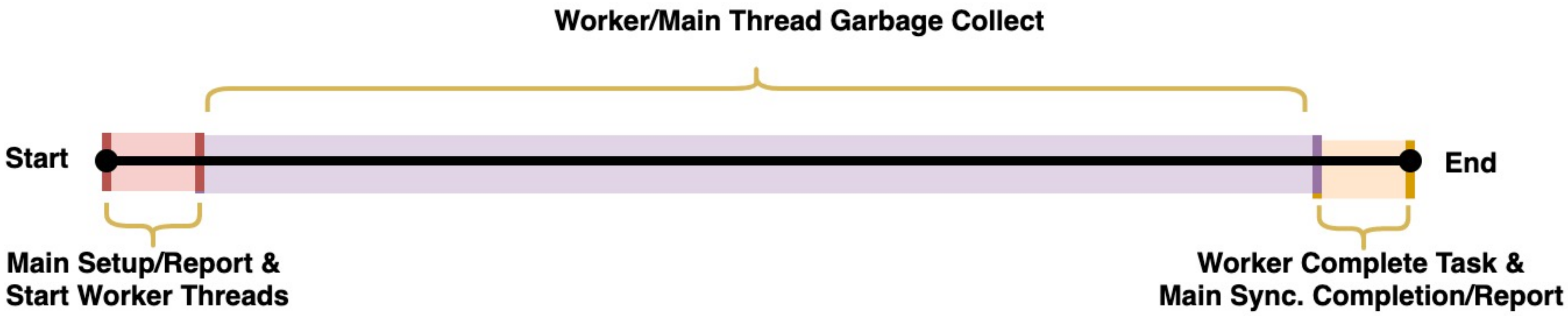


Parallelism

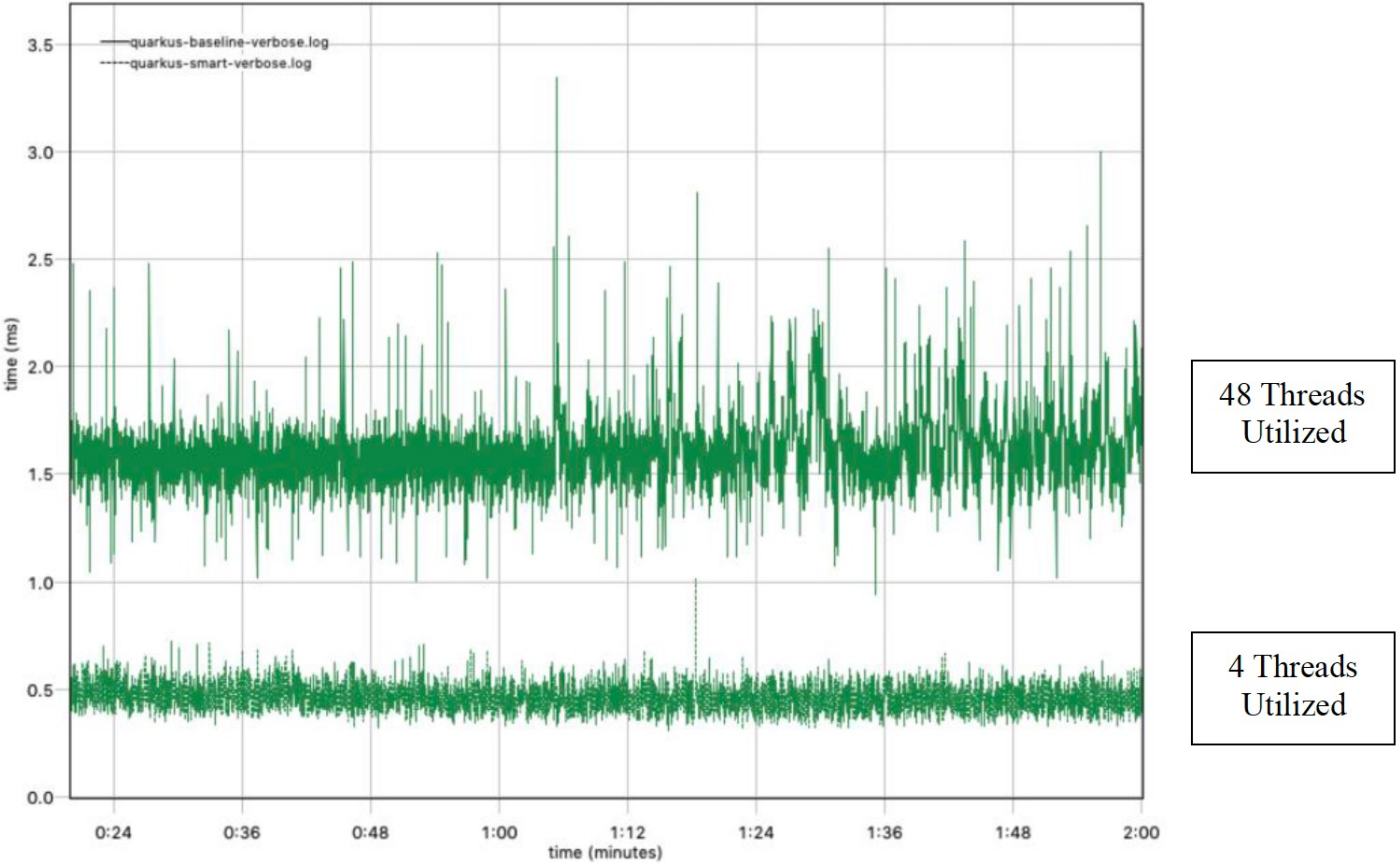
OMR

- noticeable overhead associated with parallelizing tasks
  - ***Little work to be distributed***
    - Workload
    - Object Graph
  - ***CPU usage / Multi VM scenario***

- This overhead can be significant as it increases proportionally with the number of threads utilized.



OpenJ9


quarkus-baseline-verbose.log
quarkus-smart-verbose.log
time (ms)
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0:24
0:36
0:48
1:00
1:12
1:24
1:36
1:48
2:00
time (minutes)
48 Threads Utilized
4 Threads Utilized

https://medium.com/road-less-ventured/too-many-cooks-in-the-kitchen-3ad8507af96a

OMR

Too many? Not enough?

OpenJ9

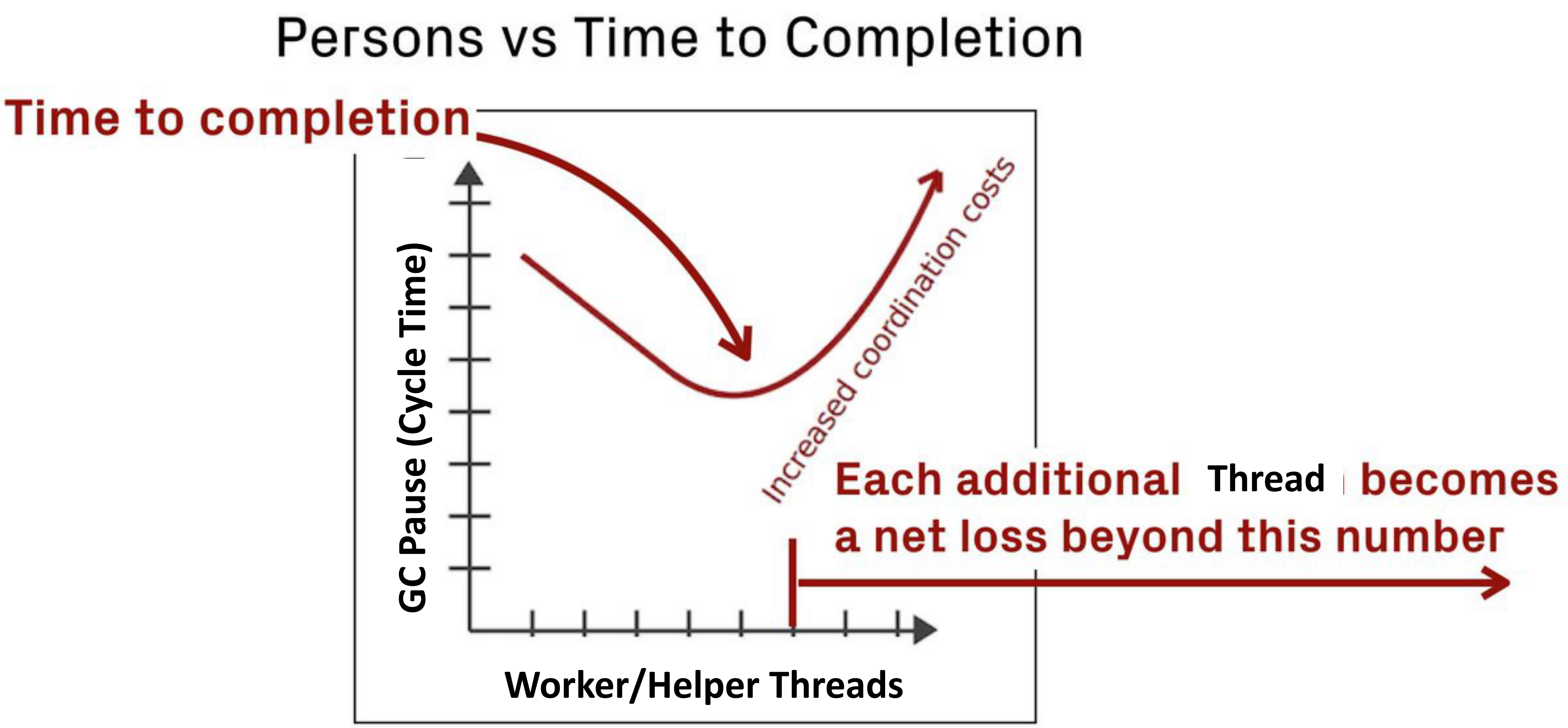
Suboptimal vs Detrimental Parallelism

- Net Loss
- Lost Gains

Threads	Score	Scav. Avg.
48	222,567	1.60 ms
8	255,611	0.60 ms
4	261,737	0.35 ms

Threads	Score
48	80,543
8	93,824
4	91,166

Adaptive Threading



https://codescene.com/blog/visualize-brooks-law/


shutterstock.com · 1719028660

Adaptive Threading

- When to adjust and how much to adjust by
- Seek equilibrium point, where parallelization results in peak performance
- Dynamic
  - workload and load distribution change
  - pick up on on threads being shared across VMs (CPU Usage)
- Recommendation must not be invasive
  - there should not be adverse effects given anomalies
- Adaptive Threading vs Traditional Tuning

Adaptive Threading

OMR

- Model and Heuristics
  - Optimal thread count can be projected
  - thread count can be adjusted between cycles

- Systematic approach based on
  - # of thread utilized
  - Overhead data (busy/stall times for managing and synchronizing threads) aggregated from utilized threads of previous GCs

OpenJ9

Busy and Stall Times

OMR

- Drives Adaptive Threading

- Busy time = time a thread is performing useful GC work which contributes to completing the cycle
  - Scanning Objects
  - Root Processing
  - RS processing
  - Copy or Marking Objects

- Stall time time a thread is doing non-useful/trivial work or time that it's idle (not doing any work).
  - Push/pop something to/from shared global list (e.g., scan list)
  - Acquire synchronization monitor (contention)
  - Idle at a synchronization point
  - Idle waiting for work
  - Wake up from idleness and start running
  - Notify idle threads (the time it takes for a thread to notify idle threads)

OpenJ9

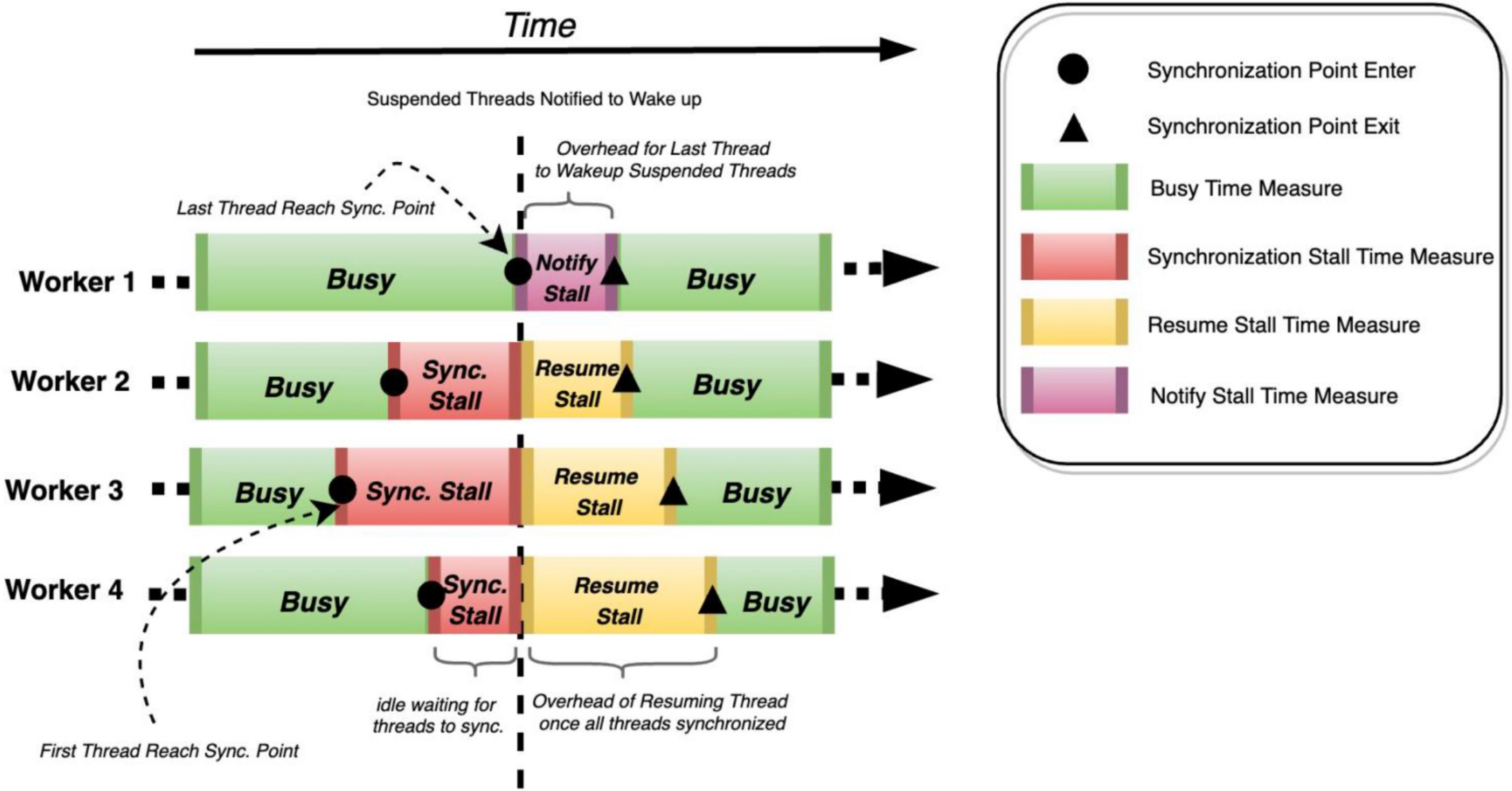
Busy and Stall Times

- Different types of stalls have different characteristics and varying dependency on utilized threads.

- we must distinguish between
  - synchronization stall (idle waiting for threads to synchronize)
  - resume stall (overhead to resume threads, also includes notify stall)
  - Idle waiting for work

These stall times respond differently when changing utilized threads

https://medium.com/road-less-ventured/too-many-cooks-in-the-kitchen-3ad8507af96a


Time
Suspended Threads Notified to Wake up
Overhead for Last Thread to Wakeup Suspended Threads
Last Thread Reach Sync. Point
Worker 1
Busy
Notify Stall
Busy
Worker 2
Busy
Sync. Stall
Resume Stall
Busy
Worker 3
Busy
Sync. Stall
Resume Stall
Busy
Worker 4
Busy
Sync. Stall
Resume Stall
Busy
idle waiting for threads to sync.
Overhead of Resuming Thread once all threads synchronized
First Thread Reach Sync. Point
Synchronization Point Enter
Synchronization Point Exit
Busy Time Measure
Synchronization Stall Time Measure
Resume Stall Time Measure
Notify Stall Time Measure

Adaptive Threading Model

One such implementation of the model can be derived by finding a minimum of the following GC time function *(used to project total duration of GC for m threads, with observed busy and stall times while performing GC with n threads)*:

$$Time_{GC}\,(m, n, b, s) = b * \left(\frac{n}{m}\right) + \ s * \left(\frac{m}{n}\right)^{X}$$

Adaptive Threading Model

$$(1) \quad Number\ of\ Optimal\ Threads = m(n,b,s) = n * \sqrt[X+1]{\frac{b}{X * s}}$$

$$(2)\ Recommended\ Threads\ For\ Next\ Cycle = \lfloor ((m(n,b,s) + H) * (1 - W)) + (n * W) \rfloor$$

$$m(n, \%\,Stall) = n * \sqrt[X+1]{\frac{1}{X} * \left(\frac{1}{\%Stall} - 1\right)}$$

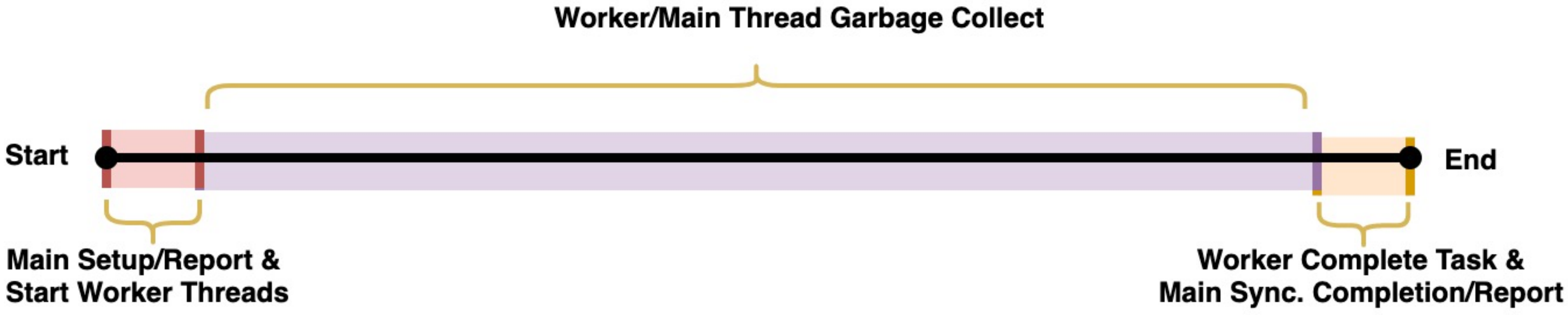
Adaptive Threading Model

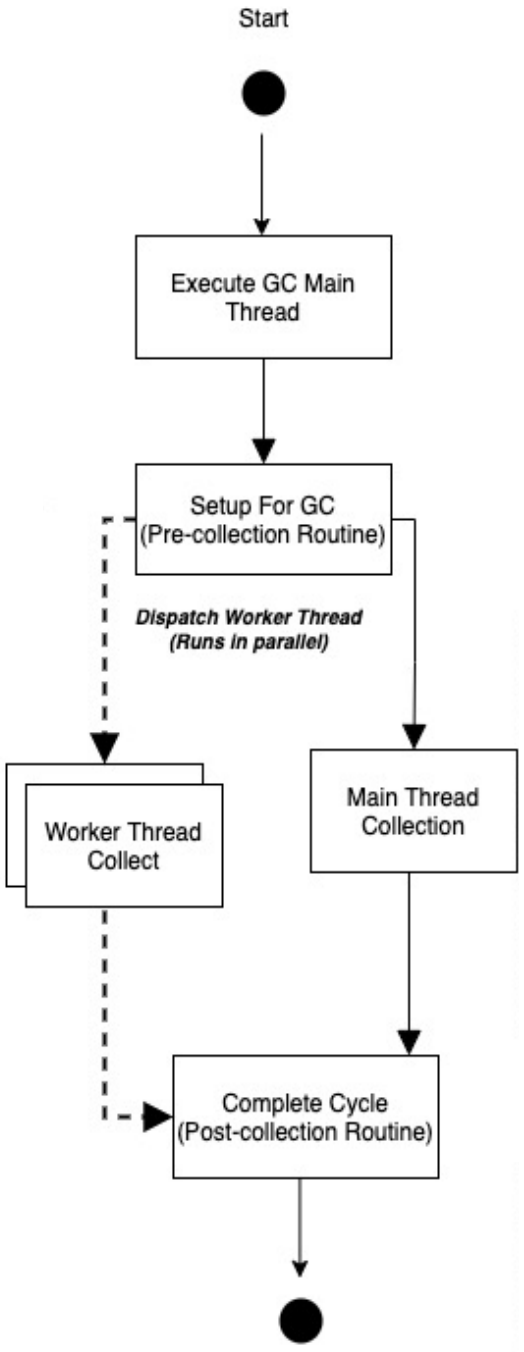
OMR

Current Working Threads (n)								
% Stall	**2**	**4**	**12**	**18**	**24**	**36**	**48**	**64**
Recommended Threads								
99%	1	3	7	10	14	20	27	36
95%	2	3	8	11	15	22	30	40
90%	2	3	8	12	16	24	32	42
85%	2	3	9	13	17	26	34	46
80%	2	3	9	14	18	27	36	48
75%	2	4	10	15	19	29	38	51
70%	2	4	10	15	20	30	40	53
65%	2	4	11	16	21	32	42	56
60%	2	4	11	17	22	33	44	58
55%	2	4	12	17	23	35	46	61
50%	2	4	12	18	24	36	48	64
45%	2	5	13	19	26	38	51	64
40%	3	5	14	20	27	40	54	64
35%	3	5	15	22	29	43	57	64
30%	3	5	16	23	31	46	61	64
25%	3	6	17	25	33	50	64	64
20%	3	6	18	27	36	54	64	64
15%	4	7	21	31	41	61	64	64
10%	4	8	24	36	48	64	64	64
5%	6	11	33	49	64	64	64	64
1%	11	22	64	64	64	64	64	64

Dynamic Threading Matrix Of Inputs and Resulting Output ($W = 50\%$, $X = 1$ & $H = 0.85$)

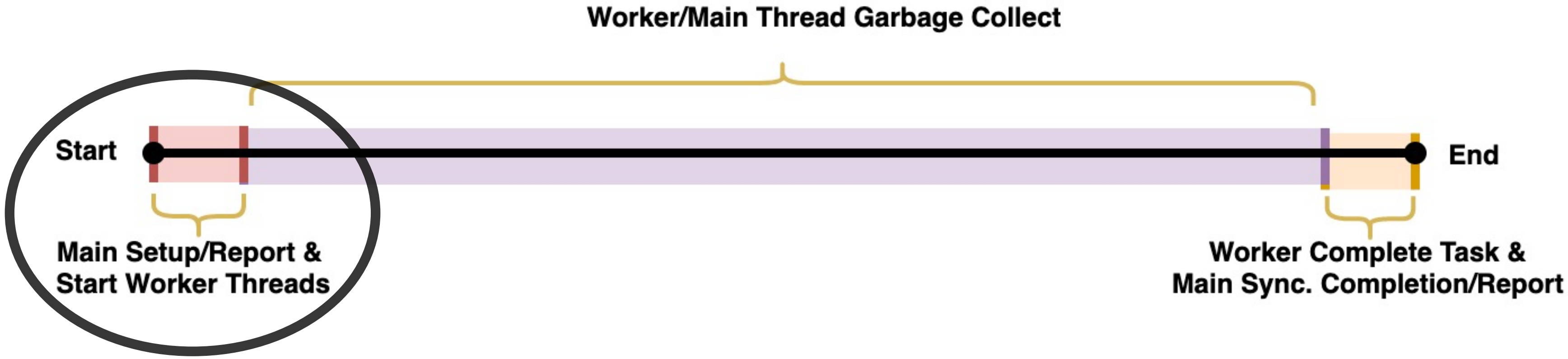
OpenJ9

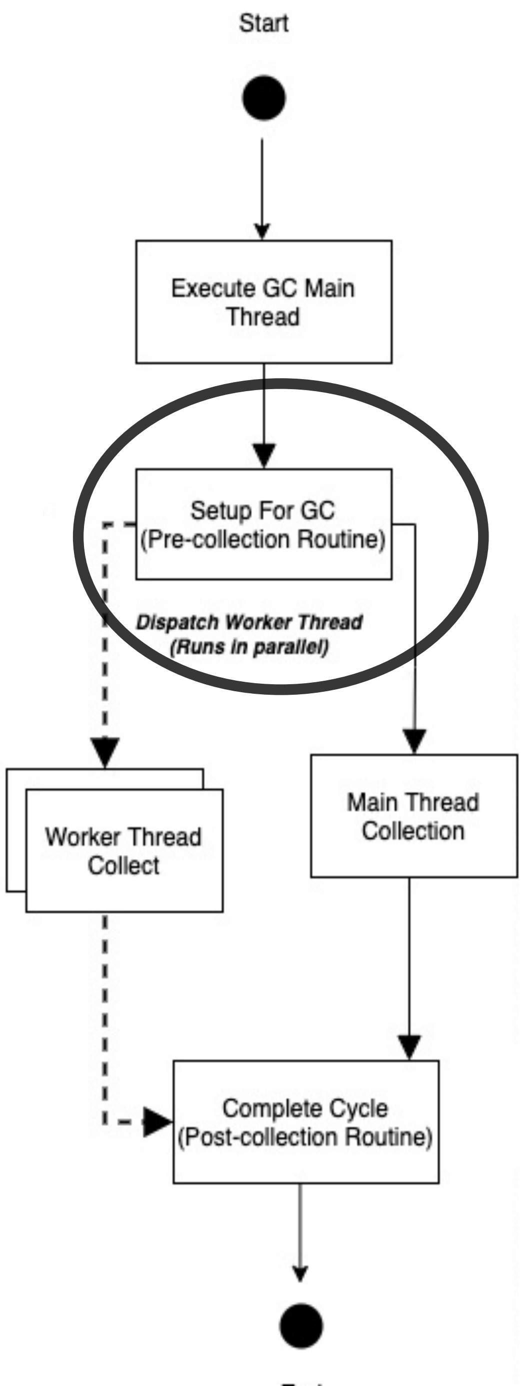

Worker/Main Thread Garbage Collect
Start
End
Main Setup/Report &
Start Worker Threads
Worker Complete Task &
Main Sync. Completion/Report


Start
Execute GC Main Thread
Setup For GC (Pre-collection Routine)
Dispatch Worker Thread (Runs in parallel)
Worker Thread Collect
Main Thread Collection
Complete Cycle (Post-collection Routine)

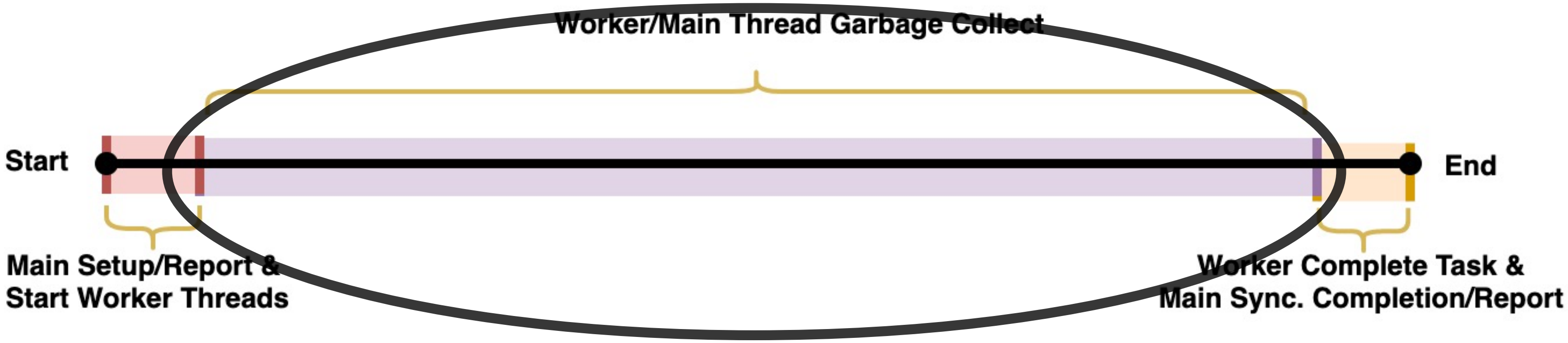
OMR

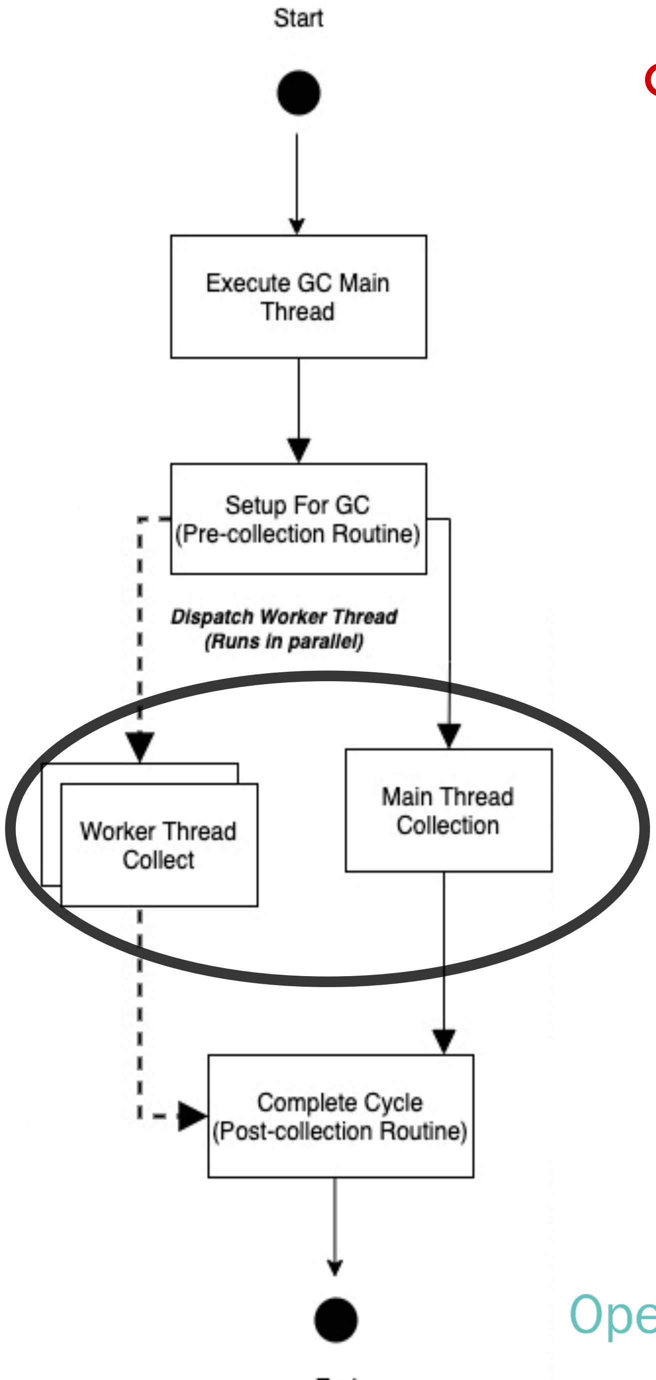
OpenJ9


Worker/Main Thread Garbage Collect
Start
End
Main Setup/Report &
Start Worker Threads
Worker Complete Task &
Main Sync. Completion/Report

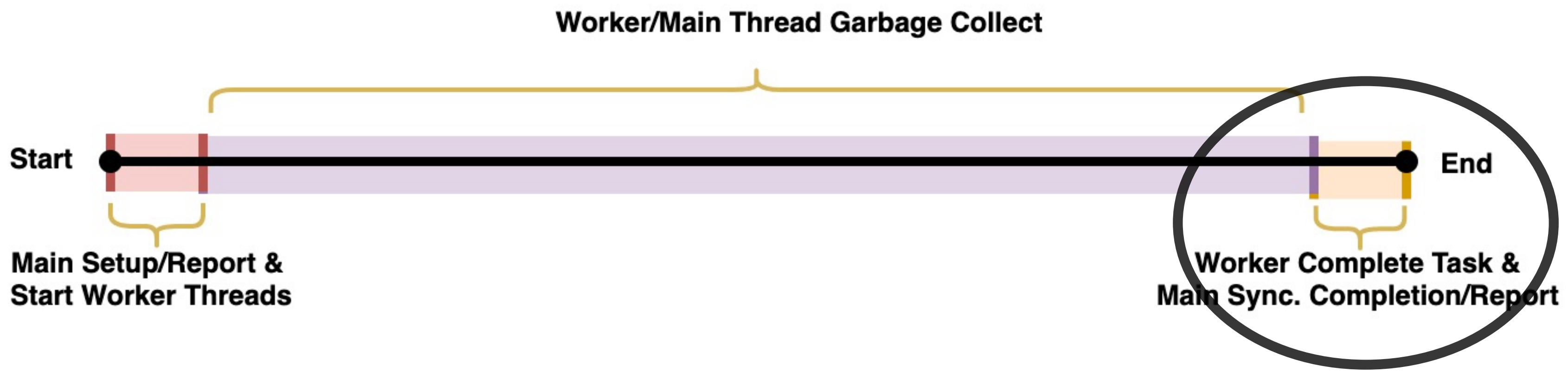

Start
Execute GC Main Thread
Setup For GC (Pre-collection Routine)
Dispatch Worker Thread (Runs in parallel)
Worker Thread Collect
Main Thread Collection
Complete Cycle (Post-collection Routine)

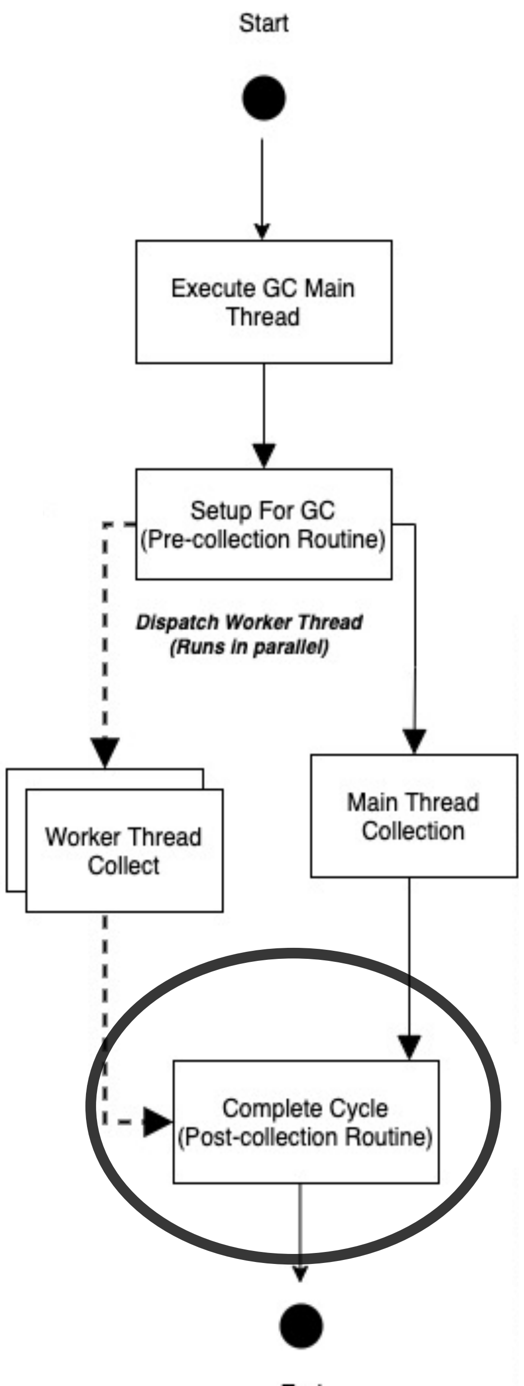
OMR

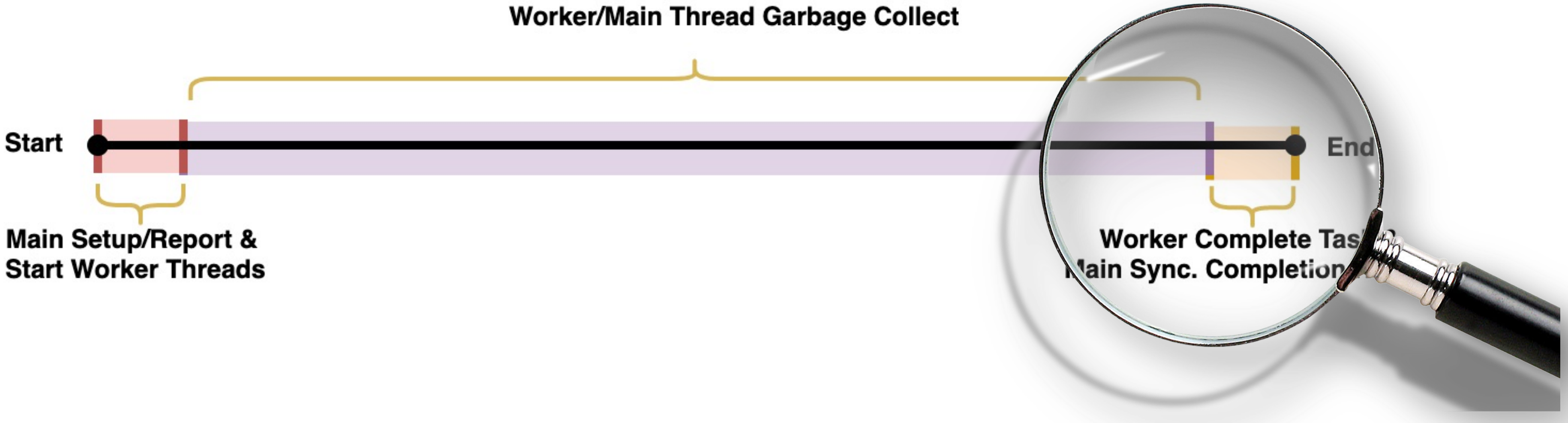

Worker/Main Thread Garbage Collect
Start
End
Main Setup/Report &
Start Worker Threads
Worker Complete Task &
Main Sync. Completion/Report

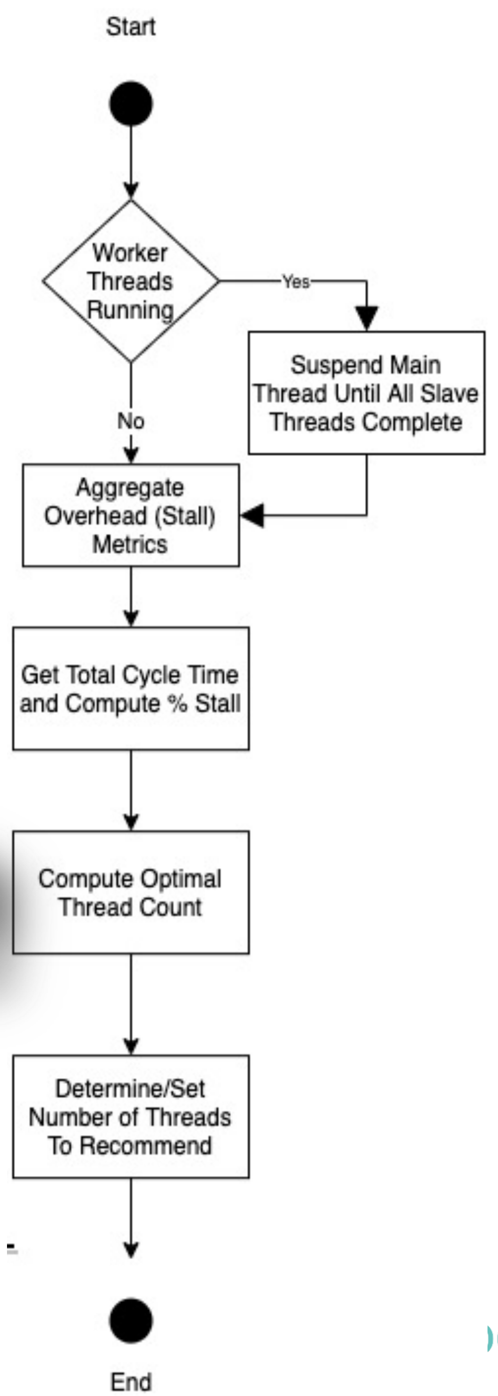

Start
Execute GC Main Thread
Setup For GC (Pre-collection Routine)
Dispatch Worker Thread (Runs in parallel)
Worker Thread Collect
Main Thread Collection
Complete Cycle (Post-collection Routine)

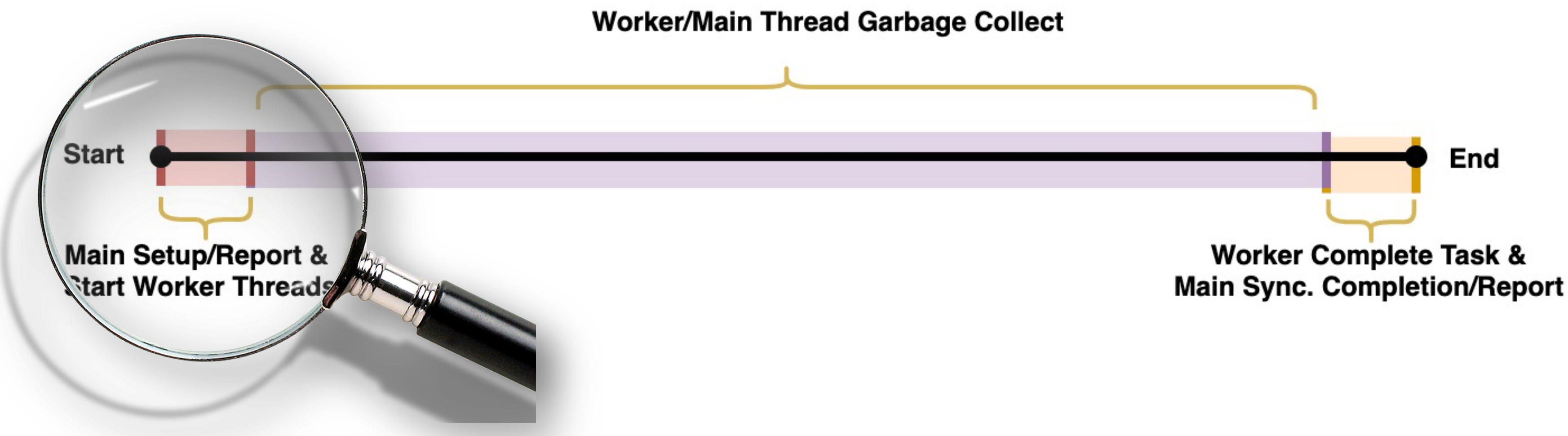
OpenJ9

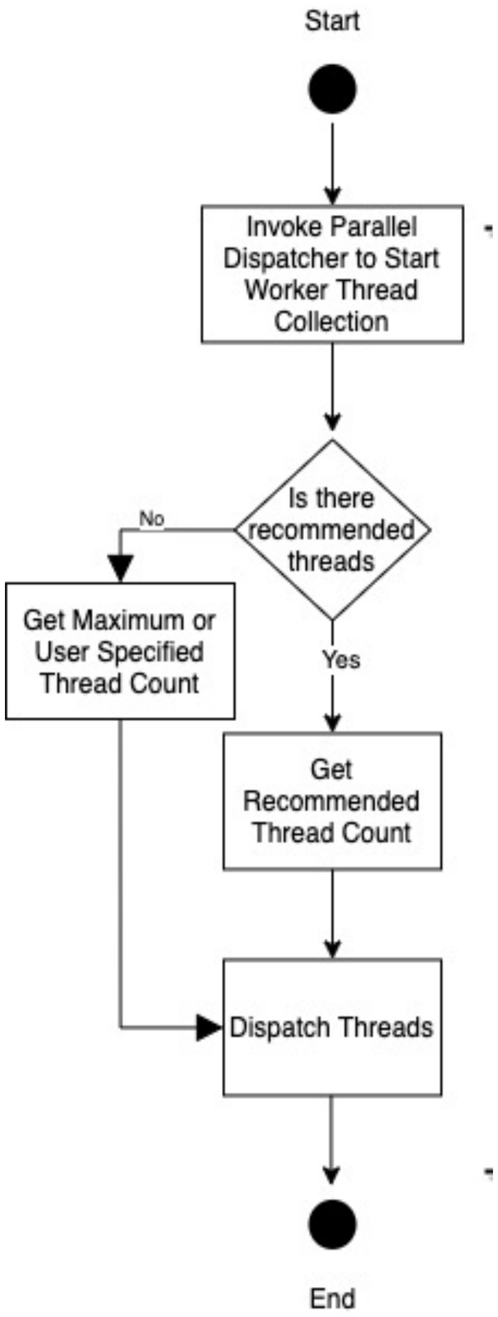

Worker/Main Thread Garbage Collect
Start
End
Main Setup/Report &
Start Worker Threads
Worker Complete Task &
Main Sync. Completion/Report


Start
Execute GC Main Thread
Setup For GC (Pre-collection Routine)
Dispatch Worker Thread (Runs in parallel)
Worker Thread Collect
Main Thread Collection
Complete Cycle (Post-collection Routine)


Worker/Main Thread Garbage Collect
Start
End
Main Setup/Report &
Start Worker Threads


Start
Worker Threads Running
Yes
Suspend Main Thread Until All Slave Threads Complete
No
Aggregate Overhead (Stall) Metrics
Get Total Cycle Time and Compute % Stall
Compute Optimal Thread Count
Determine/Set Number of Threads To Recommend
End

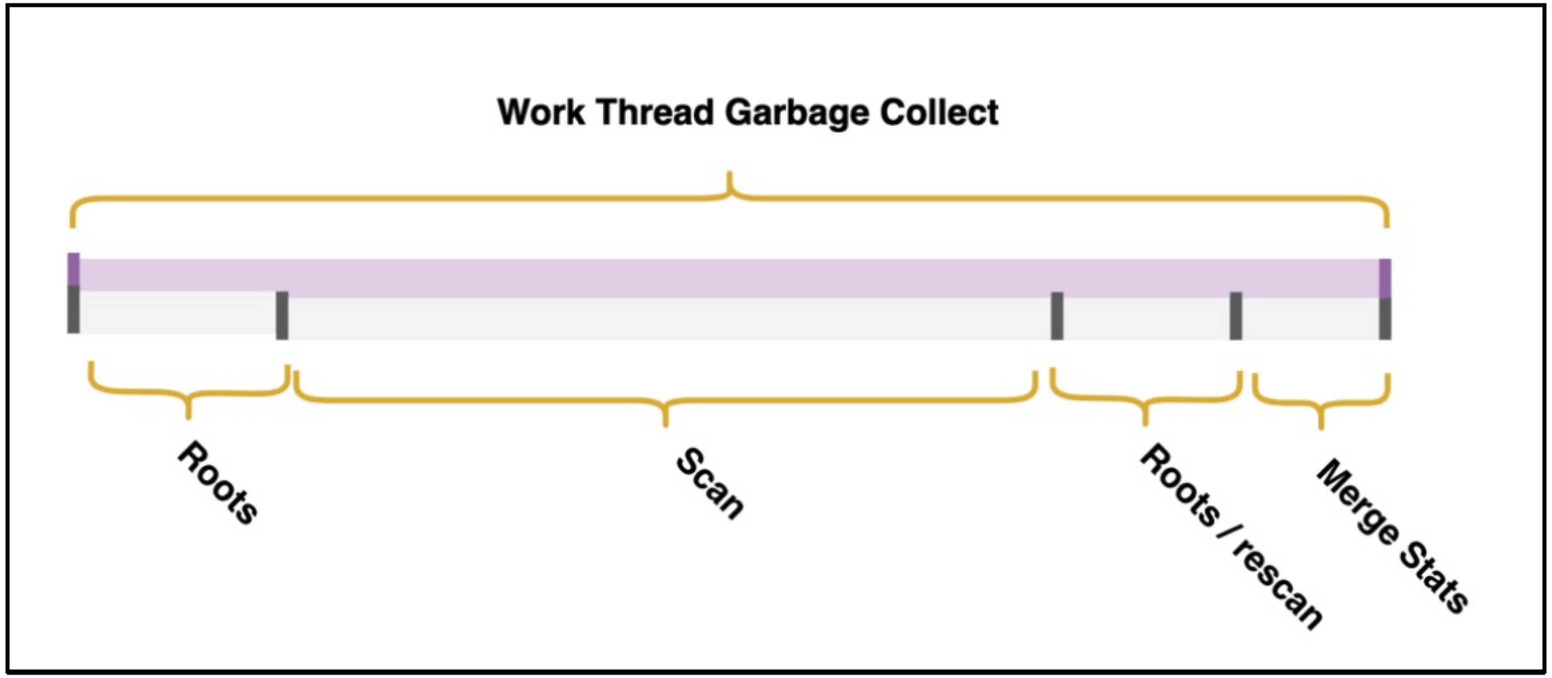

Worker/Main Thread Garbage Collect
Start
End
Main Setup/Report &
Start Worker Threads
Worker Complete Task &
Main Sync. Completion/Report

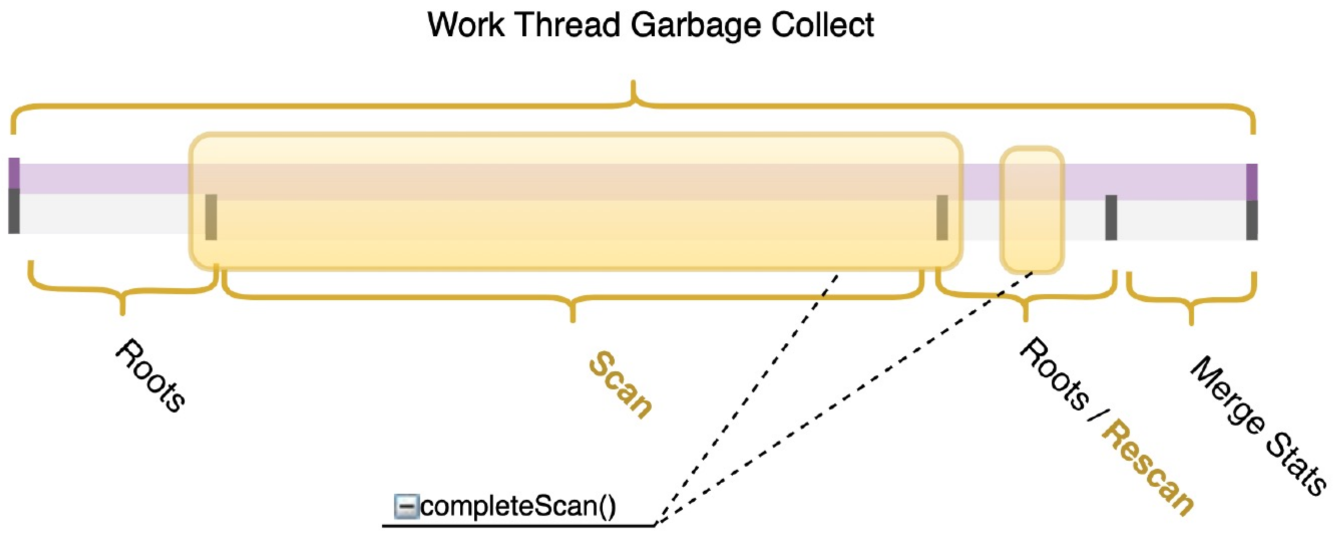

Start
Invoke Parallel Dispatcher to Start Worker Thread Collection
Is there recommended threads
No
Yes
Get Maximum or User Specified Thread Count
Get Recommended Thread Count
Dispatch Threads
End

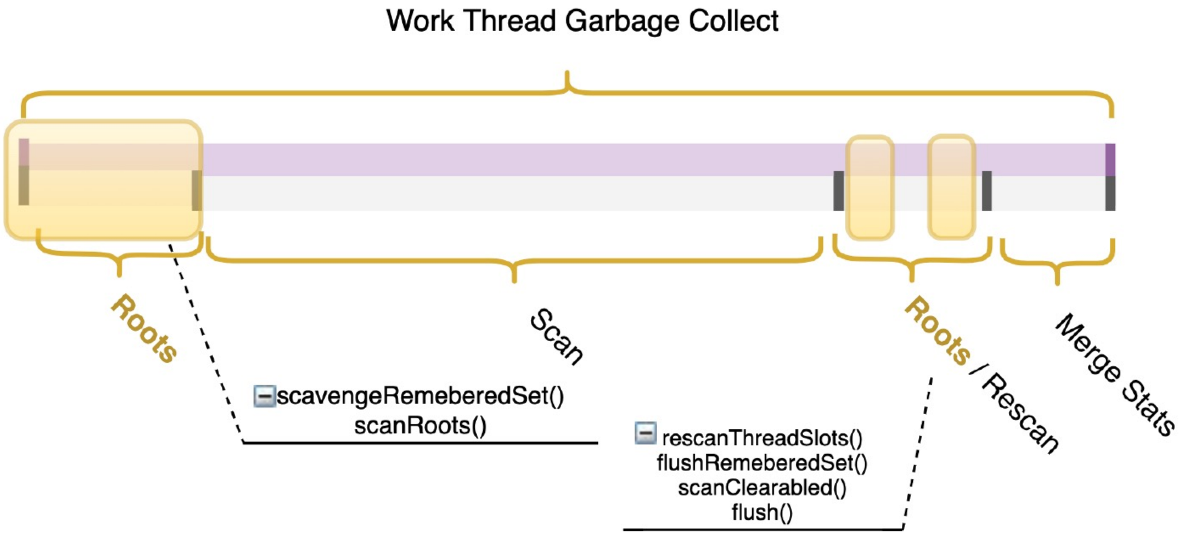
OMR

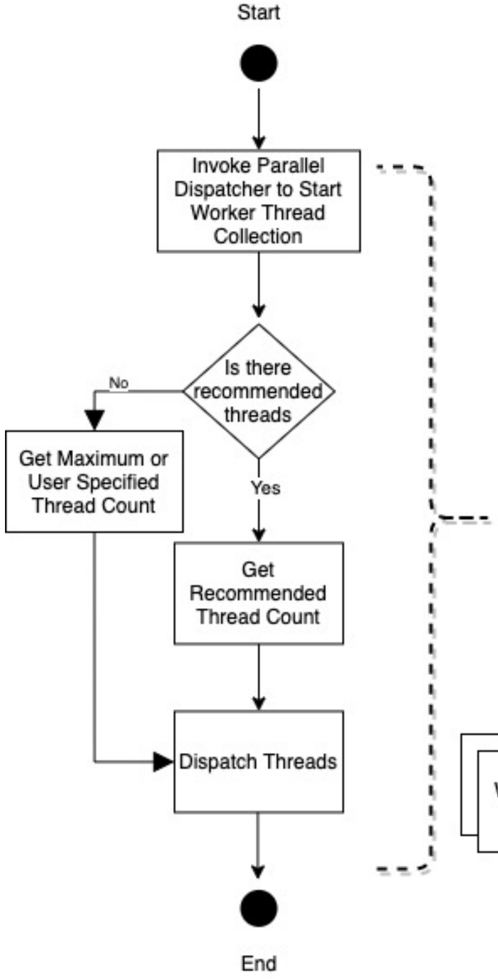

Worker/Main Thread Garbage Collect
Start
End
Main Setup/Report &
Start Worker Threads
Worker Complete Task &
Main Sync. Completion/Report

OpenJ9

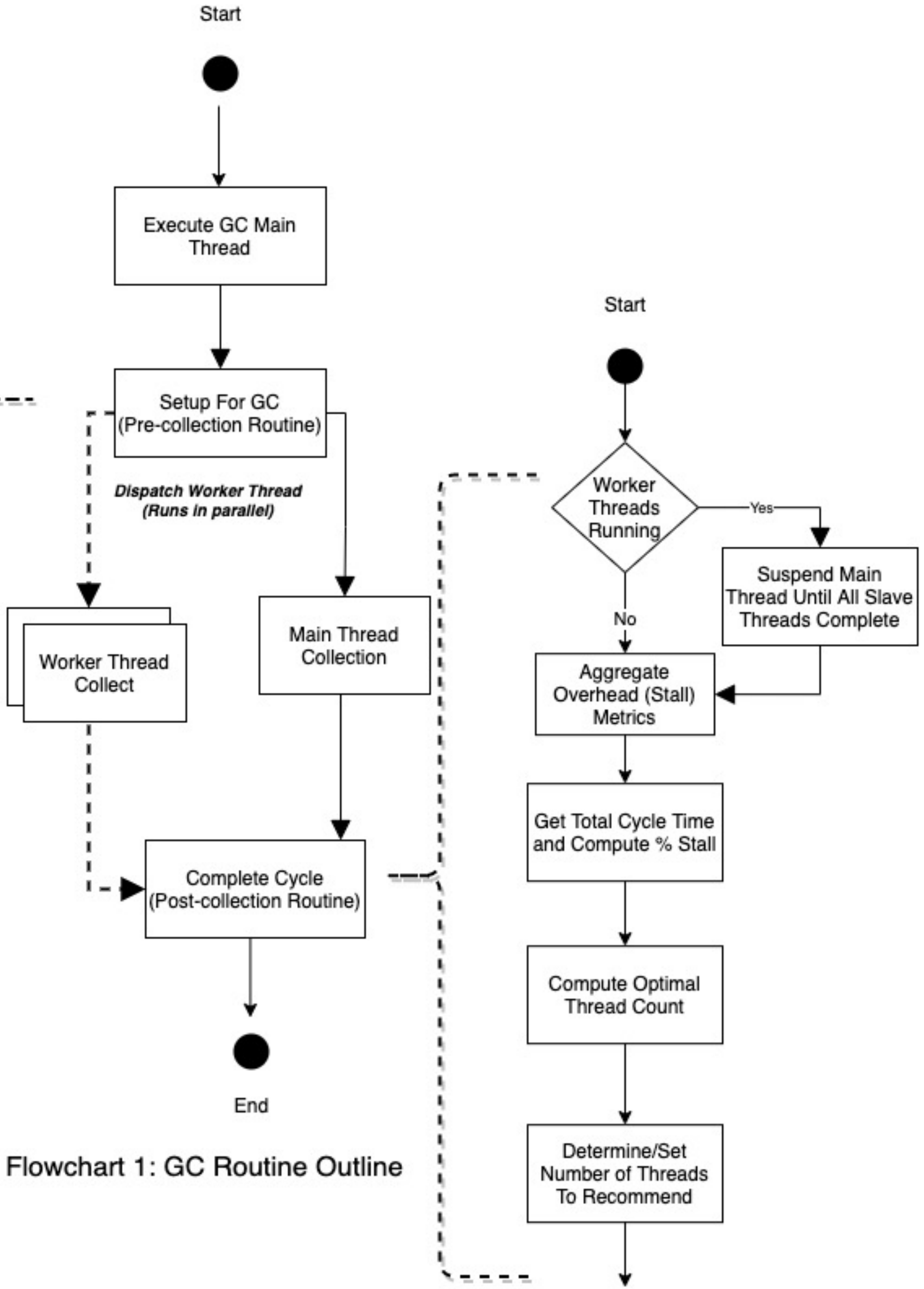

Work Thread Garbage Collect
Roots
Scan
Roots / rescan
Merge Stats


Work Thread Garbage Collect
Roots
Scan
Roots / Rescan
Merge Stats
completeScan()


Work Thread Garbage Collect
Roots
Scan
Roots / Rescan
Merge Stats
scavengeRemeberedSet()
scanRoots()
rescanThreadSlots()
flushRemeberedSet()
scanClearabled()
flush()



Flowchart 2: GC Pre-collection Routine for Adaptive Threading



Flowchart 1: GC Routine Outline

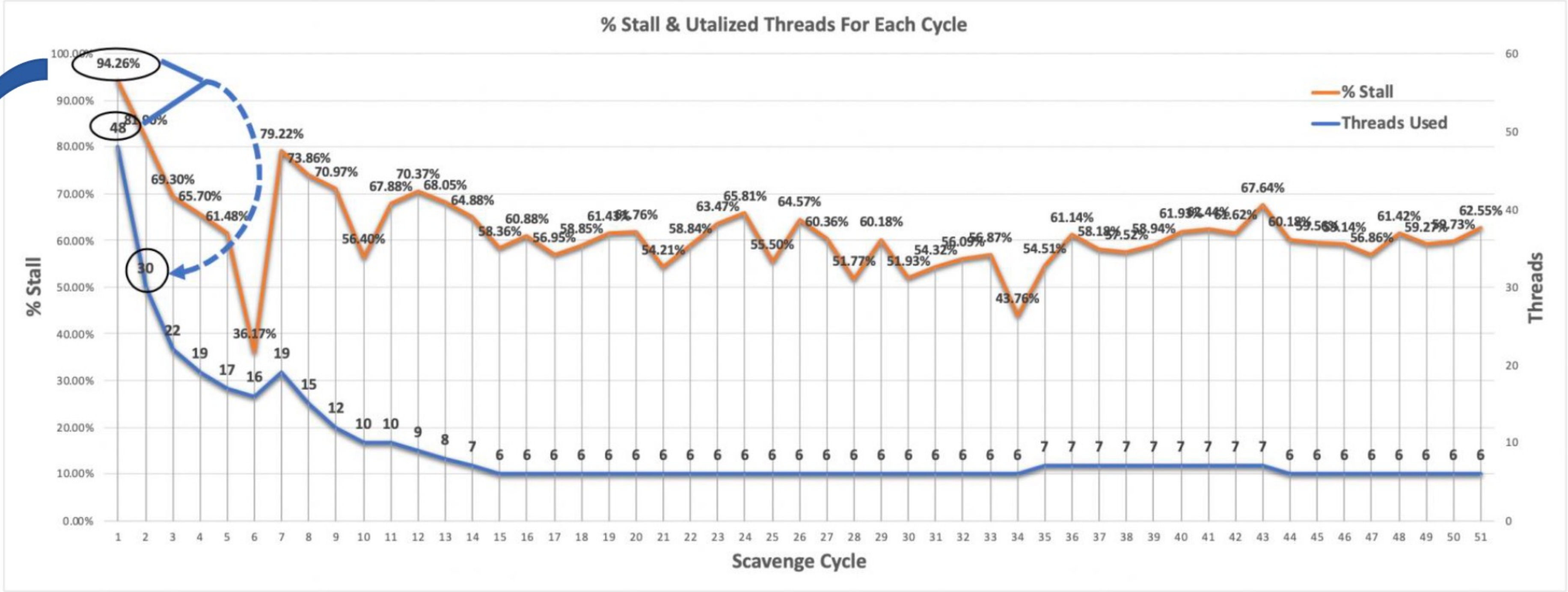
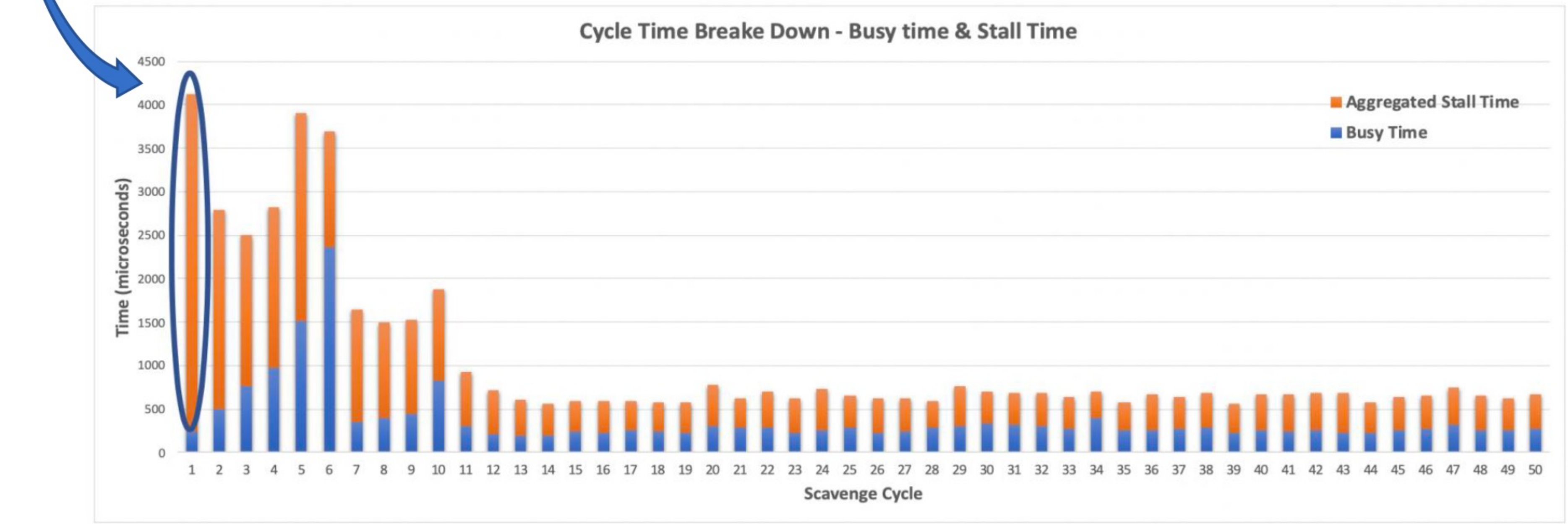


Figure 3



OMR

OpenJ9

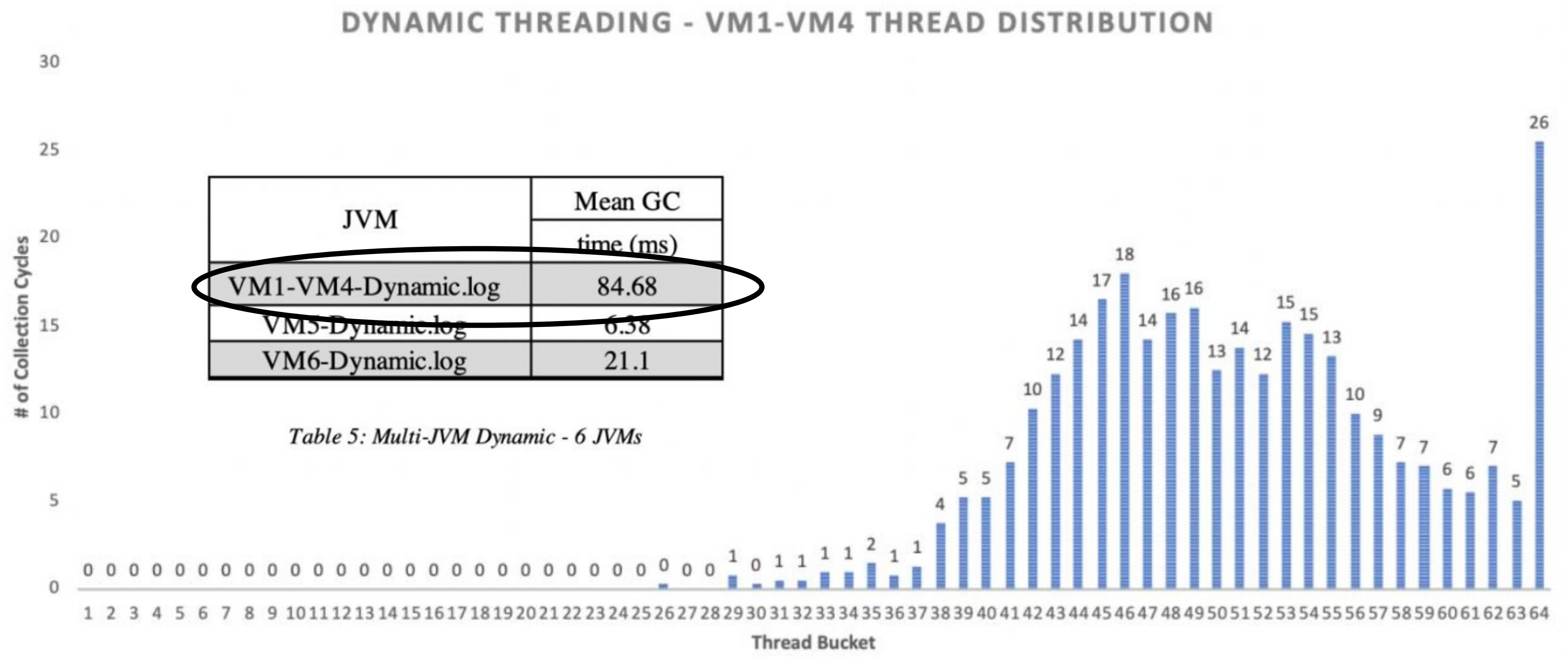
Adaptive Threading

JVM	Mean GC time (ms)
VM1-VM4-Baseline	91.63
VM5-Baseline.log	14.4
VM6-Baseline.log	22.1

Table 4: Multi-JVM Baseline - 6 JVMs

JVM	Mean GC time (ms)
VM1-VM4-Dynamic.log	84.68
VM5-Dynamic.log	6.38
VM6-Dynamic.log	21.1

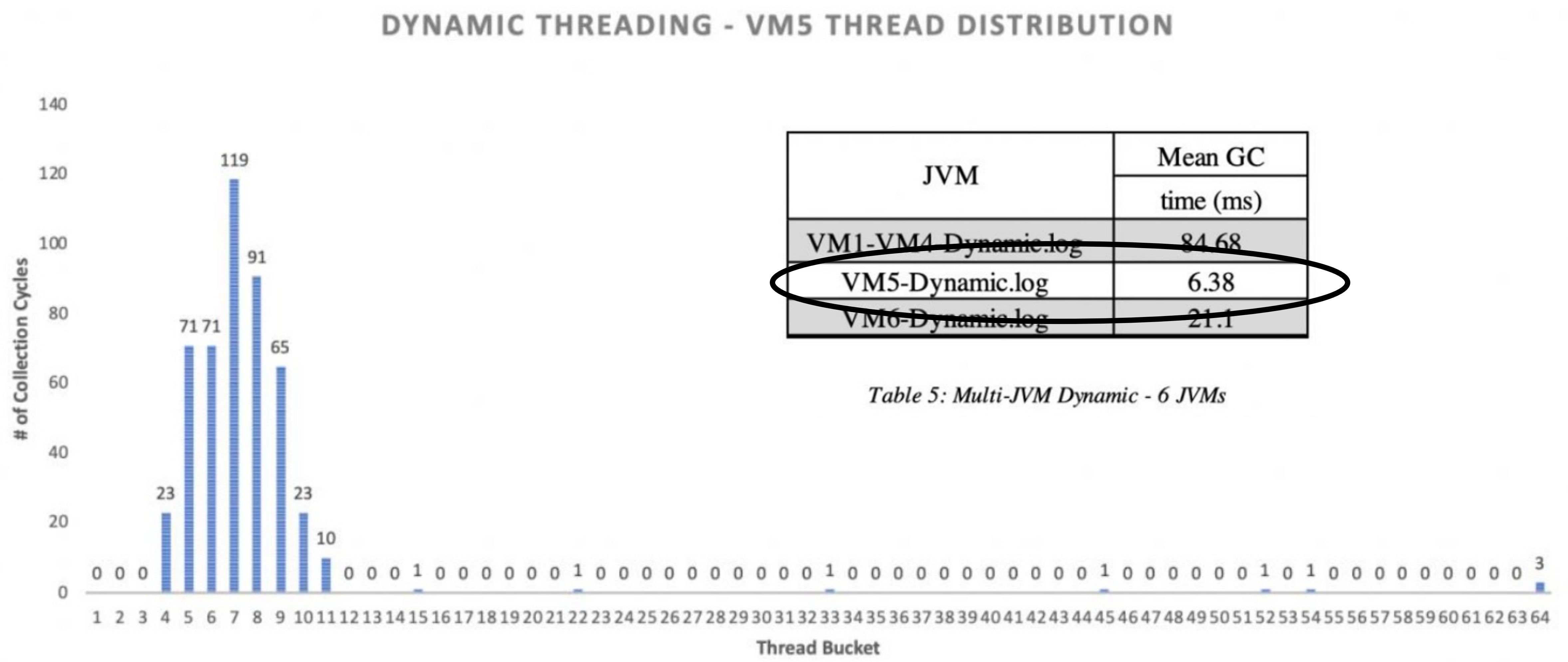
Table 5: Multi-JVM Dynamic - 6 JVMs



JVM	Mean GC time (ms)
VM1-VM4-Dynamic.log	84.68
VM5-Dynamic.log	6.38
VM6-Dynamic.log	21.1

Table 5: Multi-JVM Dynamic - 6 JVMs

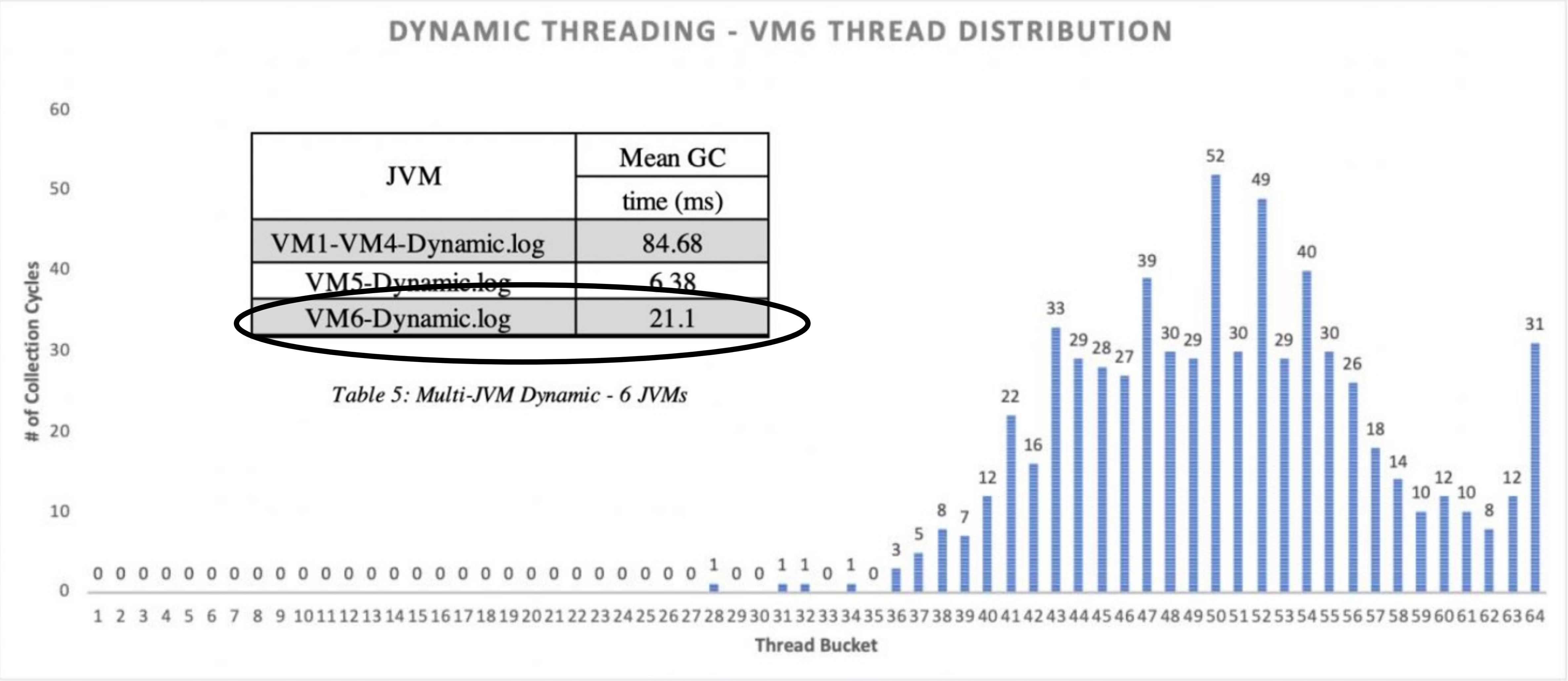
Figure 6: VM1-VM4 Thread Distribution



JVM	Mean GC time (ms)
VM1-VM4-Dynamic.log	84.68
VM5-Dynamic.log	6.38
VM6-Dynamic.log	21.1

Table 5: Multi-JVM Dynamic - 6 JVMs

Figure 7: VM5 Thread Distribution



JVM	Mean GC time (ms)
VM1-VM4-Dynamic.log	84.68
VM5-Dynamic.log	6.38
VM6-Dynamic.log	21.1

Table 5: Multi-JVM Dynamic - 6 JVMs

Figure 8: VM6 Thread Distribution

OMR

OpenJ9

Future Work

Thank You