Hierarchical Real Time Garbage Collection

Filip Pizlo, Antony Hosking, Jan Vitek
(Purdue & MSR, Purdue, Purdue & TJ Watson)

LCTES 2007 San Diego, CA

- Real Time Java (RTJ) is a growing technology for developing robust, mission-critical, hard real-time systems.

- Real Time Java (RTJ) is a growing technology for developing robust, mission-critical, hard real-time systems.

- Programming for RTJ is still made hard by memory management:

- Real Time Java (RTJ) is a growing technology for developing robust, mission-critical, hard real-time systems.
- Programming for RTJ is still made hard by memory management:
  - Java programmers are accustomed to garbage collection.

- Real Time Java (RTJ) is a growing technology for developing robust, mission-critical, hard real-time systems.

- Programming for RTJ is still made hard by memory management:

  - Java programmers are accustomed to garbage collection.

  - We would like to use real-time garbage collection (RTGC) - but sometimes performance is not good enough.

- Real Time Java (RTJ) is a growing technology for developing robust, mission-critical, hard real-time systems.

- Programming for RTJ is still made hard by memory management:

  - Java programmers are accustomed to garbage collection.

- We would like to use real-time garbage collection (RTGC) - but sometimes performance is not good enough.

- Programmers may be forced to use some form of manual memory management instead (scoped memory, object pools, eventrons, reflexes).

- RTGC introduction:

- RTGC introduction:
  - Real-time garbage collectors are designed to maintain a predictable schedule, minimize pause times and maximize utilization.

- RTGC introduction:
  - Real-time garbage collectors are designed to maintain a predictable schedule, minimize pause times and maximize utilization.

Interruptions from the collector are part of the real-time schedule.

- RTGC introduction:
  - Real-time garbage collectors are designed to maintain a predictable schedule, minimize pause times and maximize utilization.

Following interruption, the time before the mutator gets to relinquish control from the collector should be small.

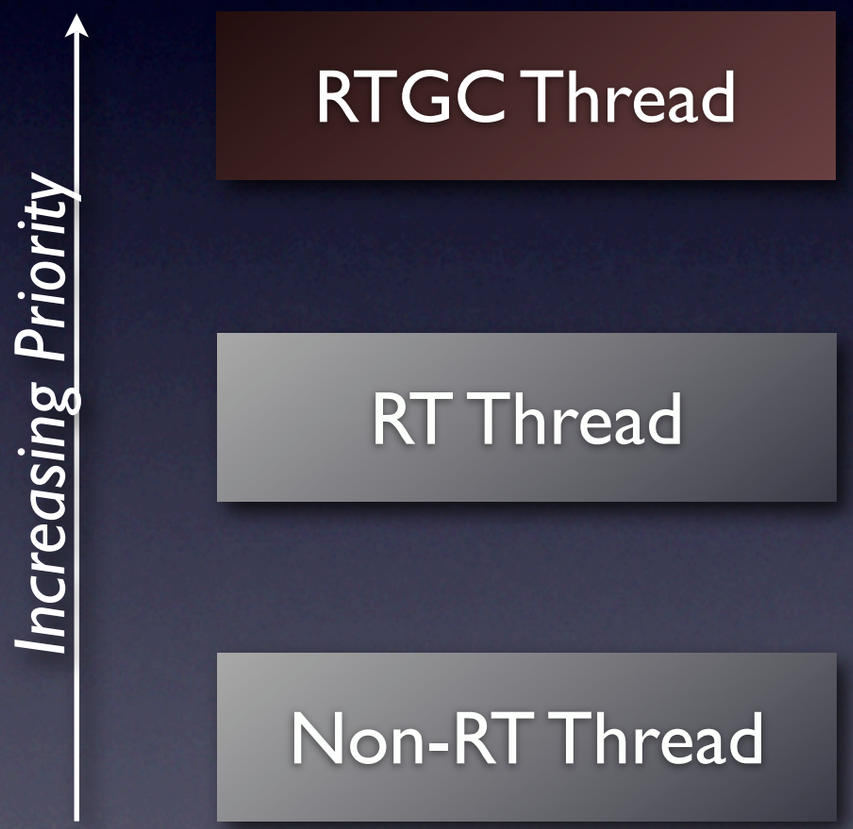
- RTGC introduction:
  - Real-time garbage collectors are designed to maintain a predictable schedule, minimize pause times and maximize utilization.

For a given timeslice, the amount of time that the mutator is *guaranteed* to utilize, is maximized.

- RTGC introduction:
  - Real-time garbage collectors are designed to maintain a predictable schedule, minimize pause times and maximize utilization.
  - *RTGCs are not primarily designed to maximize overall application throughput!*

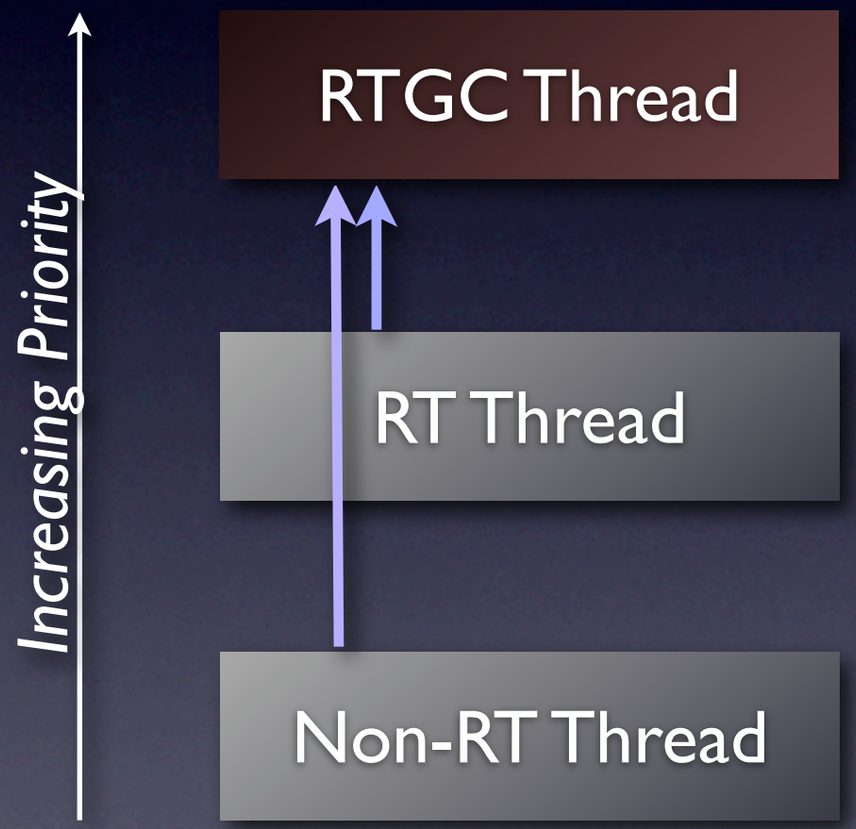
- RTGC introduction:
  - Real-time garbage collectors are designed to maintain a predictable schedule, minimize pause times and maximize utilization.
    - *RTGCs are not primarily designed to maximize overall application throughput!*
  - All RTGCs "interfere" with the mutator by either actively interrupting it (Metronome) or requiring it to occasionally yield (Henriksson).

The problem with “normal” RTGCs.



The problem with “normal” RTGCs.

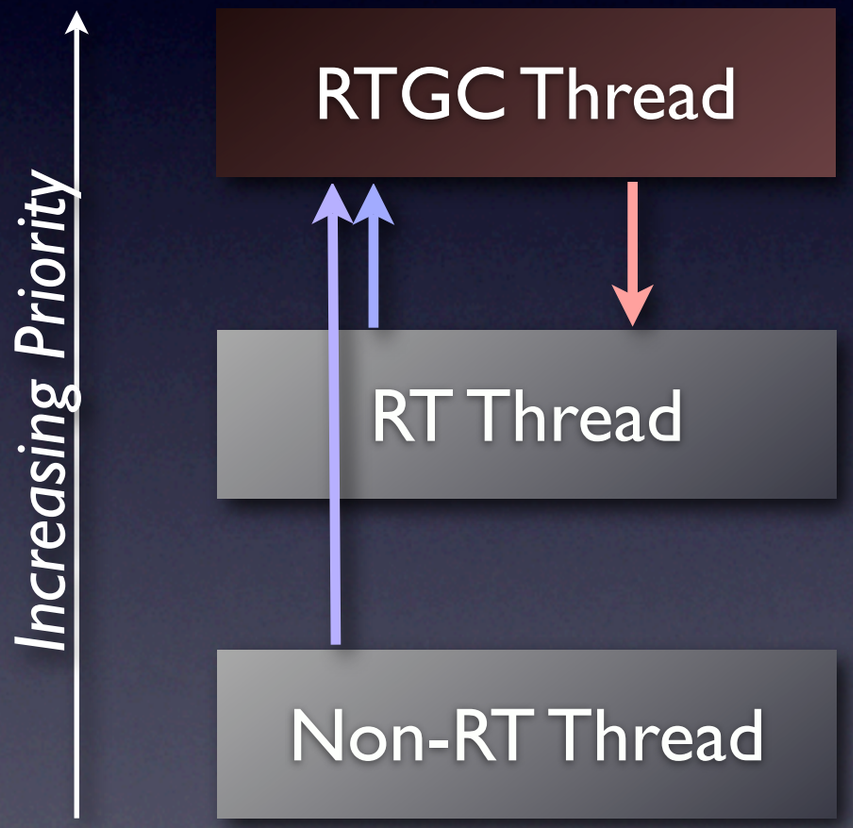
- The amount of interference from the RTGC is determined by the allocation rate of *all threads*, and the size of the *whole heap*.



The problem with "normal" RTGCs.

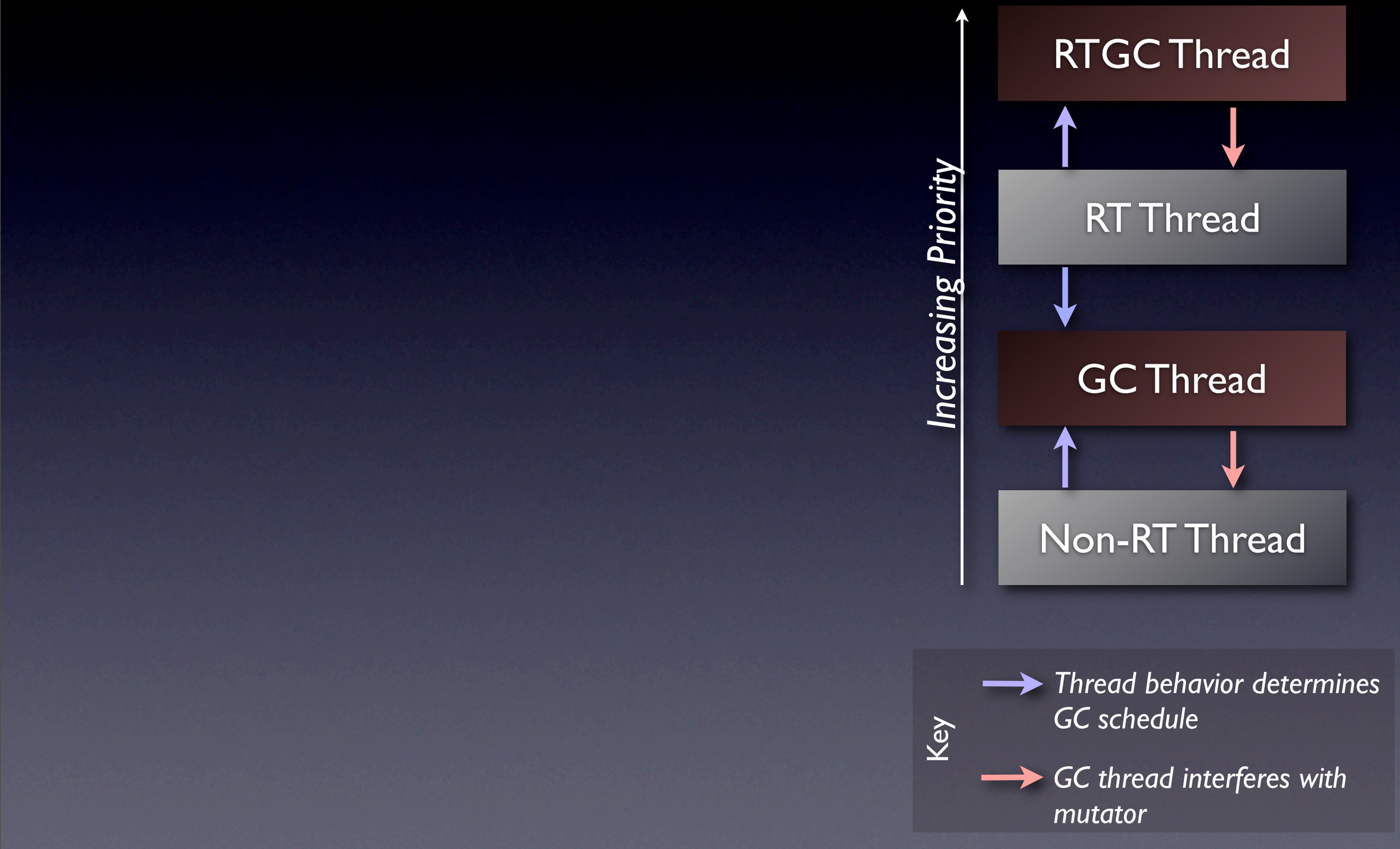
- The amount of interference from the RTGC is determined by the allocation rate of *all threads*, and the size of the *whole heap*.

- This leads to a kind of *priority inversion*: the heap usage of a non-real-time task may cause the GC to interfere with a real-time task.



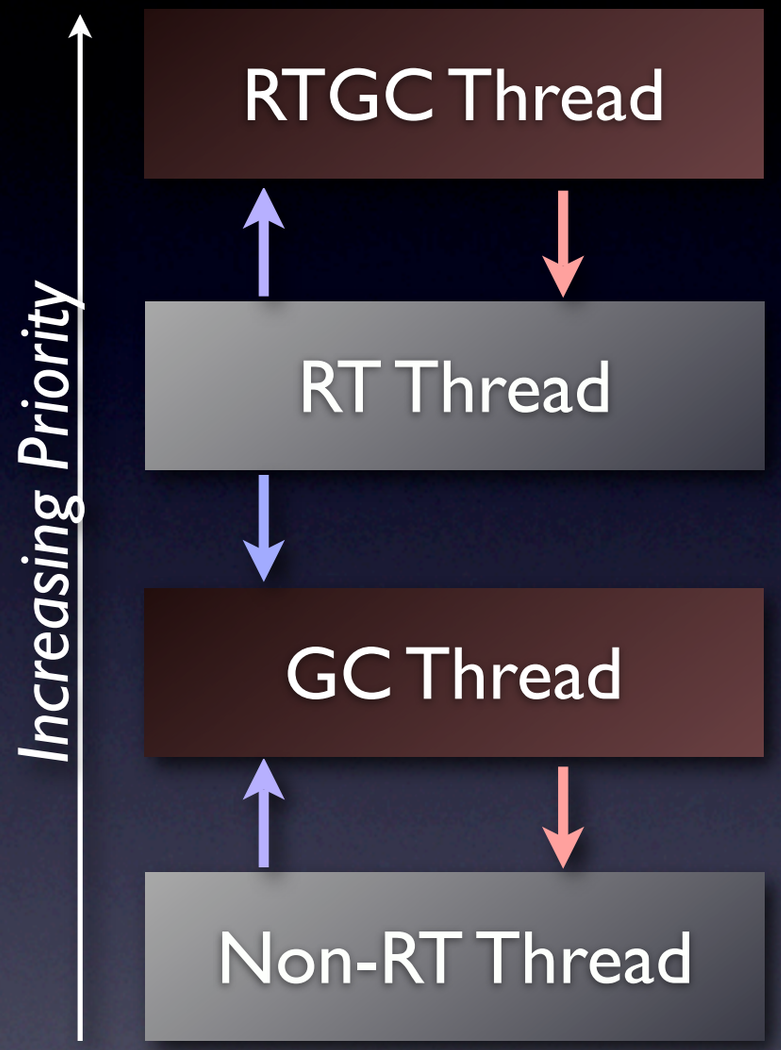
- This problem affects all styles of RTGC (time-based, work-based, Henriksson-style).
- *It can be easily avoided if the part of the heap used by the real-time tasks is segregated from the part used by non-real-time tasks.*

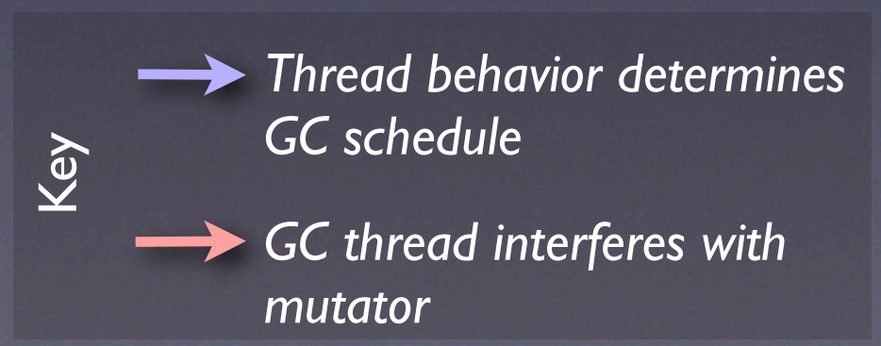
Basic Strategy



Basic Strategy

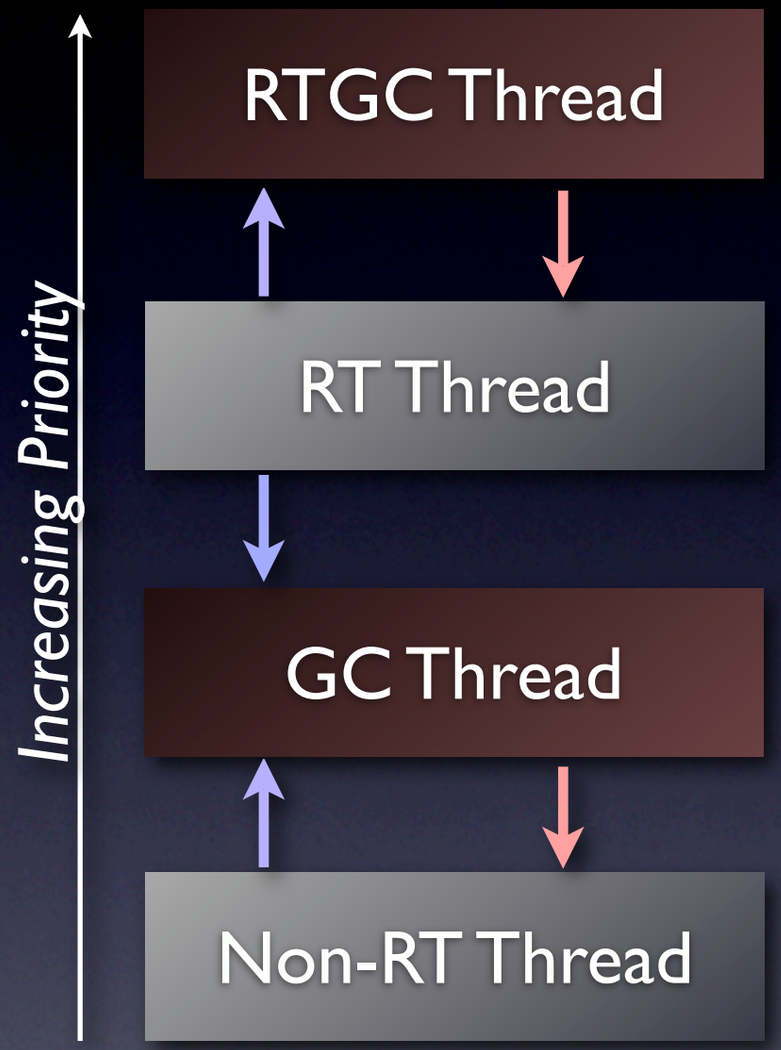
- We segregate the heap into "heaplets".

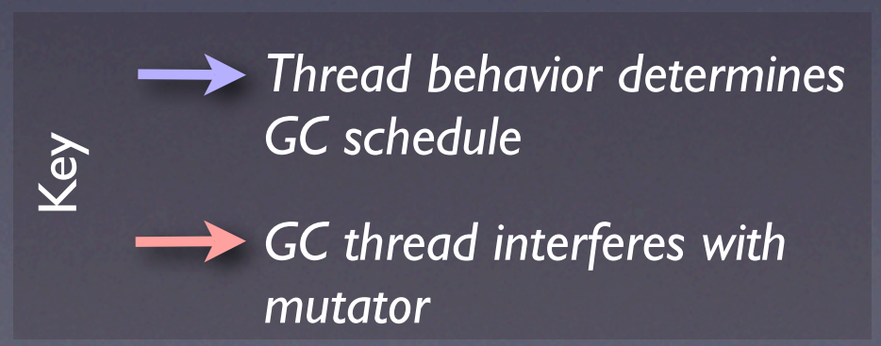




Basic Strategy

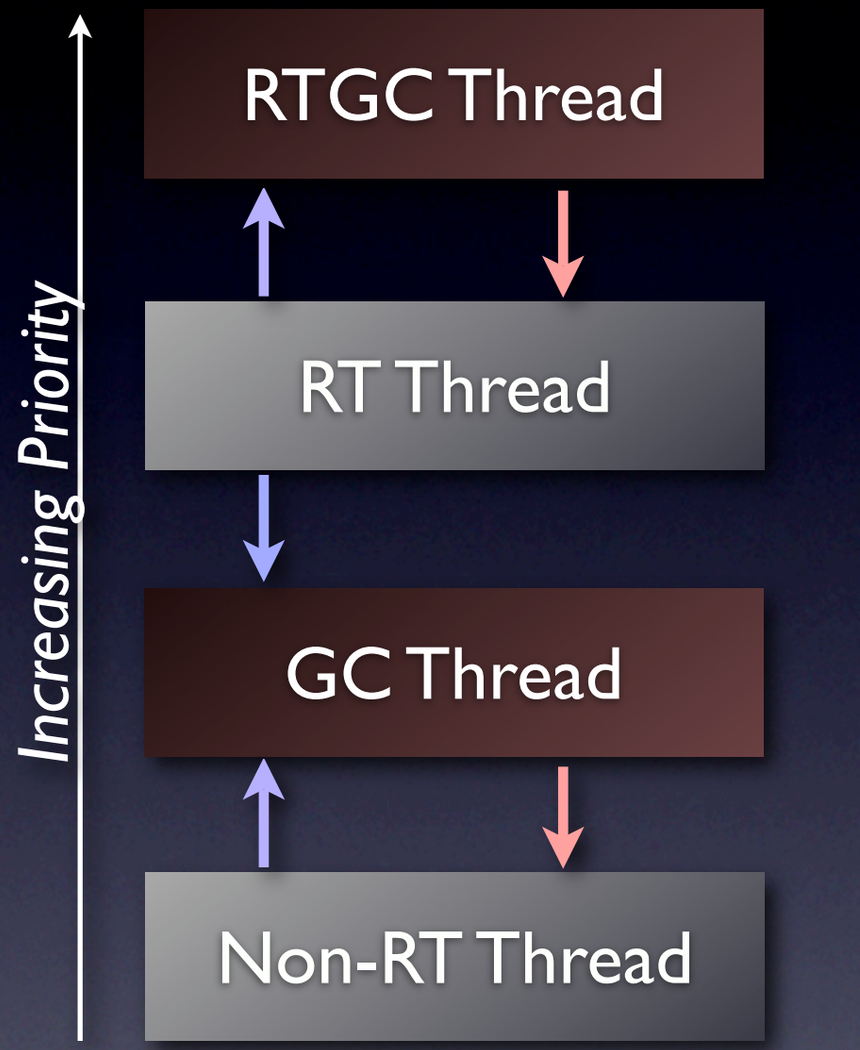
- We segregate the heap into "heaplets".
- Each heaplet gets its own collector thread.





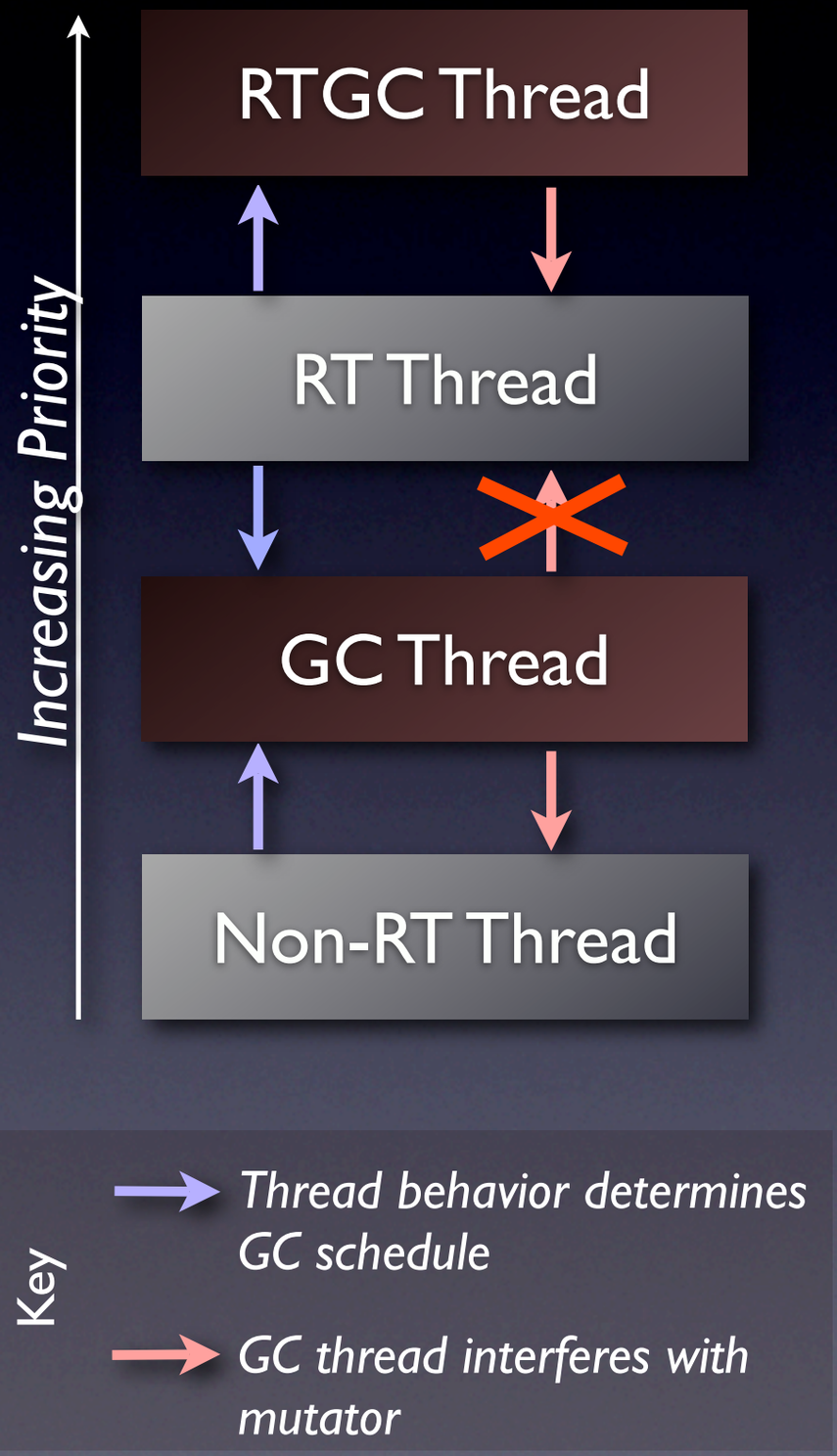
Basic Strategy

- We segregate the heap into "heaplets".
- Each heaplet gets its own collector thread.
- The collector for the non-real-time heaplets never interferes with real-time tasks.



Basic Strategy

- We segregate the heap into "heaplets".
- Each heaplet gets its own collector thread.
- The collector for the non-real-time heaplets never interferes with real-time tasks.
- Thus - real-time code will not be affected by the footprint and allocation behavior of the non-real-time code.



What are heaplets?

- A "heaplet" is a user-specified heap partition, with a user-tuned RTGC thread.
- Any thread may use any heaplet for allocation at any time. The current allocation context is determined using an RTSJ-like API.
- Any thread may have references to objects in any heaplet.
- References between heaplets are allowed.

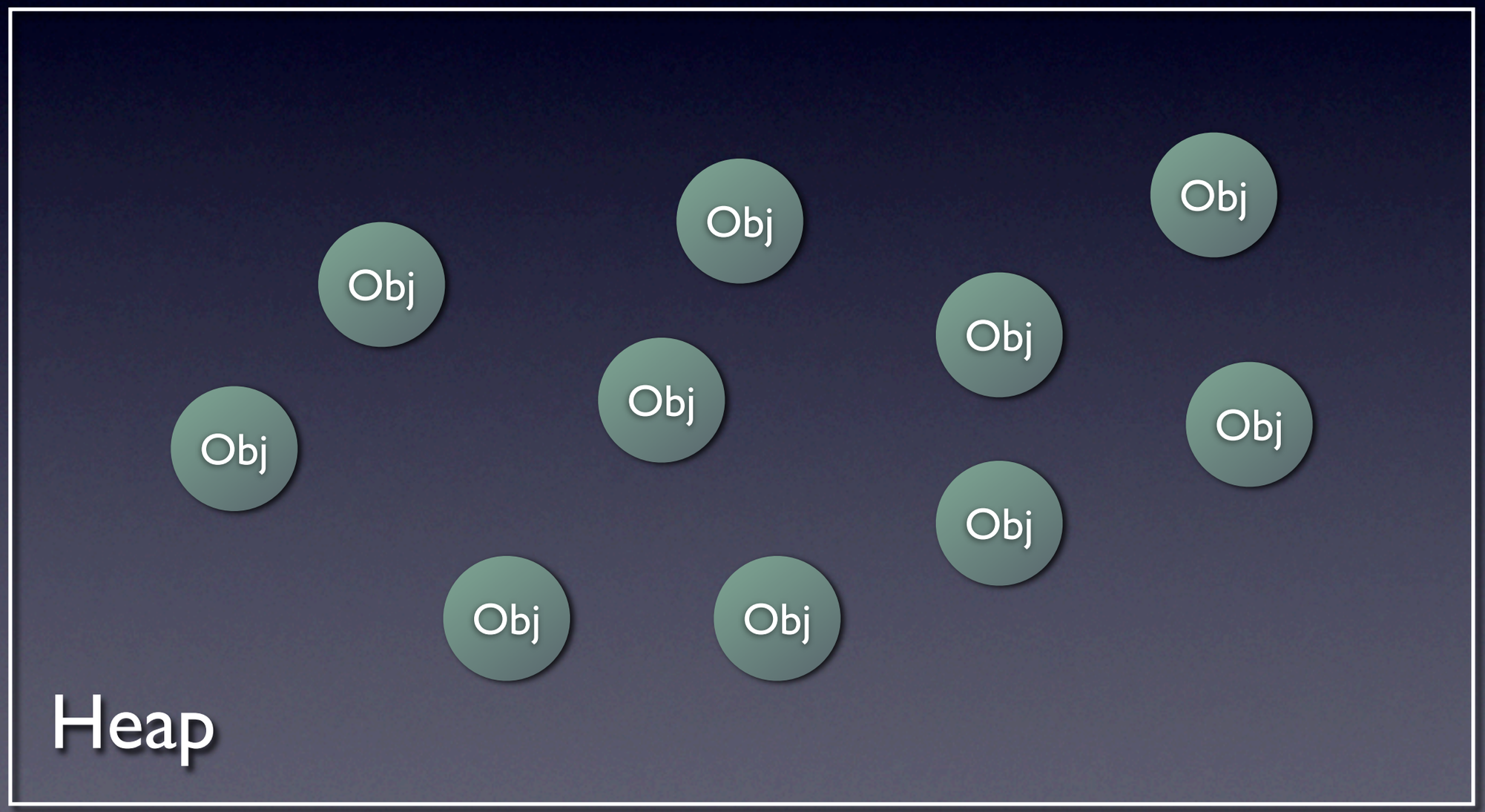
RTGC Example

Thread 1

Thread 2

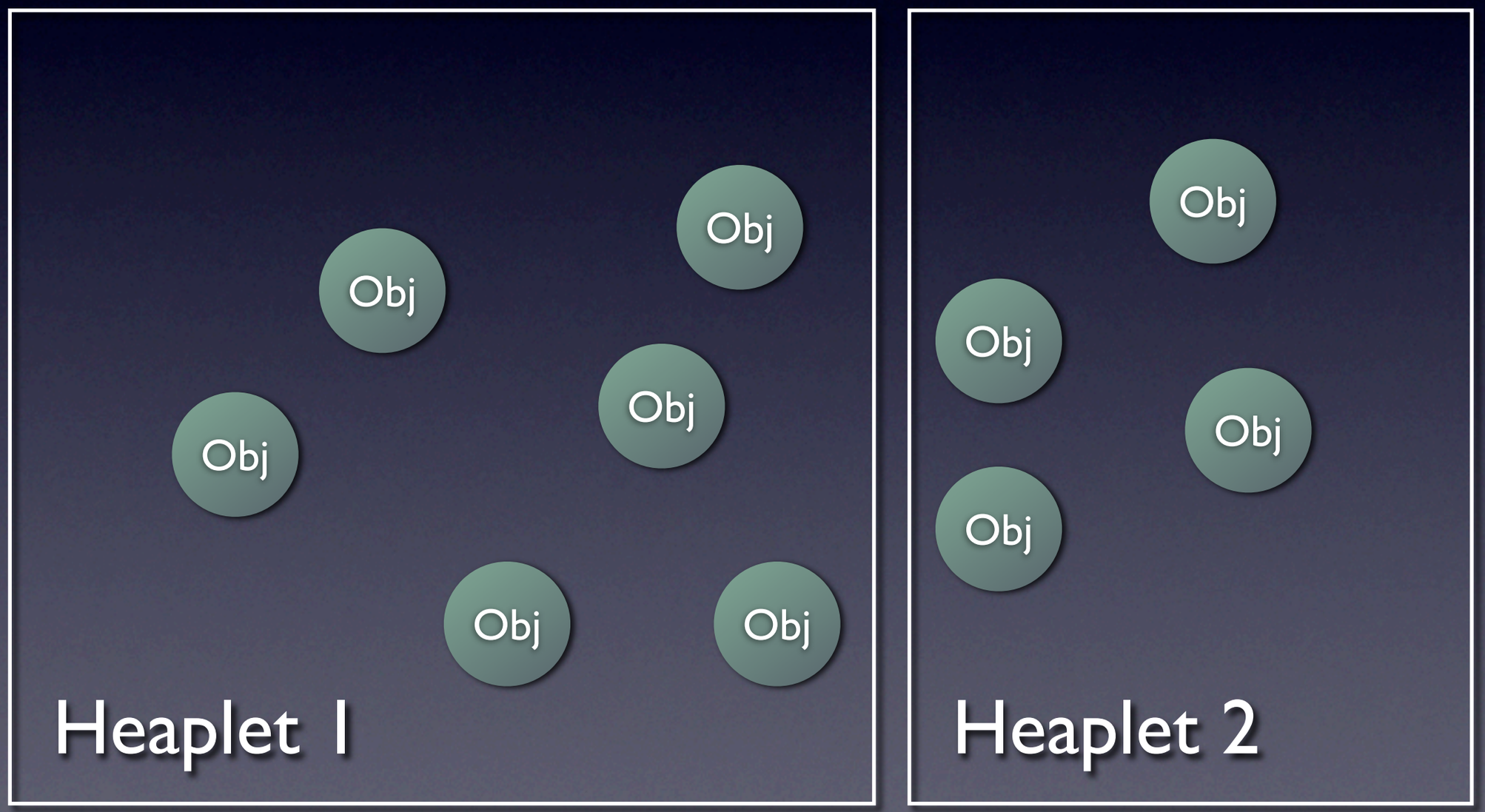
Thread 3

GC Thread

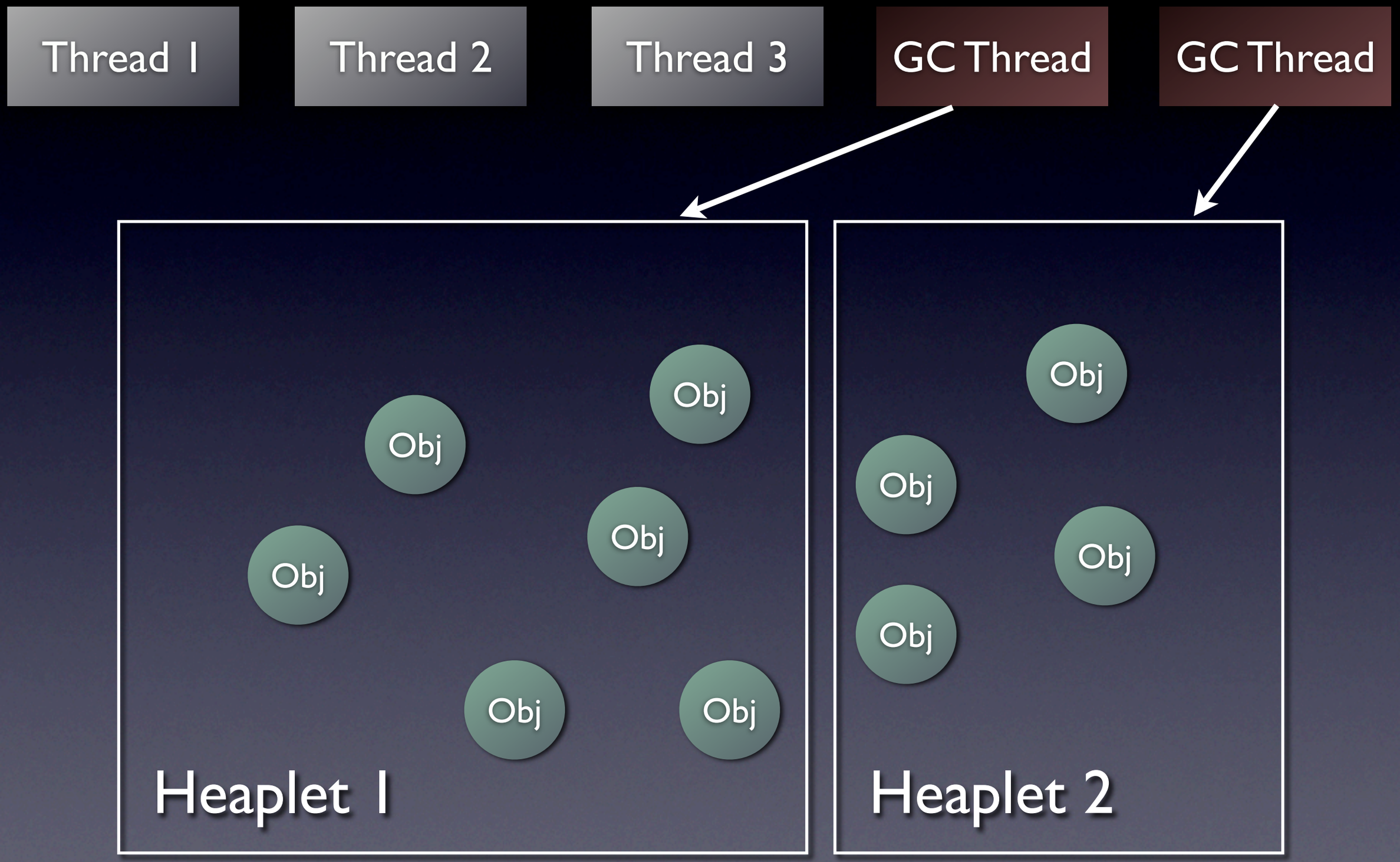


RTGC with Heaplets Example

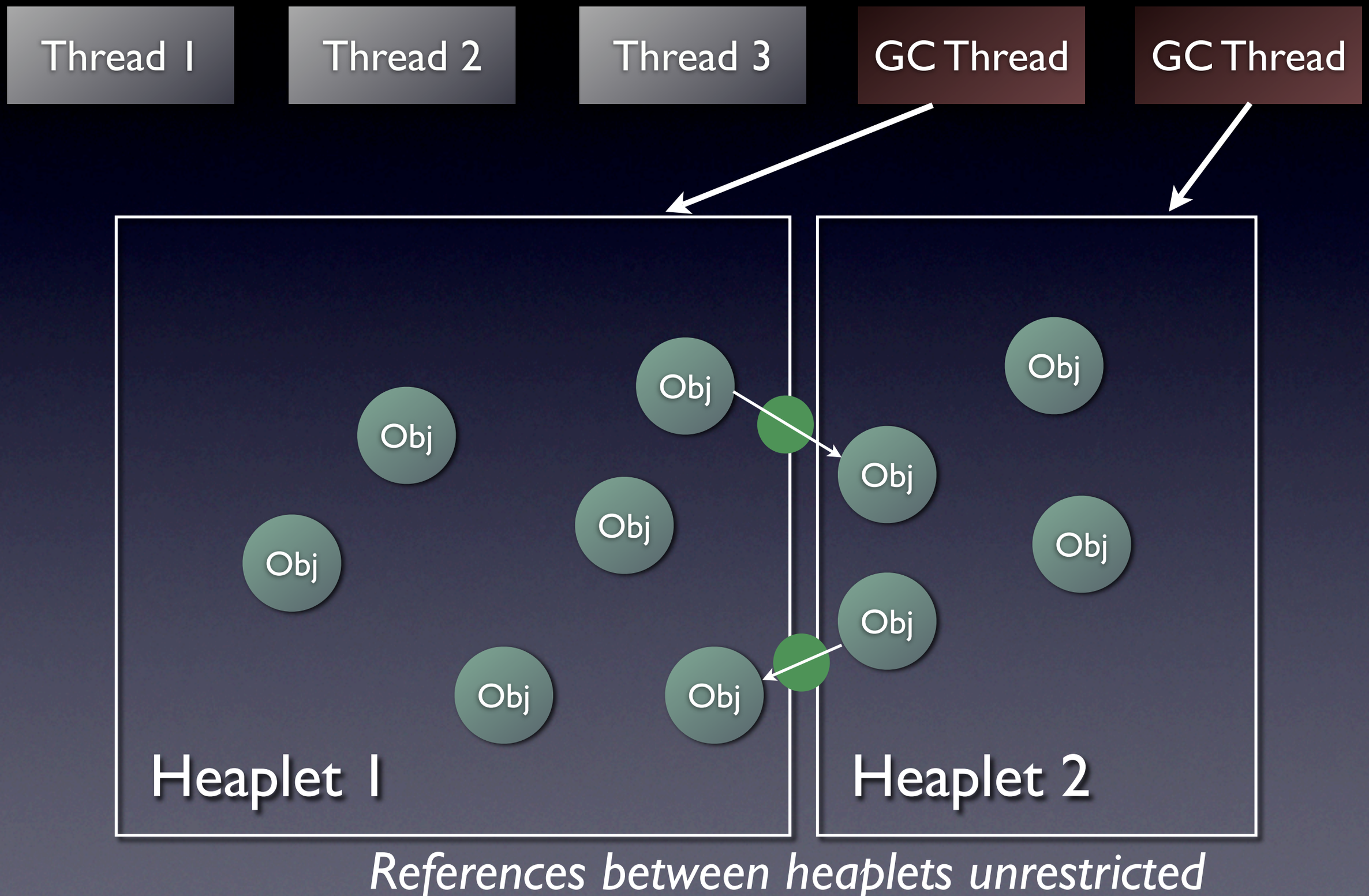
Thread 1 | Thread 2 | Thread 3 | GC Thread | GC Thread



RTGC with Heaplets Example



RTGC with Heaplets Example

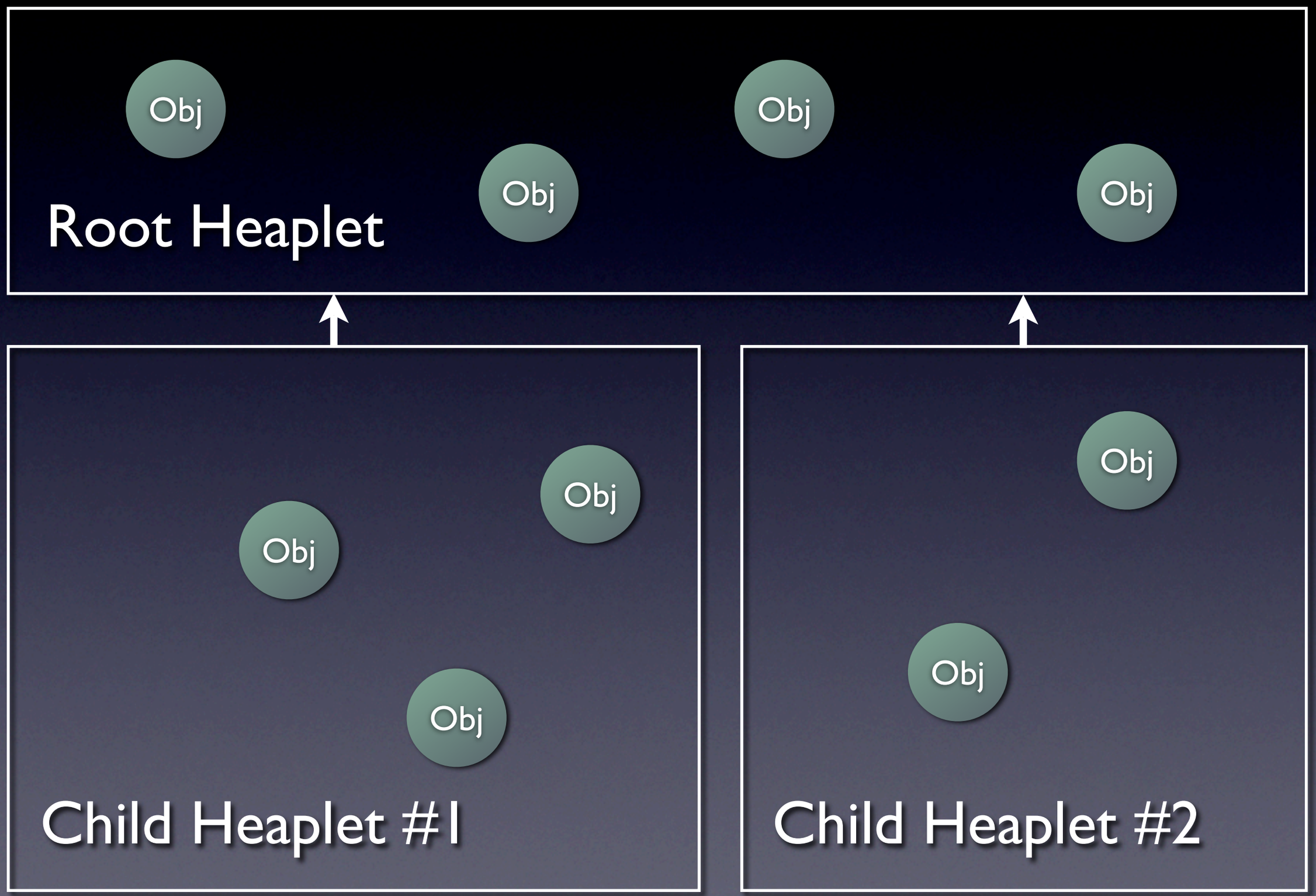


References between heaplets unrestricted

Heaplet Hierarchy

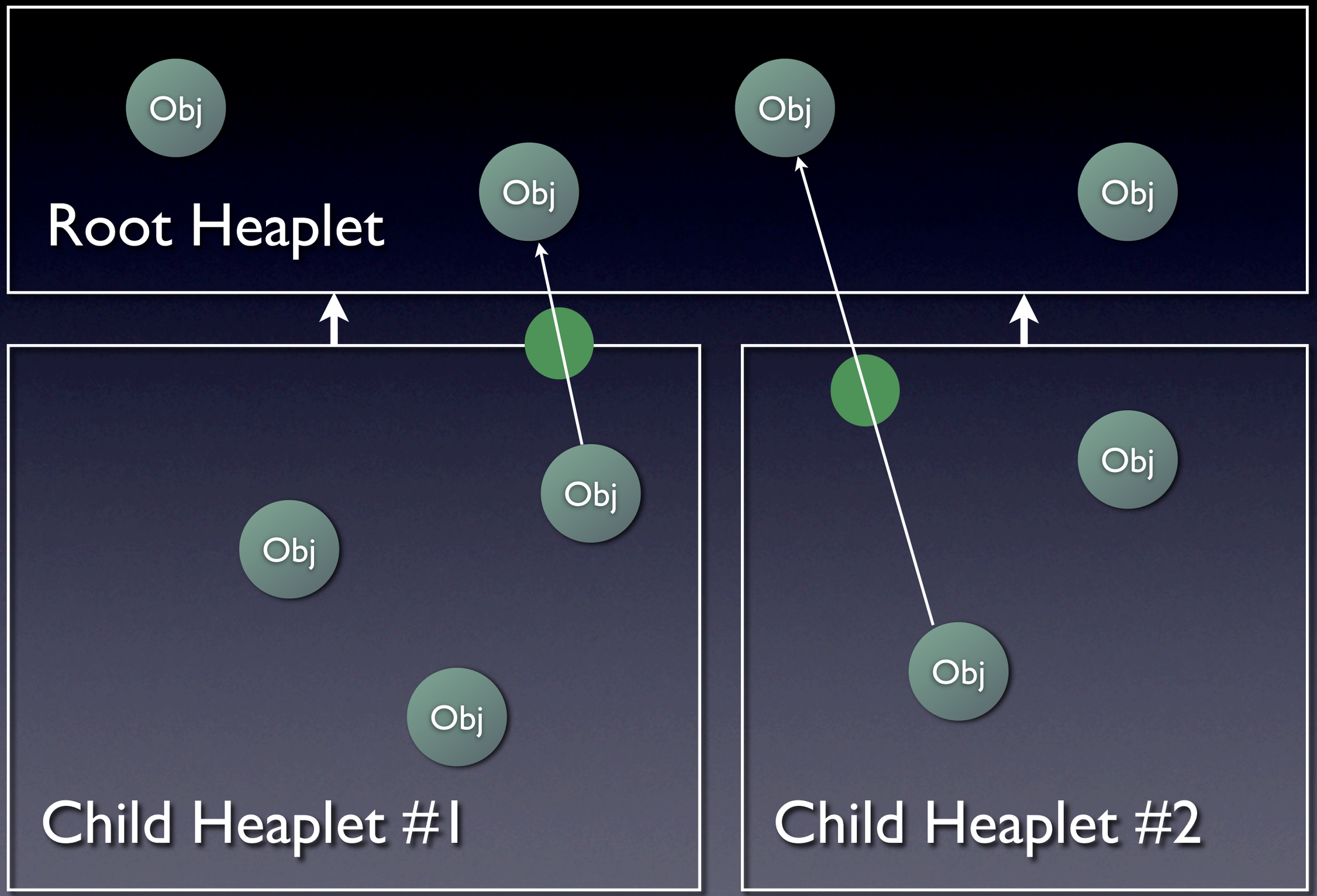
- We introduce a heaplet hierarchy to increase the performance of cross-heaplet references.
- A heaplet collector always scans child heaplets for references - thus, establishing new "up-hierarchy" references does not require barriers.
- Others cross-heaplet references are handled using a barrier and global cross-reference list ("cross-set").
  - Thus - establishing a cross-reference incurs a cost in both space and time.

Heaplet Hierarchy



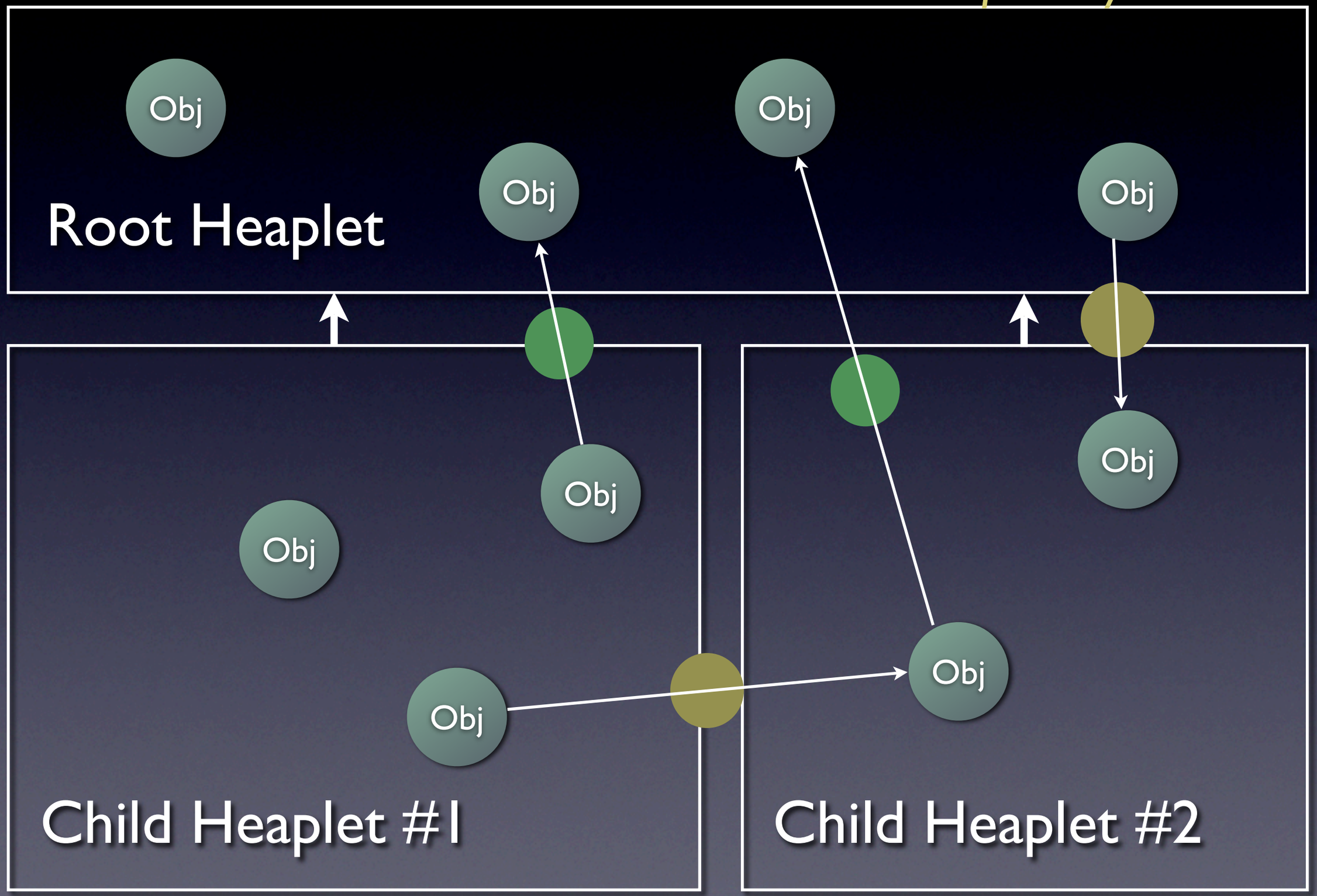
Heaplet Hierarchy

"up-references" are guaranteed fast



Heaplet Hierarchy

"up-references" are guaranteed fast
"cross-references" are allowed, but come with a penalty



Putting it Together

Putting it Together

- Heap is manually partitioned into *heaplets*.

Putting it Together

- Heap is manually partitioned into *heaplets*.
- Heaplets are manually arranged into a *hierarchy*, as a hint from the programmer about the likely directionality of references.

Putting it Together

- Heap is manually partitioned into *heaplets*.
- Heaplets are manually arranged into a *hierarchy*, as a hint from the programmer about the likely directionality of references.
- Each heaplet gets its own collector, user-tuned for the allocation and footprint behavior of the heaplet.

Putting it Together

- Heap is manually partitioned into *heaplets*.
- Heaplets are manually arranged into a *hierarchy*, as a hint from the programmer about the likely directionality of references.
- Each heaplet gets its own collector, user-tuned for the allocation and footprint behavior of the heaplet.
- *Introducting heaplets into a correct program does not make it incorrect.*

The HRTGC Algorithm

- Each heaplet gets a Metronome-style mark-sweep collector.
- Each collector is scheduled like the Metronome - but with control of schedules extended to include phasing.
- Cycles of cross-heaplet references are handled using a global cycle collector. Because garbage cycles are rare, the cycle collector runs at a very low rate - in fact it runs at a *zero rate* in our benchmarks.

Implementation
and
Evaluation

- We use the OpenVM RTJVM and J2c ahead-of-time compiler on the Linux operating system.
  - HRTGC is implemented as a memory management configuration in the OVM.
- OVM already implements a Metronome-like RTGC, which we use as a baseline.

- Two real-time Java benchmarks were used for comparing regular RTGC and HRTGC:
  - RTZen, a 202 KLoC CORBA implementation from UC Irvine, and
  - CD, a 41 KLoC benchmark developed at Purdue.
- Both benchmarks were originally written to use RTSJ scoped memory. We have previously converted both to use our Metronome-like RTGC.
- For this evaluation, we again converted the benchmarks, this time to use heaplets.

Conversion to use HRTGC

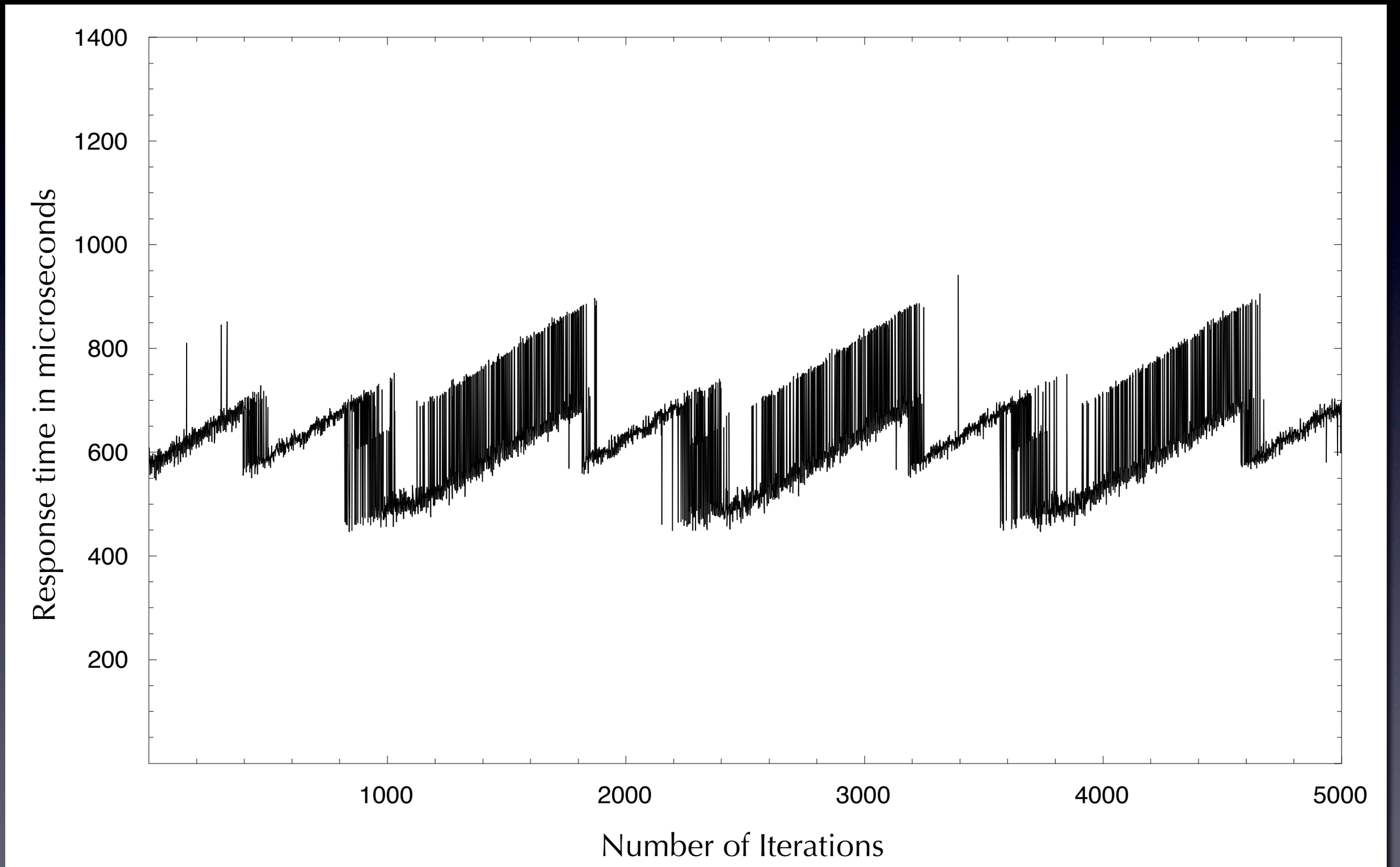
- Converting CD:
  - The CD use a producer-consumer pattern. We placed the producer and consumer in separate heaplets.
- Converting RTZen:
  - We place the core of Zen into its own heaplet.
  - The only changes were instrumentation in the main() method to create the ORB in our new heaplet.
  - *Thus, the Zen benchmark demonstrates not only the performance benefits of HRTGC, but the ease with which code can be refactored to use it effectively.*
- *Both benchmarks use 227_mtrt from SPECjvm98 as a noise maker.*

Measurements

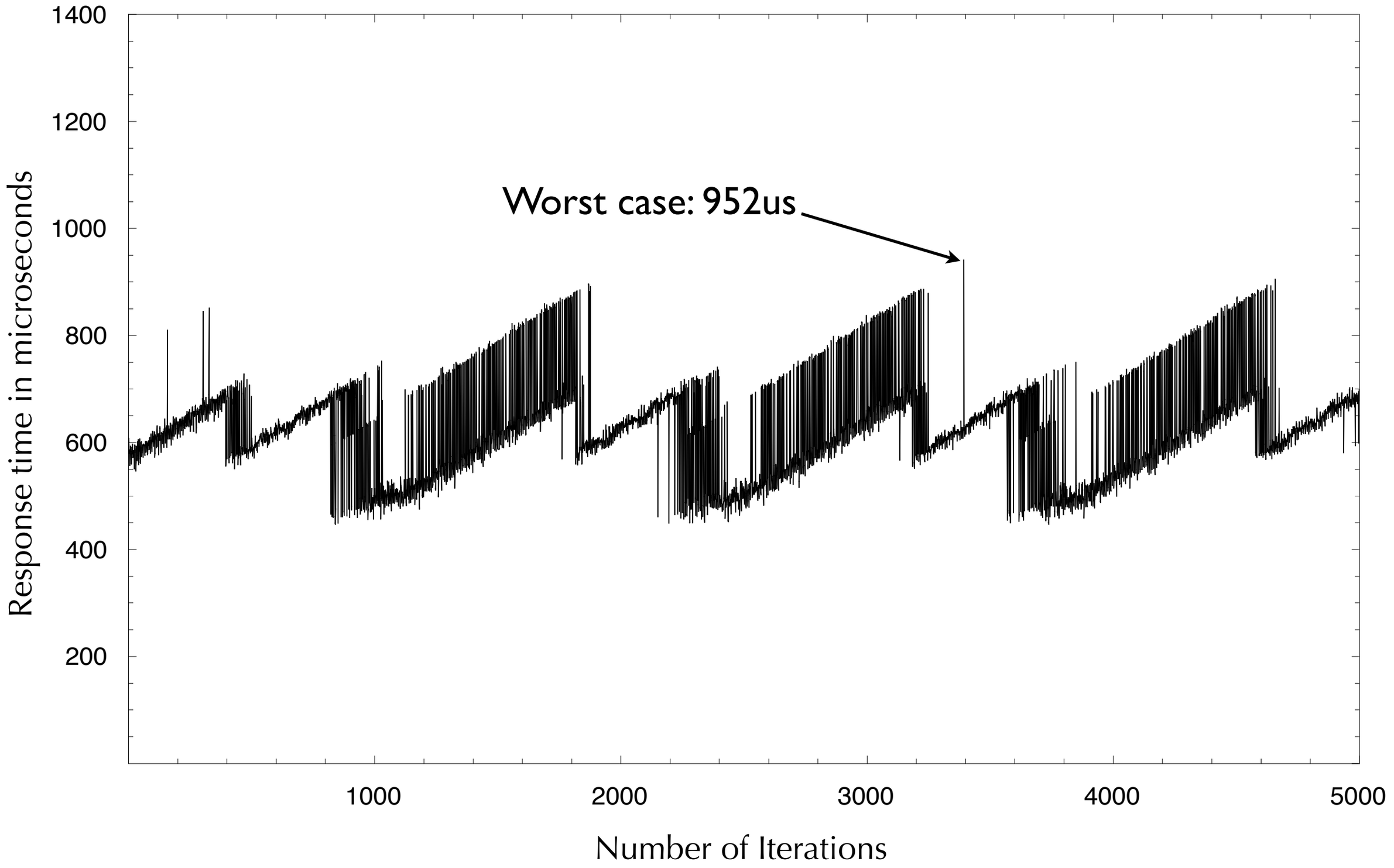
- We use a fixed total footprint for all configurations.
- The collector schedules are optimized for highest utilization while not allowing the memory usage to diverge.
- Both CD and RTZen are event-driven - thus, we record the total time required to handle each event - a quantity we call the *response time*.
- Additionally, we measure the *minimum mutator utilization* (MMU).

RTZen Response Time

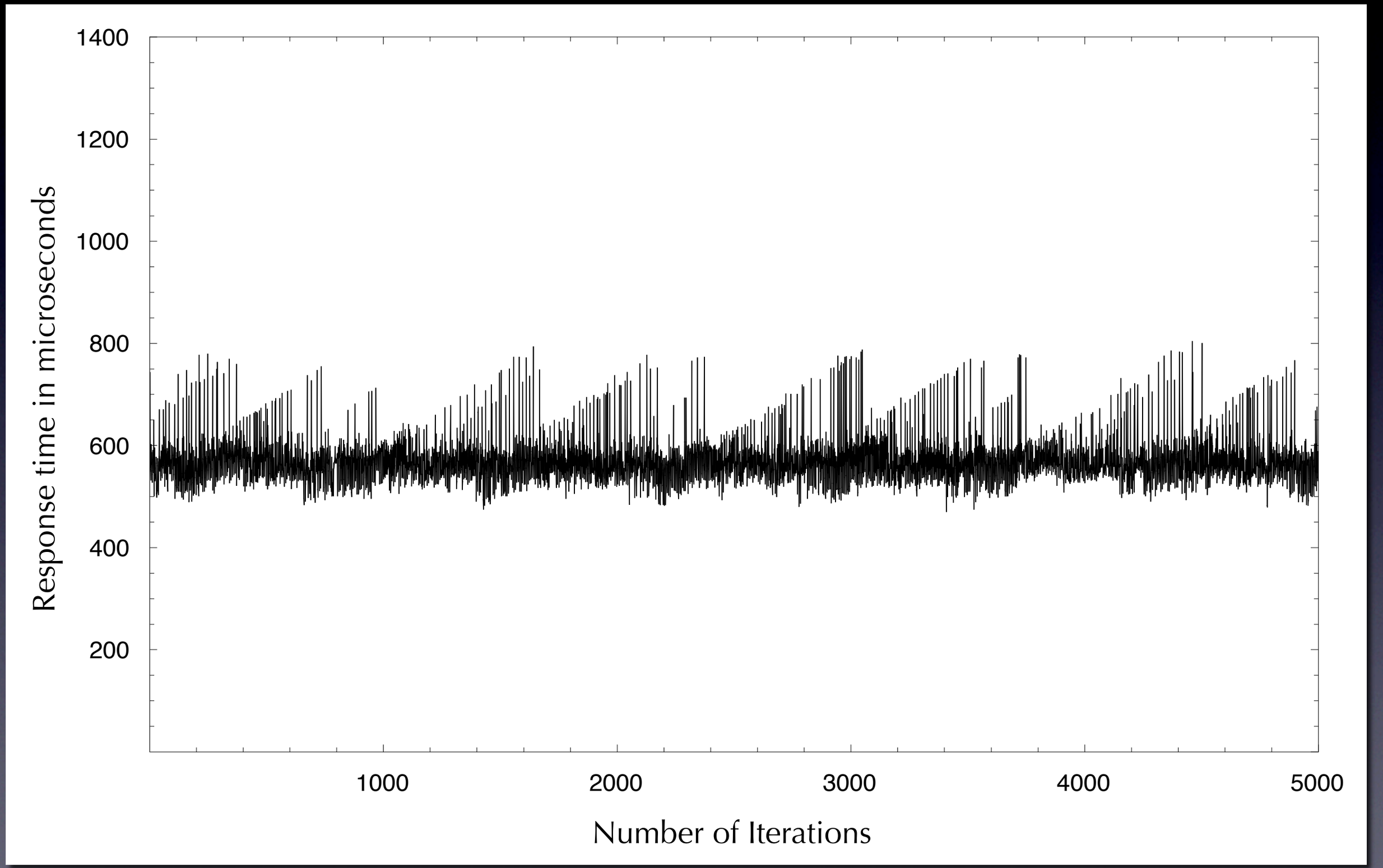
RTZen with RTGC



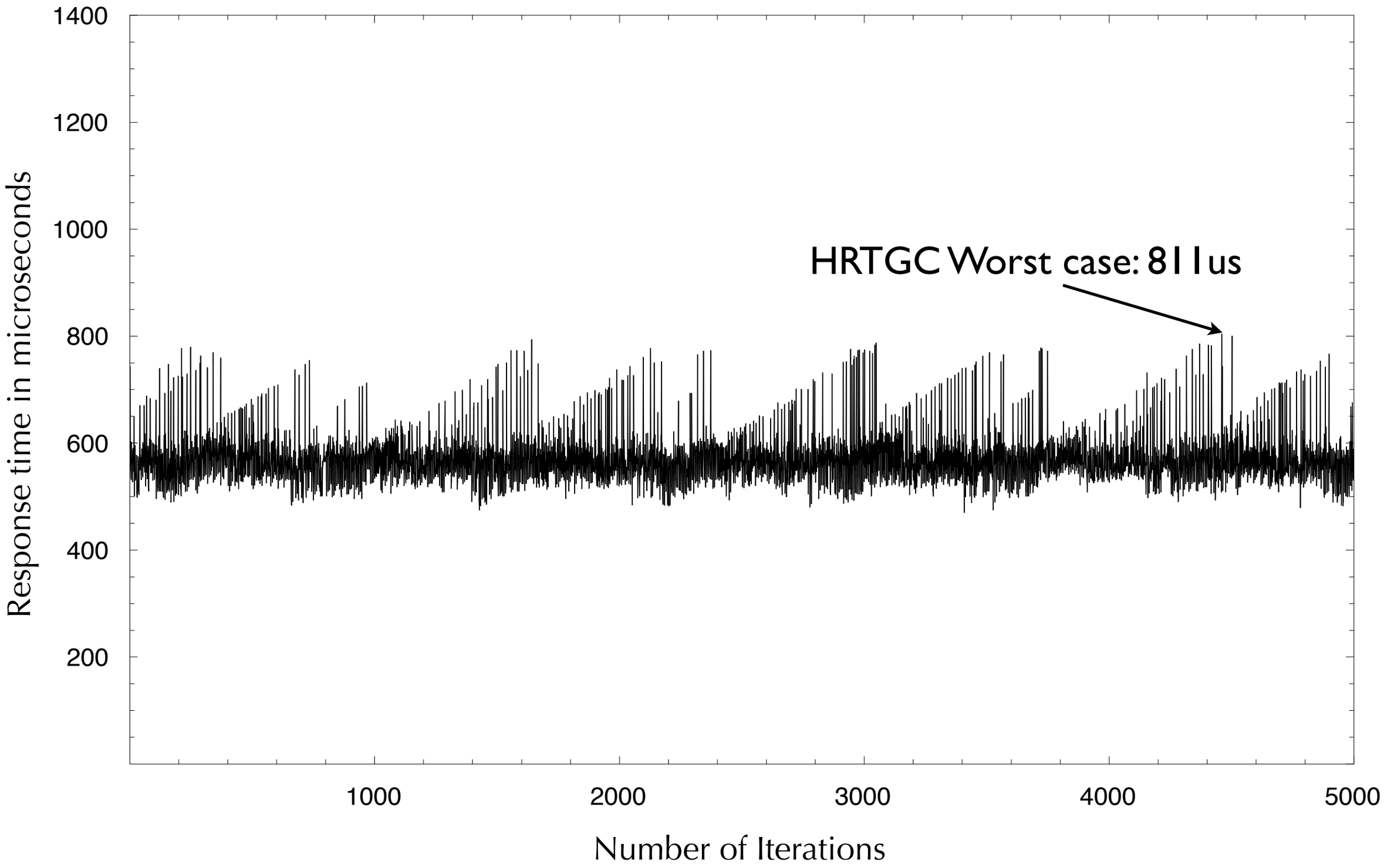
RTZen with RTGC



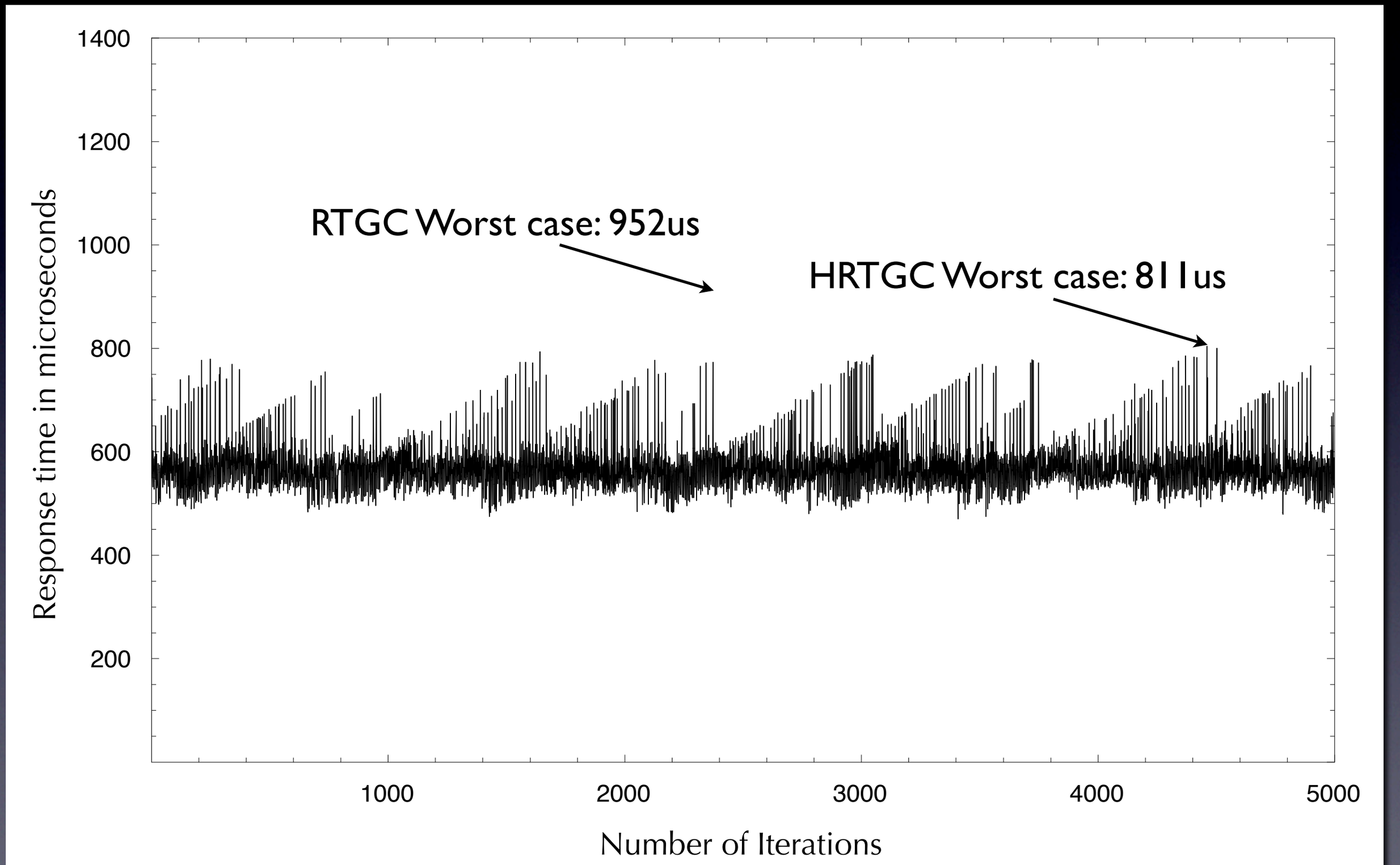
RTZen with HRTGC



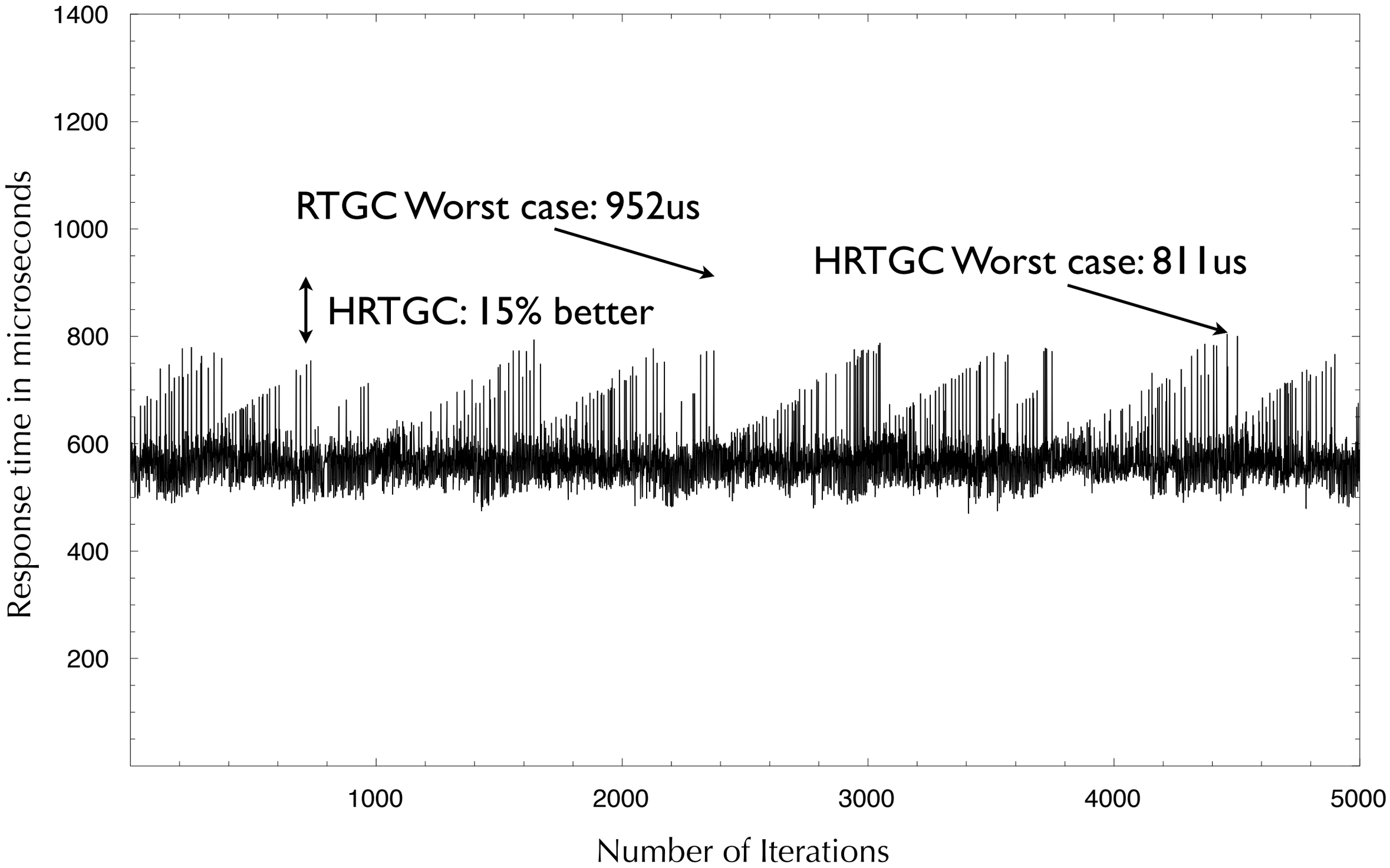
RTZen with HRTGC



RTZen with HRTGC

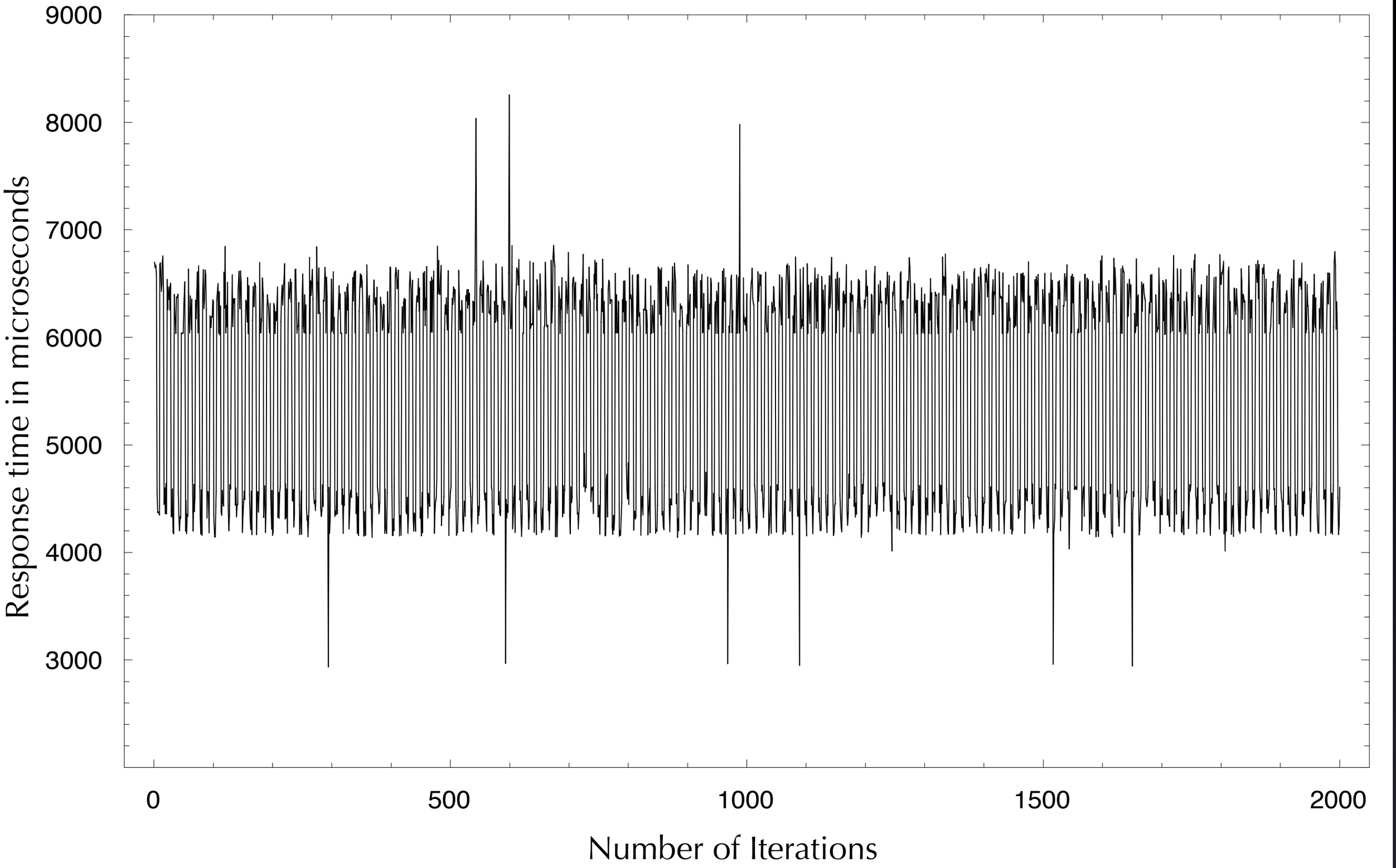


RTZen with HRTGC

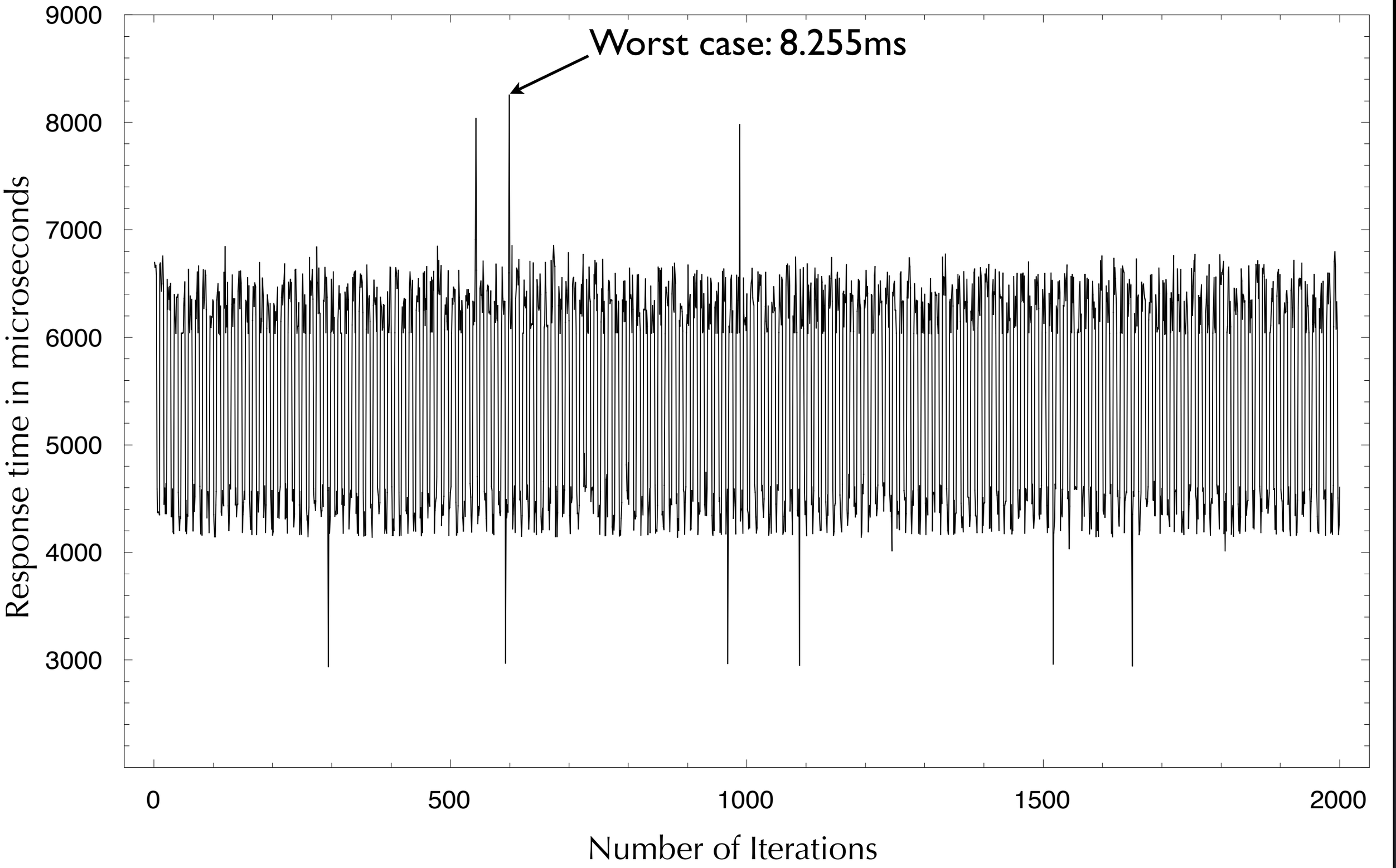


CD Response Time

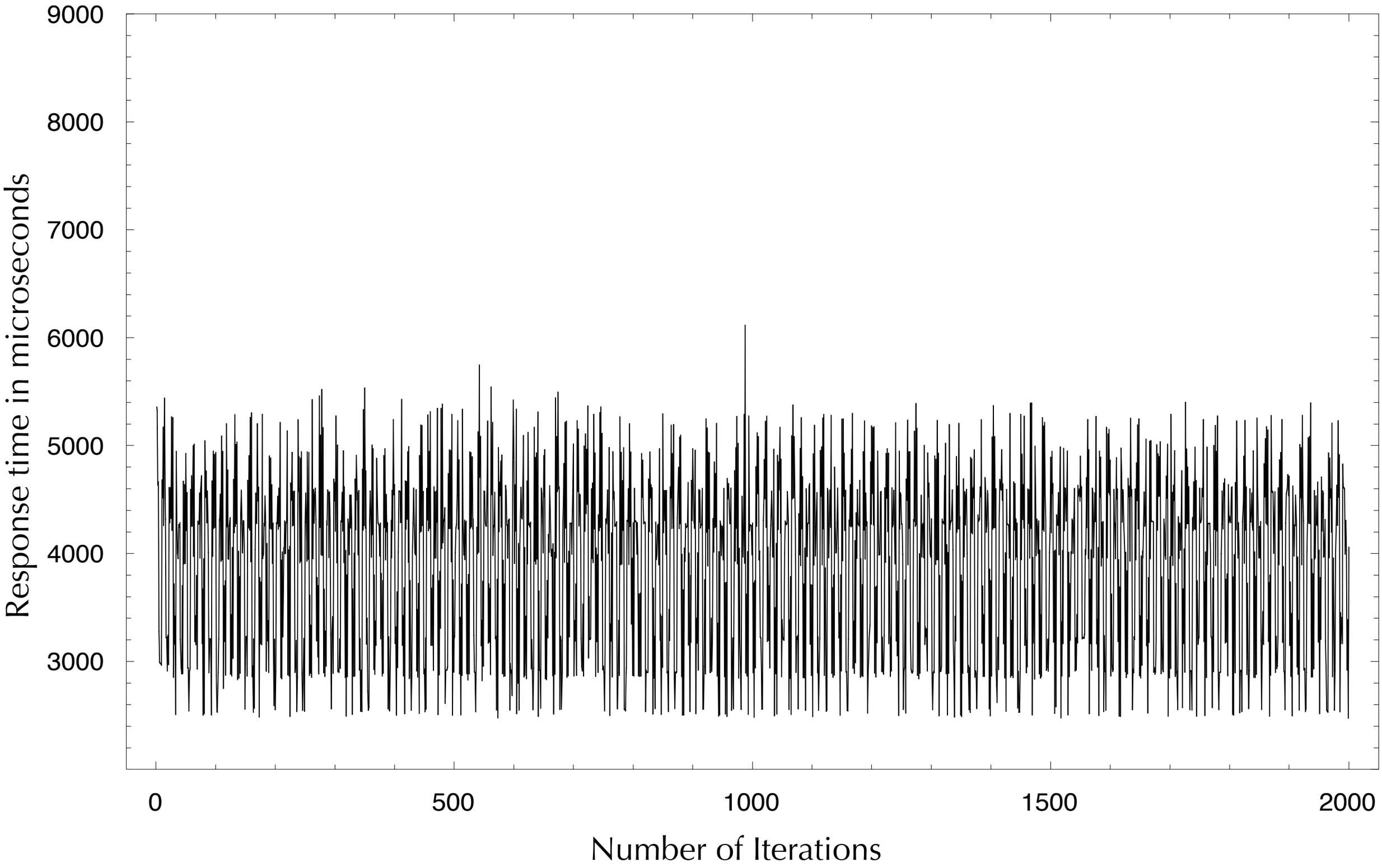
CD with RTGC


9000
8000
7000
6000
5000
4000
3000
Response time in microseconds
0
500
1000
1500
2000
Number of Iterations

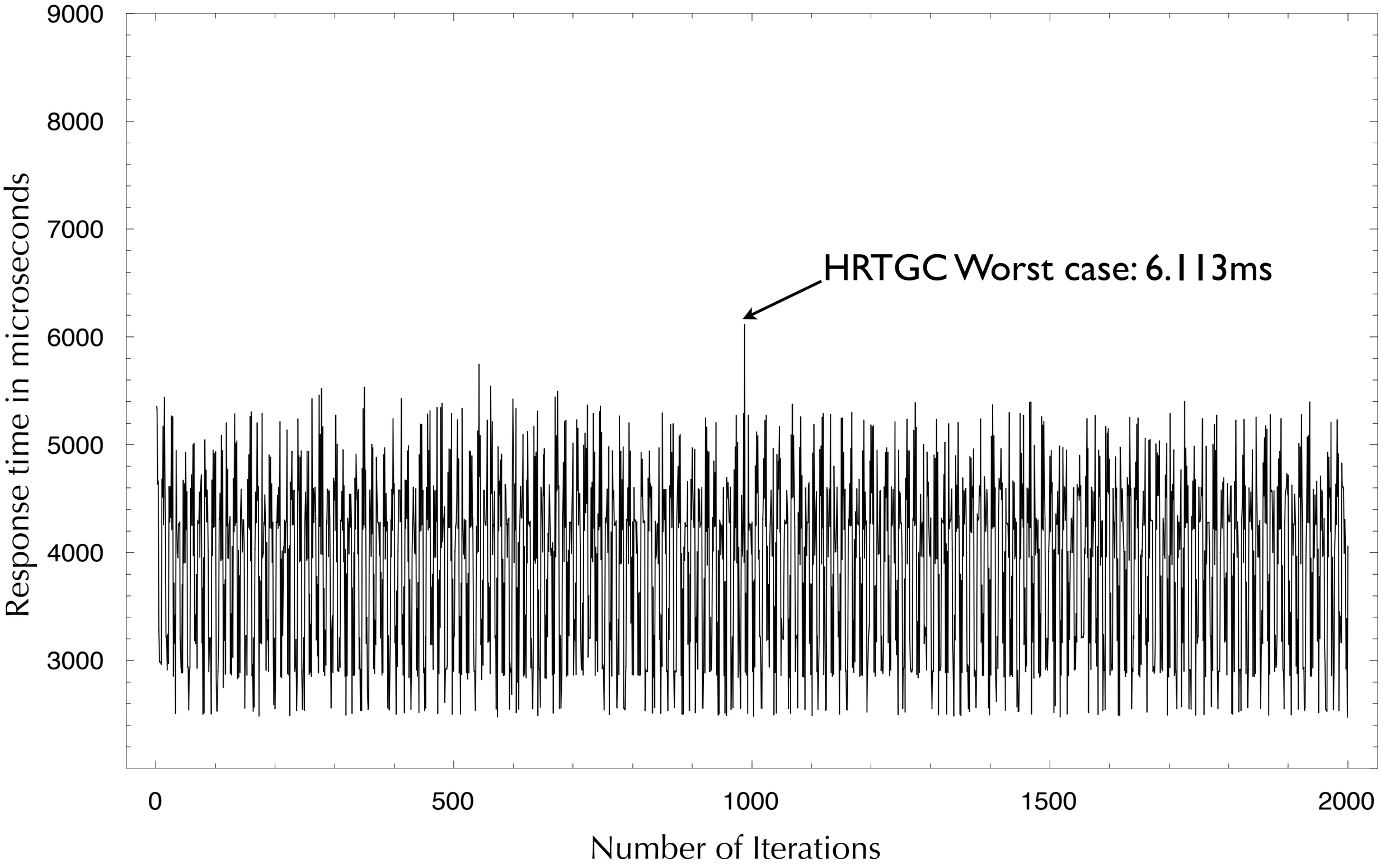
CD with RTGC


Worst case: 8.255ms
9000
8000
7000
6000
5000
4000
3000
Response time in microseconds
0
500
1000
1500
2000
Number of Iterations

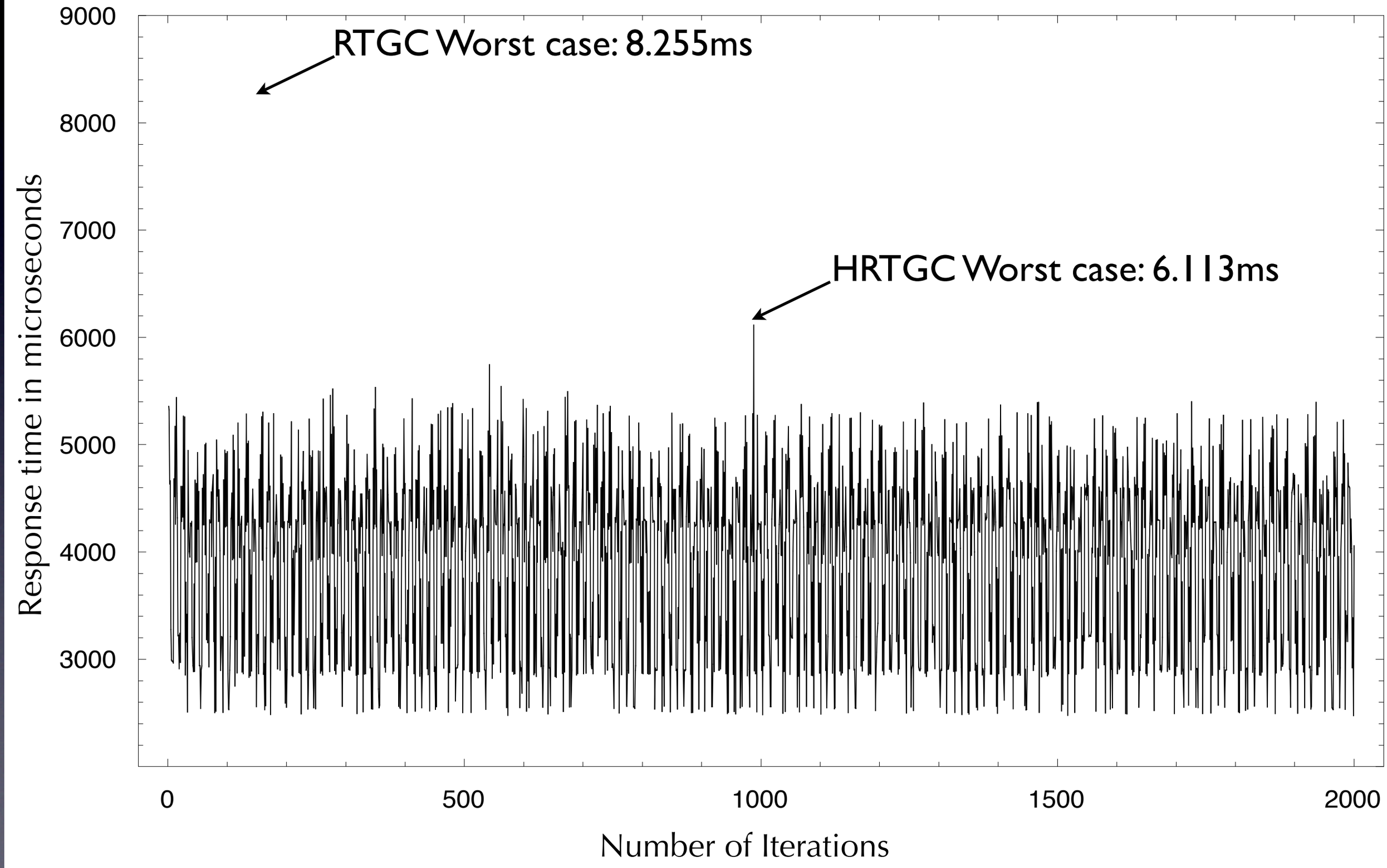
CD with HRTGC



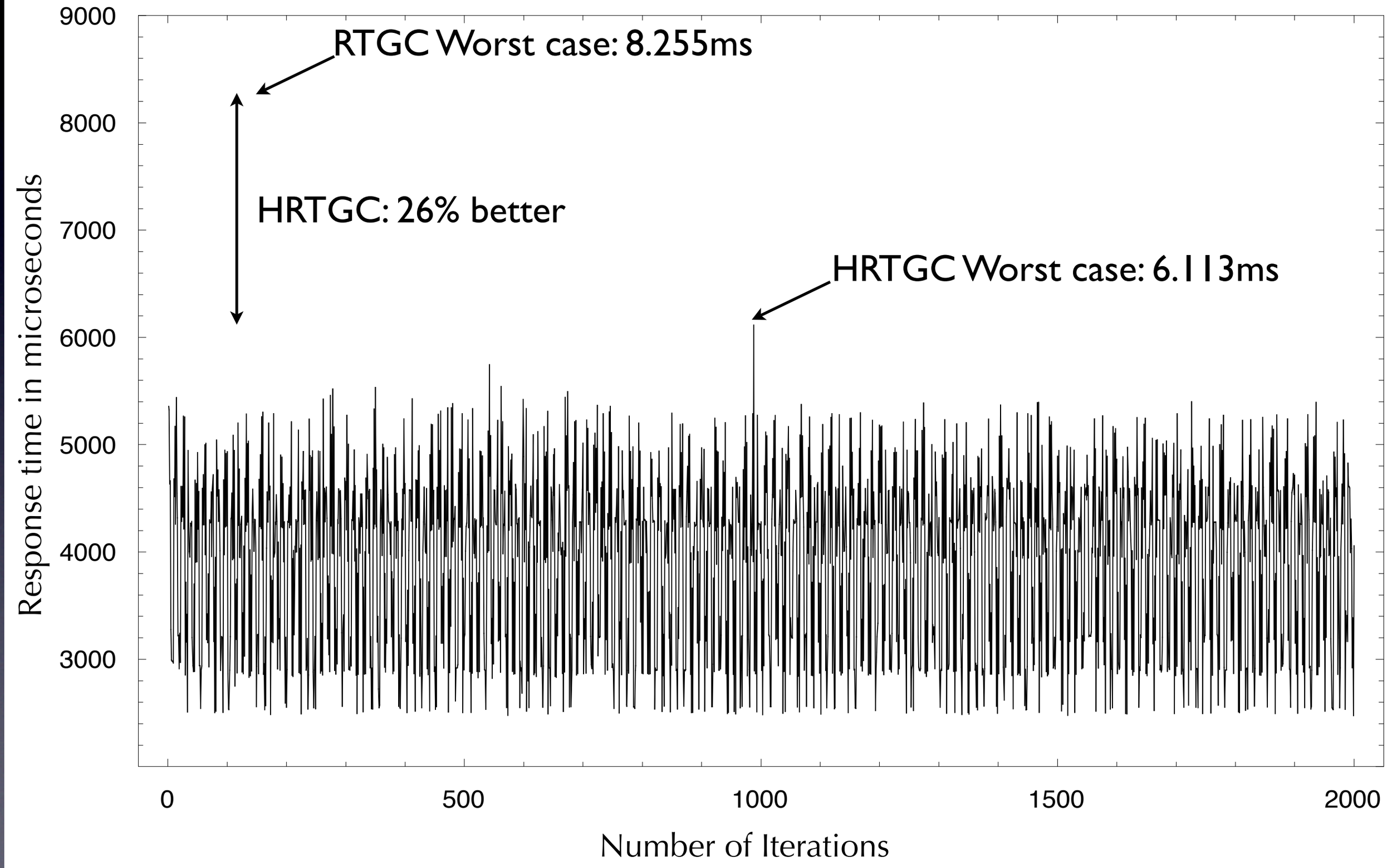
CD with HRTGC



CD with HRTGC



CD with HRTGC



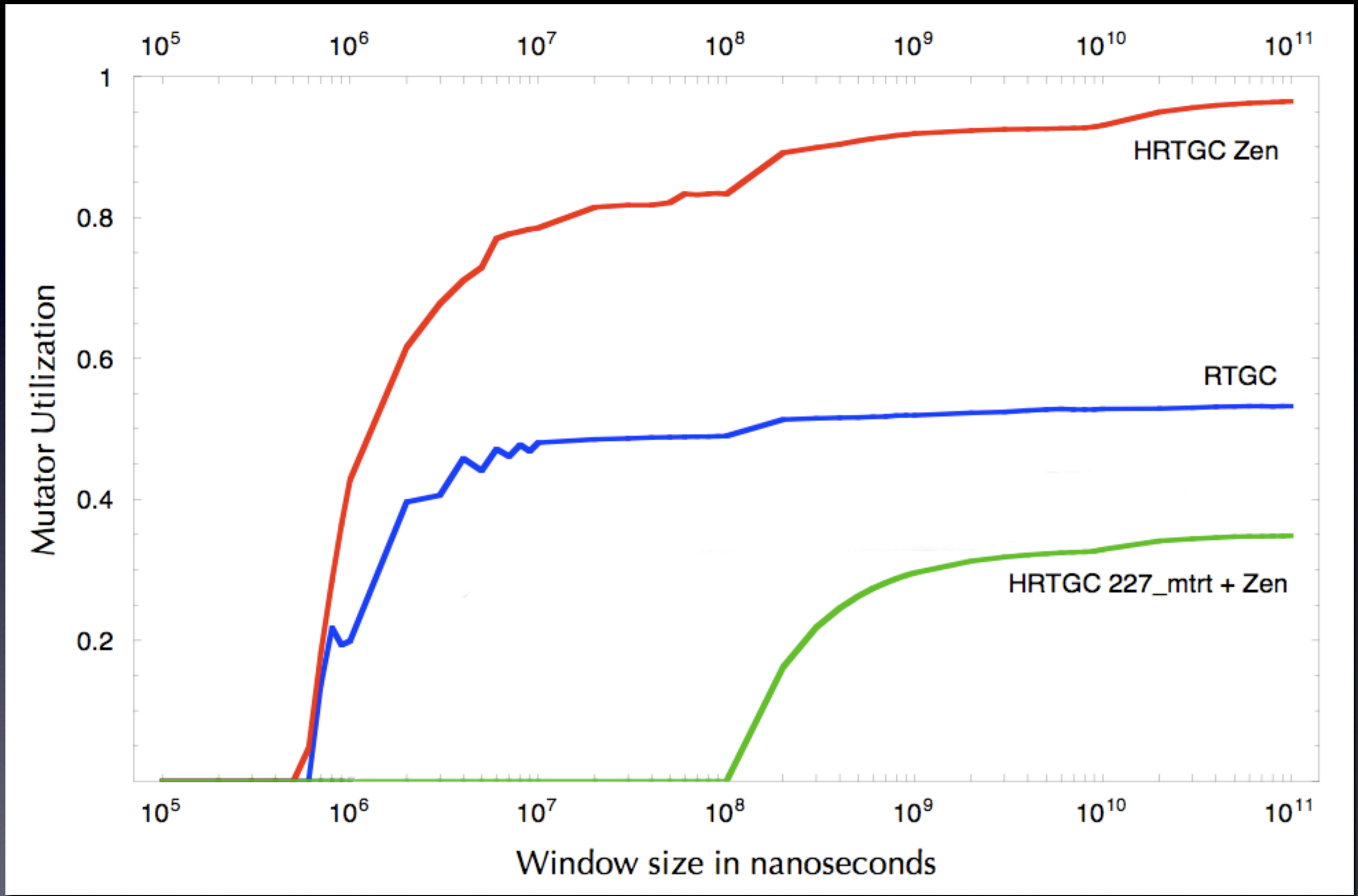
MMU

- *Minimum mutator utilization (MMU) shows the worst-case amount of time the mutator would get for a timeslice of a given length.*

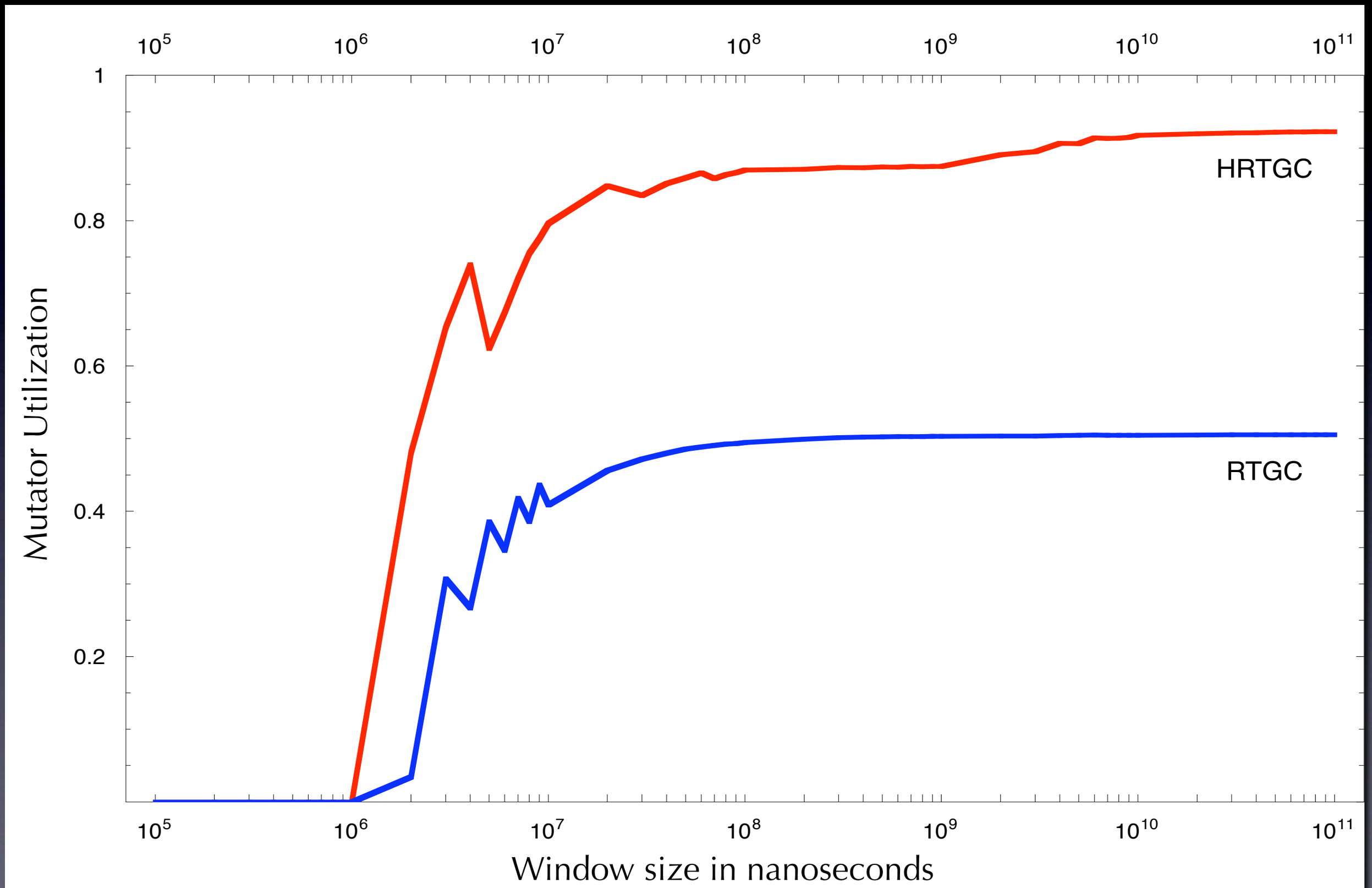
- Thus - MMU shows utilization (a number from 0 to 1, where 1 is better) versus timeslice size (in this case, in nanoseconds).

- We display MMU that has been empirically measured for our two benchmarks (RTZen and CD).

RTZen MMU


10^5
10^6
10^7
10^8
10^9
10^{10}
10^{11}
1
0.8
0.6
0.4
0.2
Mutator Utilization
HRTGC Zen
RTGC
HRTGC 227_mtrt + Zen
Window size in nanoseconds

CD MMU


10^5
10^6
10^7
10^8
10^9
10^{10}
10^{11}
1
0.8
0.6
0.4
0.2
Mutator Utilization
HRTGC
RTGC
Window size in nanoseconds

A more in-depth discussion of the algorithm, and the results, is found in the paper.

Questions/Comments