NG2C: Pretenuring Garbage Collector with Dynamic Generations for HotSpot Big Data Apps

Rodrigo Bruno*, Luís Picciochi Oliveira+, Paulo Ferreira*

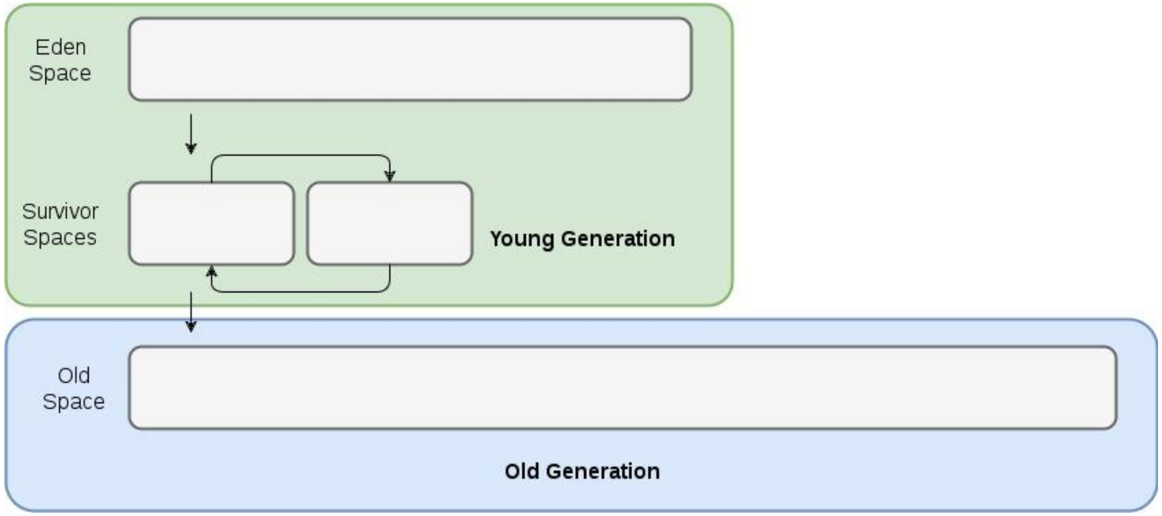
rodrigo.bruno@tecnico.ulisboa.pt, luis.oliveira@feedzai.com, paulo.ferreira@inesc-id.pt

*INESC-ID - Instituto Superior Técnico, University of Lisbon, Portugal

+Feedzai, Lisbon, Portugal

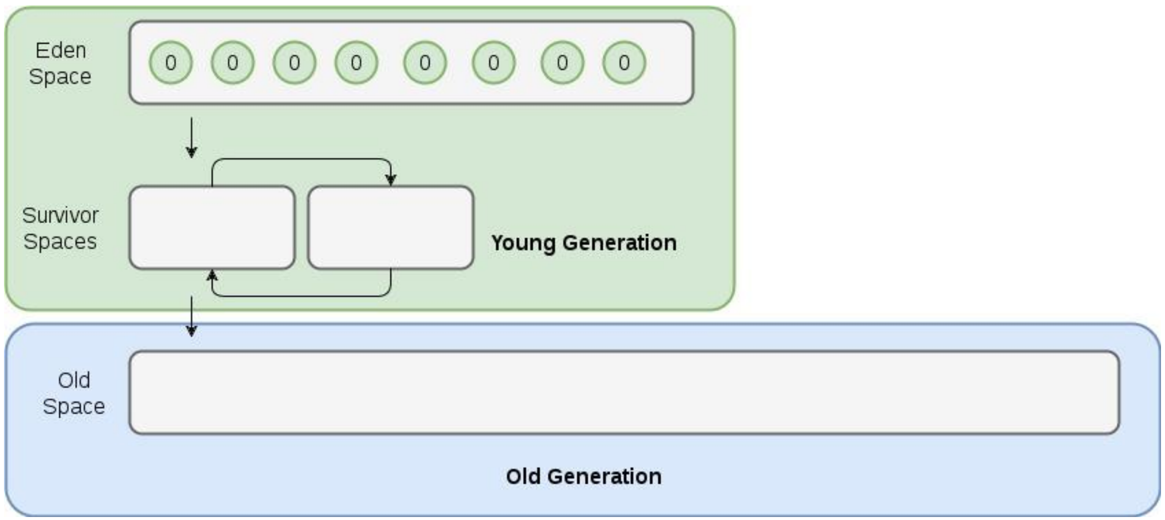
ISMM'17@Barcelona

OpenJDK HotSpot Generational GCs (PS, CMS, G1)

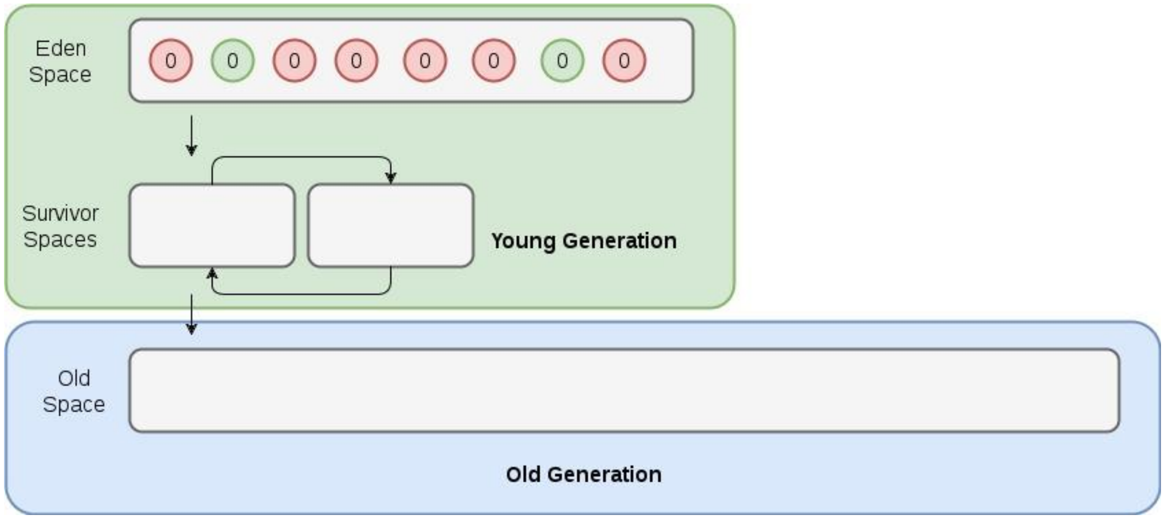


- Two generations:
  - **Young** and **Old**
- Surviving objects are copied to
  - **Survivor** spaces and then to
  - the **Old** generation.

OpenJDK HotSpot Generational GCs

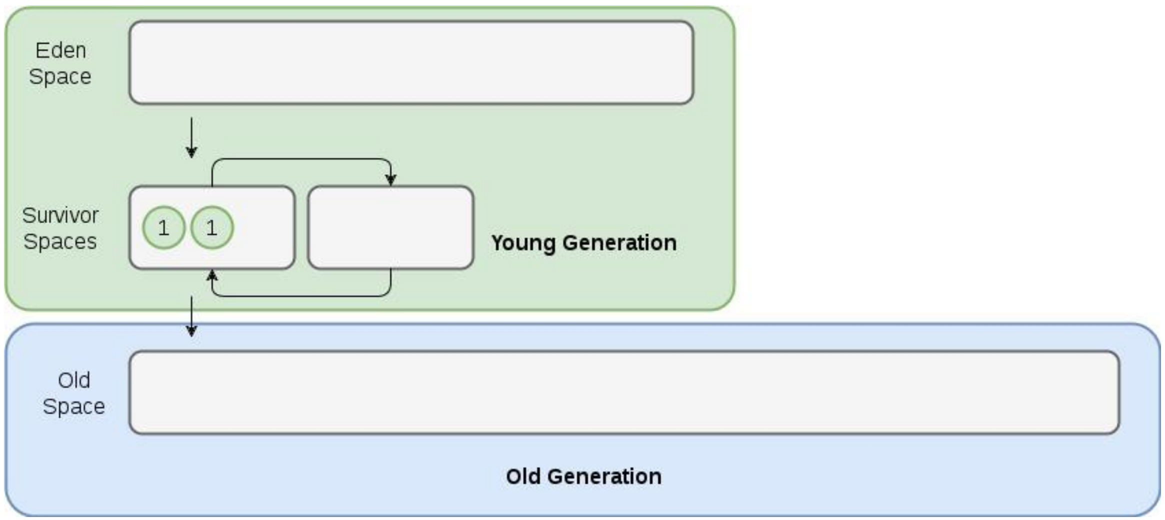


OpenJDK HotSpot Generational GCs



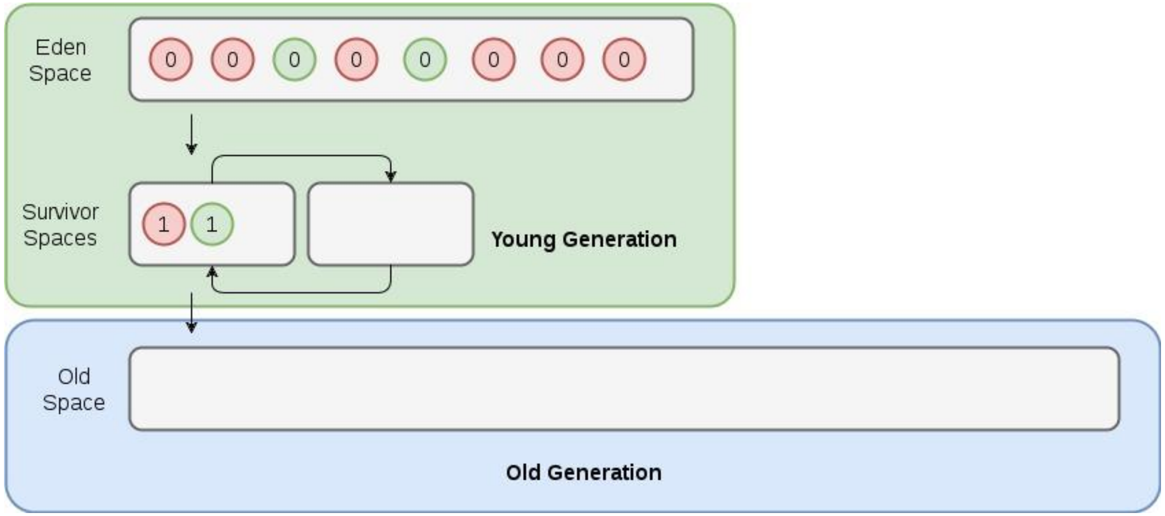
Before GC cycle 1

OpenJDK HotSpot Generational GCs



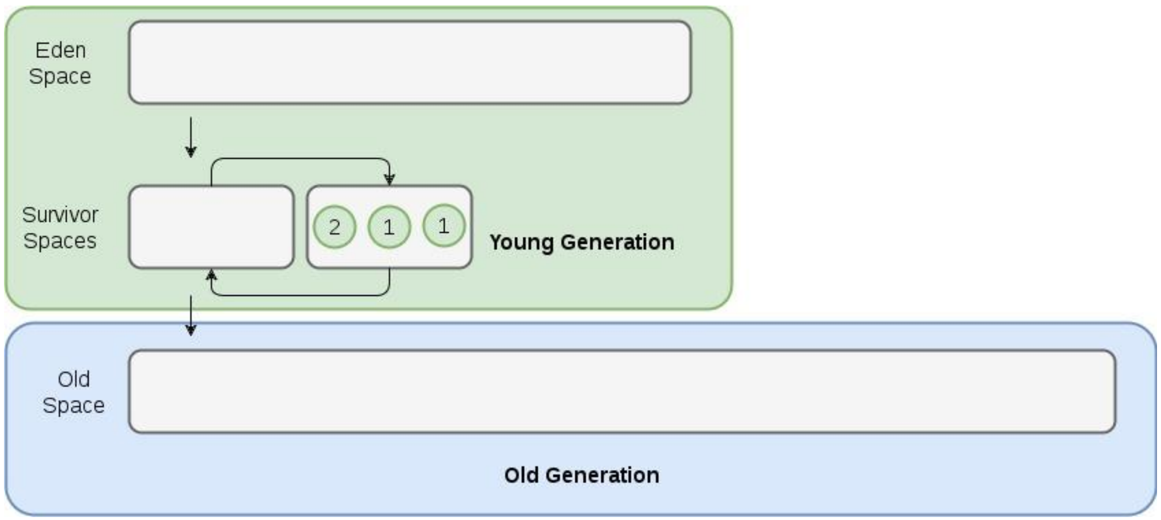
After GC cycle 1

OpenJDK HotSpot Generational GCs



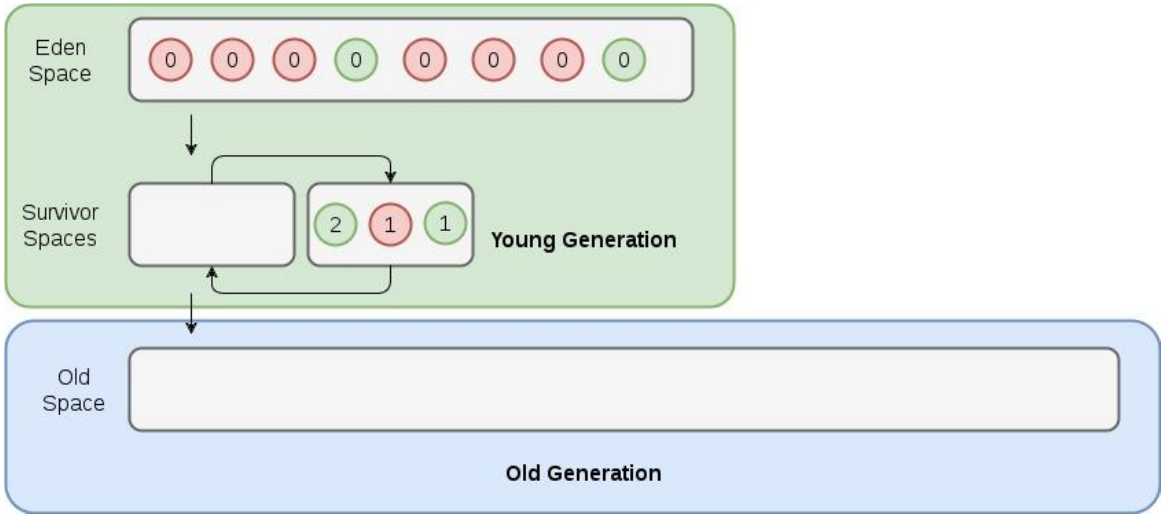
Before GC cycle 2

OpenJDK HotSpot Generational GCs



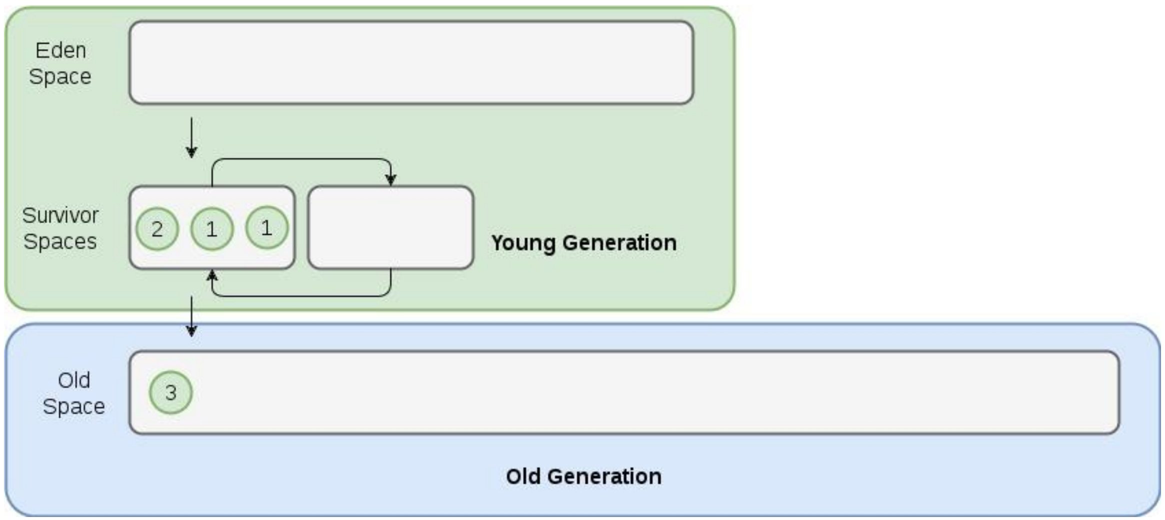
After GC cycle 2

OpenJDK HotSpot Generational GCs



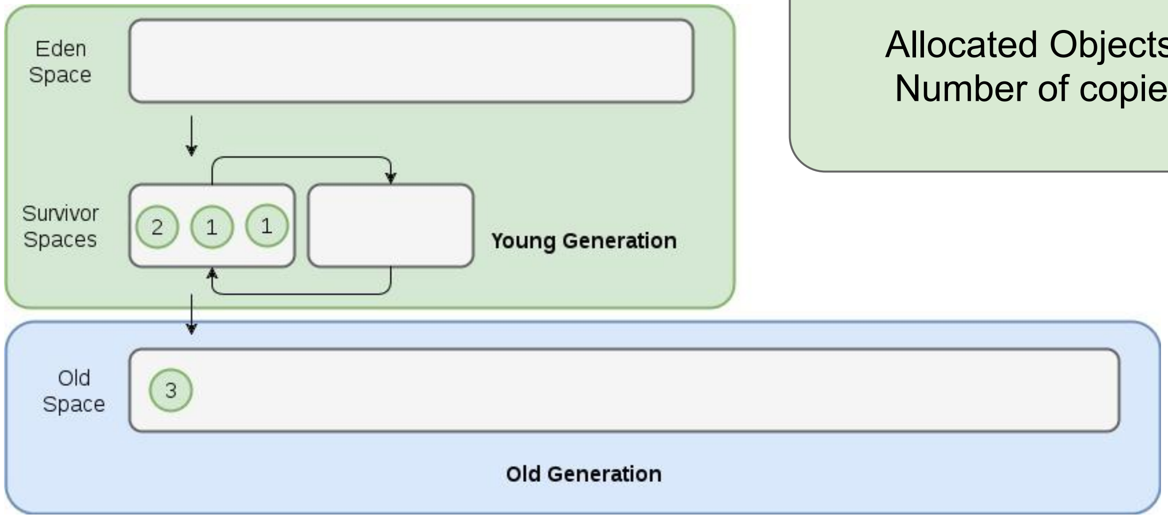
Before GC cycle 3

OpenJDK HotSpot Generational GCs



After GC cycle 3

OpenJDK HotSpot Generational GCs



Allocated Objects: **32**
Number of copies: **9**

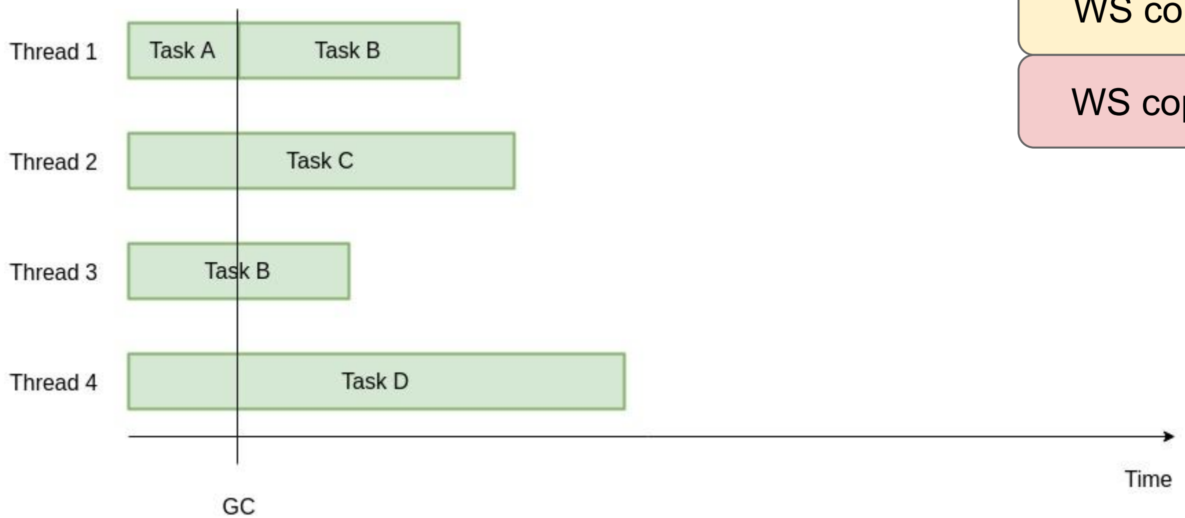
After GC cycle 3

Big Data Application (simplification)

```java
public void runTask(enum TaskType tt) {

  // Allocates memory to hold Working Set
  WorkingItem[] buffer = new WorkingItem[WS_SIZE];

  // Loads Working Set
  DataProvider.load(tt, buffer);

  // Process Working Set
  Result r = DataProcessor.process(tt, buffer);

  // Pushes results from computation
  Output.push(r);
}
```

- 4 threads (one per core), running ‘runTask’ method in loop
- Each task consumes 500 MB of memory (Working Set size)
- Eden is 2GB in size
- Tasks can take different amounts of time to finish

Big Data Application in HotSpot GCs



WS not copied

WS copied once

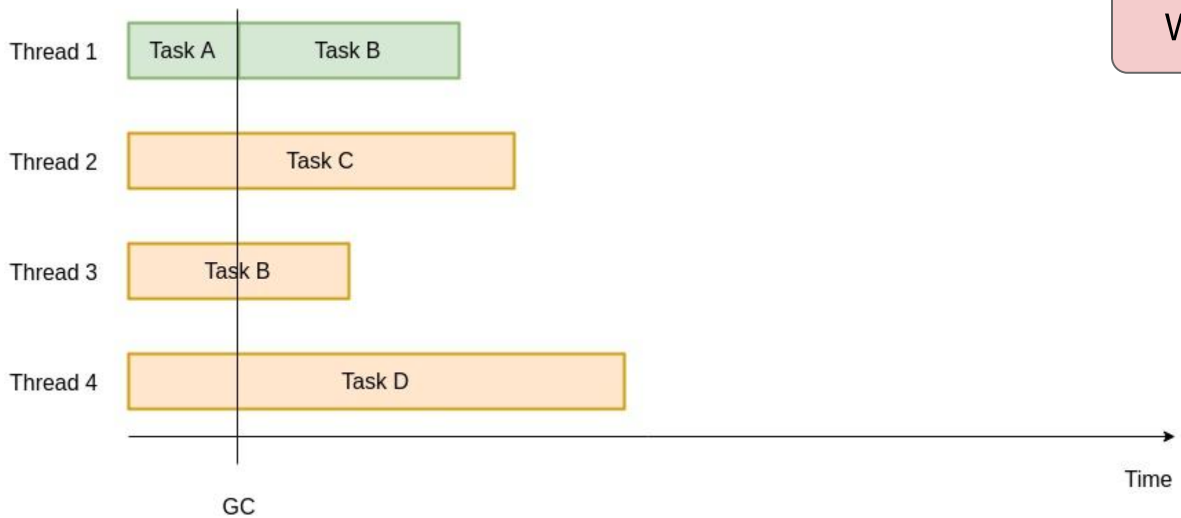
WS copied twice

Big Data Application in HotSpot GCs

WS not copied

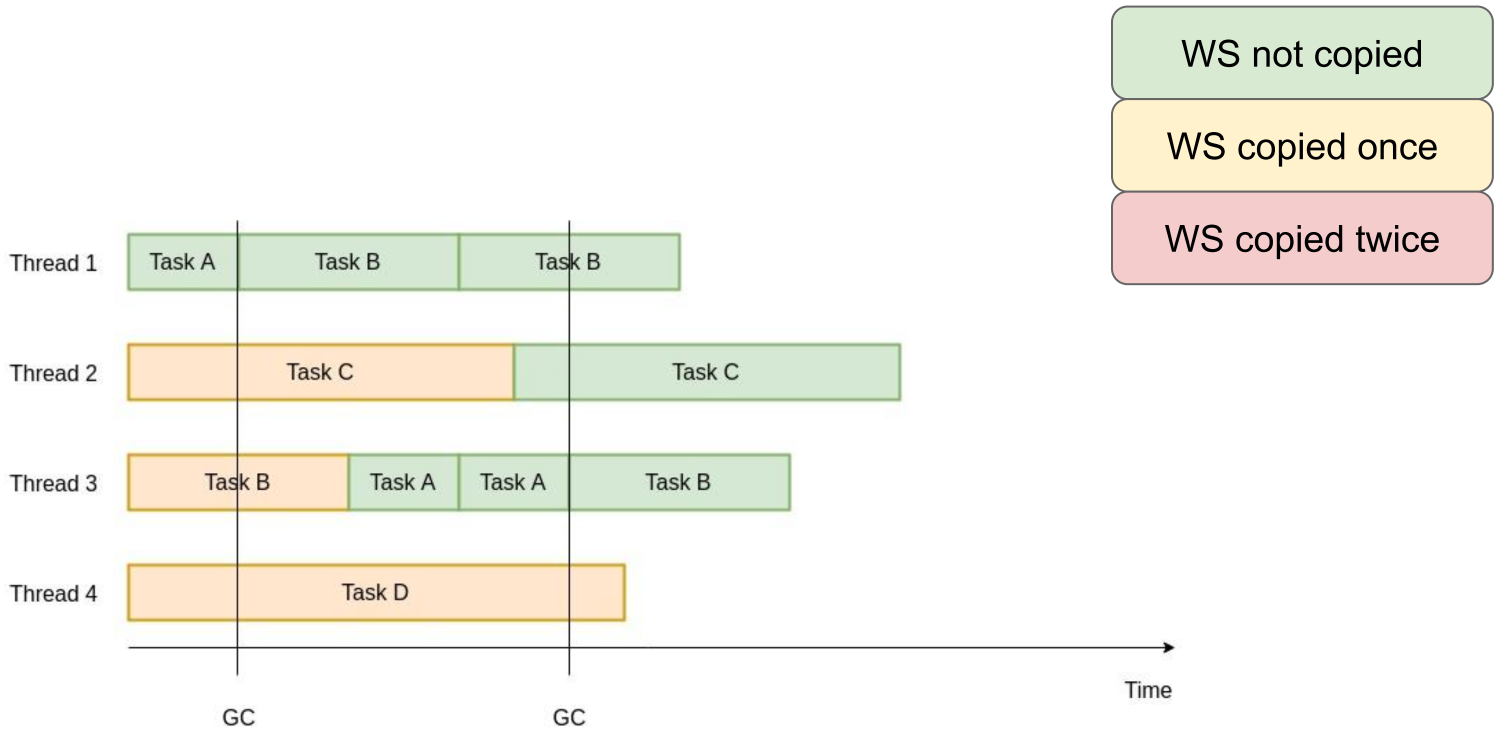
WS copied once

WS copied twice

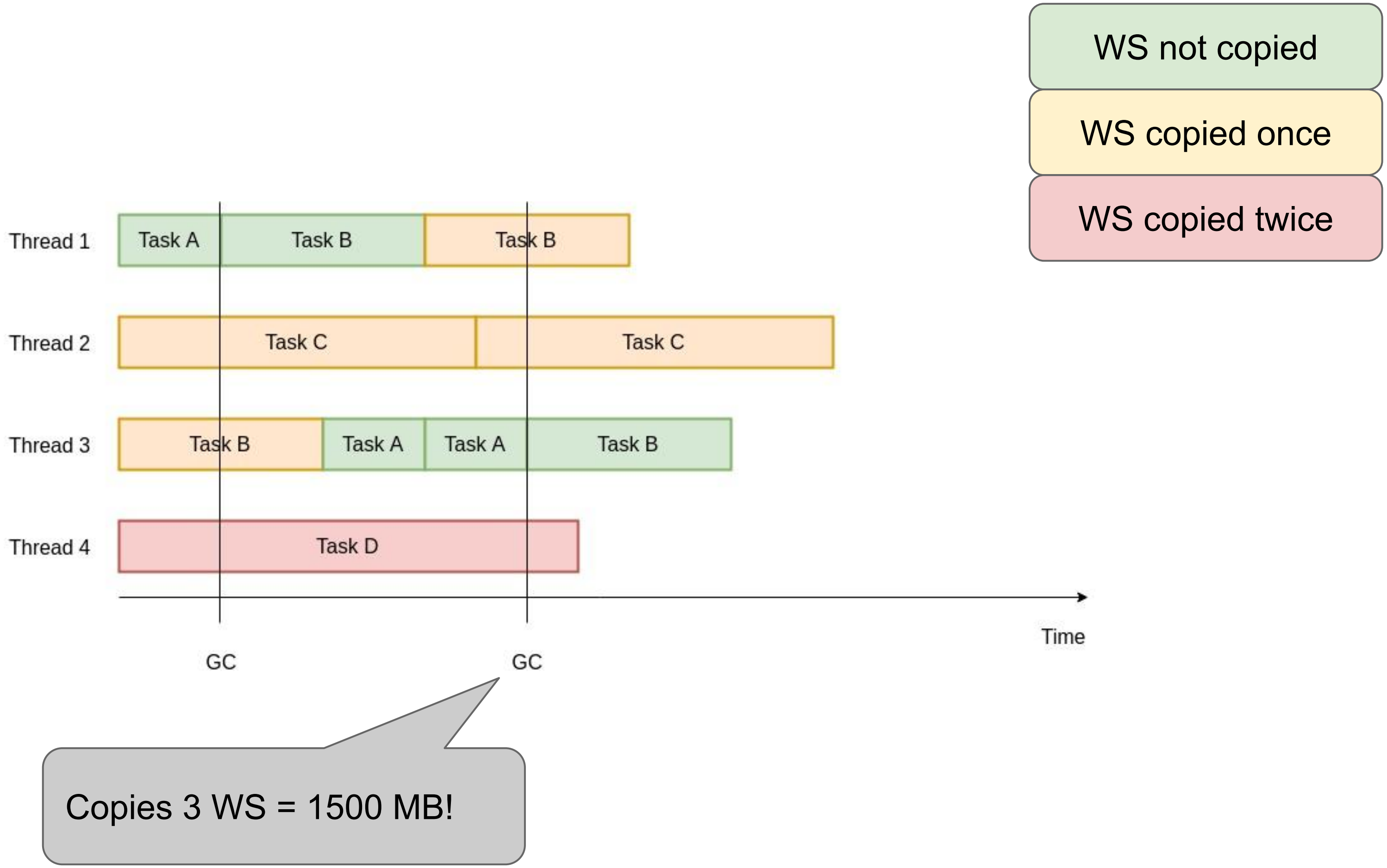


Copies 3 WS = 1500 MB!

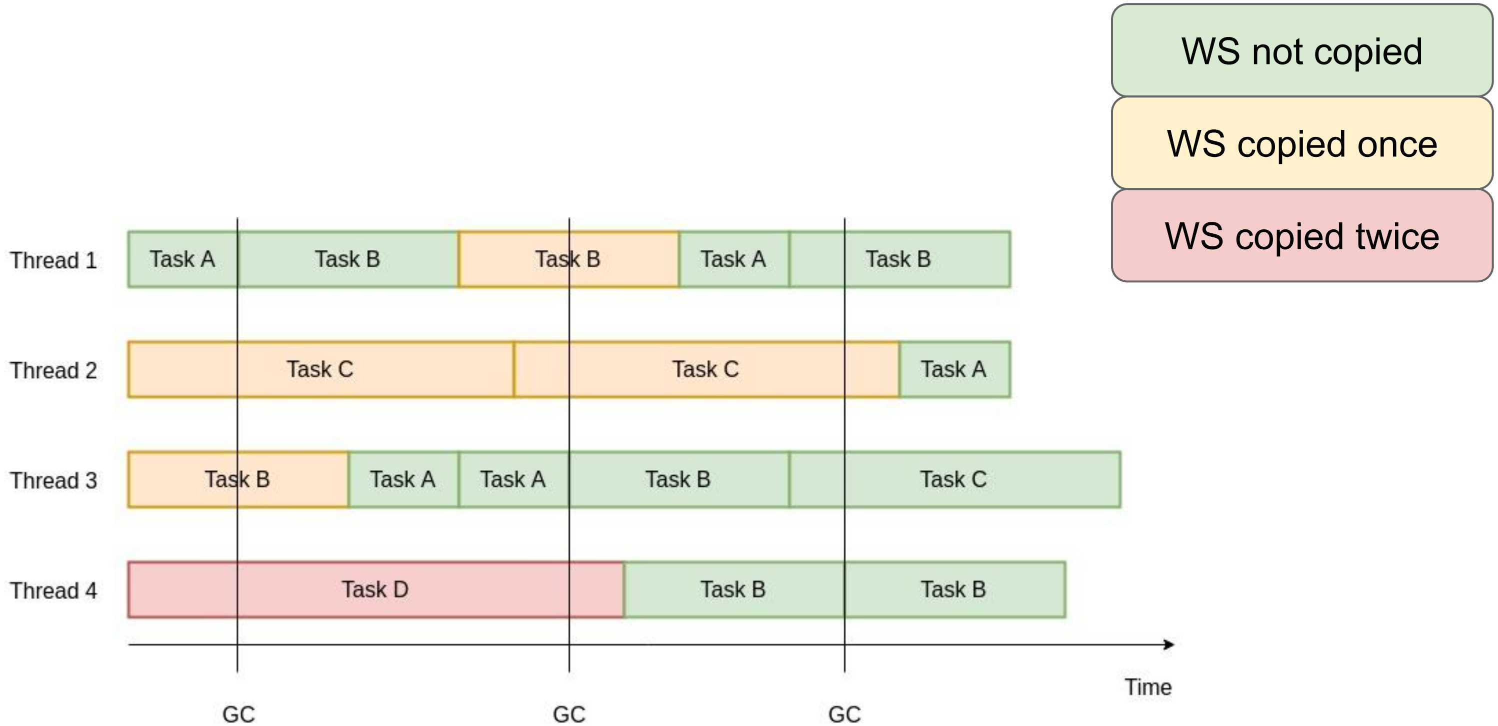
Big Data Application in HotSpot GCs



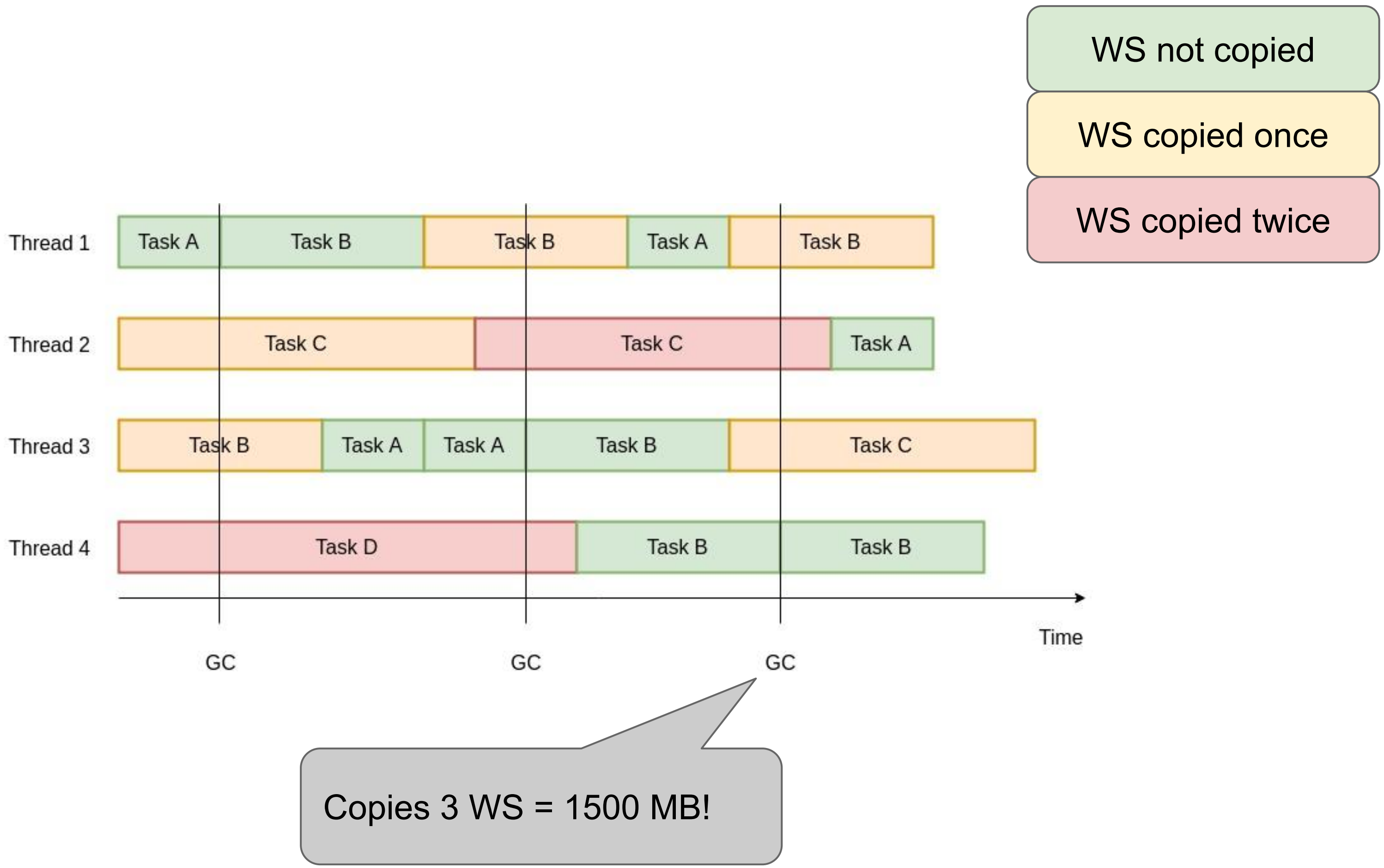
Big Data Application in HotSpot GCs



Big Data Application in HotSpot GCs

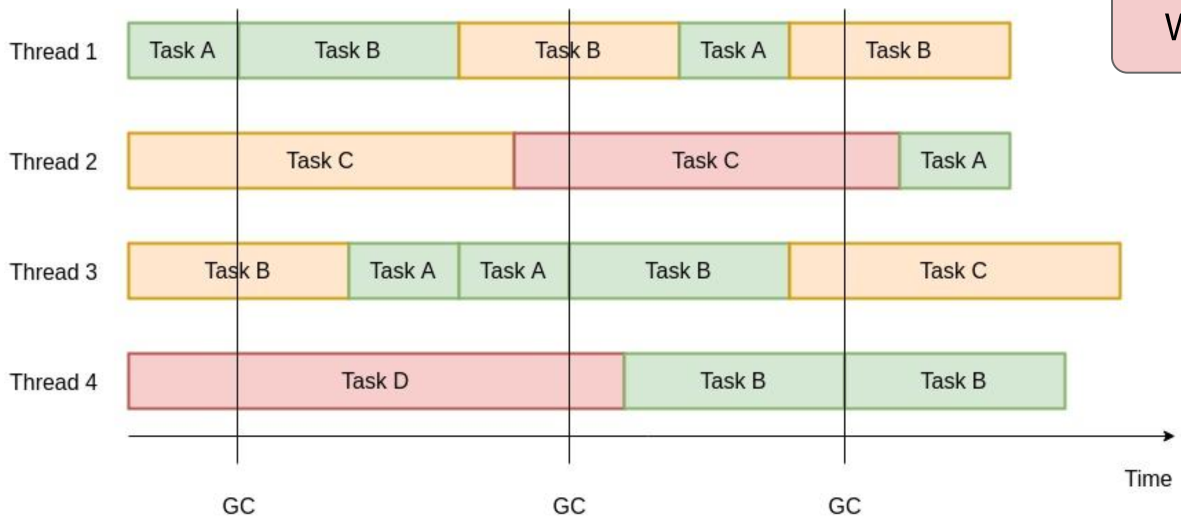


Big Data Application in HotSpot GCs



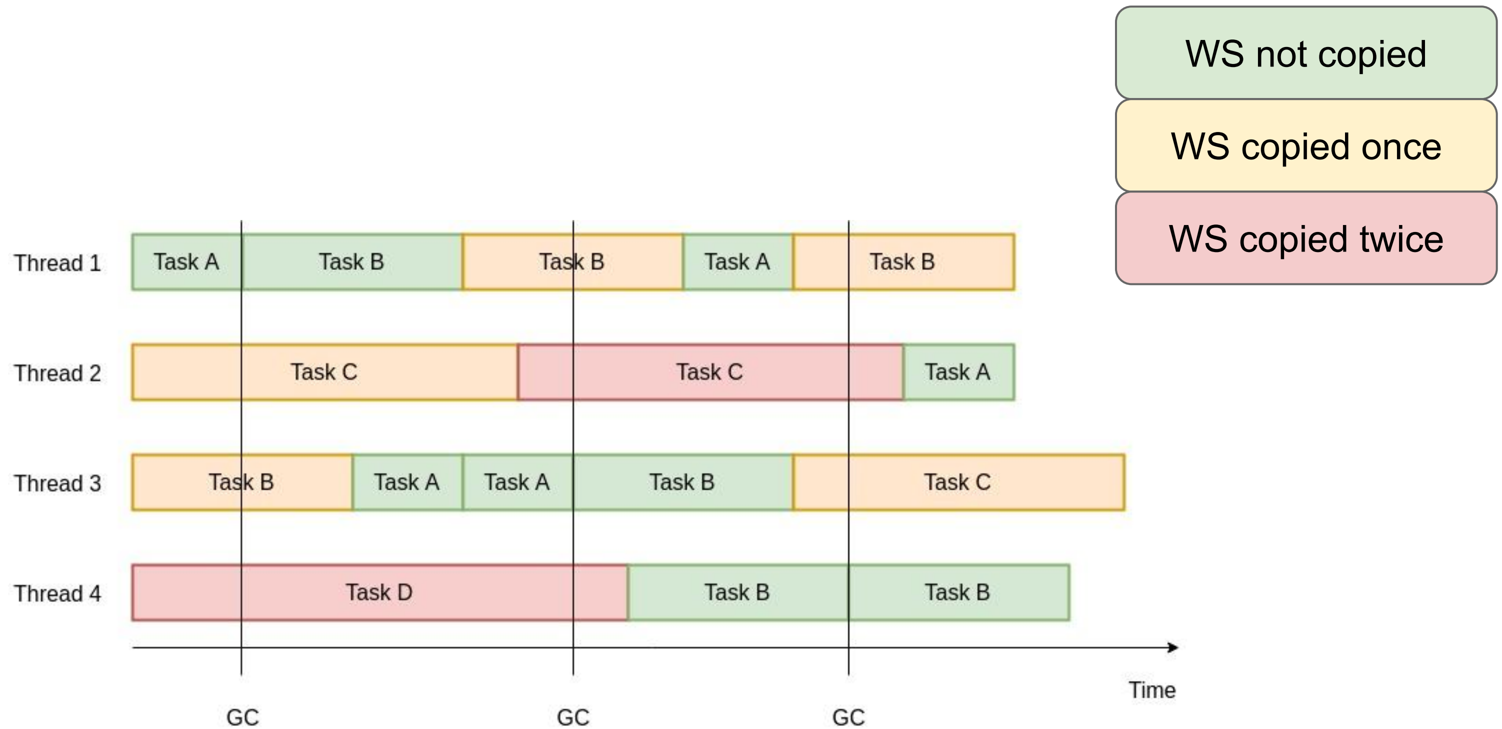
Big Data Application in HotSpot GCs

WS not copied
WS copied once
WS copied twice



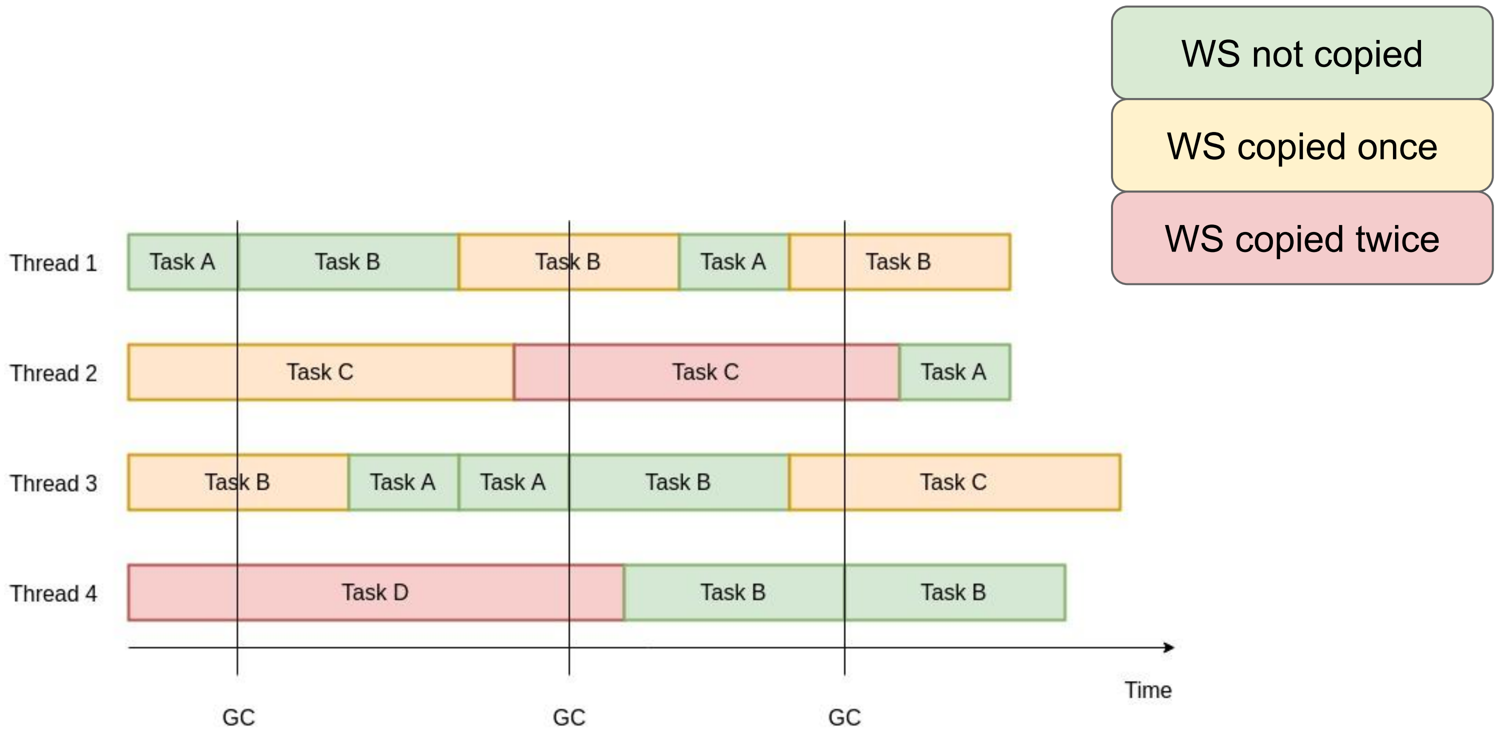
Object copy per GC cycle: 1500 MB
Total amount of object copy: 4500 MB

Big Data Application in HotSpot GCs



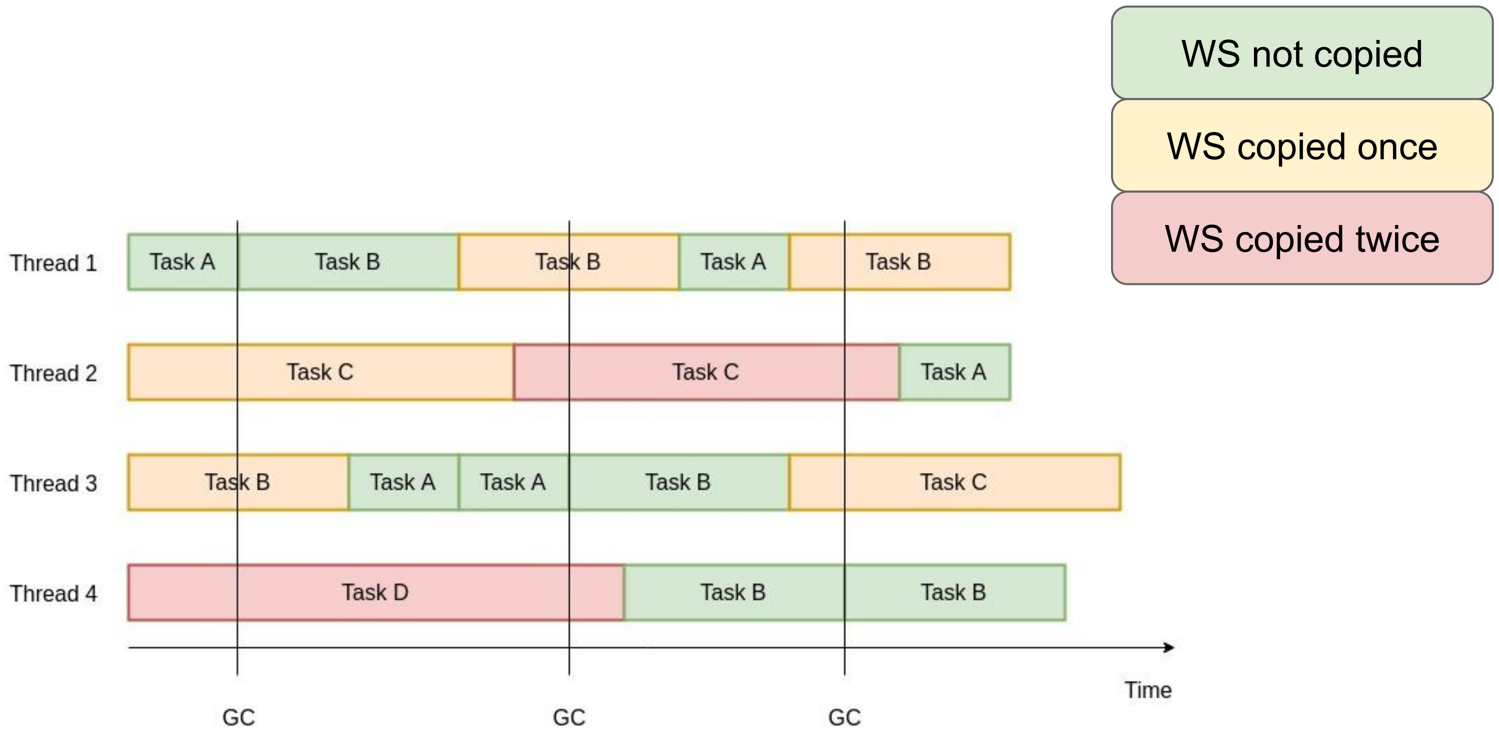
Object copy per GC cycle: 1500 MB
Total amount of object copy: 4500 MB
Assuming average RAM bandwidth of 10GB/s (DDR3)

Big Data Application in HotSpot GCs



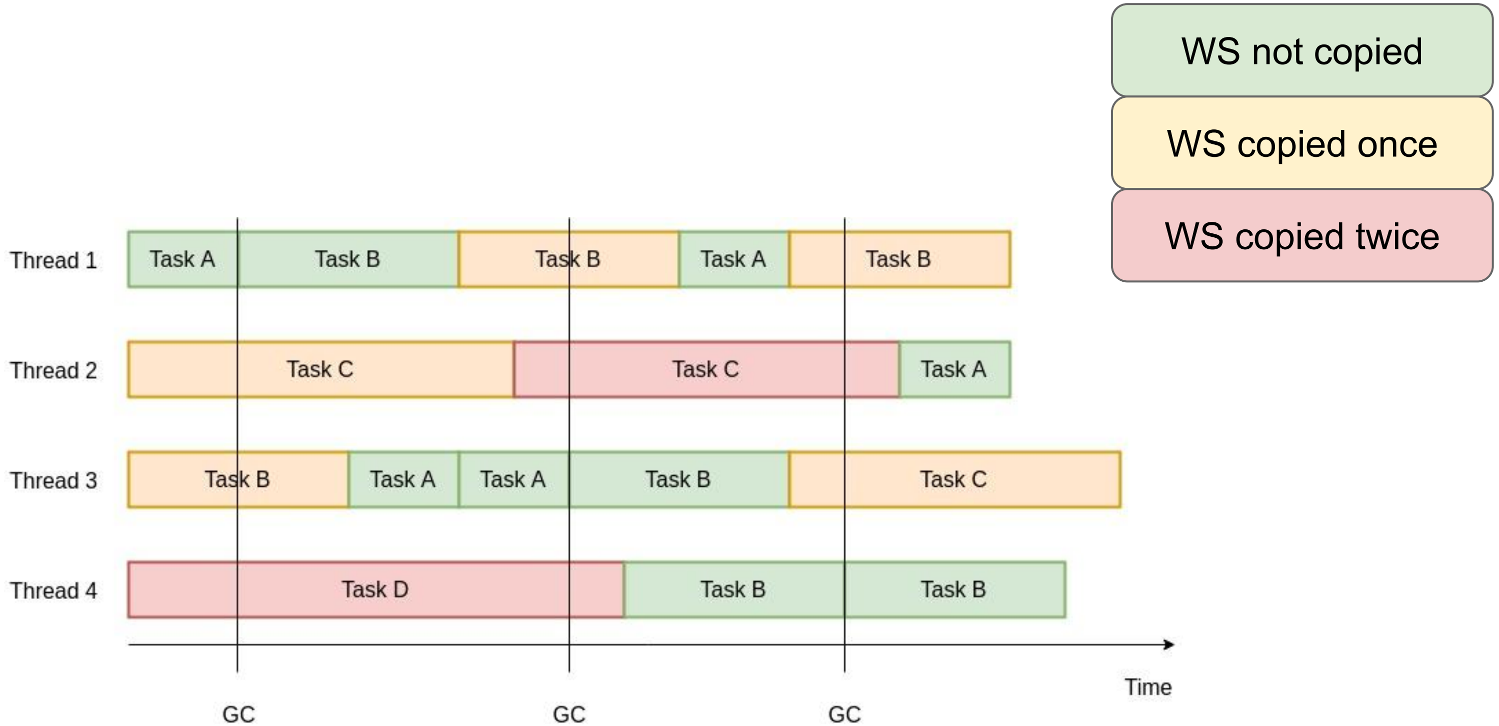
Object copy per GC cycle: 1500 MB
Total amount of object copy: 4500 MB
Assuming average RAM bandwidth of 10GB/s (DDR3)
4 Threads, Eden 2GB = copy 3 tasks (1500 MB) ~= **150 ms**

Big Data Application in HotSpot GCs



Object copy per GC cycle: 1500 MB
Total amount of object copy: 4500 MB
Assuming average RAM bandwidth of 10GB/s (DDR3)
4 Threads, Eden 2GB = copy 3 tasks (1500 MB) ~= **150 ms**
8 Threads, Eden 4GB = copy 7 tasks (3500 MB) ~= **350 ms**

Big Data Application in HotSpot GCs



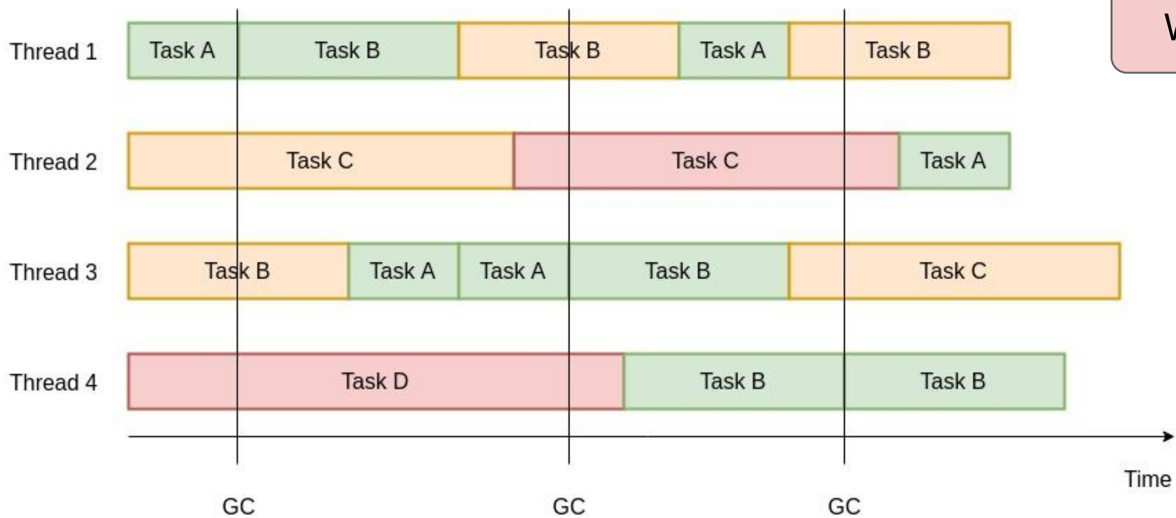
Object copy per GC cycle: 1500 MB
Total amount of object copy: 4500 MB
Assuming average RAM bandwidth of 10GB/s (DDR3)
4 Threads, Eden 2GB = copy 3 tasks (1500 MB) ~= **150 ms**
8 Threads, Eden 4GB = copy 7 tasks (3500 MB) ~= **350 ms**
16 Threads, Eden 8GB = copy 15 task (7500 MB) ~= **750 ms**

Big Data Application in HotSpot GCs

WS not copied

WS copied once

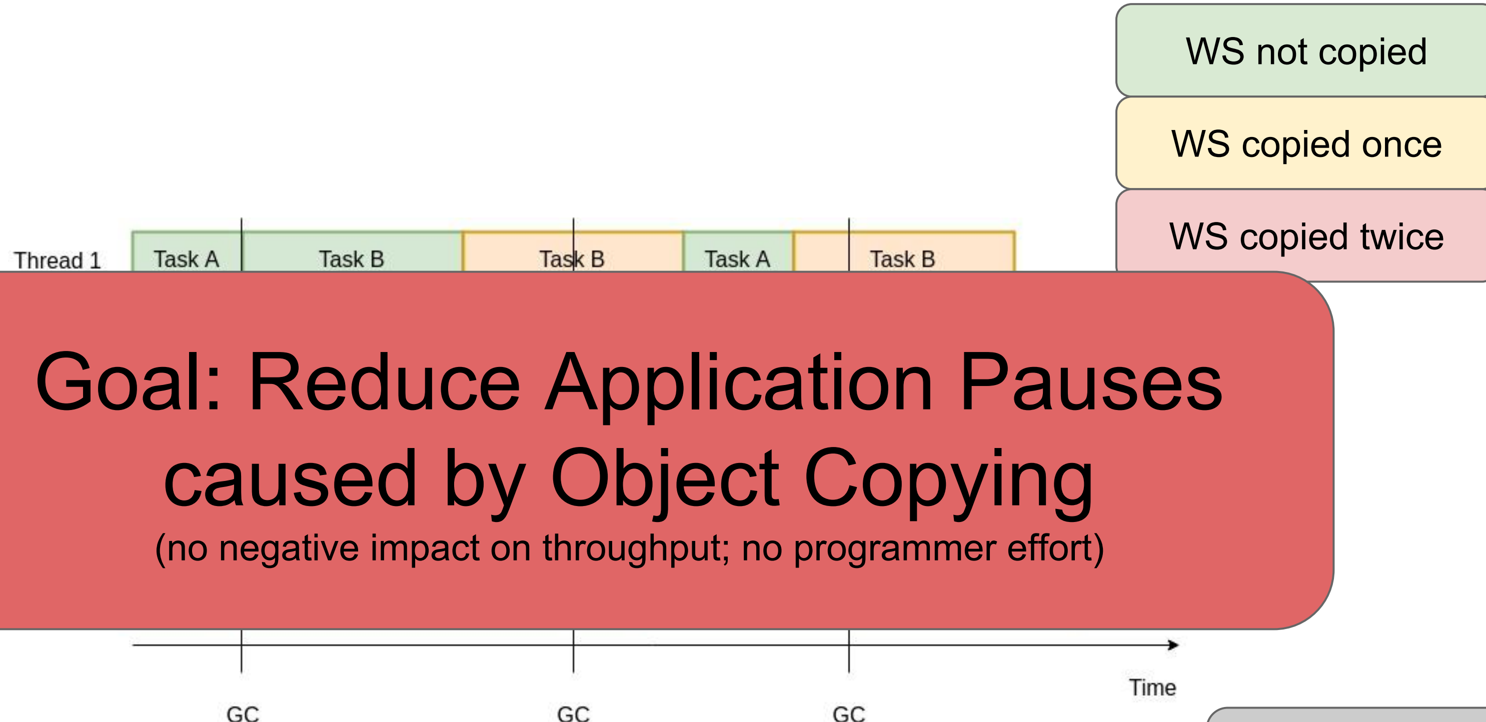
WS copied twice



Long Pauses!
Not Scalable!

Object copy per GC cycle: 1500 MB
Total amount of object copy: 4500 MB
Assuming average RAM bandwidth of 10GB/s (DDR3)
4 Threads, Eden 2GB = copy 3 tasks (1500 MB) ~= **150 ms**
8 Threads, Eden 4GB = copy 7 tasks (3500 MB) ~= **350 ms**
16 Threads, Eden 8GB = copy 15 task (7500 MB) ~= **750 ms**

Big Data Application in HotSpot GCs



Goal: Reduce Application Pauses caused by Object Copying

(no negative impact on throughput; no programmer effort)

Object copy per GC cycle: 1500 MB
Total amount of object copy: 4500 MB
Assuming average RAM bandwidth of 10GB/s (DDR3)
4 Threads, Eden 2GB = copy 3 tasks (1500 MB) ~= **150 ms**
8 Threads, Eden 4GB = copy 7 tasks (3500 MB) ~= **350 ms**
16 Threads, Eden 8GB = copy 15 task (7500 MB) ~= **750 ms**

Long Pauses!
Not Scalable!

How to Avoid en-masse Object Copying

- Attempt 1: Heap Resizing
  - ✓ Increase Young generation size;
  - ✓ Gives more time for objects to die;
  - ! Does not solve the problem, eventually the Young gen will get full and objects will be copied.
- Attempt 2: Reduce Task/Working Set size
  - ✓ Reduces the amount of object copying since the WS is smaller;
  - ! Increases overhead as more tasks and coordination is necessary to process smaller tasks.
- Attempt 3: Reuse data objects
  - ✓ Avoids allocating new memory for future Tasks;
  - ✓ Limits GC effort;
  - ! Requires major rewriting of applications combined with very unnatural Java programming style.
- Attempt 4: Off-heap memory
  - ✓ Reduces GC effort as data objects can reside in off-heap
  - ! Objects describing data objects still reside in the GC-managed heap
  - ! Requires manual memory management (defeats the purpose of running inside a managed heap).
- Attempt 5: Region-based/Scope-based memory allocation
  - ✓ Limits object's reachability by scope/region;
  - ✓ Limits GC effort as objects are automatically collected once the scope/region is discarded;
  - ! Requires major rewriting of existing applications;
  - ! Does not allow objects to freely move between scopes. Fits only to bag of tasks model.

How to Avoid en-masse Object Copying

- Attempt 1: Heap Resizing
  - ✓ Increase Young generation size;
  - ✓ Gives more time for objects to die;
  - ! Does not solve the problem, eventually the Young gen will get full and objects will be copied.
- Attempt 2: Re
  - ✓ Reduces th
  - ! Increases c aller tasks.
- Attempt 3: Re
  - ✓ Avoids allo
  - ✓ Limits GC e
  - ! Requires m gramming style.
- Attempt 4: Off-
  - ✓ Reduces GC effort as data objects can reside in off-heap
  - ! Objects describing data objects still reside in the GC-managed heap
  - ! Requires manual memory management (defeats the purpose of running inside a managed heap).
- Attempt 5: Region-based/Scope-based memory allocation
  - ✓ Limits object's reachability by scope/region;
  - ✓ Limits GC effort as objects are automatically collected once the scope/region is discarded;
  - ! Requires major rewriting of existing applications;
  - ! Does not allow objects to freely move between scopes. Fits only to bag of tasks model.

Takeaway:

- Avoiding massive object copying is non-trivial!
- Existing solutions only alleviate the problem!
- Existing solutions might work in some scenarios but do not provide a general solution.

Proposed Solution: NG2C

- Goals:
  - reduce en-masse object copying
    - From object promotion
    - From object compaction
  - avoid memory and/or throughput negative impact
  - require minimal programmer knowledge and effort.
- Overview:
  - Objects are pretenured/allocated into different dynamic generations
  - Dynamic generations
    - Memory segments that can be created and discarded at runtime
    - Hold objects with similar lifetimes

Proposed Solution: NG2C

- Goals:
  - reduce en-masse object copying
    - From object promotion
    - From object compaction
  - avoid memory and/or throughput negative impact
  - require minimal programmer knowledge and effort.
- Overview:
  - Objects are pretenured/allocated into different dynamic generations
  - Dyna… …runtime

In short: allocate objects close to each other as long as they have similar lifetimes

Outline

- NG2C - Pretenuring GC with Dynamic Generations
  - Pretenuring into Dynamic Generations
  - Application Example
  - Memory Collection
- Implementation
- Evaluation
  - Environment & Workloads
  - Programmer Effort
  - GC Pause Times
  - Throughput
- Conclusions
- Future Work

NG2C - Pretenuring into Dynamic Generations

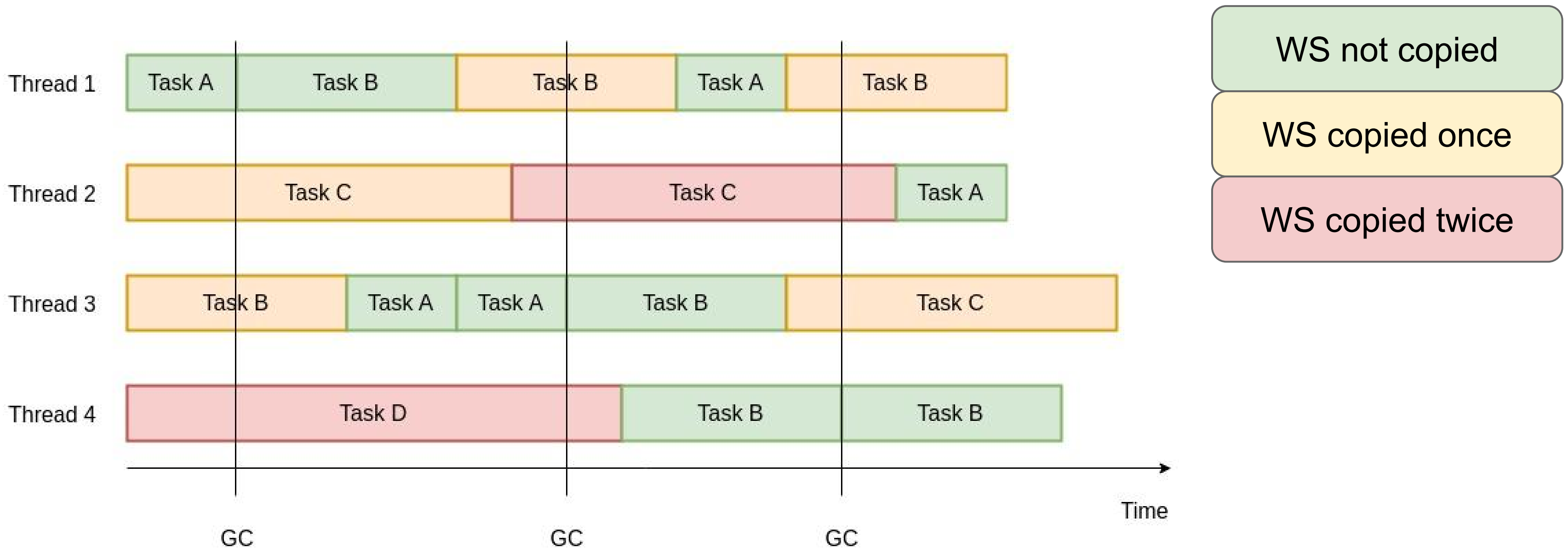
- NG2C combines:
  - **Pretenuring**: allocation of objects in older spaces;
  - **Dynamic Generations**: memory segments that hold objects with similar lifetimes. Dynamic generations can be created and destroyed at runtime.

- Pretenuring avoids costly promotion
  - Because objects are not copied around

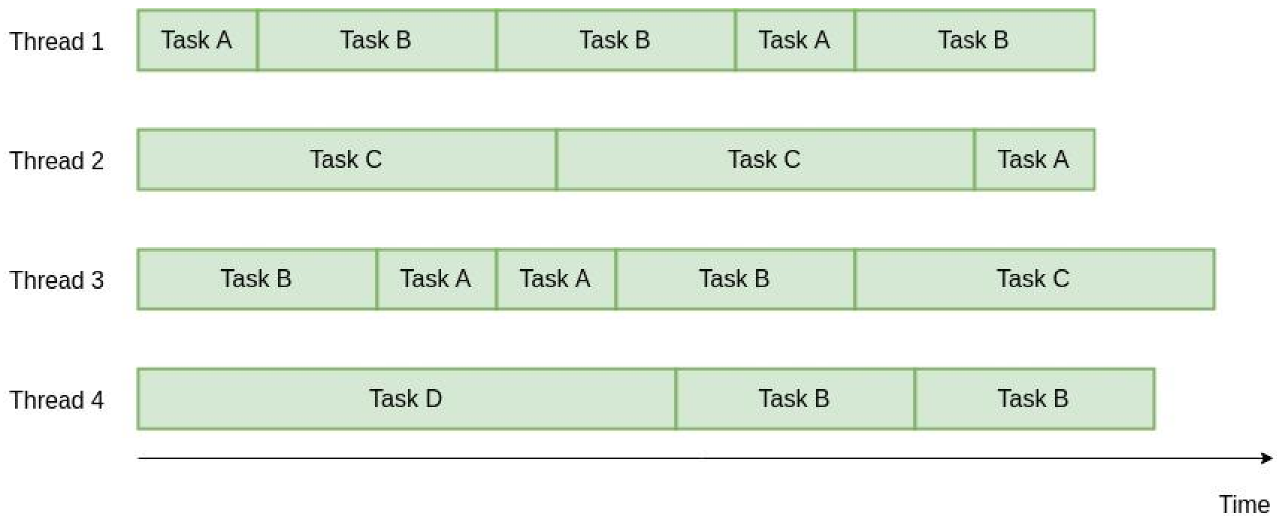
- Dynamic generations are effortlessly collected
  - Because most objects die approximately at the same time
    - I.e., no compaction needed

- NG2C provides a simple API that can be used
  - to select which objects should be pretenured
    - By using a special annotation
  - into which dynamic generation
    - By controlling the current target generation (per-thread)

NG2C - Application Example



NG2C - Application Example

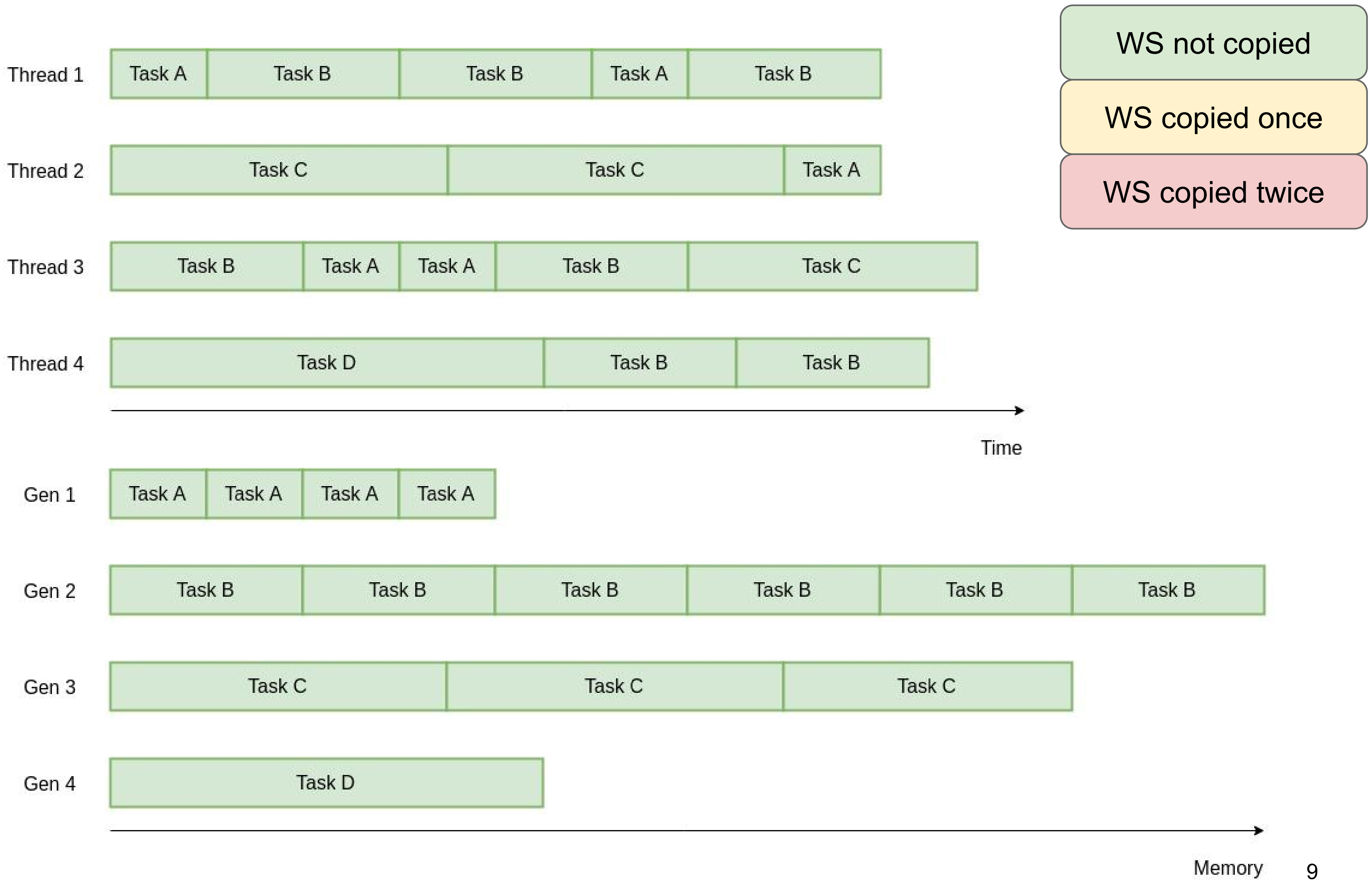


WS not copied

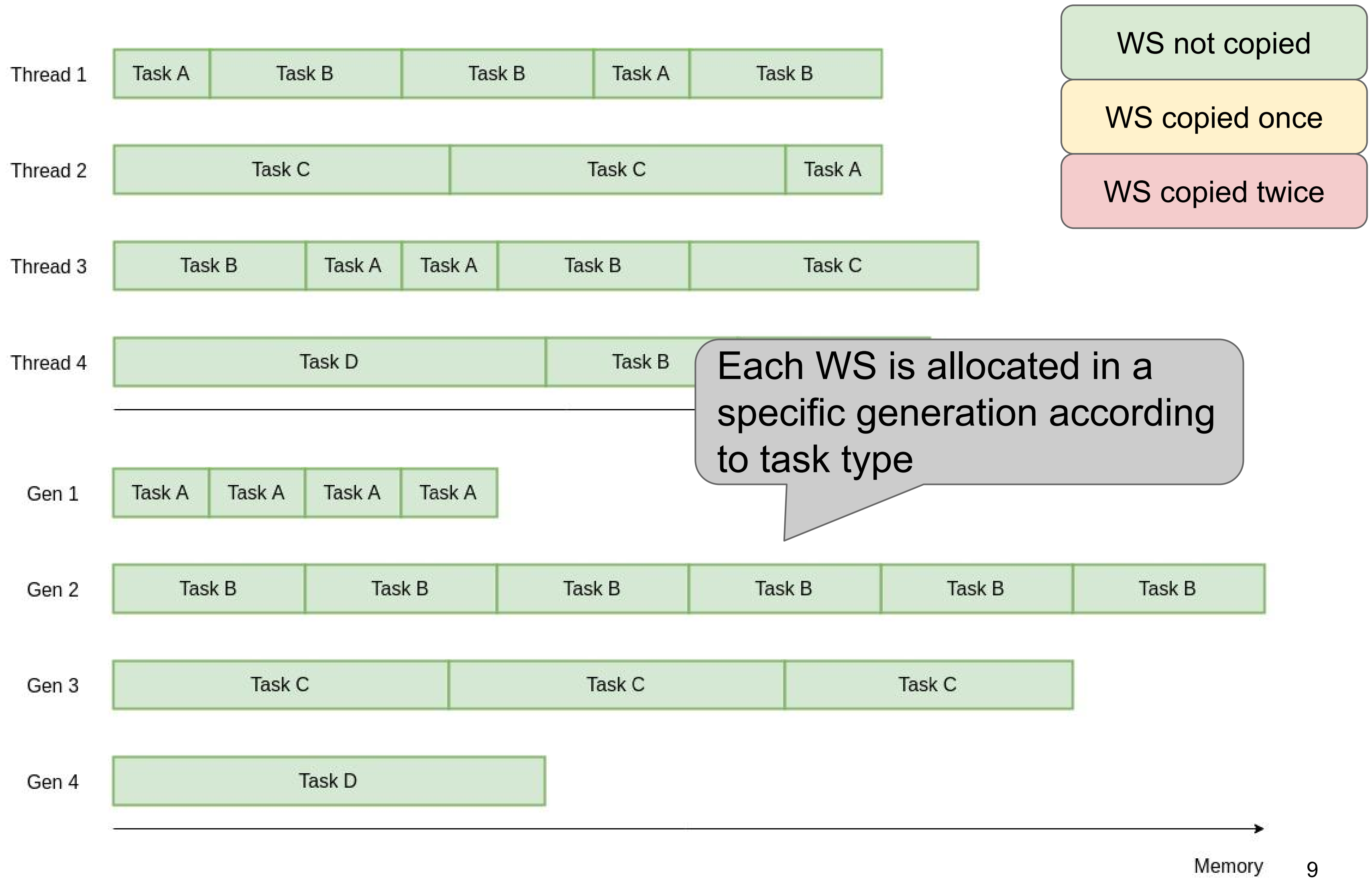
WS copied once

WS copied twice

NG2C - Application Example



NG2C - Application Example



NG2C - Application Example

```java
static Generation[] generations = new Generation[TaskType.values().length];

static {
  for (int i = 0; i < TaskType.values().length; i++) {
    generations[i] = System.newGeneration();
  }
}

public void runTask(enum TaskType tt) {

  // Selects Target Generation for current thread
  System.setGeneration(generations[tt]);

  // Allocates memory to hold Working Set
  WorkingItem[] buffer = new @Gen WorkingItem[WS_SIZE];

  // Loads Working Set
  DataProvider.load(tt, buffer);

  // Process Working Set
  Result r = DataProcessor.process(tt, buffer);

  // Pushes results from computation
  Output.push(r);
}
```

NG2C - Application Example

```
static Generation[] generations = new Generation[TaskType.values().length];

static {
  for (int i = 0; i < TaskType.values().length; i++) {
    generations[i] = System.newGeneration();
  }
}

public void runTask(enum TaskType tt) {

  // Selects Target Generation for current thread
  System.setGeneration(generations[tt]);

  // Allocates memory to hold Working Set
  WorkingItem[] buffer = new @Gen WorkingItem[WS_SIZE];

  // Loads Working Set
  DataProvider.load(tt, buffer);

  // Process Working Set
  Result r = DataProcessor.process(tt, buffer);

  // Pushes results from computation
  Output.push(r);
}
```

Creates new generation for each task type

NG2C - Application Example

```java
static Generation[] generations = new Generation[TaskType.values().length];

static {
  for (int i = 0; i < TaskType.values().length; i++) {
    generations[i] = System.newGeneration();
  }
}

public void runTask(enum TaskType tt) {

  // Selects Target Generation for current thread
  System.setGeneration(generations[tt]);

  // Allocates memory to hold Working Set
  WorkingItem[] buffer = new @Gen WorkingItem[WS_SIZE];

  // Loads Working Set
  DataProvider.load(tt, buffer);

  // Process Working Set
  Result r = DataProcessor.process(tt, buffer);

  // Pushes results from computation
  Output.push(r);
}
```

Creates new generation for each task type

Selects the correct Dynamic Generation for allocating data for this task.

TÉCNICO LISBOA

gsd inesc id lisboa

NG2C - Application Example

```java
static Generation[] generations = new Generation[TaskType.values().length];

static {
  for (int i = 0; i < TaskType.values().length; i++) {
    generations[i] = System.newGeneration();
  }
}

public void runTask(enum TaskType tt) {

  // Selects Target Generation for current thread
  System.setGeneration(generations[tt]);

  // Allocates memory to hold Working Set
  WorkingItem[] buffer = new @Gen WorkingItem[WS_SIZE];

  // Loads Working Set
  DataProvider.load(tt, buffer);

  // Process Working Set
  Result r = DataProcessor.process(tt, buffer);

  // Pushes results from computation
  Output.push(r);
}
```

Creates new generation for each task type

Selects the correct Dynamic Generation for allocating data for this task.

Informs NG2C that this allocation should go into the current generation.

NG2C - Application Example

```java
static Generation[] generations = new Generation[TaskType.values().length];

static {
  for (int i = 0; i < TaskType.values().length; i++) {
    generations[i] = System.newGeneration();
  }
}

public void runTask(enum TaskType tt) {

  // Selects Target Generation for current thread
  System.setGeneration(generations[tt]);

  // Allocates memory to hold Working Set
  WorkingItem[] buffer = new @Gen WorkingItem[WS_SIZE];

  // Loads Working Set
  DataProvider.load(tt, buffer);

  // Process Working Set
  Result r = DataProcessor.process(tt, buffer);

  // Pushes results from computation
  Output.push(r);
}
```

Creates new generation for each task type

Selects the correct Dynamic Generation for allocating data for this task.

Informs NG2C that this allocation should go into the current generation.

We provide a tool that helps the programmer to identify where and how to instrument the code.

NG2C - Memory Collection

- NG2C memory collection algorithms are inherited from
  - Garbage First (Detlefs, 2004)
- Types of GC cycles:
  - **Minor GC** (inherited from G1): Young generation is collected. Surviving objects are moved to survivor spaces or to the Old generation.
  - **Mixed GC** (adapted from G1): besides collecting the Young generation, a Mixed GC might also collect memory from other generations, including dynamic generations. Survivor objects are moved to the Old generation.
  - **Full GC** (adapted from G1): collects all generations. Survivors are moved to the Old generation. Should be avoided at all cost.
- **Concurrent Marking**:
  - Traverses the heap marking reachable objects
  - Collects unreachable memory blocks
    - Most efficient way of collecting dynamic generations

Implementation

- Implemented for the OpenJDK 8 HotSpot JVM
  - Not a toy implementation

- Built on top of G1, the new by-default collector;

- Extends JVM to allow object allocation and collection in any generation:
  - Code interpretation
  - Code JIT
  - TLAB management
  - Heap Region management
  - Remembered Set management
  - …

- Approx. 2000 LOC

TÉCNICO LISBOA

gsd inesc id lisboa

Evaluation

- Evaluate NG2C's performance compared to:
  - CMS and G1 - popular OpenJDK GCs
  - C4 - Zing GC

- Big Data Platforms & Workloads:
  - Cassandra (**Key-Value Store**)
    - Feedzai (credit-card transaction validation)
      - Real world based workload (mixes reads and writes)
    - Synthetic workloads (YCSB)
      - Write-Intensive (75% writes), Read-Intensive (75% reads)
  - Lucene (**In-Memory Indexing Tool**)
    - Read/Write transactions on Wikipedia dump (33M documents)
      - Write-intensive (80% writes)
  - GraphChi (**Graph Processing Engine**)
    - Twitter graph dump (42M vertexes, 1.5B edges) processing
      - PageRank
      - Connected Components

Evaluation

Evaluation Uses:

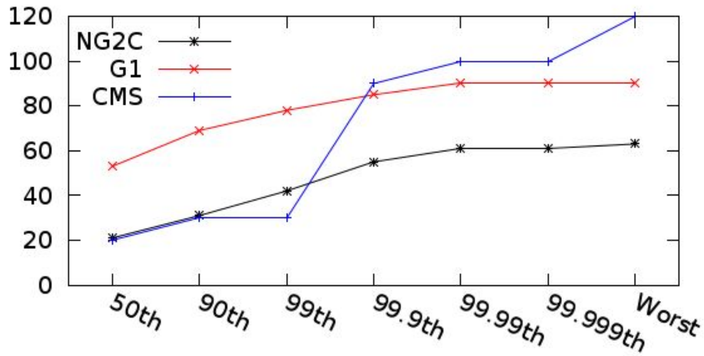
- Real world platforms (Cassandra, Lucene)
- Real data (Lucene, GraphChi)
- Real Workloads (Feedzai)

- Platforms & Workloads:
  - Cassandra (Key-Value Store)
    - Feedzai (credit-card transaction validation)
      - Real world based workload (mixes reads and writes)
    - Synthetic workloads (YCSB)
      - Write-Intensive (75% writes), Read-Intensive (75% reads)
  - Lucene (In-Memory Indexing Tool)
    - Read/Write transactions on Wikipedia dump (33M documents)
      - Write-intensive (80% writes)
  - GraphChi (Graph Processing Engine)
    - Twitter graph dump (42M vertexes, 1.5B edges) processing
      - PageRank
      - Connected Components

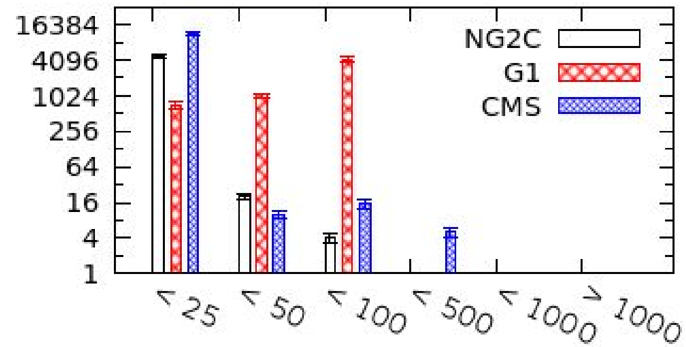
Evaluation - Environment

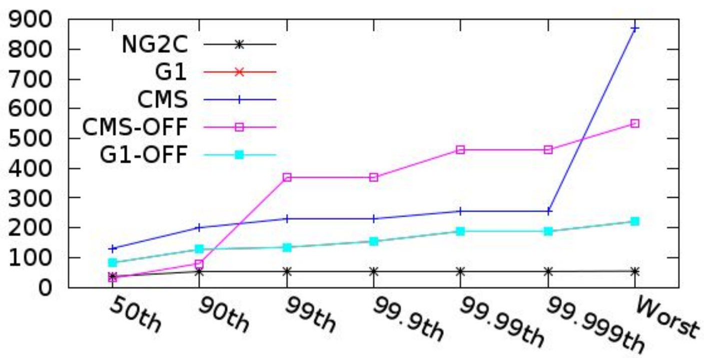
Platform	Workload	CPU	RAM	OS	Heap Size	Young Size
Cassandra	Feedzai	Intel Xeon E5-2680	64GB	CentOS 6.7	30GB	4GB
Cassandra	RI,WI	Intel Xeon E5505	16GB	Linux 3.13	12GB	2GB
Lucene	RW	AMD Opteron 6168	128GB	Linux 3.16	120GB	2GB
GraphChi	PR,CC	AMD Opteron 6168	128GB	Linux 3.16	120GB	6GB

Evaluation - Pause Times (Cassandra)

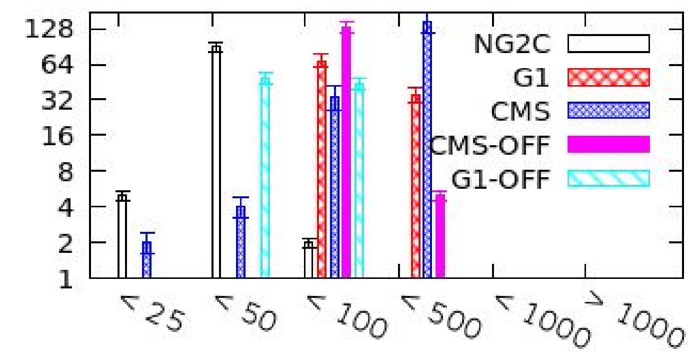


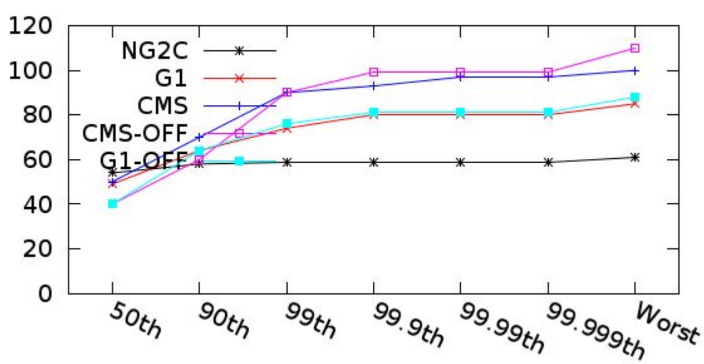
Feedzai



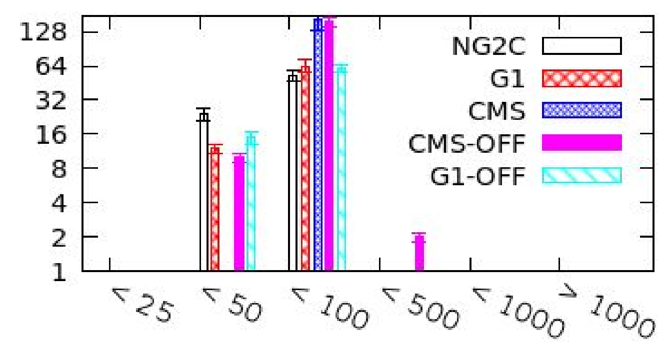


Write-Intensive

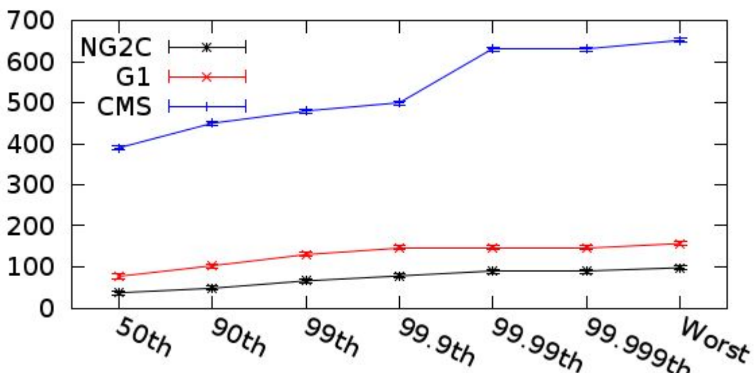




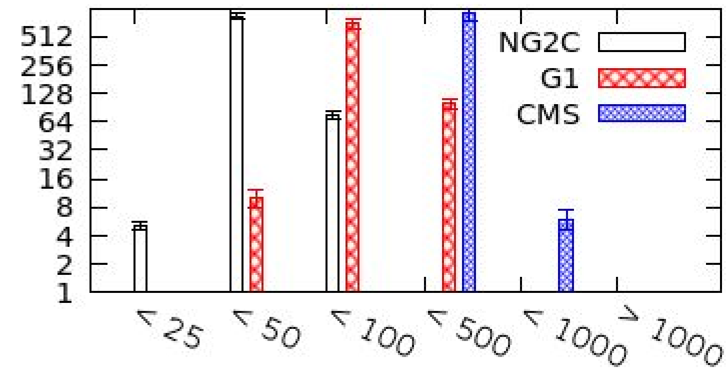
Read-Intensive

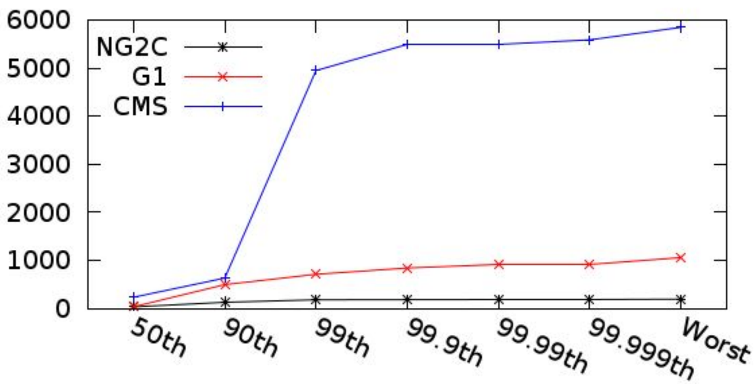


Evaluation - Pause Times (Lucene and GraphChi)

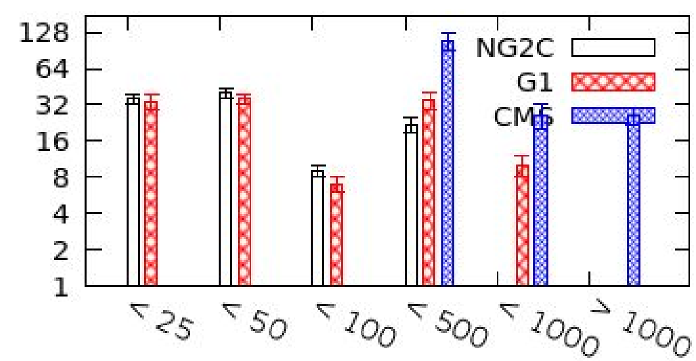


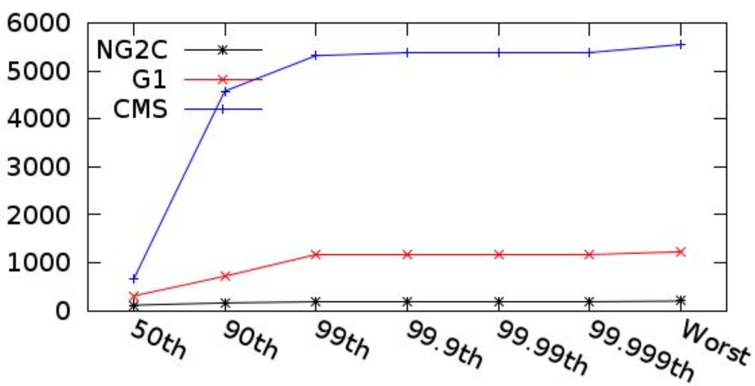
Lucene



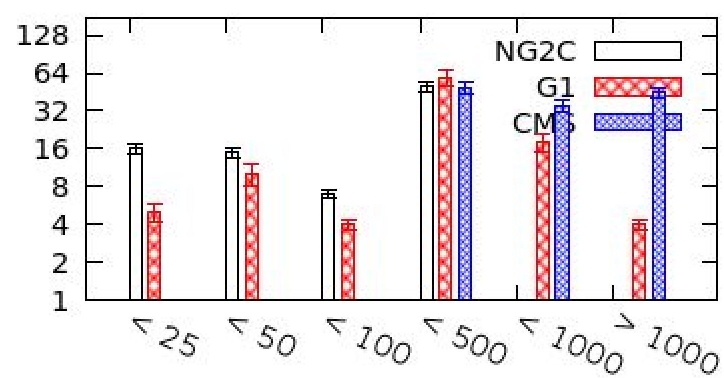


GraphChi CC

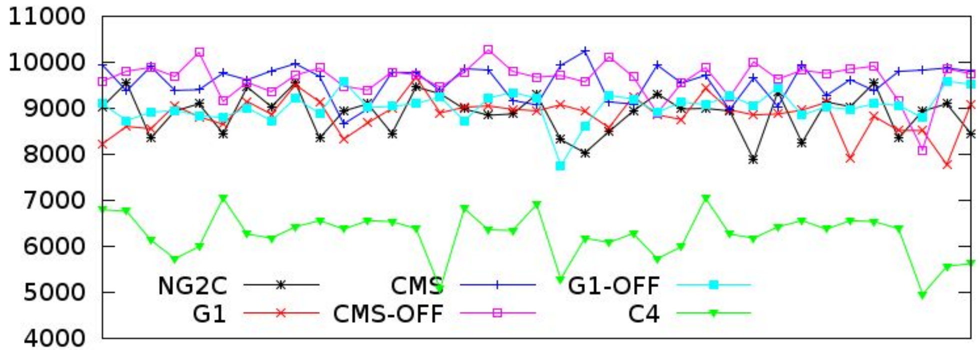




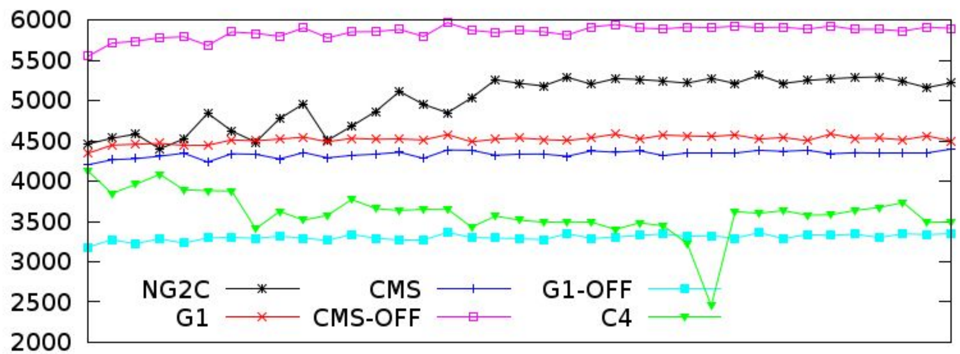
GraphChi PR



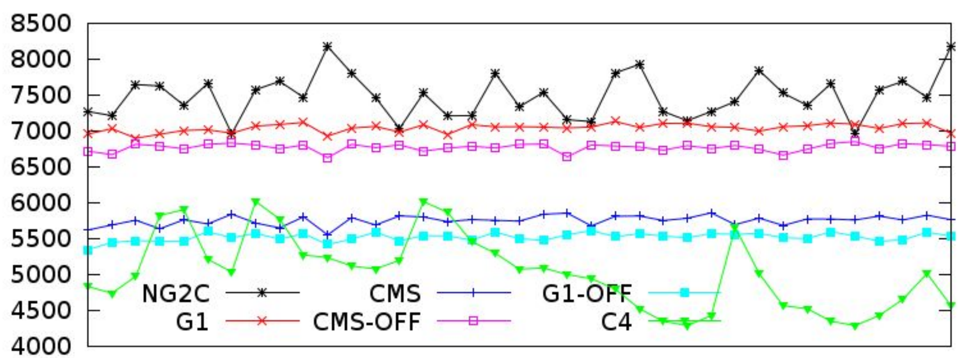
Evaluation - Throughput (Cassandra) - 10 min sample



Write-Intensive



Read-Intensive



Read-Write

More results in the paper

Evaluation - Programmer Effort

- Code changes to use NG2C:
    - Cassandra
        - **11** code locations with @Gen
        - **11** code locations NG2C API calls
    - Lucene
        - **8** code locations with @Gen
    - GraphChi
        - **9** code locations with @Gen
- Code changes suggested by the Object Lifetime Recorder (OLR)
    - We profiled each platform for 10 mins
        - Enough for the workload to stabilize

More details in the paper

Conclusions

- NG2C provides a realistic approach to improve Big Data application memory management in HotSpot
  - It decreases pause times by avoiding object copying
  - It requires minimal programmer effort and knowledge
  - It does not compromise throughput
- Results are very encouraging
- NG2C is implemented for HotSpot 8
  - Code is available at github.com/rodrigo-bruno/ng2c

Future Work

- Improve Object Lifetime Recorder and automatically rewrite bytecode at load time to incorporate NG2C API calls and annotation
  - Completely replaces programmer effort and knowledge
  - Work is being peer-reviewed
- Provide in-JVM support for dynamic generations and pretenuring
  - JVM must internally estimate the appropriate generation for each alloc. site
  - JVM must dynamically change the target generation for each alloc. site
  - Work in progress
    - Current prototype leads to up to 6% performance degradation for Cassandra
    - There are still several performance improvements to be done

Thank you for your time.
Questions?

Rodrigo Bruno
email: rodrigo.bruno@tecnico.ulisboa.pt
webpage: www.gsd.inesc-id.pt/~rbruno
ng2c's github: github.com/rodrigo-bruno/ng2c

NG2C - Object Lifetime Recorder