RT-LAGC: Fragmentation-Tolerant Real-Time Memory Management Revisited

Isabella Stilkerich, Michael Strotz, **Christoph Erhardt**, Michael Stilkerich

Department of Computer Science 4
Distributed Systems and Operating Systems

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

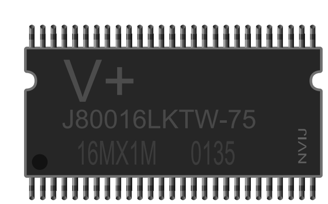
Real-Time Embedded Java

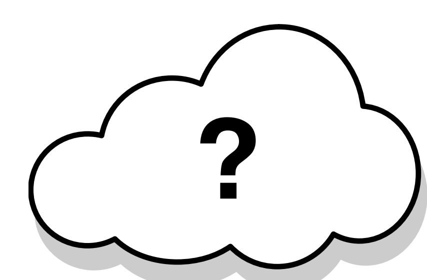
- **Productivity**
- **Safety**
  - Software-based memory protection
- **Efficiency**
  - Ahead-of-time compilation
  - Static configuration



Real-Time Embedded Java

- **Productivity**
- **Safety**
  - Software-based memory protection
- **Efficiency**
  - Ahead-of-time compilation
  - Static configuration
- **… Memory management?**
  - In particular: predictability?





Real-Time Memory Management

- **Pseudo-static allocation** (`ImmortalMemory`)
  - Bump pointer
  - No memory release

Real-Time Memory Management

- **Pseudo-static allocation** (`ImmortalMemory`)
  - Bump pointer
  - No memory release
- **Region-based allocation** (`ScopedMemory`)
  - Bump pointer
  - Region released as a whole

Real-Time Memory Management

- **Pseudo-static allocation** (`ImmortalMemory`)
  - Bump pointer
  - No memory release
- **Region-based allocation** (`ScopedMemory`)
  - Bump pointer
  - Region released as a whole

Simple, predictable, special-purpose

Real-Time Memory Management

- **Pseudo-static allocation** (`ImmortalMemory`)
  - Bump pointer
  - No memory release
- **Region-based allocation** (`ScopedMemory`)
  - Bump pointer
  - Region released as a whole

} Simple, predictable, special-purpose

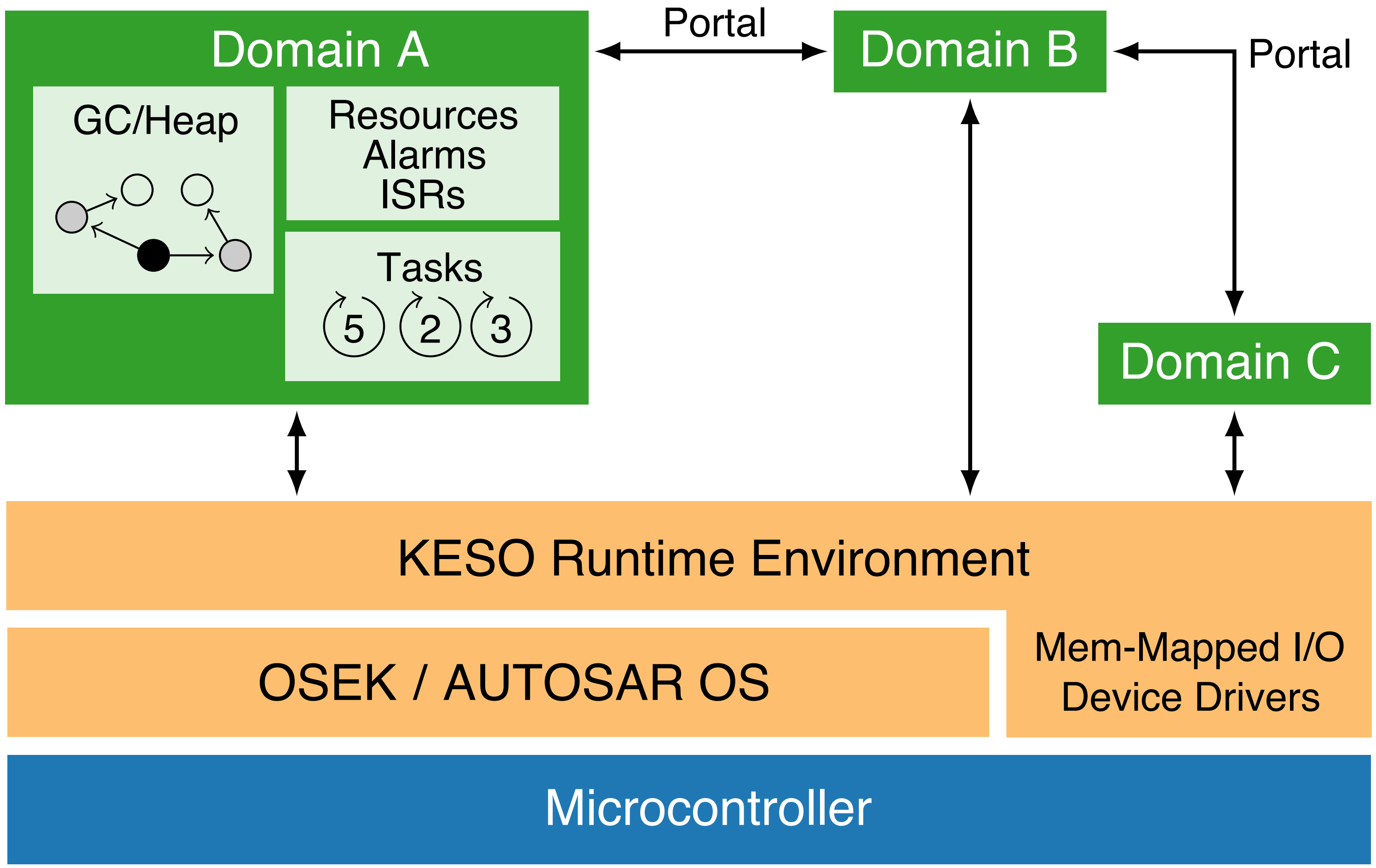
- **Automatic memory management**
  - Complex, list-like heap structure
  - Garbage collection
  - → Heap fragmentation, unpredictable allocation times
  - → Unpredictable latencies

Agenda

- **Goal: real-time-capable automatic memory management**
  - Bounded allocation times
  - Bounded latencies
  - Good throughput

- **Implementation in the KESO JVM**
  - ***R**eal-**T**ime **L**atency-**A**ware **G**arbage **C**ollection*
- **Evaluation**

The KESO JVM



- Ahead-of-time compilation to C code
- Static configuration
- "Pay only for what you use"

Garbage Collectors in KESO

1. **No GC**
2. **Slack-based mark-and-sweep GC**
    a) Stop-the-world
        - Not preemptible
    b) Incremental
        - Preemptible by application tasks
        - Synchronisation, write barriers

- **Configurable per domain**

Garbage Collectors in KESO

1. **No GC**
2. **Slack-based mark-and-sweep GC**
   a) Stop-the-world
      - Not preemptible

   b) Incremental
      - Preemptible by application tasks
      - Synchronisation, write barriers

- **Configurable per domain**

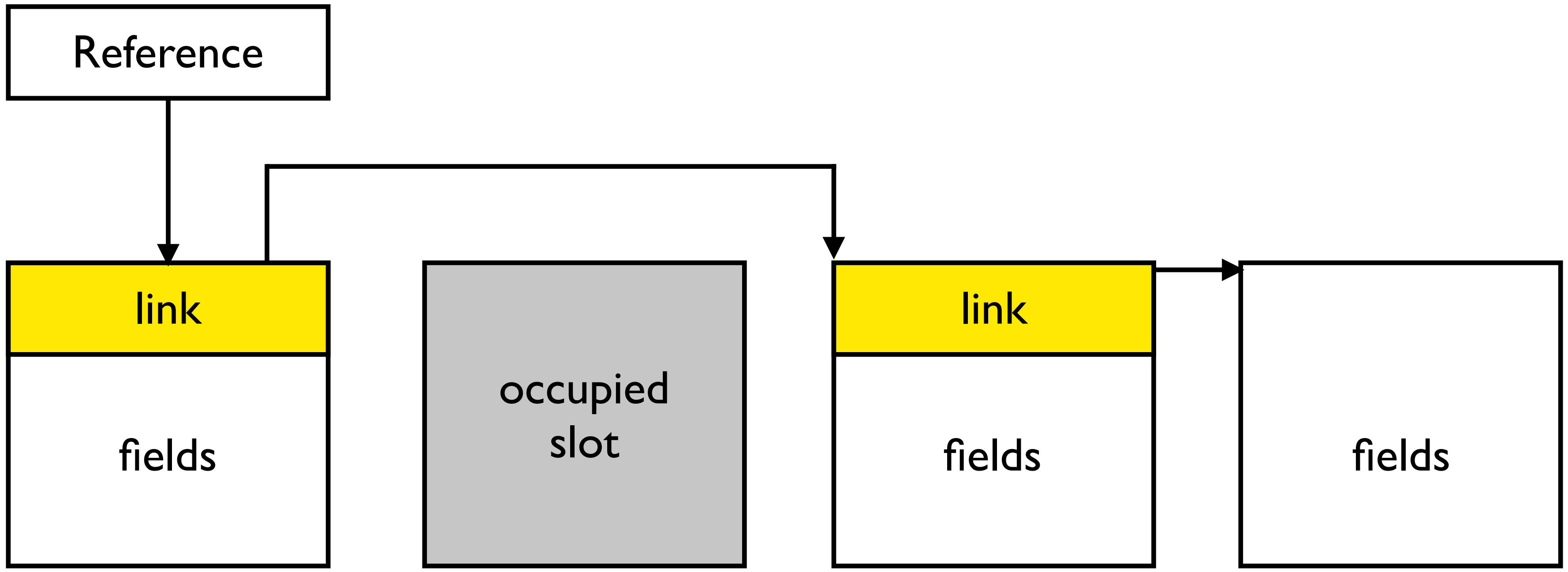
Extend for better predictability

Fragmented Objects

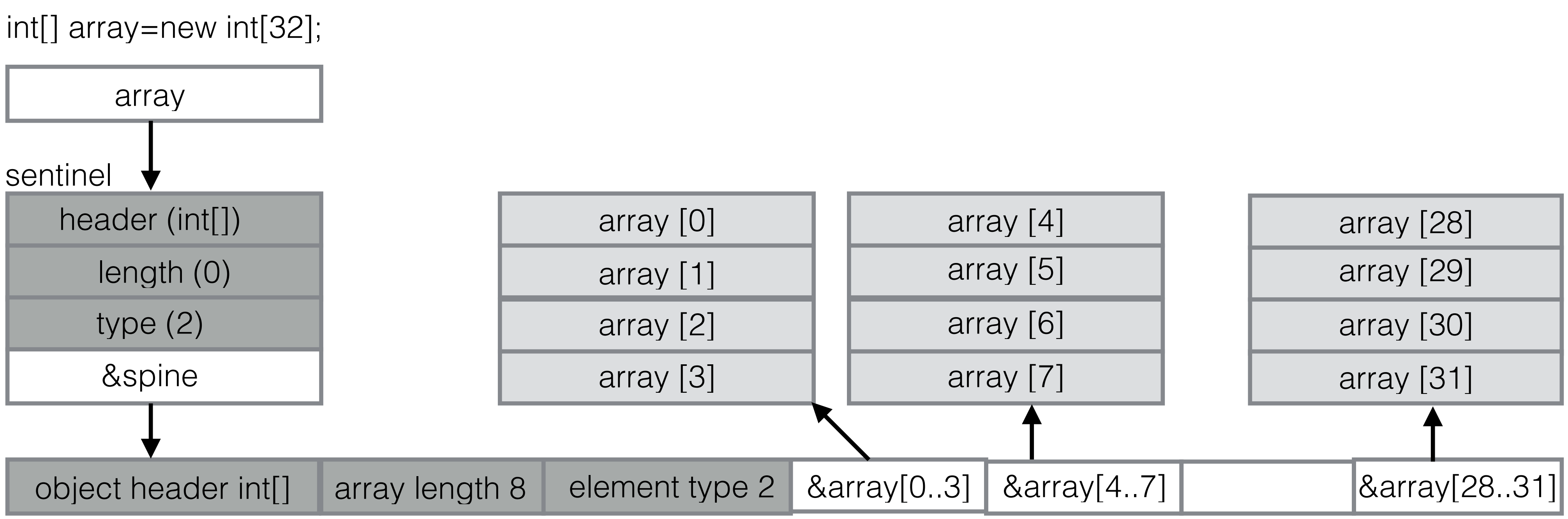
- **Traditional heap**
  - Variable-size chunks → heap fragmentation
  - Unpredictable allocation time

Fragmented Objects

- **Traditional heap**
  - Variable-size chunks → heap fragmentation
  - Unpredictable allocation time
- **Fragmented allocation**
  - Fixed-size slots → no heap fragmentation possible
  - Fragmented objects
  - Predictable allocation time

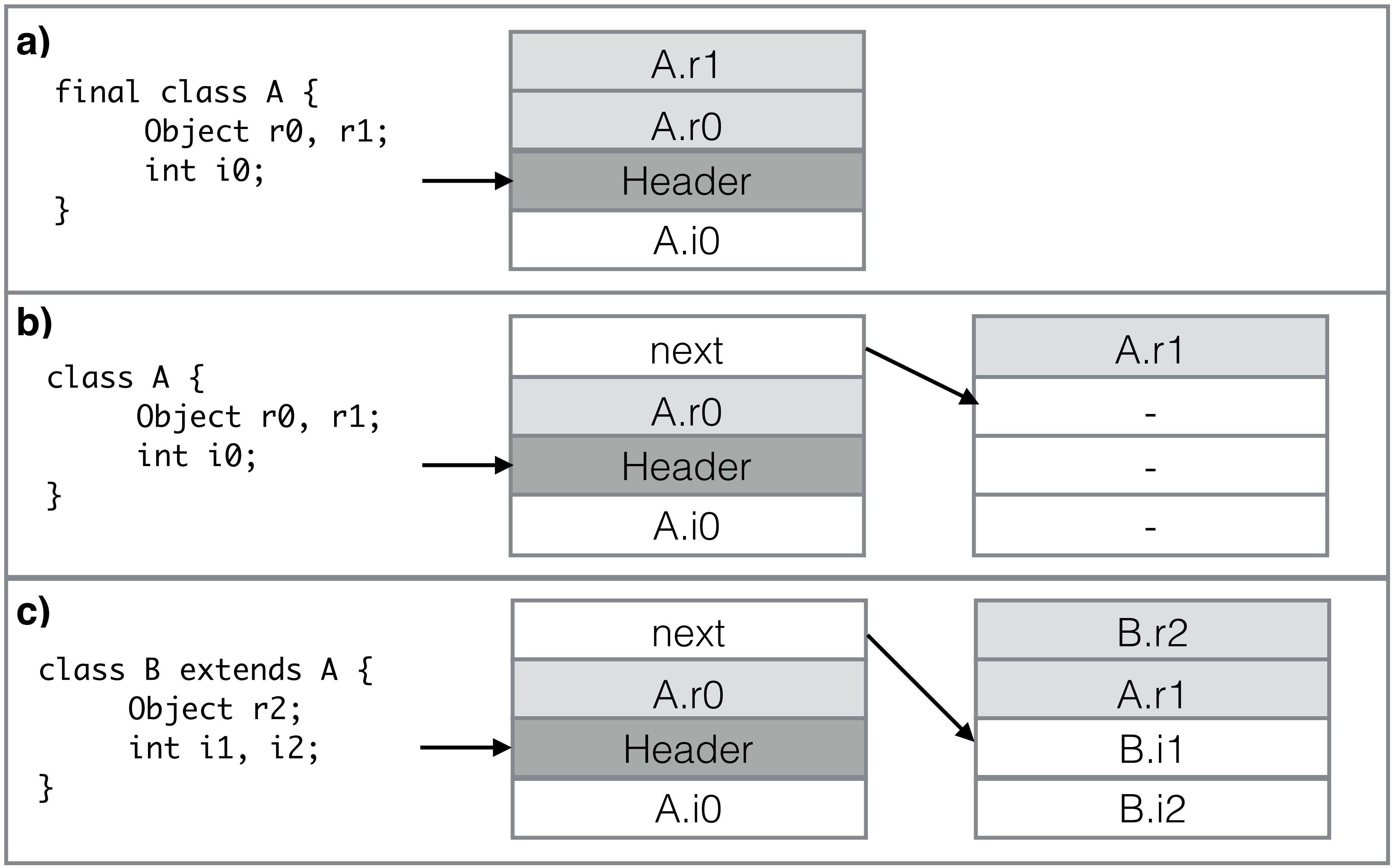


Fragmented Arrays



- **Spines**
  - Constant access complexity
  - Allocated in separate heap space
    - Bump pointer
    - Replicating or generational GC
- **Optimisation: omit spine if all fragments are contiguous**

Object Layout



- Reference grouping for efficient scanning by GC
- ***F**ragmented **B**idirectional **O**bject **L**ayout*

Scanning the Object Graph

- **Static references**
  - Grouped together in `.data` section
- **Non-static reference fields**
  - Grouped together in object fragments
- **Local references**
  - Henderson’s linked stack frames
    - Only generated for blocking methods
  - Only scan stacks of currently blocked tasks

Write Barriers

- **Whenever a reference is written, mark its pointee grey**
  - Significant application overhead

Write Barriers

- **Whenever a reference is written, mark its pointee grey**
  - Significant application overhead
- **Trade-off: latency vs. performance**
  - Omit write barriers for local references
  - During stack scanning, raise GC's priority above stack owner's
    - Short, bounded time (sans recursion)
    - GC still preemptible by higher-priority tasks
  - Also possible: omit write barriers for static fields

Improving GC Predictability

- **Observation: GC runtime dominated by mark phase**
  - Object-graph traversal expensive
  - Only surviving heap objects contribute
- **Survivability analysis**
  - Per-object lifespan categorisation
  - Based on escape analysis and call-graph analysis
  - Object may survive GC if reference …
    - … escapes globally
    - … is live beyond blocking system calls

Survivability Classification

- **Method-local**
  - Automatic stack allocation

Survivability Classification

- **Method-local**
  - Automatic stack allocation
- **Region-local**
  - Automatic region-based allocation

Survivability Classification

- **Method-local**
  - Automatic stack allocation
- **Region-local**
  - Automatic region-based allocation
- **Task-local**
  - Per-task heaps

Survivability Classification

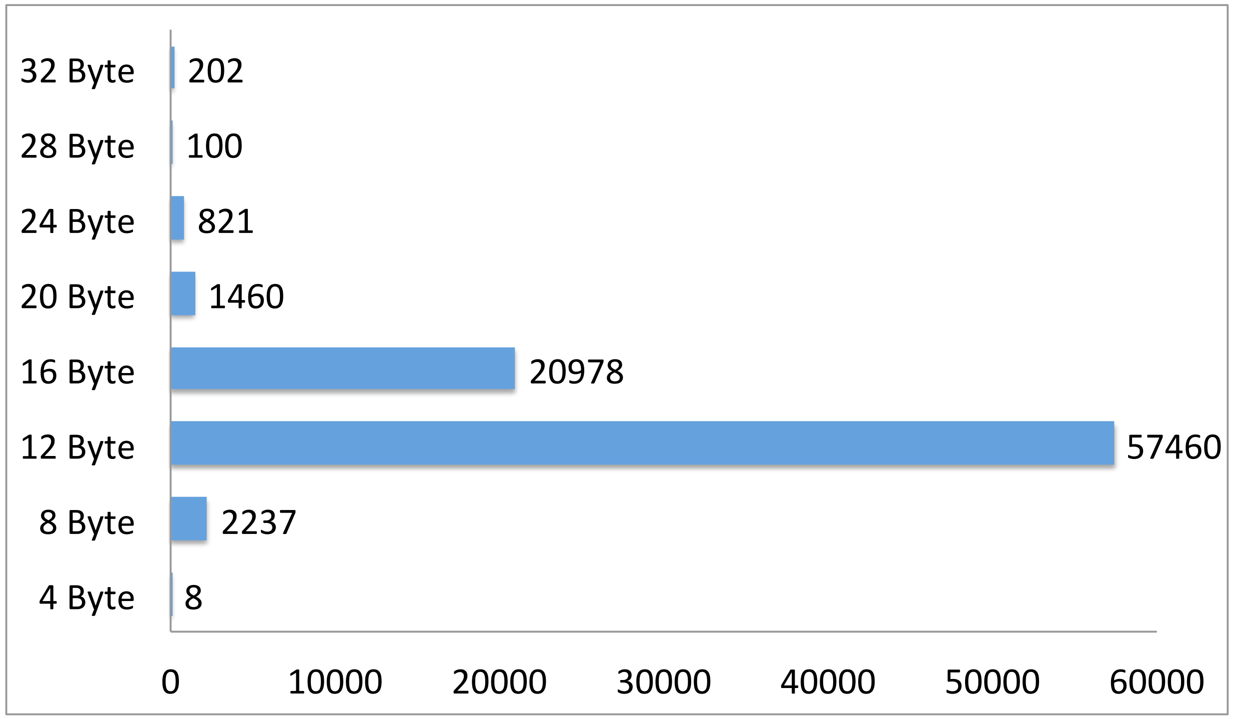
- **Method-local**
  - Automatic stack allocation
- **Region-local**
  - Automatic region-based allocation
- **Task-local**
  - Per-task heaps
- **Task-escaping**
  - Regular heap

Evaluation

- **Collision Detector (CD_j) 1.2**
  - Real-time air-traffic simulator and collision detector
  - CiAO OS
  - TriCore TC1796 @ 150 MHz

Evaluation

- **Collision Detector (CD_j) 1.2**
  - Real-time air-traffic simulator and collision detector
  - CiAO OS
  - TriCore TC1796 @ 150 MHz
- **Object sizes**
  - Regular objects are small due to by-reference semantics
  - Chain length determines execution time of
    - Allocation
    - Field access



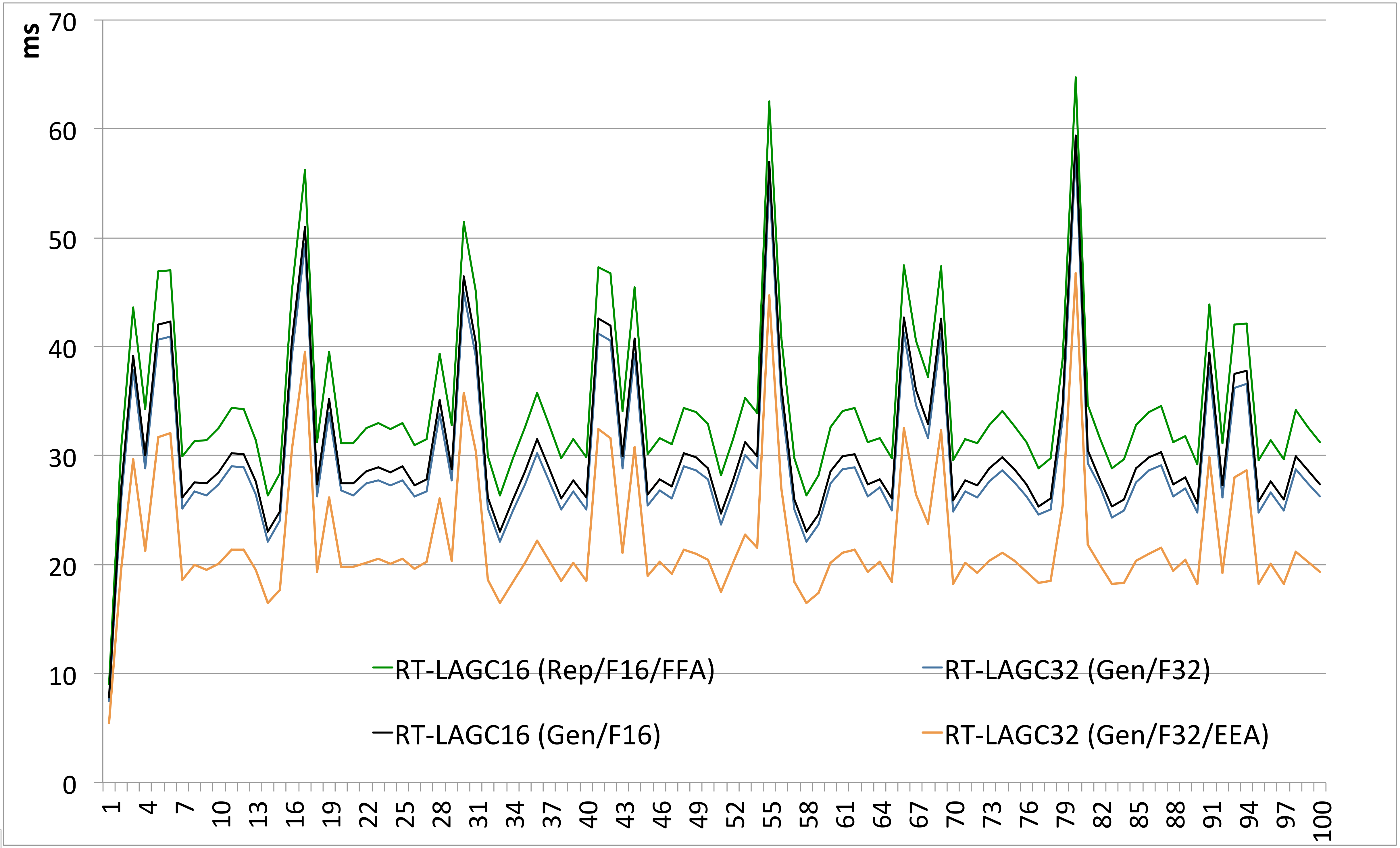
Memory Footprint / Heap Usage

- **Memory footprint**

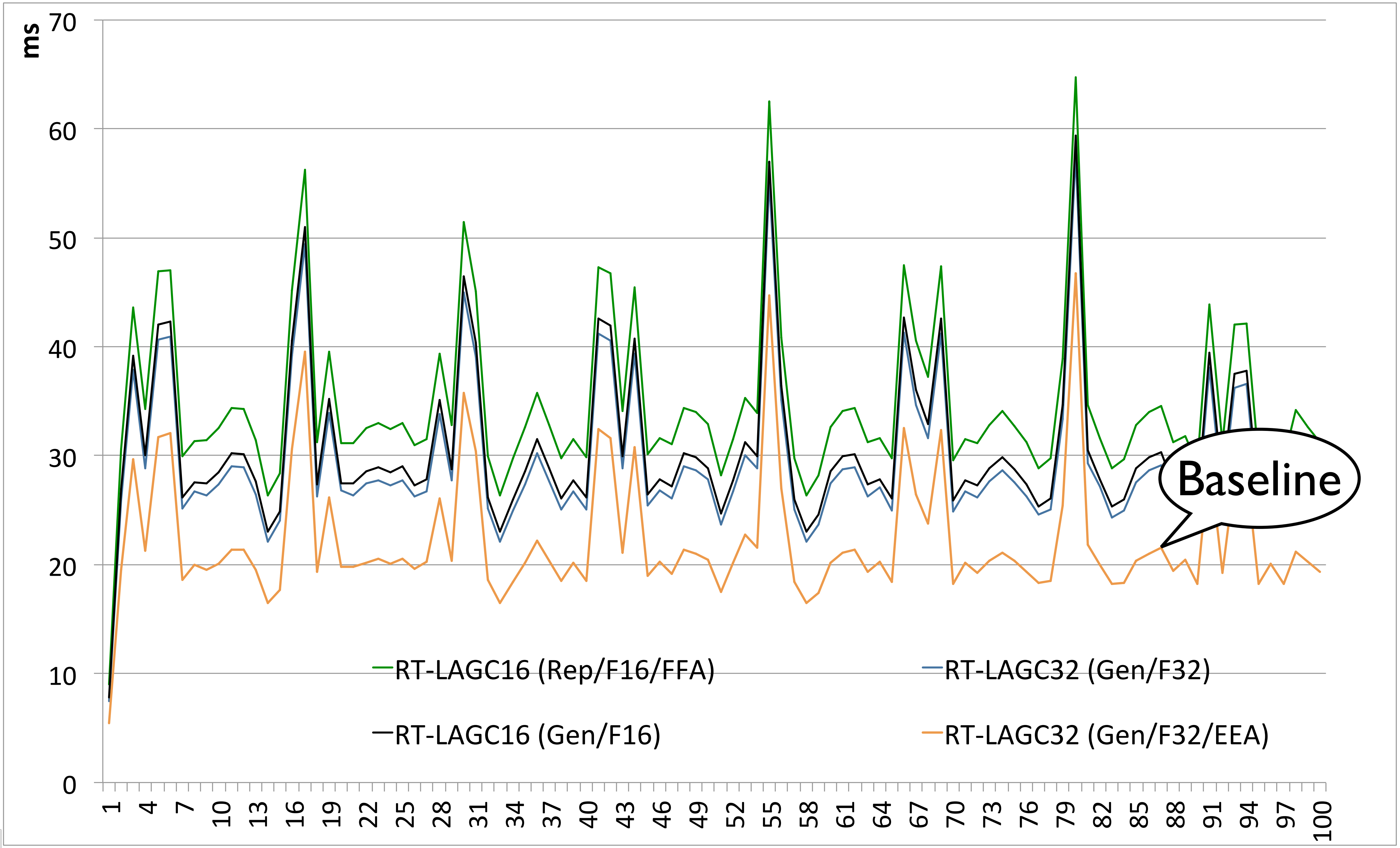
	`.text`	`.data + .bss`
Incremental	50 KiB	660 KiB
RT-LAGC	58-60 KiB	820 KiB
Δ	*+ 16-20 %*	*+ 23 %*

- **Heap usage**
  - Generational vs. replicating spines: spine memory reduced by 50 %
  - Stack allocation reduces heap usage by 43 %
  - 32-byte vs. 16-byte fragments: usage raised by 45 % (internal fragmentation)

Runtime

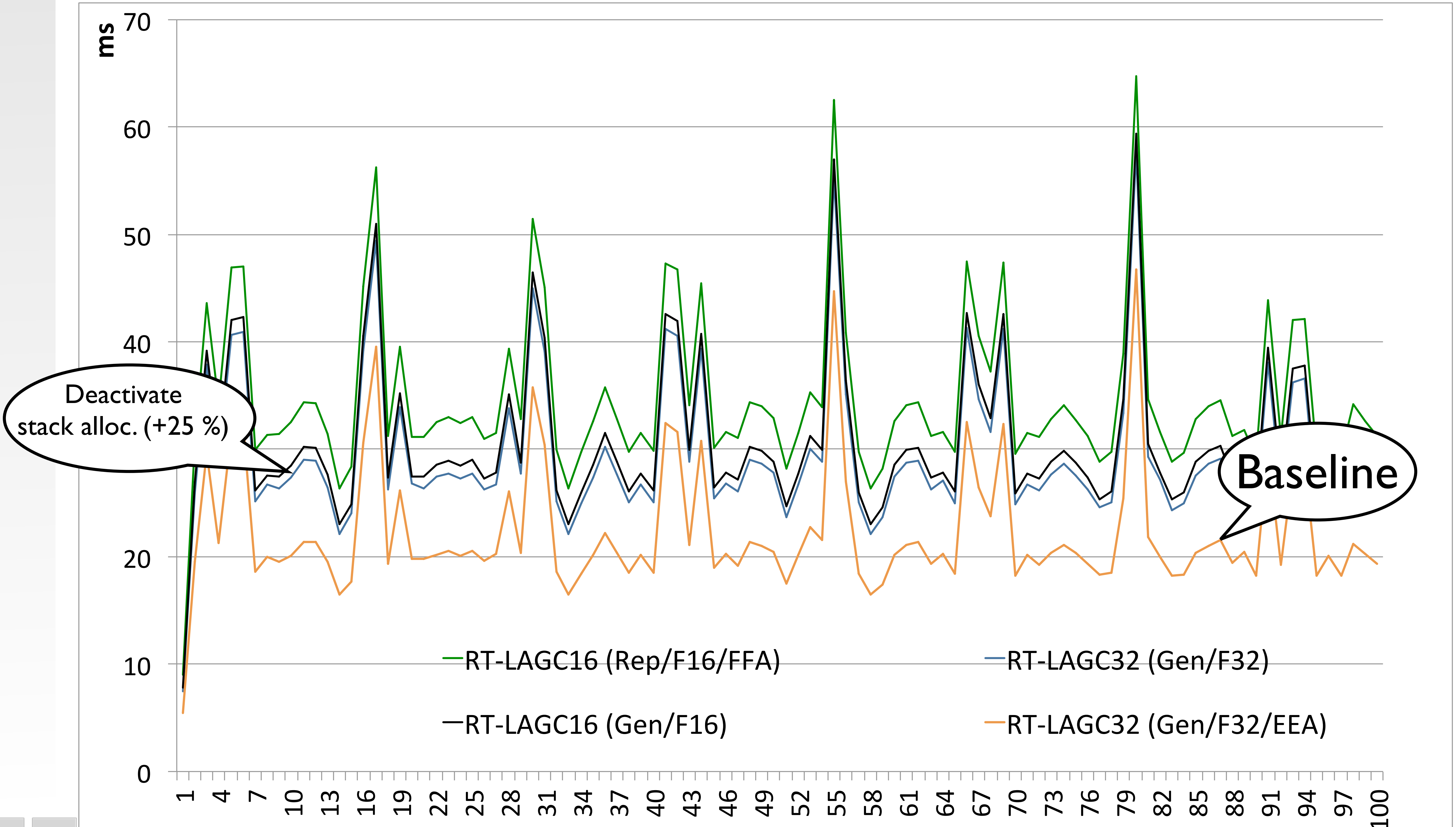

ms
70
60
50
40
30
20
10
0
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58
61
64
67
70
73
76
79
82
85
88
91
94
97
100
RT-LAGC16 (Rep/F16/FFA)
RT-LAGC32 (Gen/F32)
RT-LAGC16 (Gen/F16)
RT-LAGC32 (Gen/F32/EEA)

Runtime

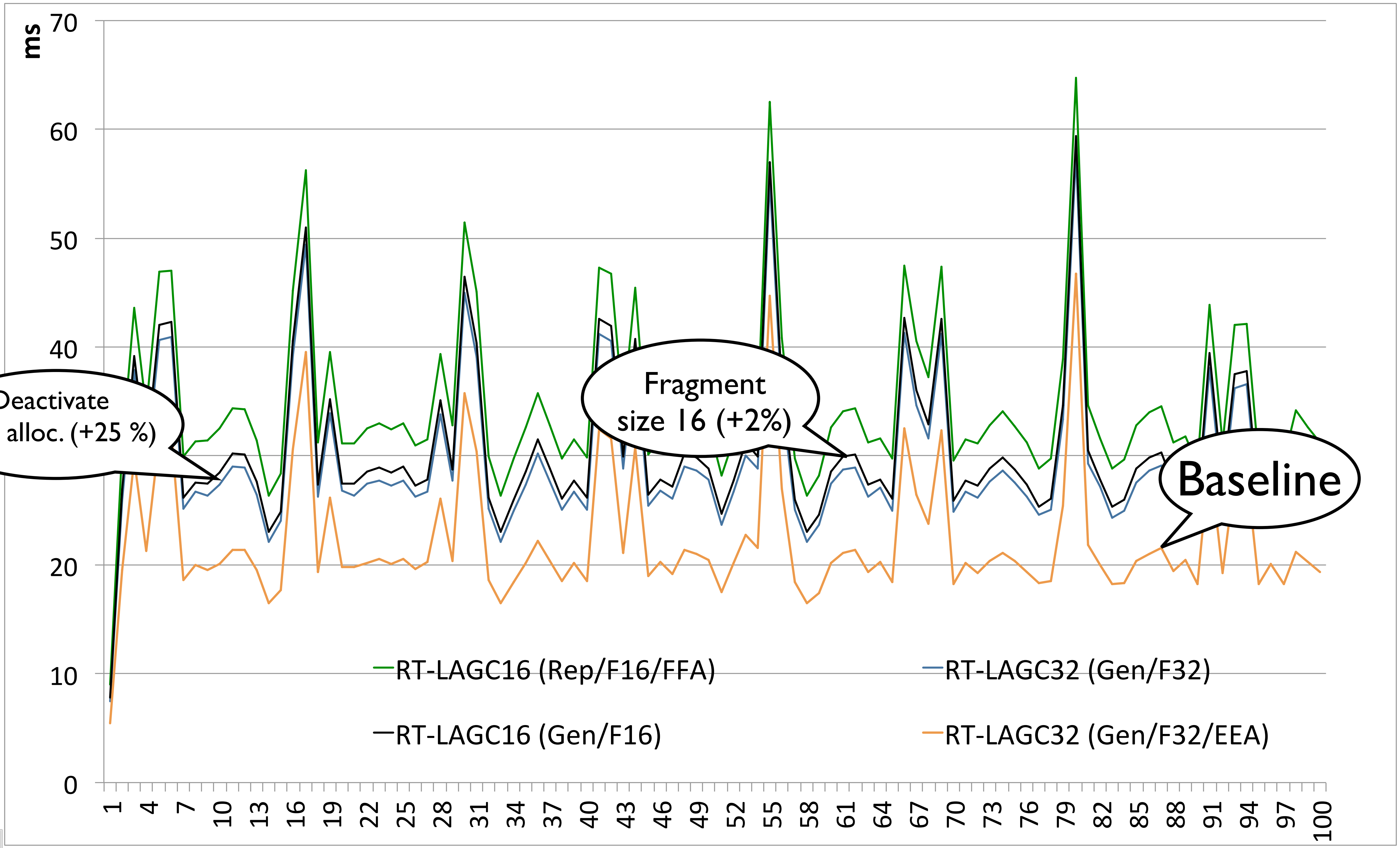


ms
70
60
50
40
30
20
10
0
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58
61
64
67
70
73
76
79
82
85
88
91
94
97
100
RT-LAGC16 (Rep/F16/FFA)
RT-LAGC32 (Gen/F32)
RT-LAGC16 (Gen/F16)
RT-LAGC32 (Gen/F32/EEA)
Baseline

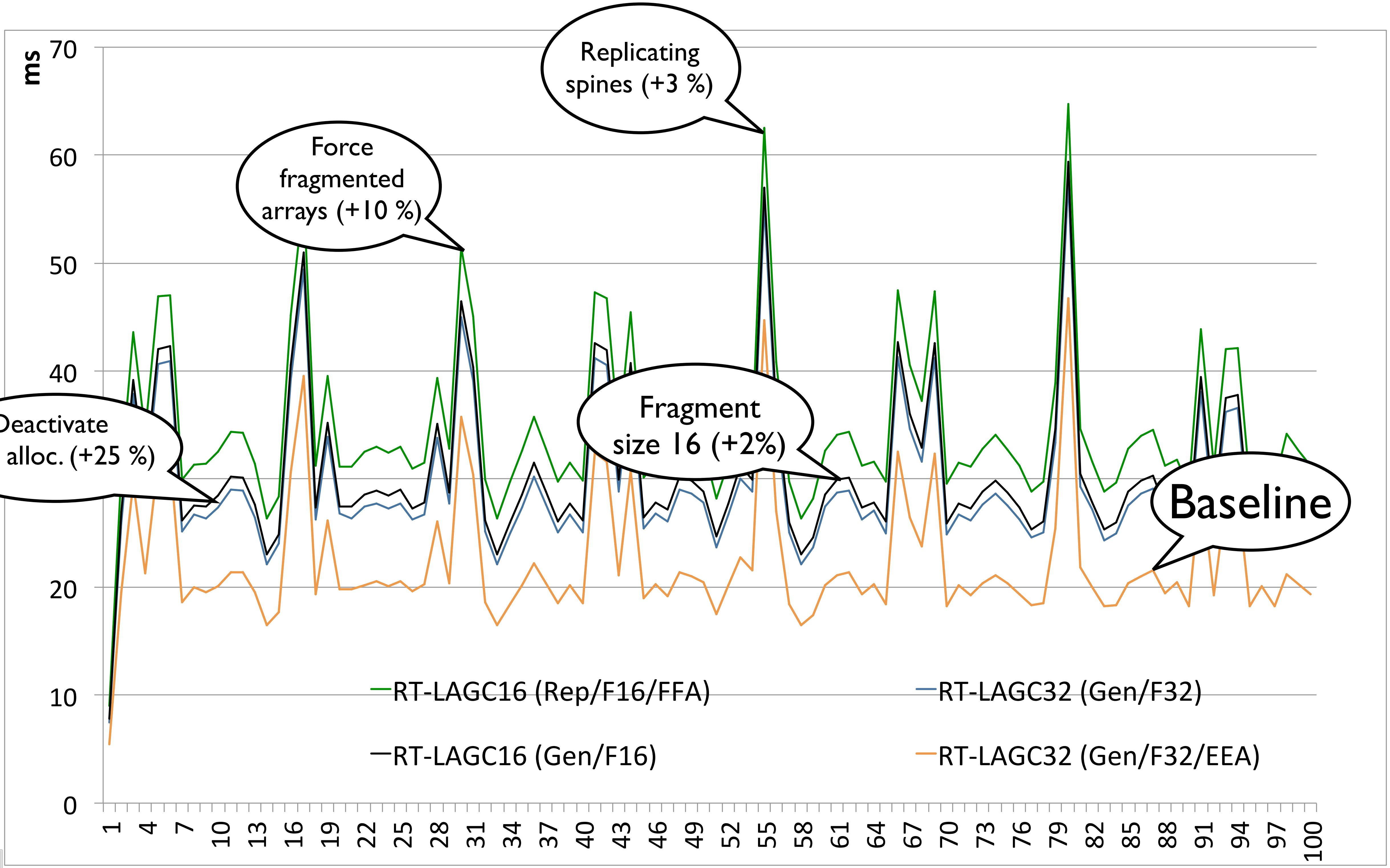
Runtime


ms
70
60
50
40
20
10
0
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58
61
64
67
70
73
76
79
82
85
88
91
94
97
100
RT-LAGC16 (Rep/F16/FFA)
RT-LAGC32 (Gen/F32)
RT-LAGC16 (Gen/F16)
RT-LAGC32 (Gen/F32/EEA)
Deactivate stack alloc. (+25 %)
Baseline

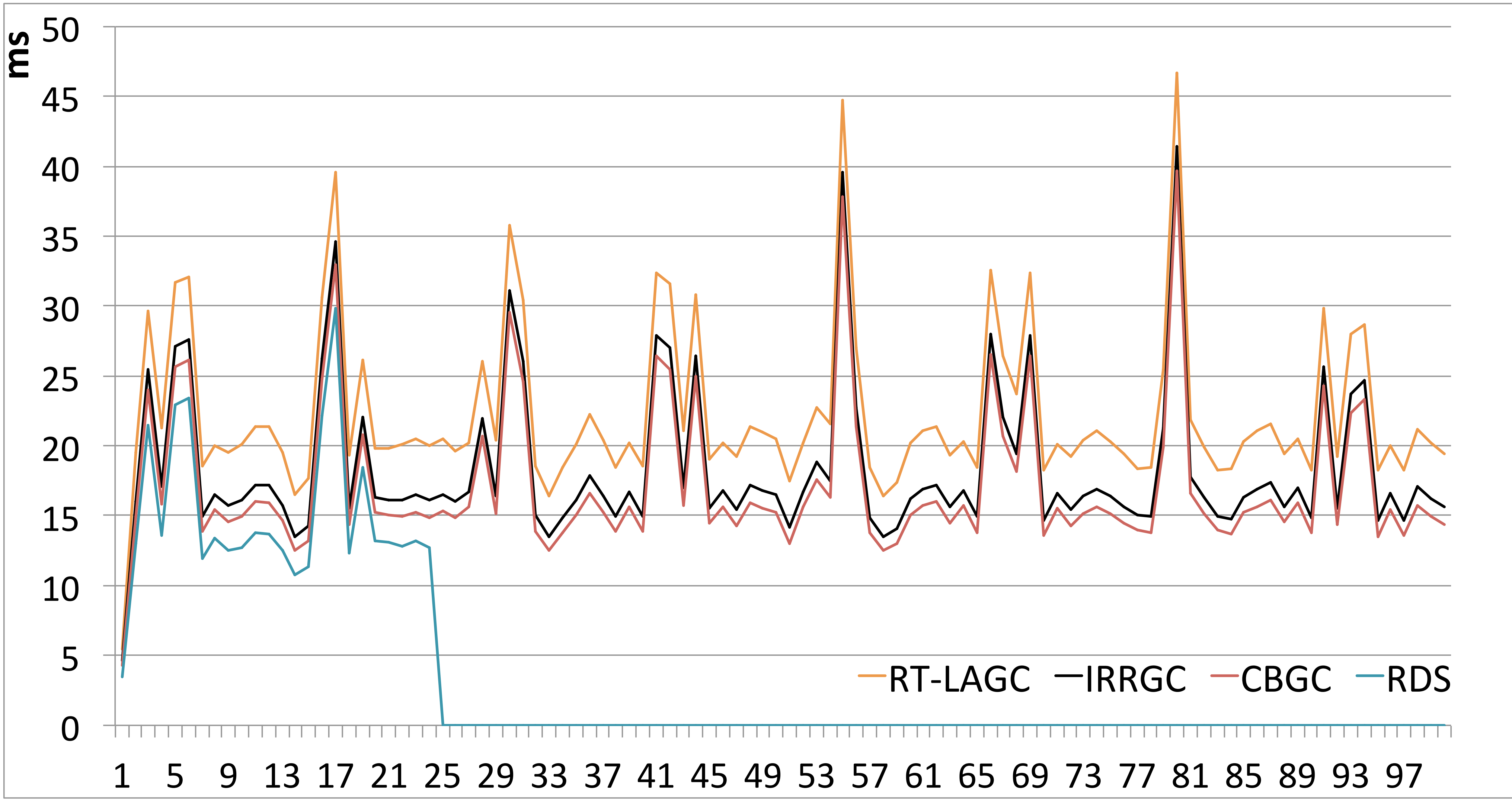
Runtime


ms
70
60
50
40
20
10
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100
Deactivate stack alloc. (+25 %)
Fragment size 16 (+2%)
Baseline
RT-LAGC16 (Rep/F16/FFA)
RT-LAGC32 (Gen/F32)
RT-LAGC16 (Gen/F16)
RT-LAGC32 (Gen/F32/EEA)

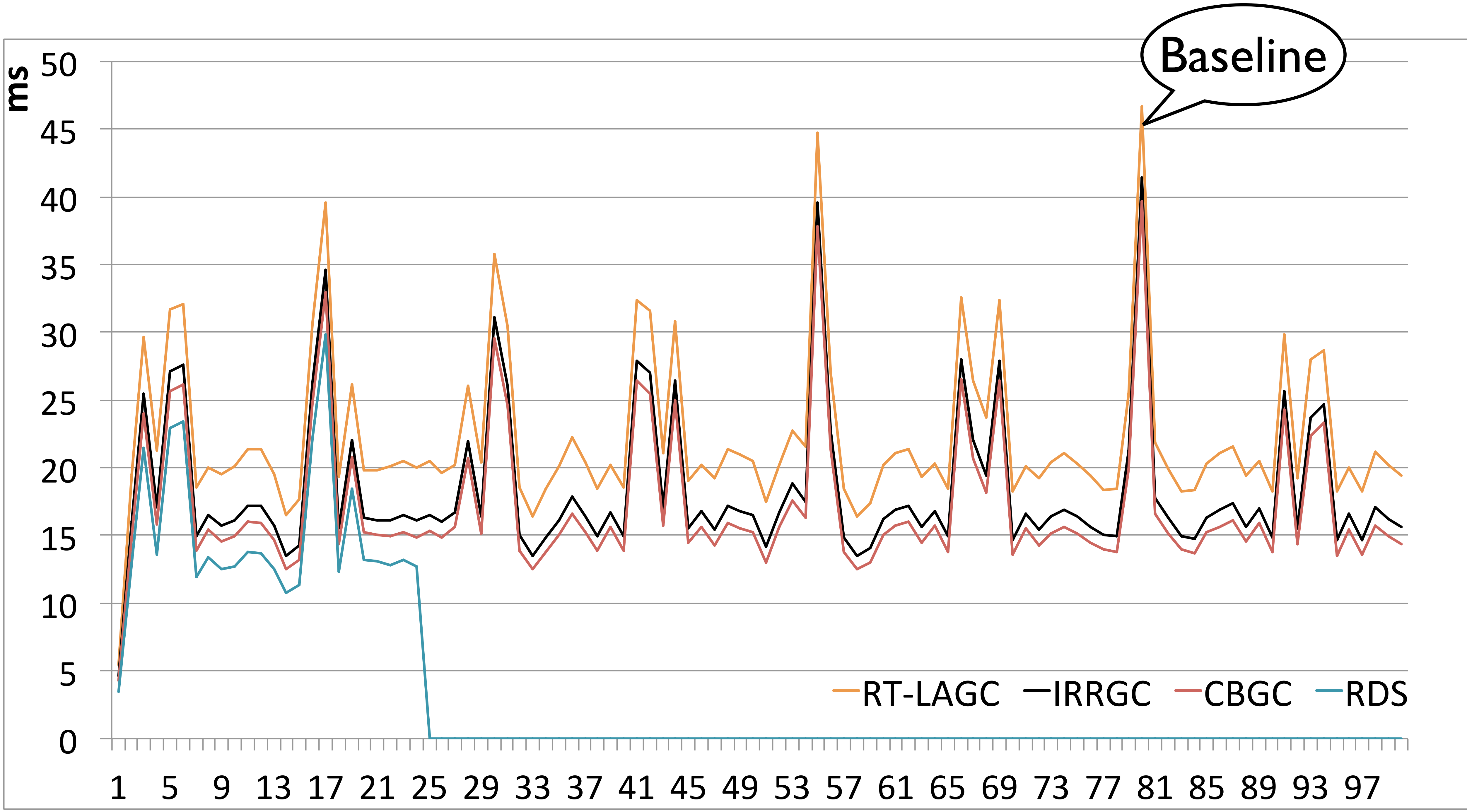
Runtime


ms
70
60
50
40
20
10
0
Replicating spines (+3 %)
Force fragmented arrays (+10 %)
Deactivate stack alloc. (+25 %)
Fragment size 16 (+2%)
Baseline
RT-LAGC16 (Rep/F16/FFA)
RT-LAGC32 (Gen/F32)
RT-LAGC16 (Gen/F16)
RT-LAGC32 (Gen/F32/EEA)
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100

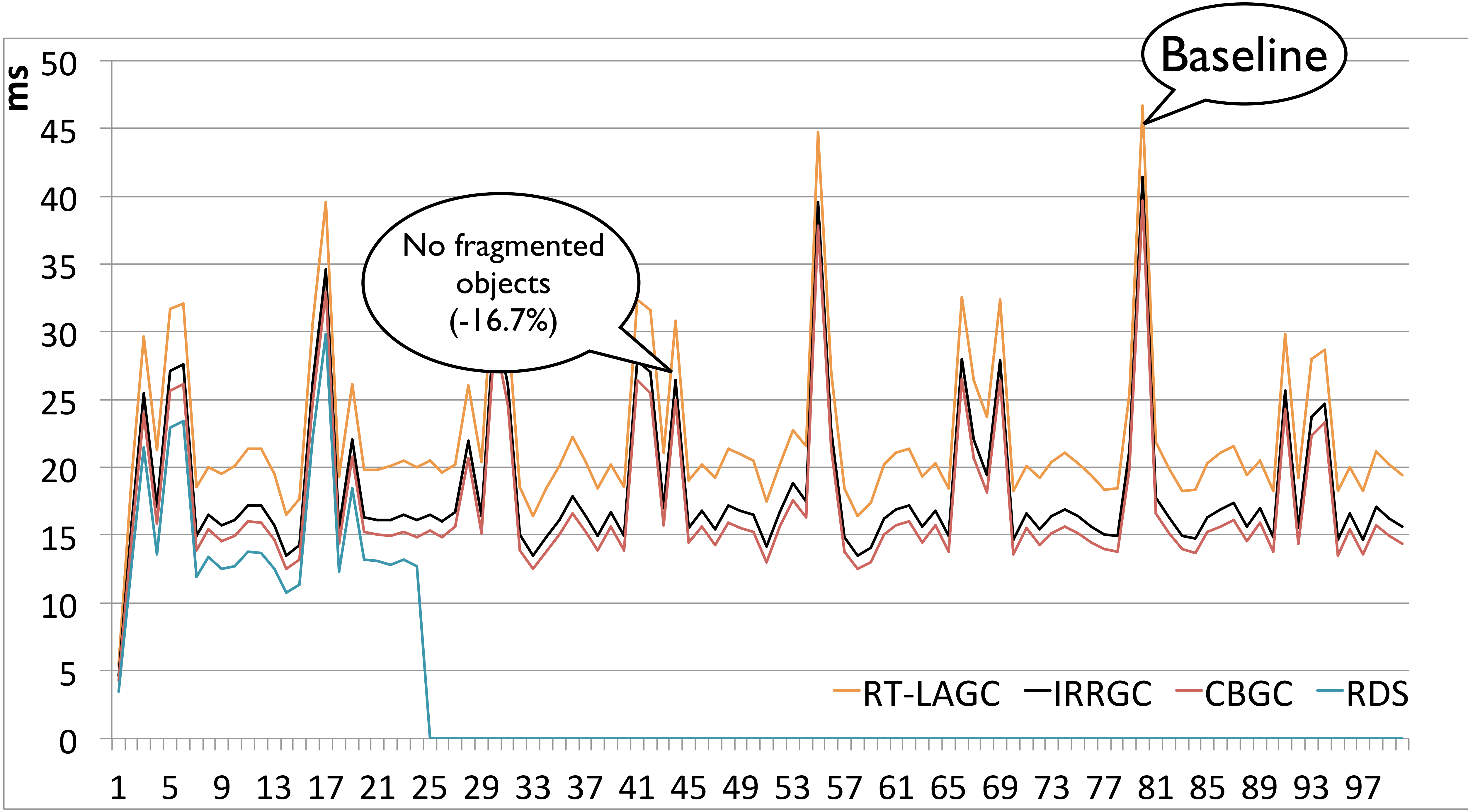
Runtime


ms
50
45
40
35
30
25
20
15
10
5
0
RT-LAGC
IRRGC
CBGC
RDS
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97

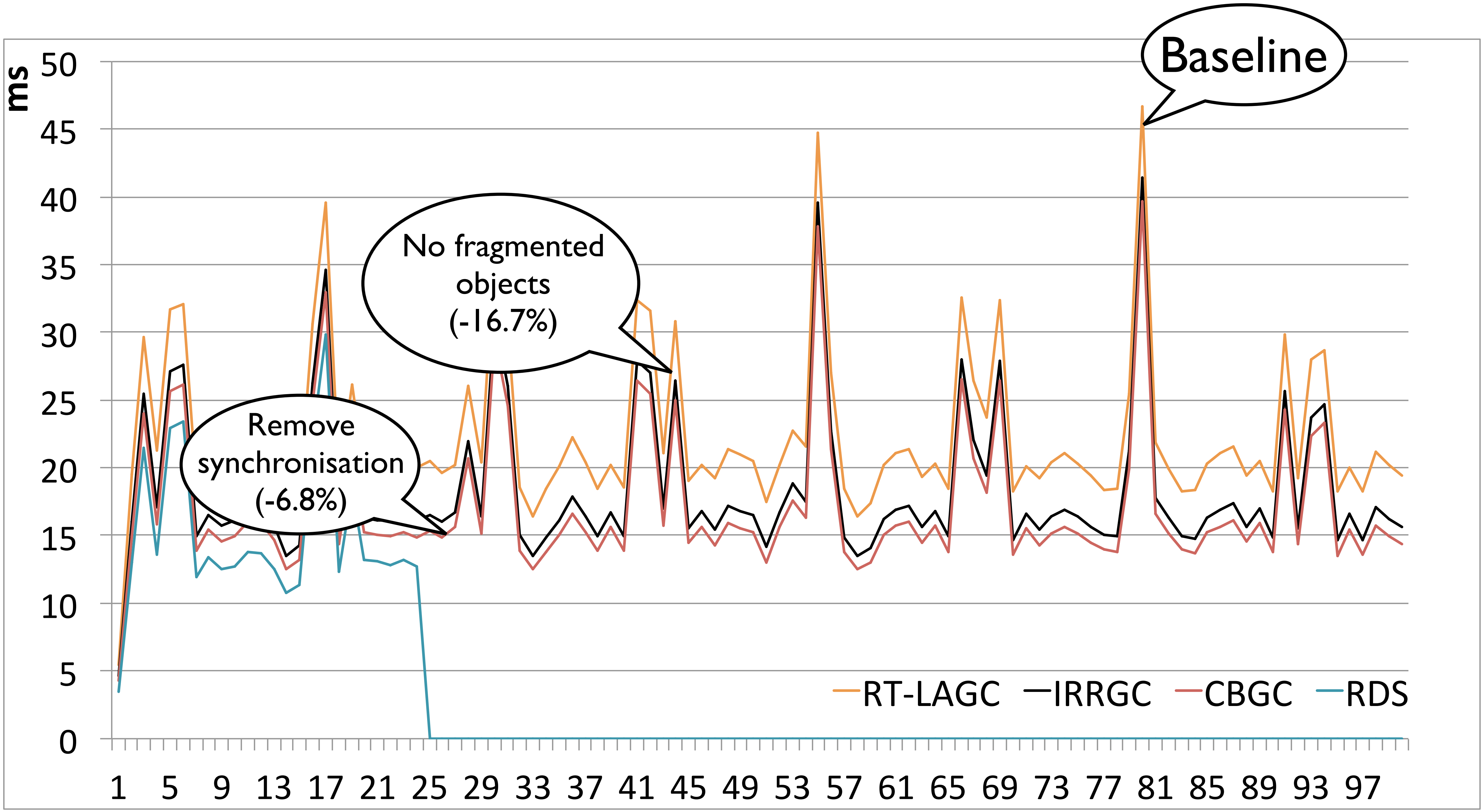
Runtime


Baseline
ms
50
45
40
35
30
25
20
15
10
5
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97
RT-LAGC
IRRGC
CBGC
RDS

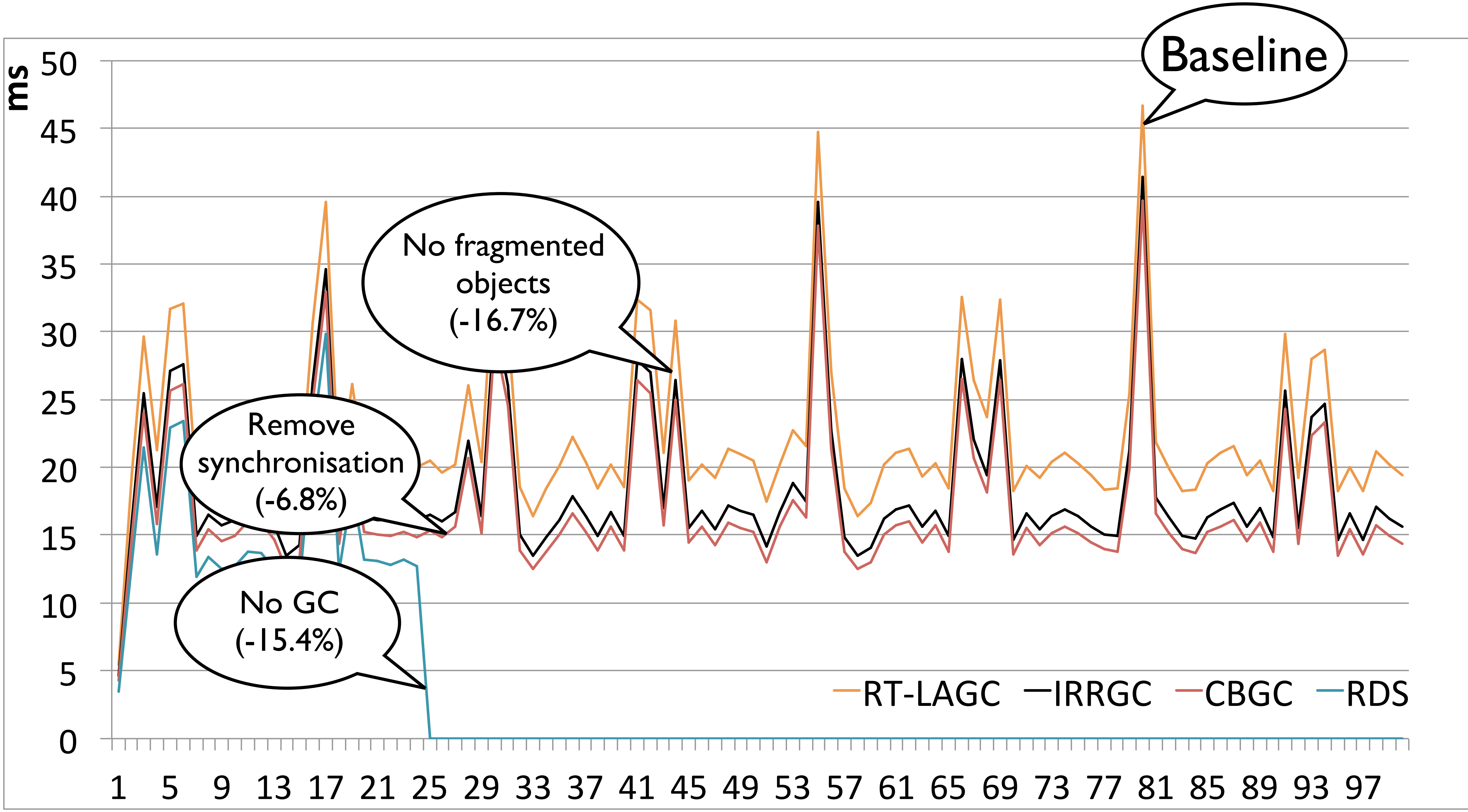
Runtime


ms
50
45
40
35
30
25
20
15
10
5
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97
No fragmented objects (-16.7%)
Baseline
RT-LAGC
IRRGC
CBGC
RDS

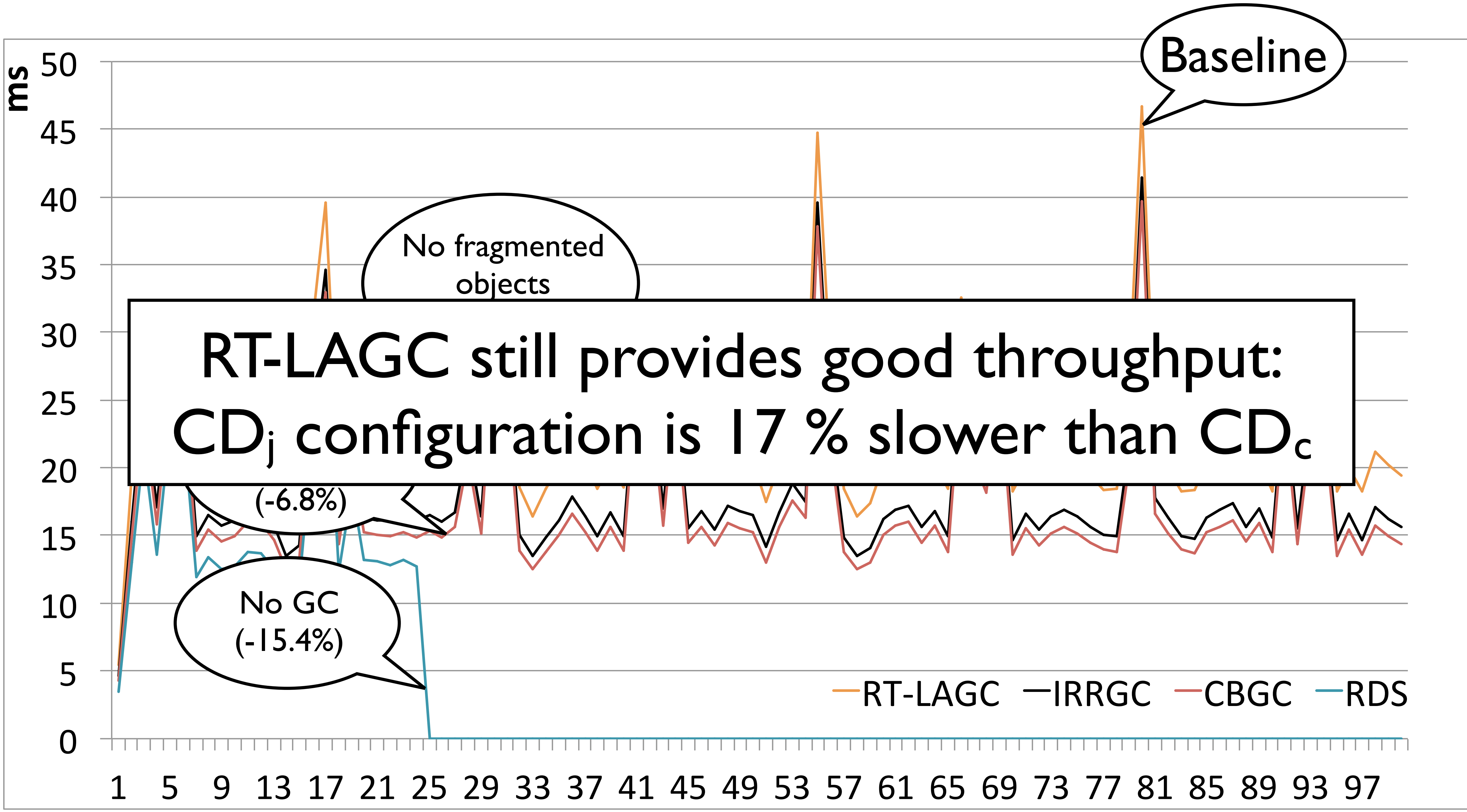
Runtime


ms
50
45
40
35
30
25
20
15
10
5
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97
Baseline
No fragmented objects (-16.7%)
Remove synchronisation (-6.8%)
RT-LAGC
IRRGC
CBGC
RDS

Runtime


ms
50
45
40
35
30
25
20
15
10
5
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97
Baseline
No fragmented objects (-16.7%)
Remove synchronisation (-6.8%)
No GC (-15.4%)
RT-LAGC
IRRGC
CBGC
RDS

Runtime



RT-LAGC still provides good throughput:
CD_j configuration is 17 % slower than CD_c

Conclusion

- **RT-LAGC**
  - Cooperative approach
  - Predictable allocation times
  - Low reaction time to external events
  - Good throughput
  - Fine-grained synchronisation code

Conclusion

- **RT-LAGC**
  - Cooperative approach
  - Predictable allocation times
  - Low reaction time to external events
  - Good throughput
  - Fine-grained synchronisation code
- **Developer assistance during integration**
  - Survivability analysis
  - Object sizes