Cooperative Memory Management in Embedded Systems

Isabella Stilkerich, Philip Taffner, Christoph Erhardt, Christian Dietrich, Michael Stilkerich

Department of Computer Science 4
Distributed Systems and Operating Systems

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Motivation

- Automated memory management
  - Safety-critical systems
  - Challenge: Resource-efficiency
- Embedded microcontrollers are heterogeneous, for example:
  - Memories (e.g. ROM, RAM)
  - Memory layout
  - Occurrence of random soft errors (bit flips)

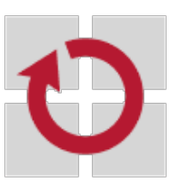
Motivation

- Automated memory management
  - Safety-critical systems
  - Challenge: Resource-efficiency
- Embedded microcontrollers are heterogeneous, for example:
  - Memories (e.g. ROM, RAM)
  - Memory layout
  - Occurrence of random soft errors (bit flips)

Motivation

Static knowledge about the employed operating system and hardware characteristics is useful for application-tailored automated memory management!

- Memory layout
- Occurrence of random soft errors (bit flips)



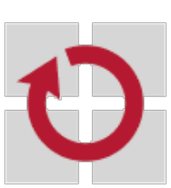
Agenda

How can static knowledge about the application, operating system and hardware-specifics be used to create an automated memory management in embedded systems?

Can automated and safe memory management efficiently be applied in resource-constraint embedded systems?

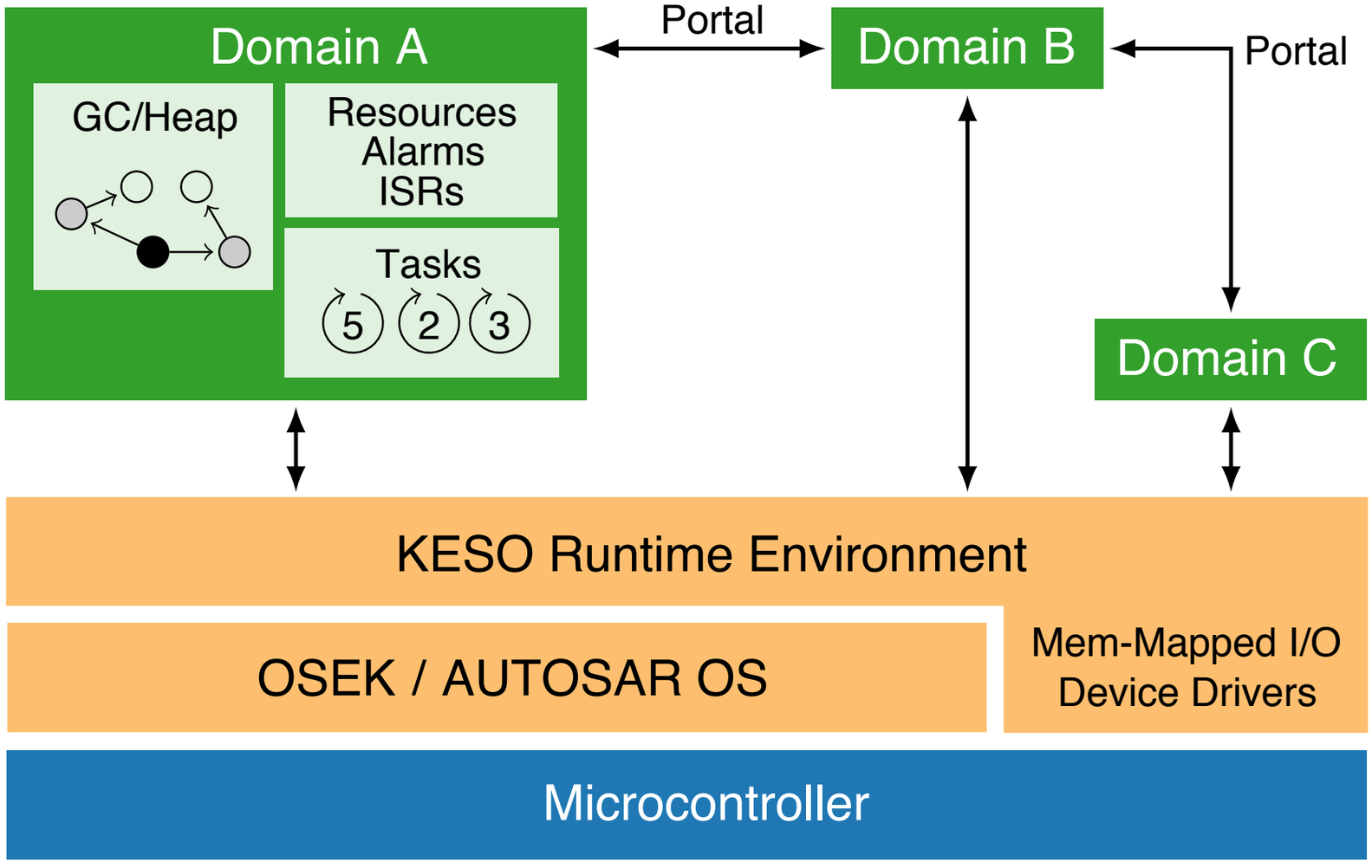
Case study with the embedded KESO Java Virtual Machine

- Cooperative Memory Management
  - Compiler Support
  - Safe and Latency-Aware Garbage Collection
- Evaluation



The KESO JVM

- Java-to-C ahead-of-time compiler
- VM tailoring, static configuration

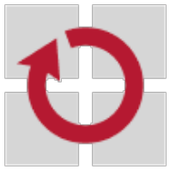


Memory Safety

```
public class Average {

  protected int sum, count;

  public synchronized void addValues( int values[] ) {
    for(int i=0; i < values.length; i++) {
      sum += values[i];
    }
    count += values.length;
  }

  public synchronized int average() {
    return (sum / count);
  }
}
```

"Memory safety ensures the **validity of memory references** by preventing `null` pointer references, references outside an array's bounds, or references to deallocated memory."

Aiken et. al (Microsoft Research)



Memory Safety

```
public class Average {

  protected int sum, count;

  public synchronized void addvalues( int values[] ) {
    for (int i = 0; i < values.length; i++) {
      sum += values[i];
    }
    count += values.length;
  }

  public synchronized int average() {
    return (sum / count);
  }
}
```

null != values

null != values

0 <= i < values.length

null != values

count != 0

"**Memory safety** ensures the **validity of memory references** by preventing `null` pointer references, references outside an array's bounds, or references to deallocated memory."

Aiken et. al (Microsoft Research)

Type Safety

"**Type safety** ensures that the only operations applied to a value are those defined for instances of its type."

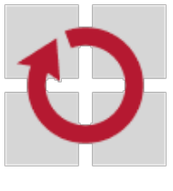
Aiken et. al (Microsoft Research)

Type Safety

```
public class Average {

  protected int sum, count;

  public synchronized void addValues( int values[] ) {
    for(int i=0; i < values.length; i++) {
      sum += values[i];
    }
    count += values.length;
  }

  public synchronized int average() {
    return (sum / count);
  }
}
```

Value of an instance of *Average*

Operations on type *Average*



"Type safety ensures that the only operations applied to a value are those defined for instances of its type."

Aiken et. al (Microsoft Research)

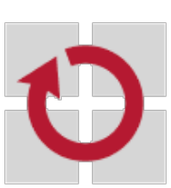
Agenda

How can static knowledge about the application, operating system and hardware-specifics be used to create an automated memory management in embedded systems?

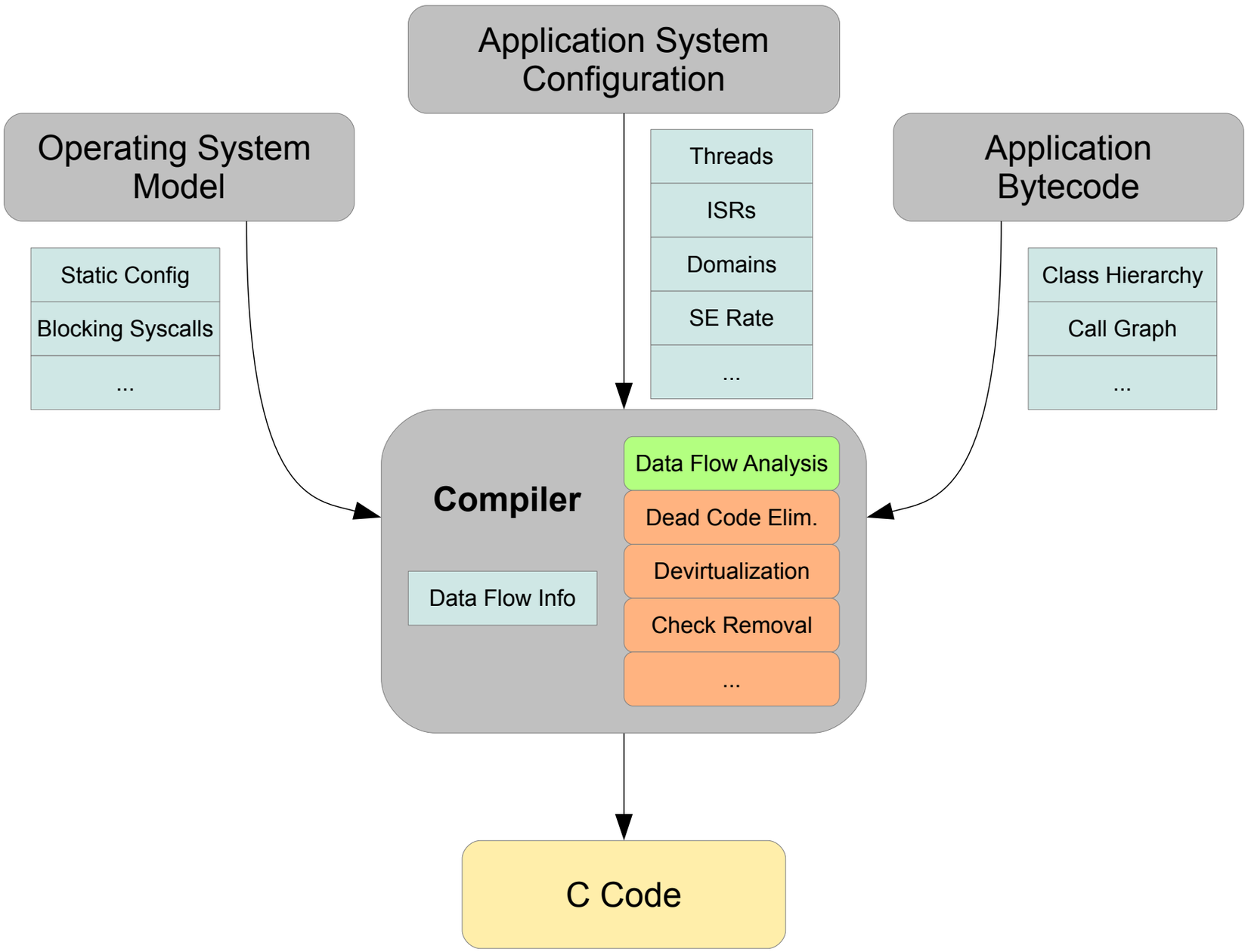
Can automated and safe memory management efficiently be applied in resource-constraint embedded systems?

Case study with the embedded KESO Java Virtual Machine

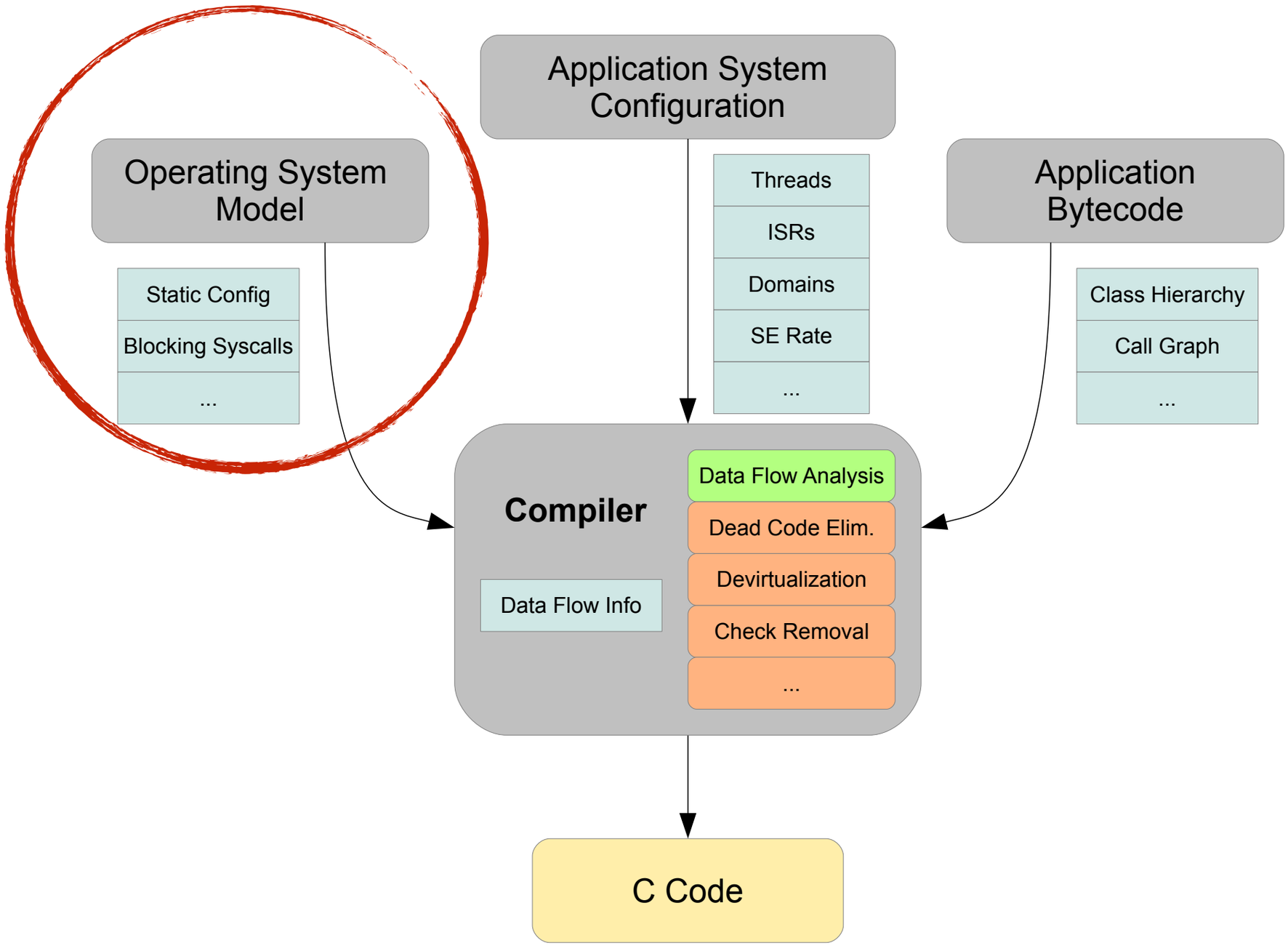
- Cooperative Memory Management
  - Compiler Support
  - Safe and Latency-Aware Garbage Collection
- Evaluation



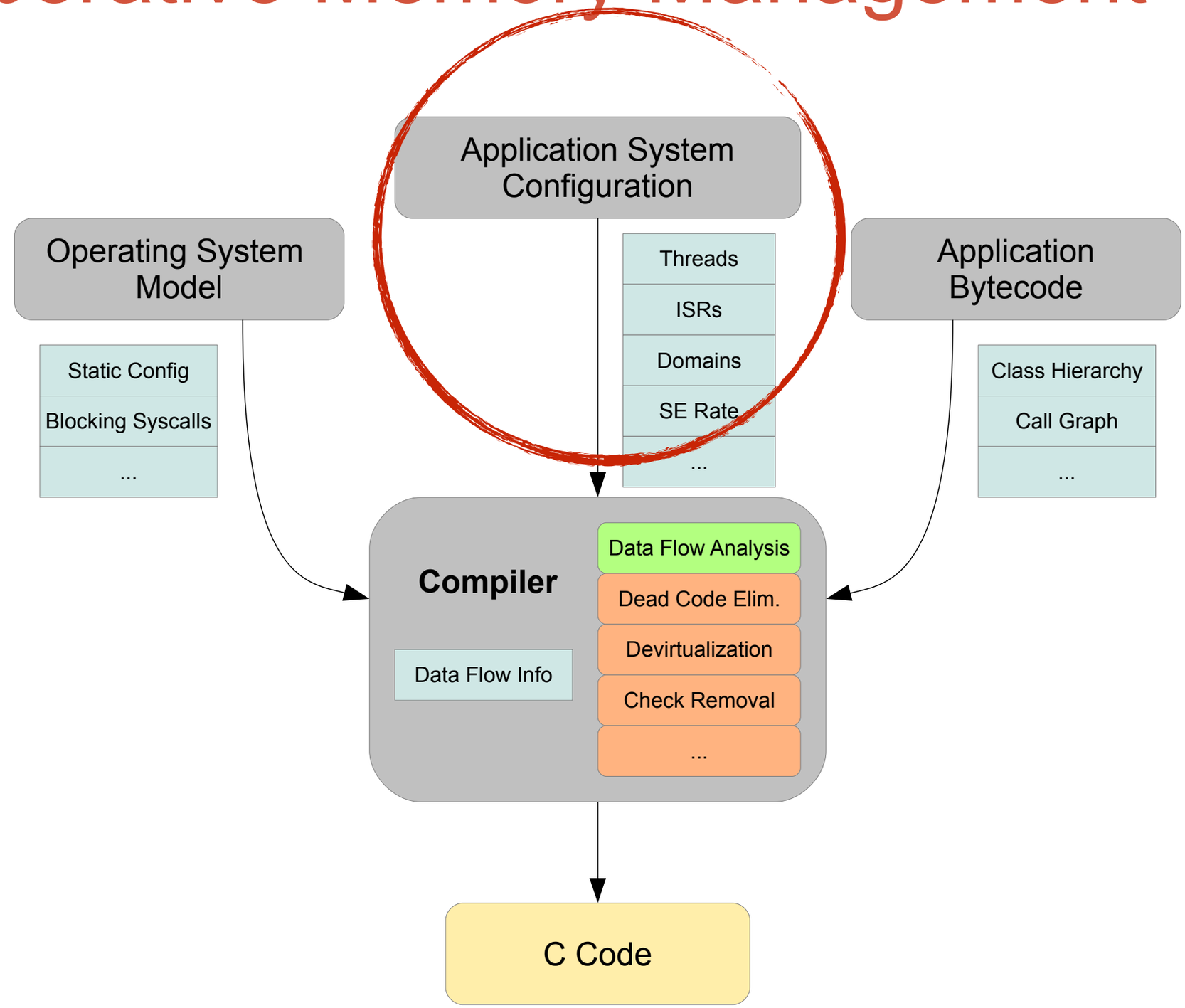
Cooperative Memory Management



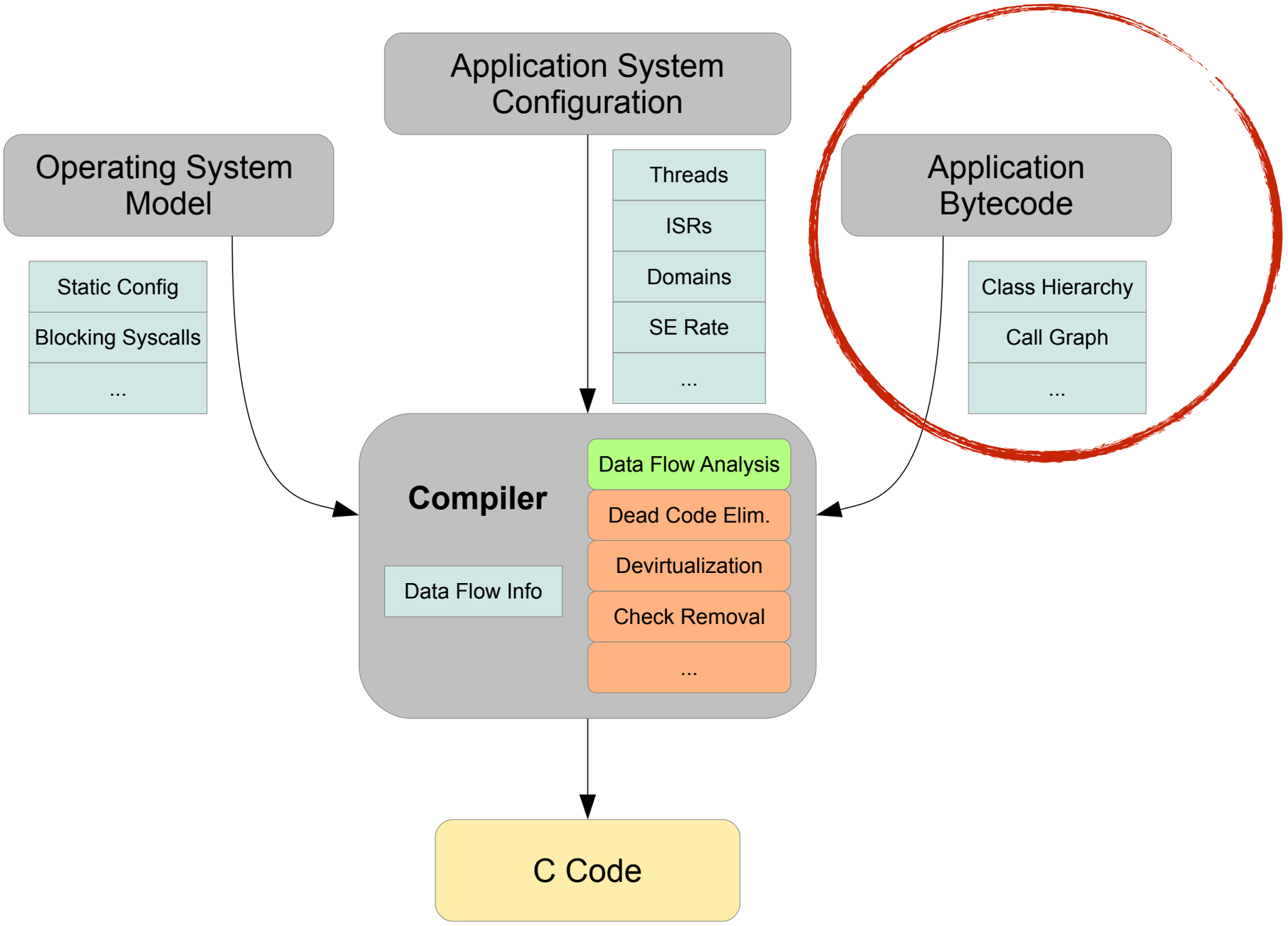
Cooperative Memory Management



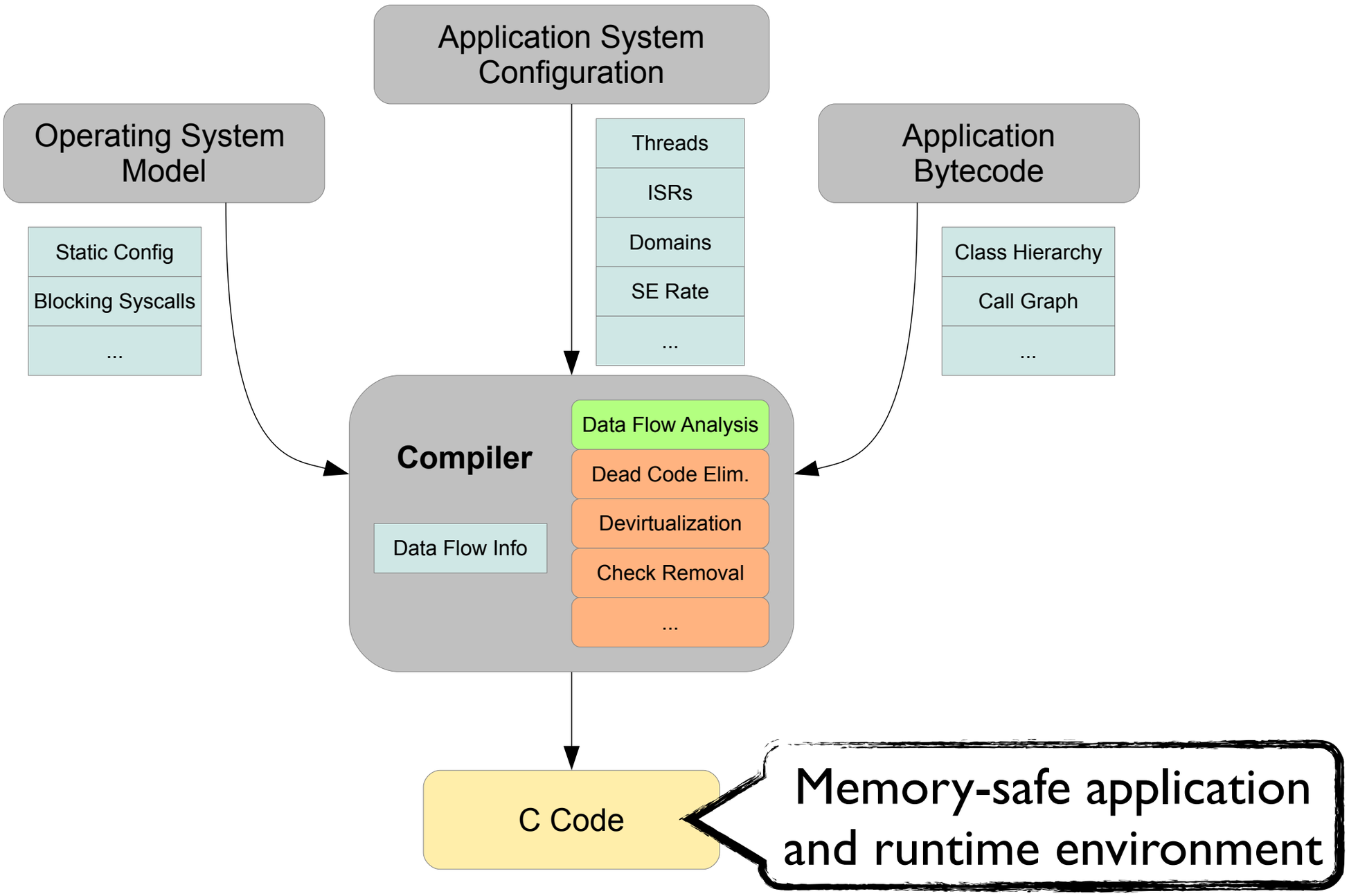
Cooperative Memory Management



Cooperative Memory Management



Cooperative Memory Management



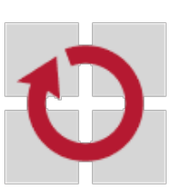
Agenda

How can static knowledge about the application, operating system and hardware-specifics be used to create an automated memory management in embedded systems?

Can automated and safe memory management efficiently be applied in resource-constraint embedded systems?

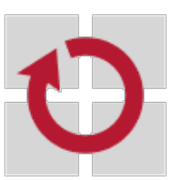
Case study with the embedded KESO Java Virtual Machine

- Cooperative Memory Management
  - Compiler Support
  - Safe and Latency-Aware Garbage Collection
- Evaluation



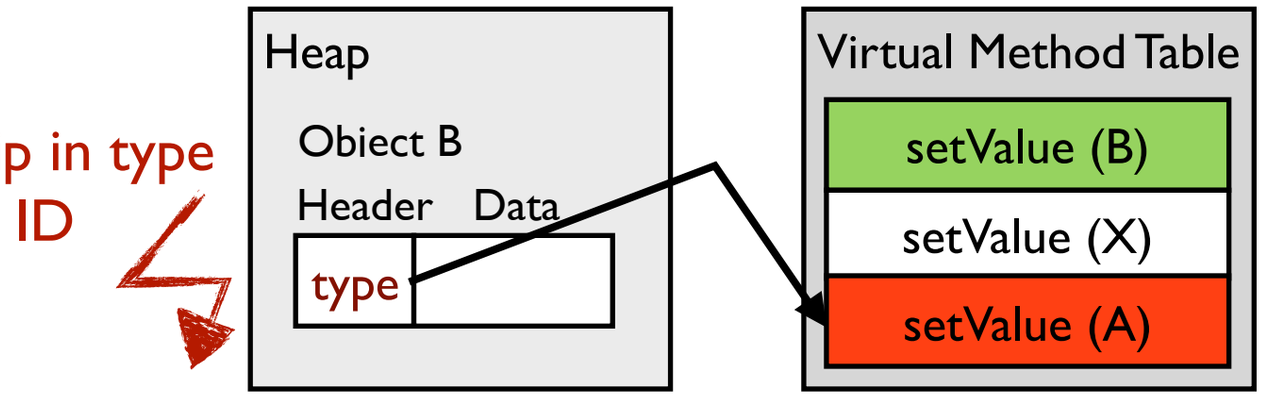
Escape Analysis

- Escape analysis is used to determine the escape state of an object
  - By using information from alias analysis and reachability of references
- Analysis results prove useful in the context of cooperative approach
  - Stack allocation and region inference
  - Allocation and deallocation is very efficient
  - Estimation of GC effort (real-time systems)
  - In the context of soft errors
    - Reduction of replication sphere
    - Efficient protection of type system via reference checking



Reference Integrity Checks

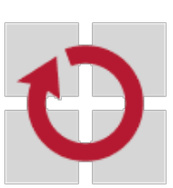
- Bit flips can break the soundness of the type system!
- Preserve software-based isolation via integrity checks of references
  - A controlflow-sensitive analysis determines the insertion of checks
  - Dereference check (**DRC**)
- Integrity check is two-fold
  - Validity of address
  - Check of header information (e.g. dynamic type)
  - **No wild references!**
- Protection information can be incorporated in the reference
  - Attack surface of the program is not increased
  - Protection information size is depended on the used hardware device



Immortal Data / Runtime-Final Analyses

- *Automatic inference of constant and runtime-final data (alias info)*

1. Determination of constant data is essential
   - Data can automatically be placed in read-only memory (ROM) e.g. flash
   - Flash memory is much less susceptible to soft errors
   - Garbage collection overhead can be reduced
2. Runtime-final analysis
   - Constant folding
   - Null checks on effectively-final data can be eliminated



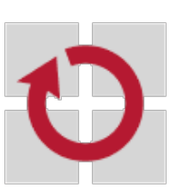
Agenda

How can static knowledge about the application, operating system and hardware-specifics be used to create an automated memory management in embedded systems?

Can automated and safe memory management efficiently be applied in resource-constraint embedded systems?

Case study with the embedded KESO Java Virtual Machine

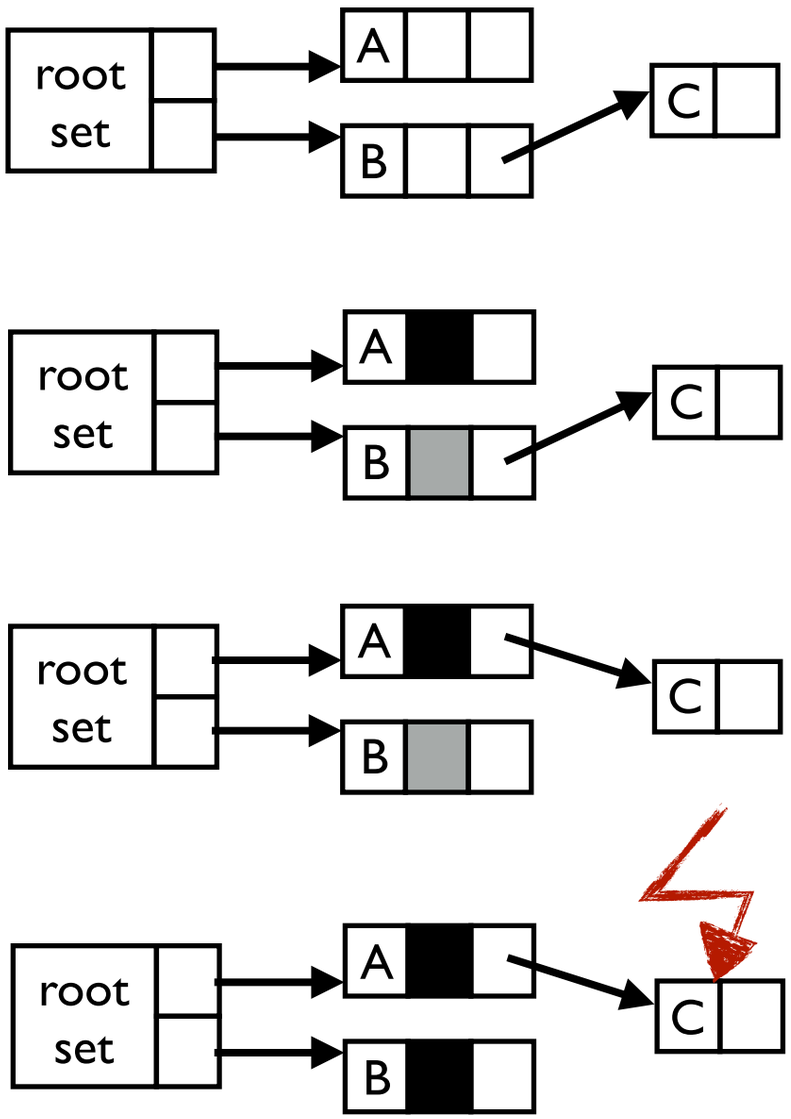
- Cooperative Memory Management
  - Compiler Support
  - Safe and Latency-Aware Garbage Collection
- Evaluation



Latency-Aware Garbage Collection (LAGC)

- Incremental and slack-based *mark-and-sweep* GC algorithm
- Restrict each critical section to constant complexity
- Dedicated (low-priority) thread for garbage collection
- ***Scan-and-mark***:
  - Traditional tricolor scheme for marking (white, grey, black)
  - Reference scanning: start at root set
    - Static references: global array
    - Local references: linked stack frames
    - Object layout: grouping of references
- Sweep:
  - Memory of unreachable (white) objects is inserted into free memory list

Synchronization Needed

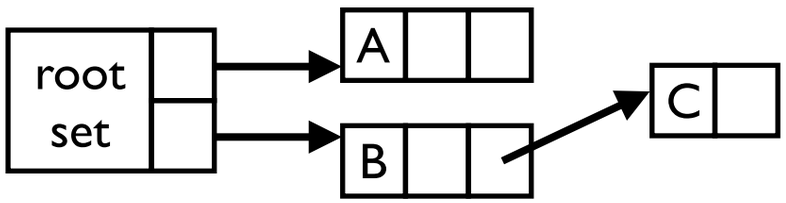


Scanned Reachable object

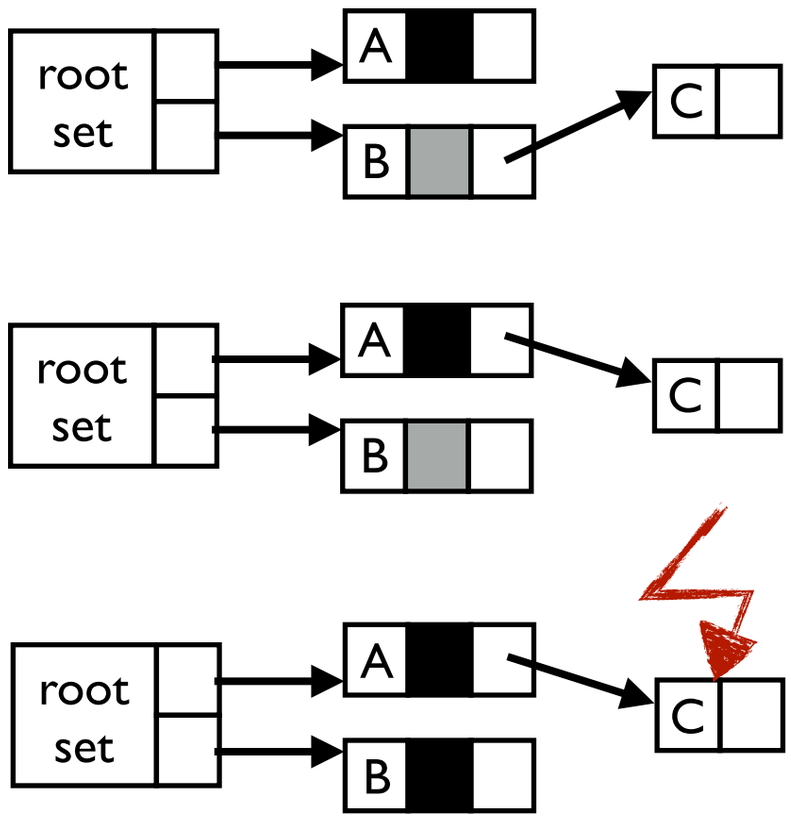
Unvisited/unreachable object

Unscanned reachable object

Synchronization Needed

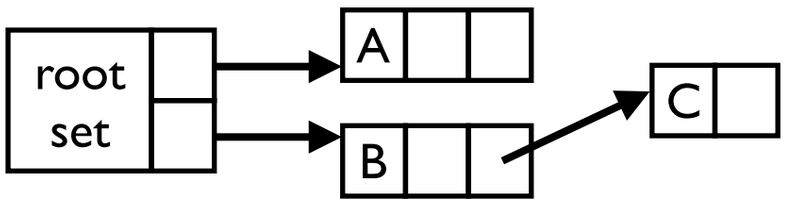


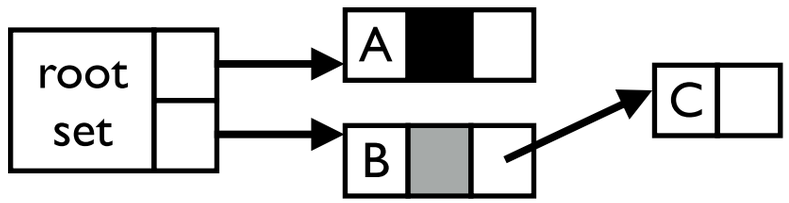
Objects A, B and C are reachable via root set
Garbage Collector (GC) starts marking



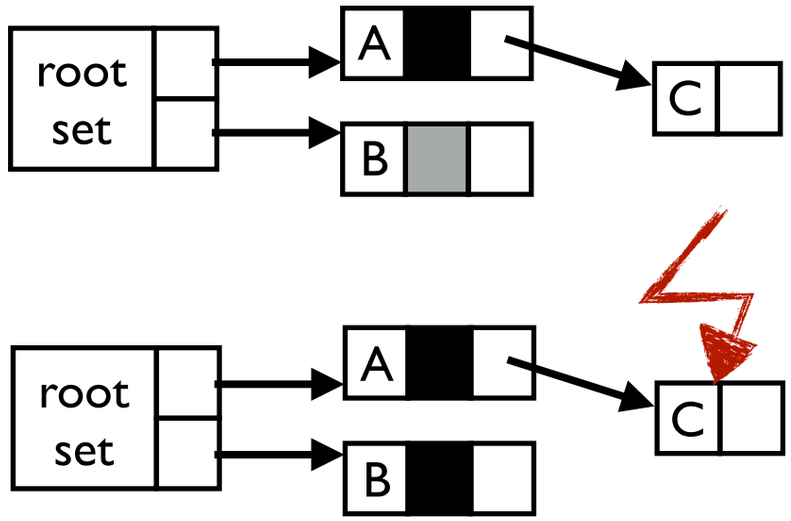
Scanned Reachable object Unvisited/unreachable object Unscanned reachable object

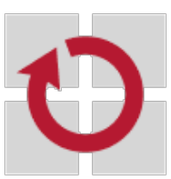
Synchronization Needed





A is marked reachable (black)



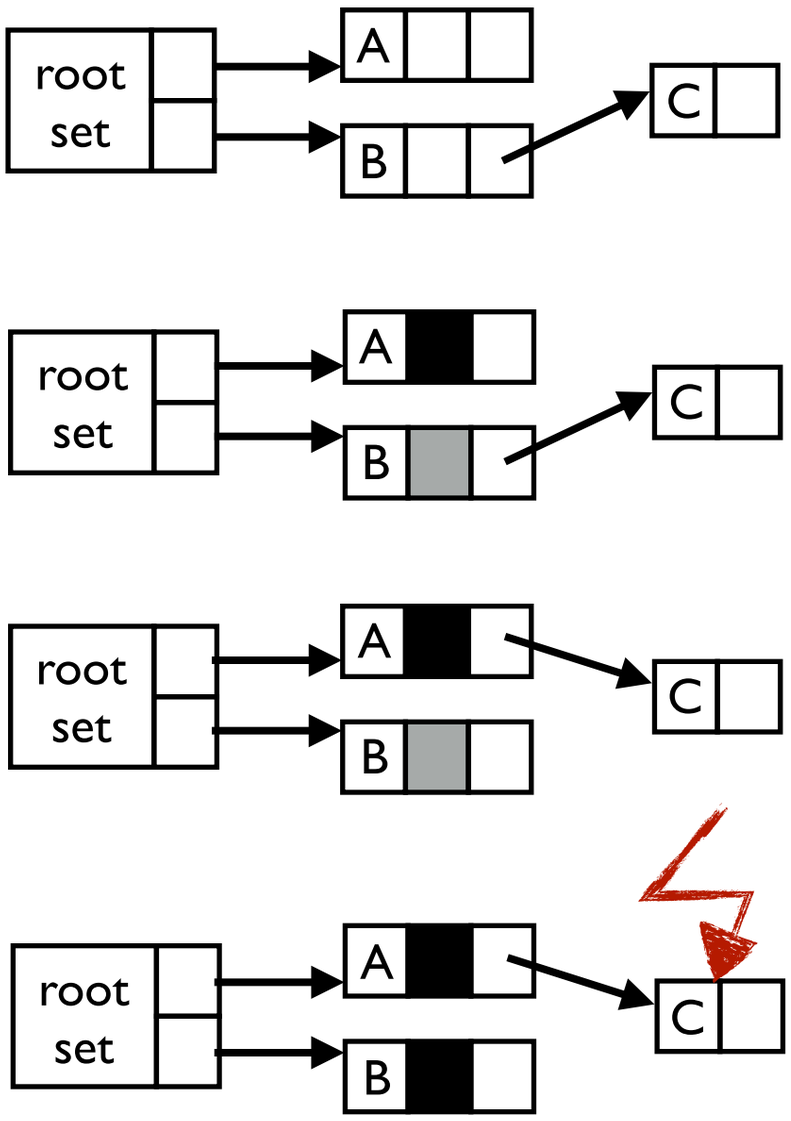


Scanned Reachable object

Unvisited/unreachable object

Unscanned reachable object

Synchronization Needed



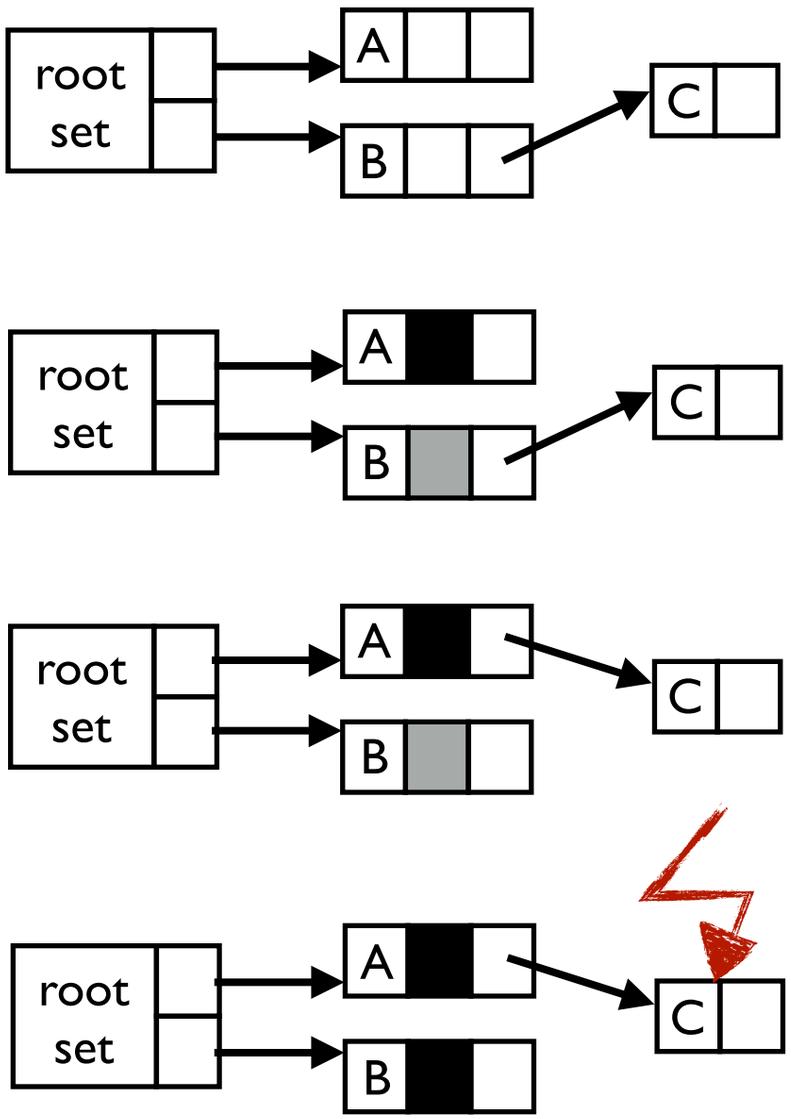
After marking A reachable, the GC is preempted by the higher-priority application, which sets a reference from A to C. The interrupted GC marking is resumed

Scanned Reachable object

Unvisited/unreachable object

Unscanned reachable object

Synchronization Needed



C is not marked reachable by the GC, though it is used by the application. A dangling reference has been forged

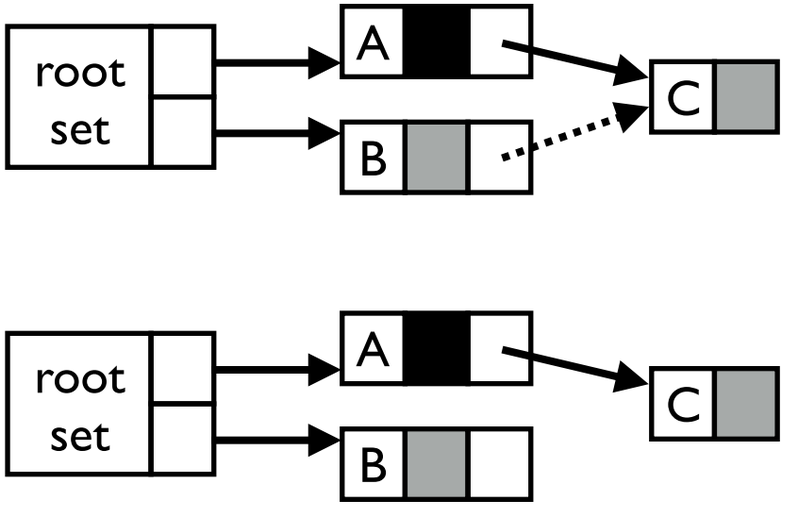
Scanned Reachable object

Unvisited/unreachable object

Unscanned reachable object

Write Barriers

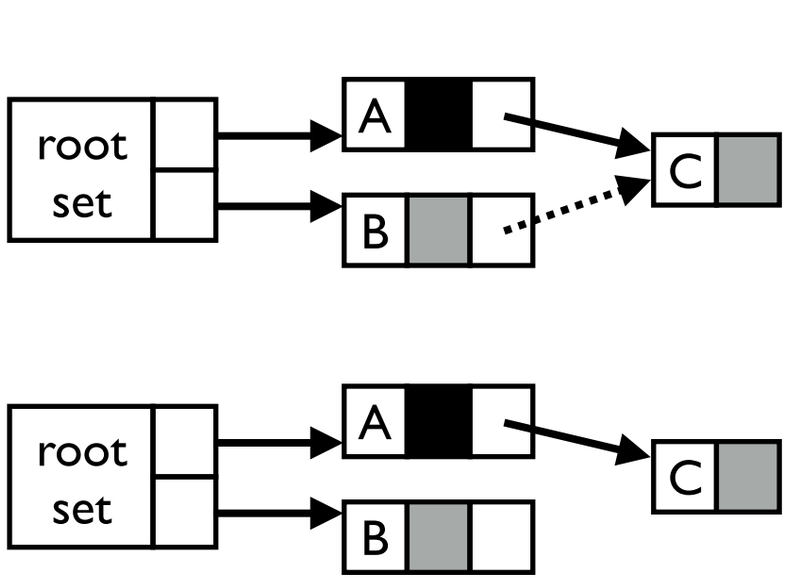
- Whenever a reference to a white object is overwritten, no matter if the reference field was already scanned or not, the object referenced by the overwritten reference is colored gray



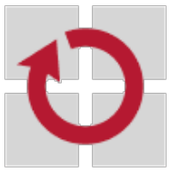
Scanned Reachable object Unvisited/Unreachable object Unscanned reachable object

Write Barriers

- Whenever a reference to a white object is overwritten, no matter if the reference field was already scanned or not, the object referenced by the overwritten reference is colored gray



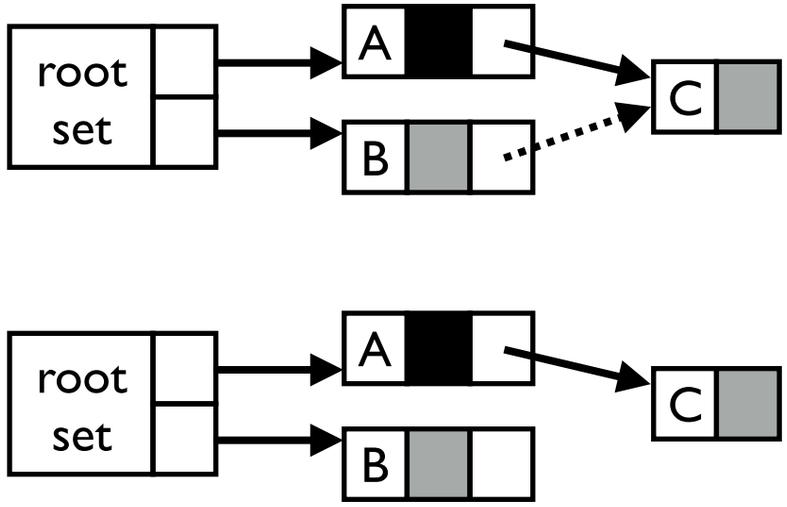
Object B's reference shall be overwritten with null, but currently it refers to object C. C is colored gray by the barrier



Scanned Reachable object Unvisited/Unreachable object Unscanned reachable object

Write Barriers

- Whenever a reference to a white object is overwritten, no matter if the reference field was already scanned or not, the object referenced by the overwritten reference is colored gray



Scanned Reachable object Unvisited/Unreachable object Unscanned reachable object

Latency-Aware Scan-and-Mark

- Write barriers are activated by the GC
- Write barriers are expensive, inserted by the compiler only for
  - Static references
  - Reference fields
  - Field writes to references of object array
- Atomic scanning of task stacks
  - Blocked tasks
  - Complexity linear in its stack size
  - The task currently being scanned is delayed
  - Discovered references are colored gray (reachable, unscanned)

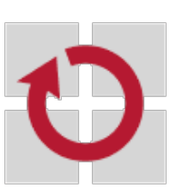
Agenda

How can static knowledge about the application, operating system and hardware-specifics be used to create an automated memory management in embedded systems?

Can automated and safe memory management efficiently be applied in resource-constraint embedded systems?

Case study with the embedded KESO Java Virtual Machine

- Cooperative Memory Management
  - Compiler Support
  - Safe and Latency-Aware Garbage Collection
- Evaluation



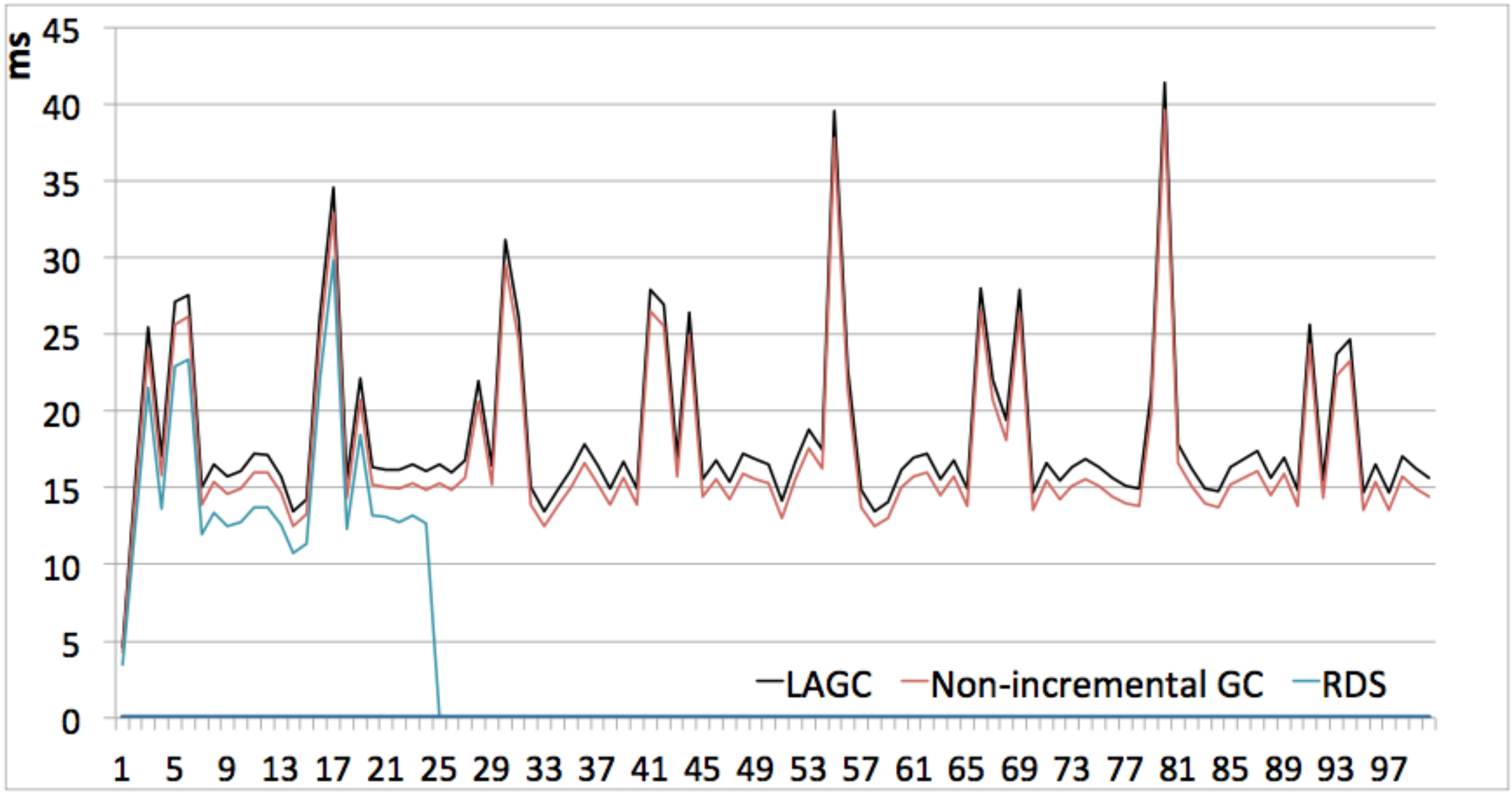
Evaluation

- Real-time Java application benchmark Collision Detector (CDx)
- Built KESO CDj variants
  - Incremental LAGC
  - **Incremental Safe-LAGC** (Protection against soft errors)
- Influence of compiler-assisted memory management
- Effectiveness of integrity checks
- Measured overhead of Safe-LAGC in contrast to LAGC
  - TriCore TC1796 (32-bit, 1 MiB SRAM, 2 MiB ROM)
  - Heap usage at runtime
  - Runtime

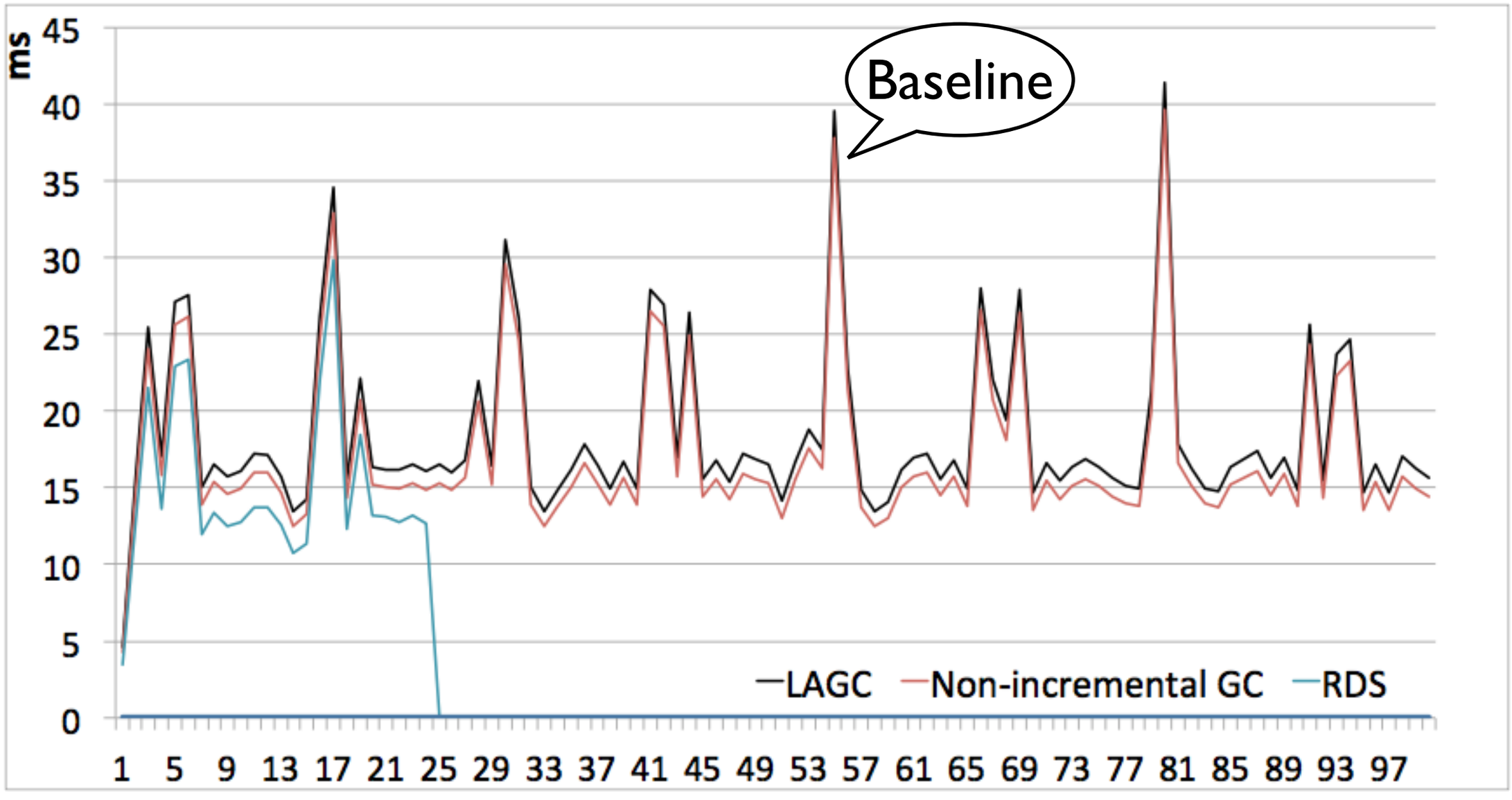
Compiler-Assisted Memory Management

- Escape analysis
  - 30% of all allocations are eligible for stack allocation
  - Reduces heap usage by 45%
  - Application runtime is boosted by 9.5%
- Immortal and runtime final analyses
  - Reduces segment sizes: .text (-14%), .data (-41%)
  - Reduces runtime of application by 12%
  - Placing immortal data in flash: runtime increases by 8%

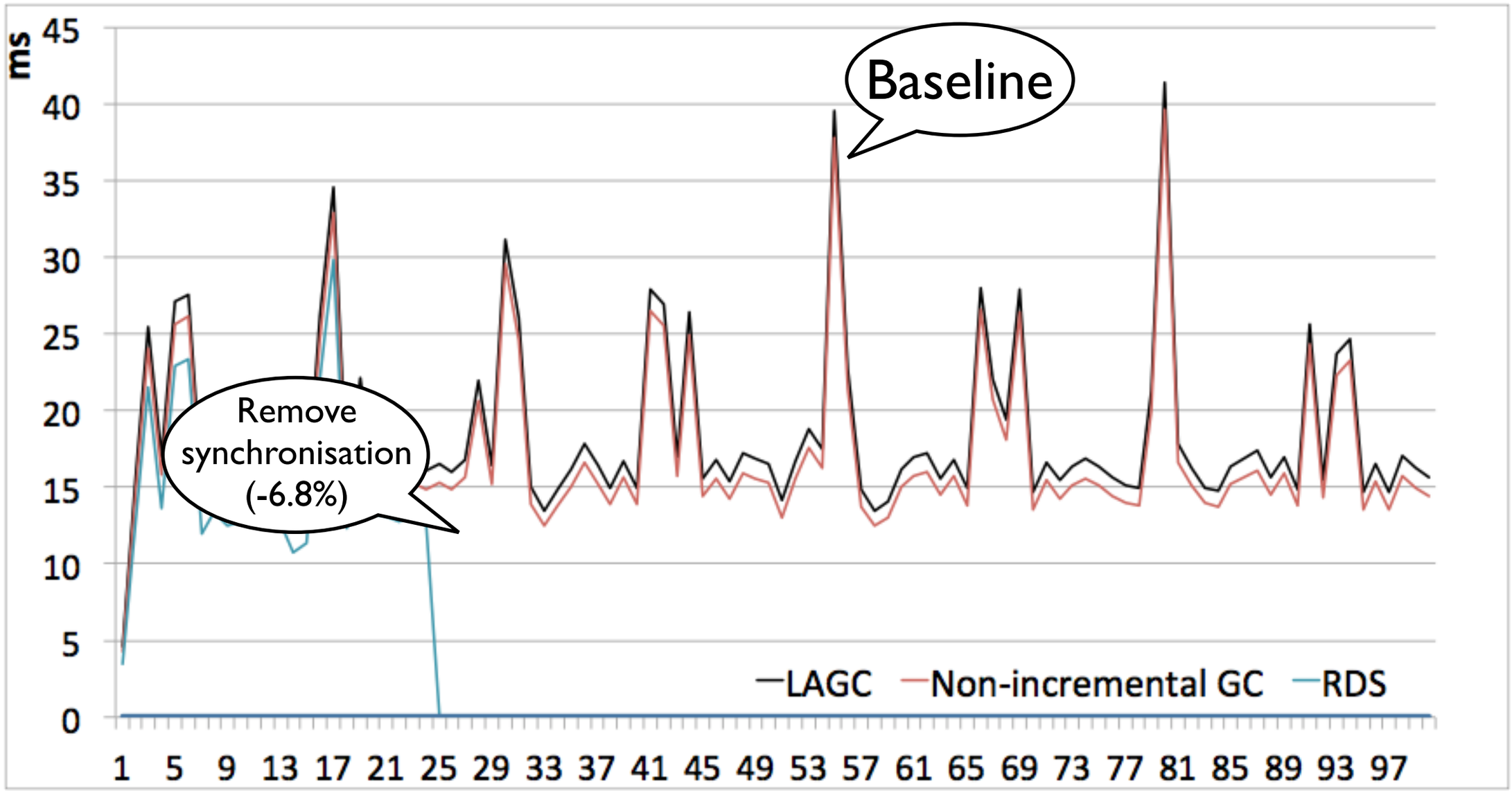
Runtime


ms
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
LAGC
Non-incremental GC
RDS

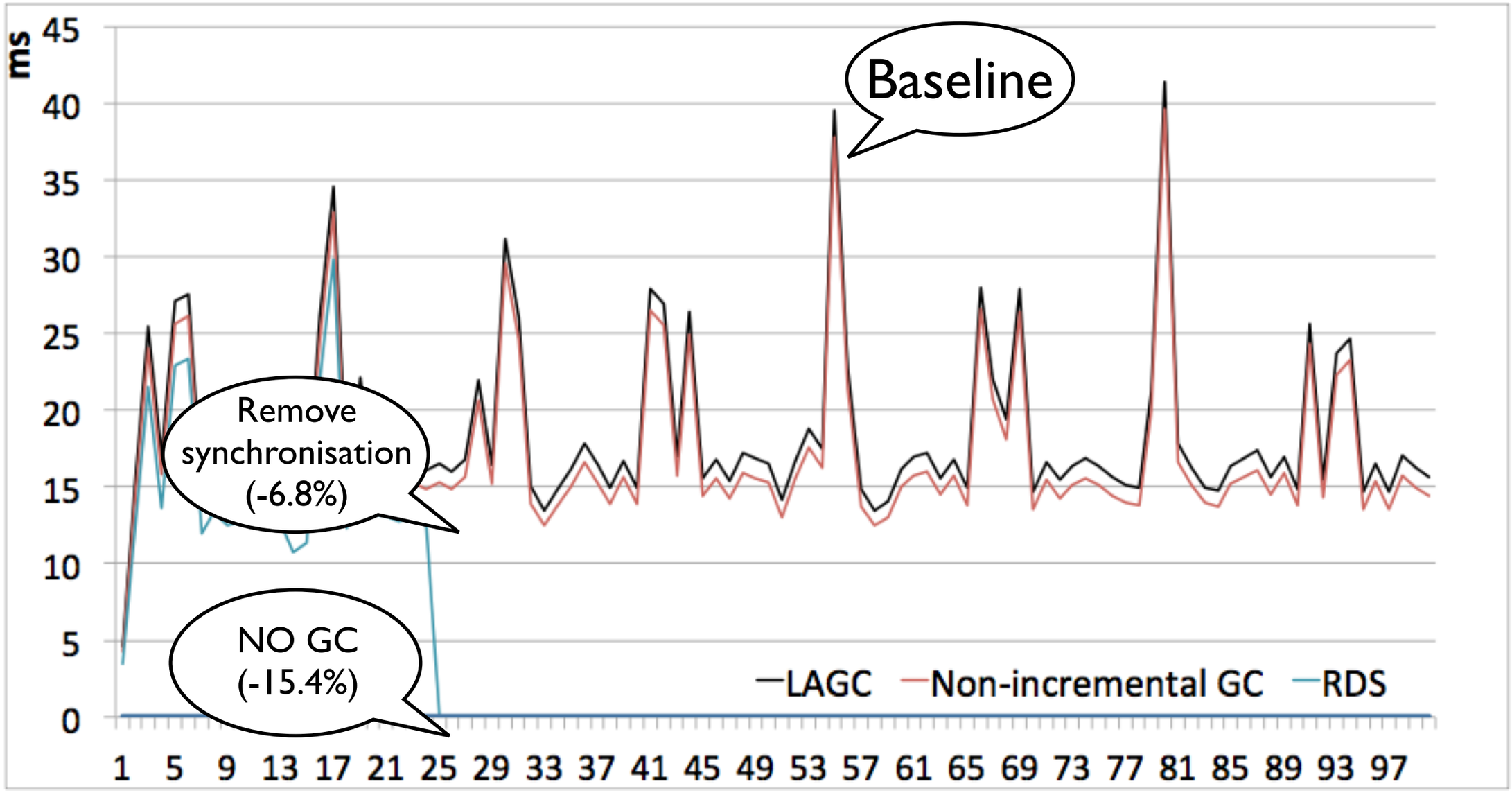
Runtime


ms
45
40
35
30
25
20
15
10
5
0
Baseline
LAGC
Non-incremental GC
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

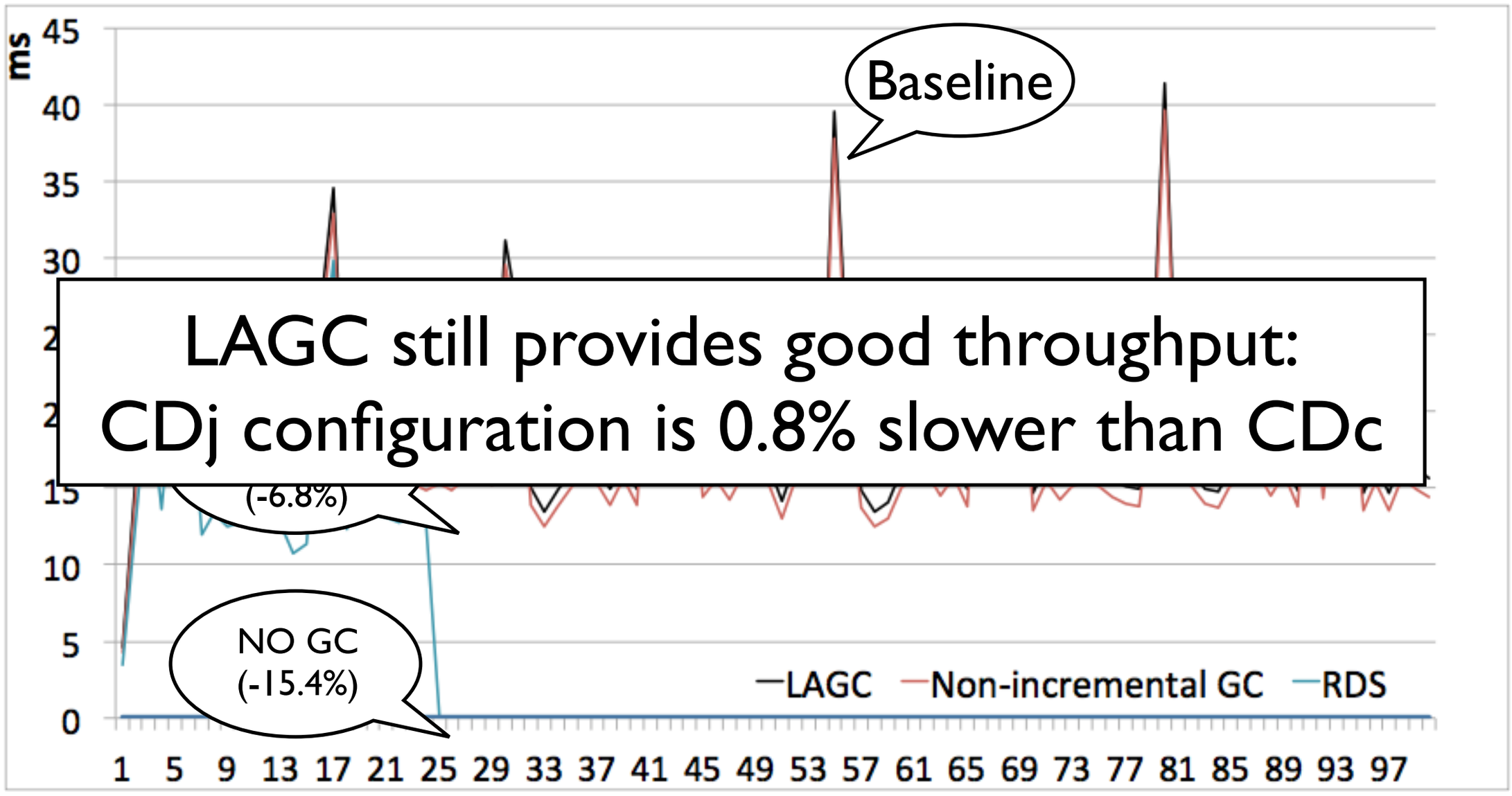
Runtime


ms
45
40
35
30
25
20
15
10
5
0
Baseline
Remove synchronisation (-6.8%)
LAGC
Non-incremental GC
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

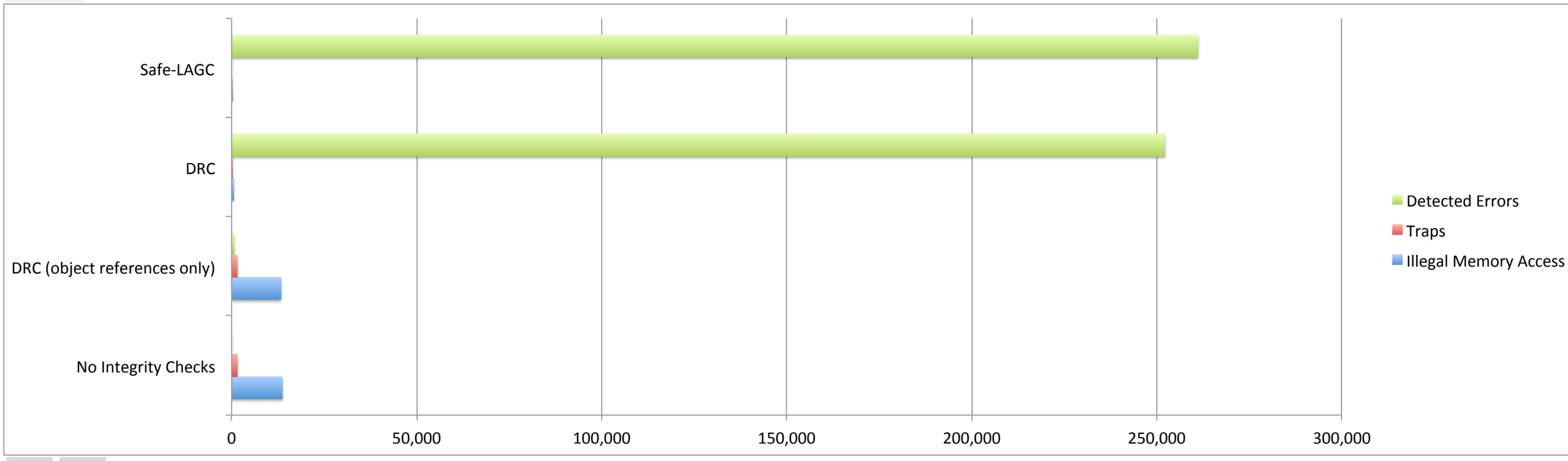

ms
45
40
35
30
25
20
15
10
5
0
Baseline
Remove synchronisation (-6.8%)
NO GC (-15.4%)
LAGC
Non-incremental GC
RDS
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97

Runtime


ms
45
40
35
30
15
10
5
0
Baseline
(-6.8%)
NO GC
(-15.4%)
LAGC
Non-incremental GC
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
LAGC still provides good throughput:
CDj configuration is 0.8% slower than CDc

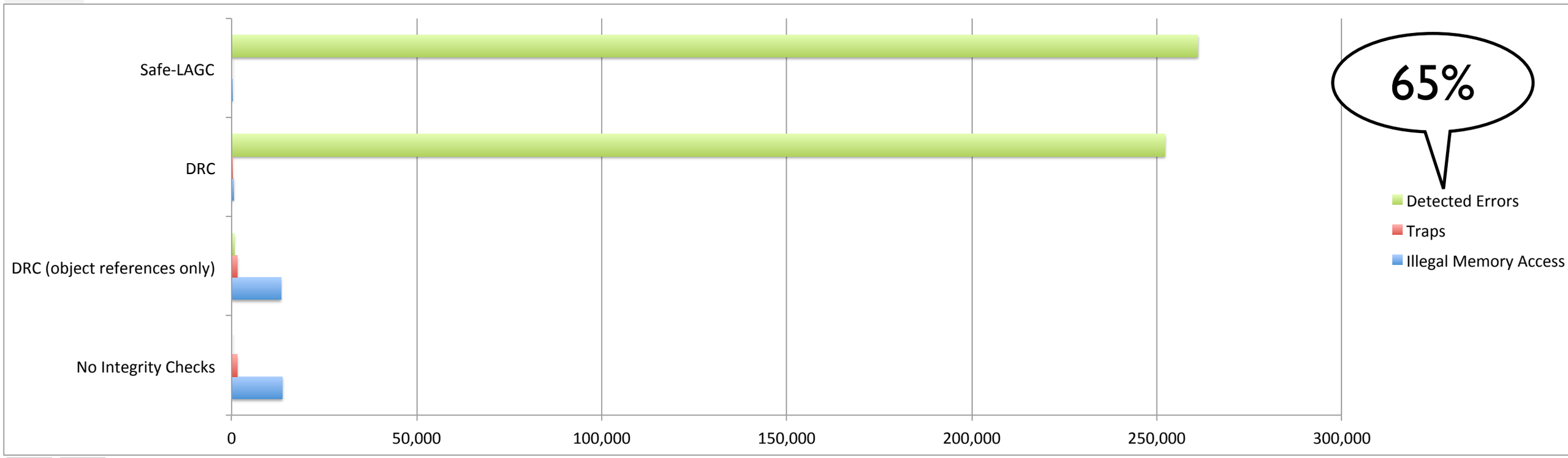
Effectiveness of Integrity Checks

- FAIL*: Fault injection tool, application of fault space pruning
- Injected bit flips into each position of a memory word
- Application data is physically grouped by KESO's reachability analysis



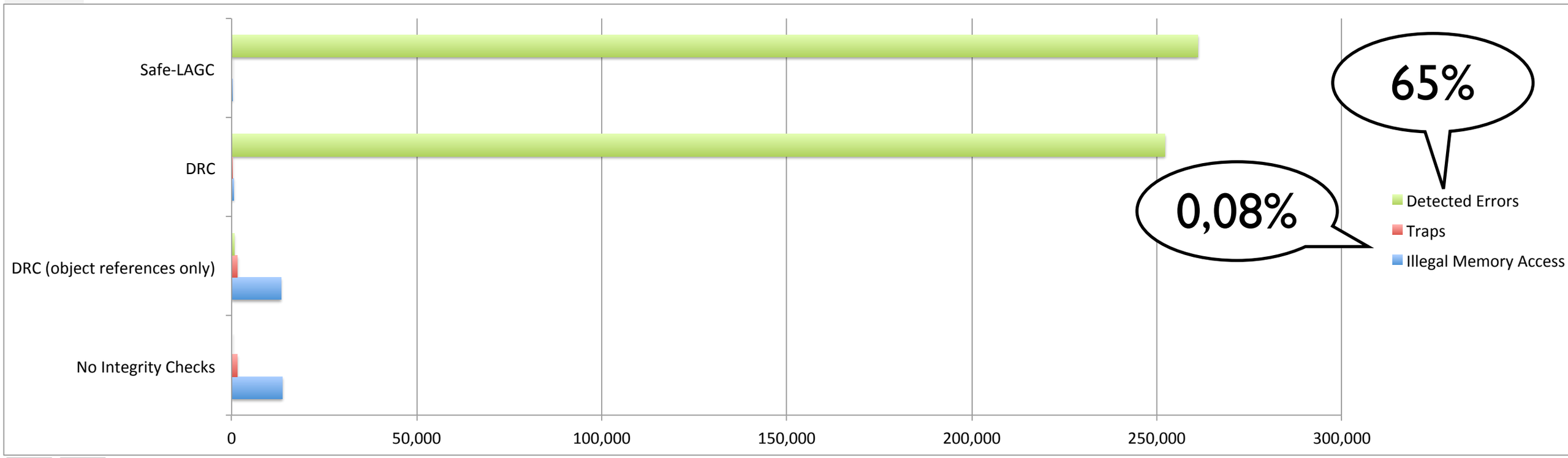
Effectiveness of Integrity Checks

- FAIL*: Fault injection tool, application of fault space pruning
- Injected bit flips into each position of a memory word
- Application data is physically grouped by KESO's reachability analysis



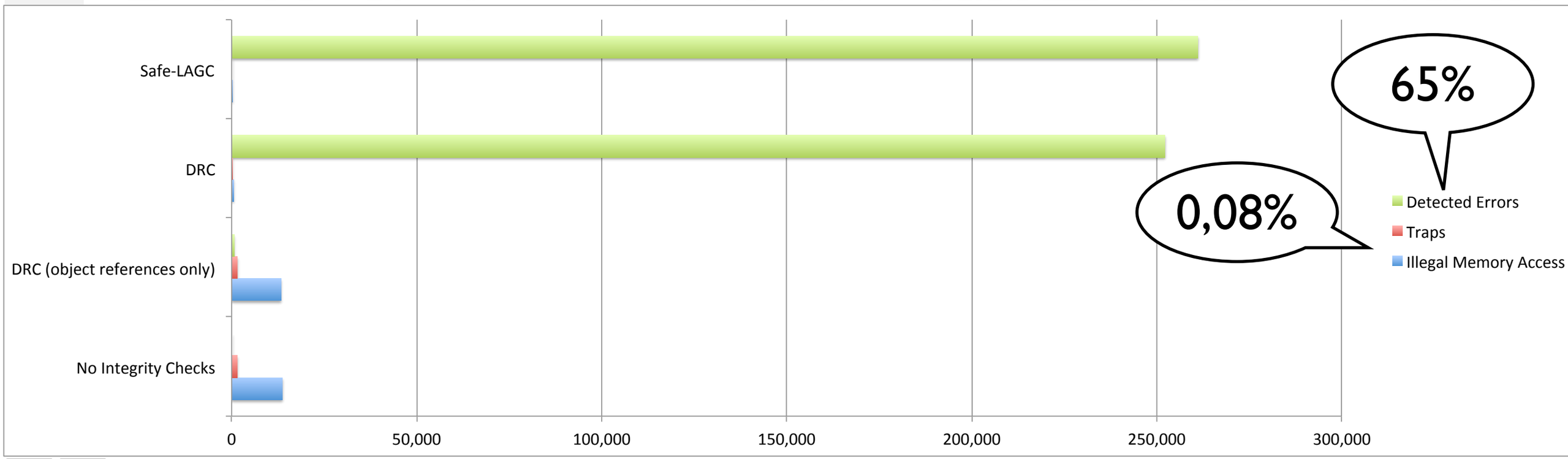
Effectiveness of Integrity Checks

- FAIL*: Fault injection tool, application of fault space pruning
- Injected bit flips into each position of a memory word
- Application data is physically grouped by KESO's reachability analysis



Effectiveness of Integrity Checks

Establishment of software-based memory protection: Foundation to built replication for the protection of application data



Overhead of Protection

- Runtime:
  - Safe-LAGC
    - 55%(DRC)-95%(full protection) for one GC execution instance
    - GC's execution time is short in contrast to application's
  - Application
    - 44% increase
- Memory needed at runtime: no increase

Conclusion

How can static knowledge about the application, operating system and hardware-specifics be used to create an automated memory management in embedded systems?

- Cooperative approach
  - Type-safe language and compiler assistance
  - Assistance of the application developer during the integration
  - Incorporation of application, OS model and hardware specifics
  - Latency-aware and safe garbage collection

Can automated and safe memory management efficiently be applied in resource-constraint embedded systems?

- Evaluation results show that generated system can compete with unsafe and manual approaches

Questions?

- http://www4.cs.fau.de/Research/KESO/
- KESO: distributed under the terms of the GNU LGPL, version 3