Applications of Escape Analysis in Embedded Real-Time Systems

Isabella Stilkerich, Clemens Lang, Christoph Erhardt,
Michael Stilkerich

Department of Computer Science 4
Distributed Systems and Operating Systems

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Motivation

- Type-safe languages such as Java are beneficial
  - Enhanced productivity
  - Safety benefits attributed to memory safety
- To ensure memory safety, memory management is implicit
  - Heap management (via e.g. garbage collectors (GC))
  - Compiler-assisted management (e.g. region inference, escape analysis)
- Escape analysis
  - Automated stack allocation
  - Lock elision
  - Other applications?

Agenda

How can the information collected by escape analysis help to support the use of Java in the domain of embedded systems?

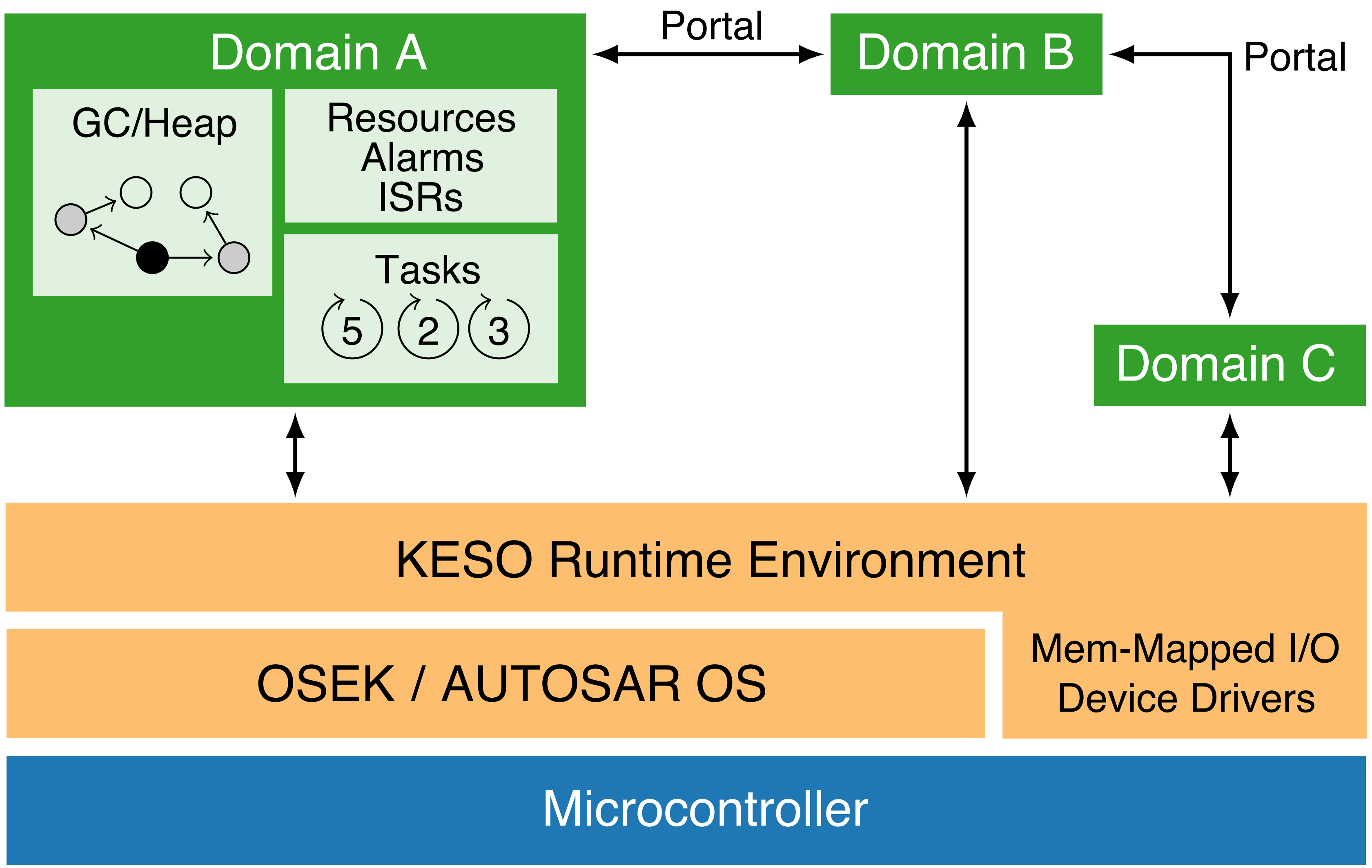
Are the alternative applications of escape analysis beneficial for the non-functional properties of an application?

<u>Case study with the embedded KESO Java Virtual Machine</u>

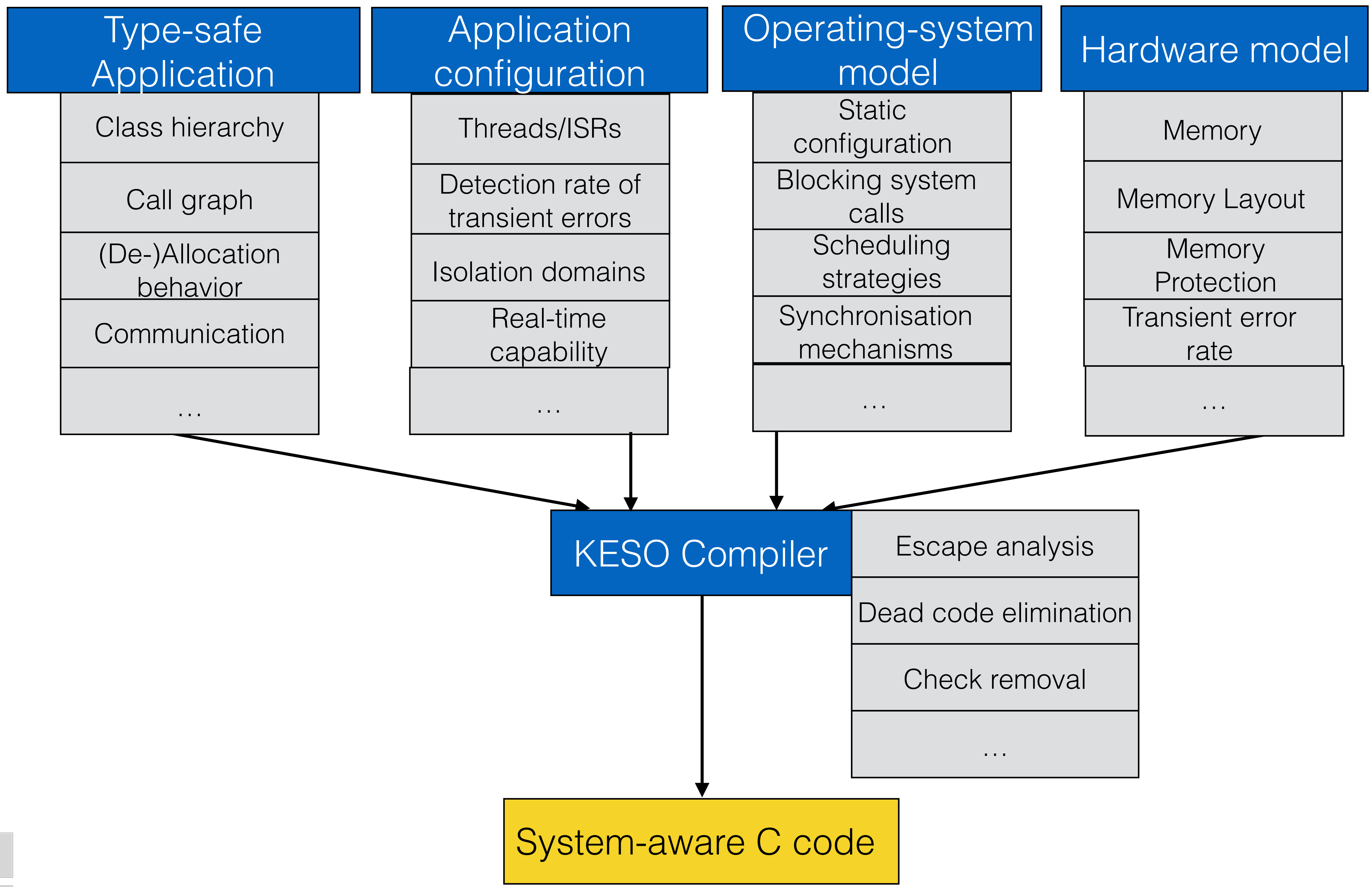
- Alternative applications of escape analysis
- Evaluation

The KESO JVM

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



KESO’s Compiler



Escape Analysis (EA) in Java

- From a conceptual point of view, all Java object are heap-allocated
  - No dedicated language support
  - Preservation of the soundness of Java's type system
- EA is a static analysis to determine an object's **escape state**
  - By using information from alias analysis and reachability of references
  - Automated and safe stack allocation
    - (De-)allocation is very efficient and predictable
    - No fragmentation
  - Garbage collection effort is reduced
  - Synchronization lock optimization

Applications of Escape Analysis

We use EA to provide more support for embedded systems

1. Fast remote-procedure-call support for software-isolated components
2. Scope extension and stack allocation
3. Scope extension and thread-local heaps
4. Automated inference of immutable data
5. Determine survivability for real-time heap management
6. Explicit, safe manual memory management
7. Resource-efficient mitigation of transient errors
8. Object inlining

Applications of Escape Analysis

We use EA to provide more support for embedded systems

1. Fast remote-procedure-call support for software-isolated components
2. Scope extension and stack allocation
3. Scope extension and thread-local heaps
4. Automated inference of immutable data
5. Determine survivability for real-time heap management
6. Explicit, safe manual memory management
7. Resource-efficient mitigation of transient errors
8. Object inlining

In the paper!

Escape Analysis in a Nutshell

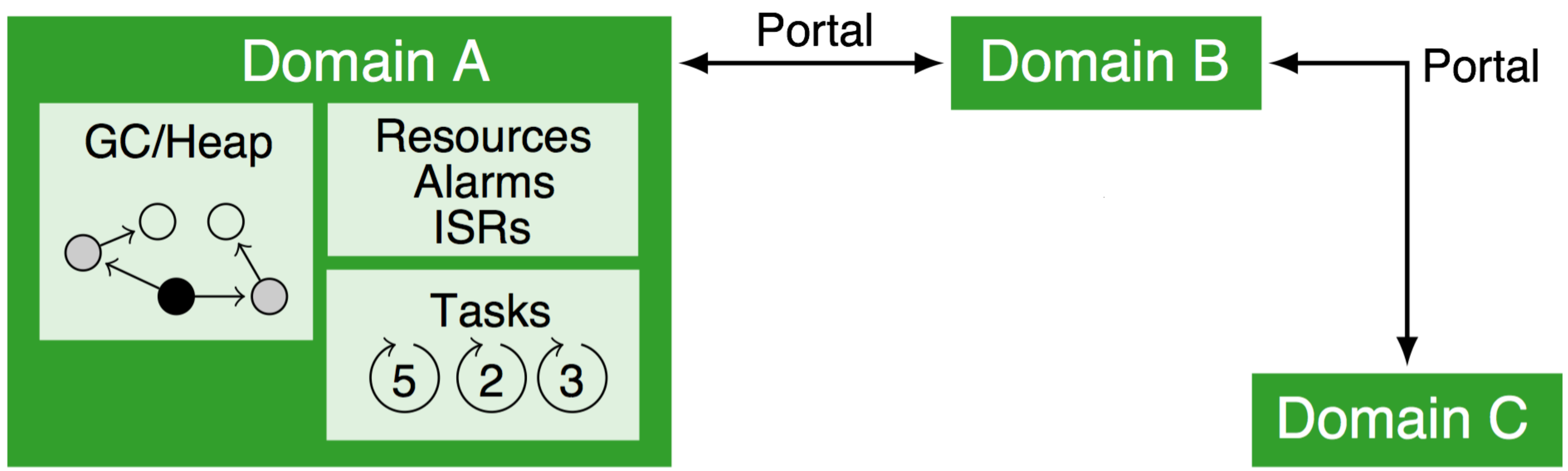
- Our escape analysis is based on Choi et al., TOPLAS ’03: „Stack allocation and synchronization optimizations for Java using escape analysis“
  - Control-flow sensitive analysis
  - Focus is on preciseness of analysis results
  - Ahead-of time compilation
  - Compile times are reasonable (perform graph compression inspired by Steensgard’s *“Points-To Analysis in Almost Linear Time"*)
- Alias information is gathered from the application
  - Method-local analysis
  - Global analysis
  - *Connection graphs (CG)* hold alias information

Applications of Escape Analysis

We use EA to provide more support for embedded systems

1. **Fast remote-procedure-call support for software-isolated components**
2. Scope extension and stack allocation
3. Scope extension and thread-local heaps
4. Automated inference of immutable data
5. Determine survivability for real-time heap management
6. Explicit, safe manual memory management
7. Resource-efficient mitigation of transient errors
8. Object inlining

Remote Procedure Calls via Portals



- Software-isolated applications may need to communicate
- Usually, reference values must never be propagated
- Deep copy of parameters can affect the program's runtime
  - Similar to *Exotask* model (Auerbach et al., TOPLAS '09)
  - Execution time / memory consumption
  - In the service domain, a GC is needed to get rid of copies

Remote-Procedure-Call Support

- Escape analysis can determine, if deep copy is needed
  1. Global escape state in callee’s connection graph
  2. Modification by the callee
- Provides *copy-on-write* and *copy-on-escape* semantics
- Time and memory consumption is significantly improved for particular communication scenarios

Remote-Procedure-Call Support

Deep copy can be omitted

Remote-Procedure-Call Support

Deep copy can be omitted

1. No global escape state in callee’s connection graph

- Portal parameter has a complement in callee’s domain
- The complement and its members do not have a global escape state
- Computed via worklist algorithm

Remote-Procedure-Call Support

Deep copy can be omitted

1. No global escape state in callee’s connection graph

- Portal parameter has a complement in callee’s domain
- The complement and its members do not have a global escape state
- Computed via worklist algorithm

2. No modification by the callee

- Construct mapping between CG representation of objects and index of portal parameter
- Code reachable from portal handler is searched write operations whose operands feature this mapping
- If mapping does not exist, the deep copy can be omitted

Applications of Escape Analysis

We use EA to provide more support for embedded systems

1. Fast remote-procedure-call support for software-isolated components
2. **Scope extension and stack allocation**
3. Scope extension and thread-local heaps
4. Automated inference of immutable data
5. Determine survivability for real-time heap management
6. Explicit, safe manual memory management
7. Resource-efficient mitigation of transient errors
8. Object inlining

Scope Extension (SE)

```
public class Factory {
  class Builder {
  // ...
  }
  protected Builder getBuilder() {
    return new Builder();
  }
}
class Simulation implements Runnable {
  public void run() {
    Factory f=new Factory();
    while (true) {
      Builder b=f.getBuilder();
      for (Aircraft a : getAircraft()) {
        b.addPosition(a, getPositionForAircraft(a));
      }
      // b's last reference
      SimFrame frame=b.makeFrame();
      simulate(frame);
    }
  }
// …
}
```

Scope Extension (SE)

```java
public class Factory {
  class Builder {
  // ...
  }
  protected Builder getBuilder() {
    return new Builder();
  }
}
class Simulation implements Runnable {
  public void run() {
    Factory f=new Factory();
    while (true) {
      Builder b=f.getBuilder();
      for (Aircraft a : getAircraft()) {
        b.addPosition(a, getPositionForAircraft(a));
      }
      // b's last reference
      SimFrame frame=b.makeFrame();
      simulate(frame);
    }
  }
// …
}
```

SE and Stack Allocation

(Stack) scope extension in a nutshell

- Method-escaping objects
- Non-virtual methods (supported by devirtualization)
- Copy of the allocation into all callers
- Pass a reference at the invocation of the source method
- Allocation is substituted by a read of the parameter values

SE and Stack Allocation

(Stack) scope extension in a nutshell

- Method-escaping objects
- Non-virtual methods (supported by devirtualization)
- Copy of the allocation into all callers
- Pass a reference at the invocation of the source method
- Allocation is substituted by a read of the parameter values

Problems imposed by extended stack scopes

- Virtual methods (e.g. adapt all method signatures)
- Objects allocated by mutually exclusive control flows
- Increased footprint
- Overhead imposed by parameter passing

Applications of Escape Analysis

We use EA to provide more support for embedded systems

1. Fast remote-procedure-call support for software-isolated components
2. Scope extension and stack allocation
3. **Scope extension and thread-local heaps**
4. Automated inference of immutable data
5. Determine survivability for real-time heap management
6. Explicit, safe manual memory management
7. Resource-efficient mitigation of transient errors
8. Object inlining

SE + Thread-Local Heaps (TLH)

- Stack allocation might not be a viable solution
  - E.g., increased worst-case stack usage
  - Variables with overlapping liveness regions
- Use of a special heap region that is managed by the compiler
  - Region is not subject to GC sweeps, but co-exists with the GC
- Logical region for each method
  - Regions are organized in a stack-like manner
  - Each thread has its own heap

SE + Thread-Local Heaps (TLH)

- Each TLH has a fill marker and a maximum fill level
  - Fill marker is saved upon method entry (static analysis can avoid that)
  - Objects are allocated by moving the fill marker
  - Deallocation is done by resetting the fill marker
- Synchronization upon allocations is no longer needed
- Checks are inserted to prevent heap overflows
- Liveness-interference avoidance can be disabled
  - Reduces GC load

Evaluation

- Microbenchmark for *Remote-Procedure-Call Support*
- Real-time Java application benchmark *Collision Detector (CDj)*
- Built KESO CDj variants (including stop-the-world GC, 600kB heap)
  - *Scope Extension and Stack Allocation*
  - *Scope Extension and Thread-Local Heaps*
- Influence of Escape Analysis measured on real-world setup
  - CiAO AUTOSAR OS version 4c19874
  - TriCore TC1796 (32-bit processor)
    - 1 MiB external SRAM, 2 MiB internal flash
    - CPU clocked at 150MHz, 75 MHz Bus
  - Compiled with GCC 4.5.2
  - Heap / execution time usage at runtime

Remote-Procedure-Call Support

Call Type	Execution Time
portal call	3.76 μs
regular virtual method call	3.09 μs
AUTOSAR non-trusted function	32.91 μs
portal call (2 int params)	4.17 μs
portal call (3 int params)	4.70 μs
portal call (1 element linked list)	31.47 μs
portal call (2 element linked list)	56.08 μs
portal call (3 element linked list)	84.37 μs
portal call (escape analysis, 1 element linked list)	3.99 μs
portal call (escape analysis, 2 element linked list)	4.19 μs
portal call (escape analysis, 3 element linked list)	4.65 μs

- In case parameters are not modified and do not escape, the execution time is reduced significantly
- Memory needed for deep copy is saved

Remote-Procedure-Call Support

Call Type	Execution Time
portal call	3.76 μs
regular virtual method call	3.09 μs
AUTOSAR non-trusted function	32.91 μs
portal call (2 int params)	4.17 μs
portal call (3 int params)	4.70 μs
portal call (1 element linked list)	31.47 μs
portal call (2 element linked list)	56.08 μs
portal call (3 element linked list)	84.37 μs
portal call (escape analysis, 1 element linked list)	3.99 μs
portal call (escape analysis, 2 element linked list)	4.19 μs
portal call (escape analysis, 3 element linked list)	4.65 μs

22%

- In case parameters are not modified and do not escape, the execution time is reduced significantly
- Memory needed for deep copy is saved

Remote-Procedure-Call Support

Call Type	Execution Time
portal call	3.76 μs
regular virtual method call	3.09 μs
AUTOSAR non-trusted function	32.91 μs
portal call (2 int params)	4.17 μs
portal call (3 int params)	4.70 μs
portal call (1 element linked list)	31.47 μs
portal call (2 element linked list)	56.08 μs
portal call (3 element linked list)	84.37 μs
portal call (escape analysis, 1 element linked list)	3.99 μs
portal call (escape analysis, 2 element linked list)	4.19 μs
portal call (escape analysis, 3 element linked list)	4.65 μs

HW-based protection

- In case parameters are not modified and do not escape, the execution time is reduced significantly
- Memory needed for deep copy is saved

Remote-Procedure-Call Support

Call Type	Execution Time
portal call	3.76 μs
regular virtual method call	3.09 μs
AUTOSAR non-trusted function	32.91 μs
portal call (2 int params)	4.17 μs
portal call (3 int params)	4.70 μs
portal call (1 element linked list)	31.47 μs
portal call (2 element linked list)	56.08 μs
portal call (3 element linked list)	84.37 μs
portal call (escape analysis, 1 element linked list)	3.99 μs
portal call (escape analysis, 2 element linked list)	4.19 μs
portal call (escape analysis, 3 element linked list)	4.65 μs

Primitive Values

- In case parameters are not modified and do not escape, the execution time is reduced significantly
- Memory needed for deep copy is saved

Remote-Procedure-Call Support

Call Type	Execution Time
portal call	3.76 μs
regular virtual method call	3.09 μs
AUTOSAR non-trusted function	32.91 μs
portal call (2 int params)	4.17 μs
portal call (3 int params)	4.70 μs
portal call (1 element linked list)	31.47 μs
portal call (2 element linked list)	56.08 μs
portal call (3 element linked list)	84.37 μs
portal call (escape analysis, 1 element linked list)	3.99 μs
portal call (escape analysis, 2 element linked list)	4.19 μs
portal call (escape analysis, 3 element linked list)	4.65 μs

Deep Copy

- In case parameters are not modified and do not escape, the execution time is reduced significantly
- Memory needed for deep copy is saved

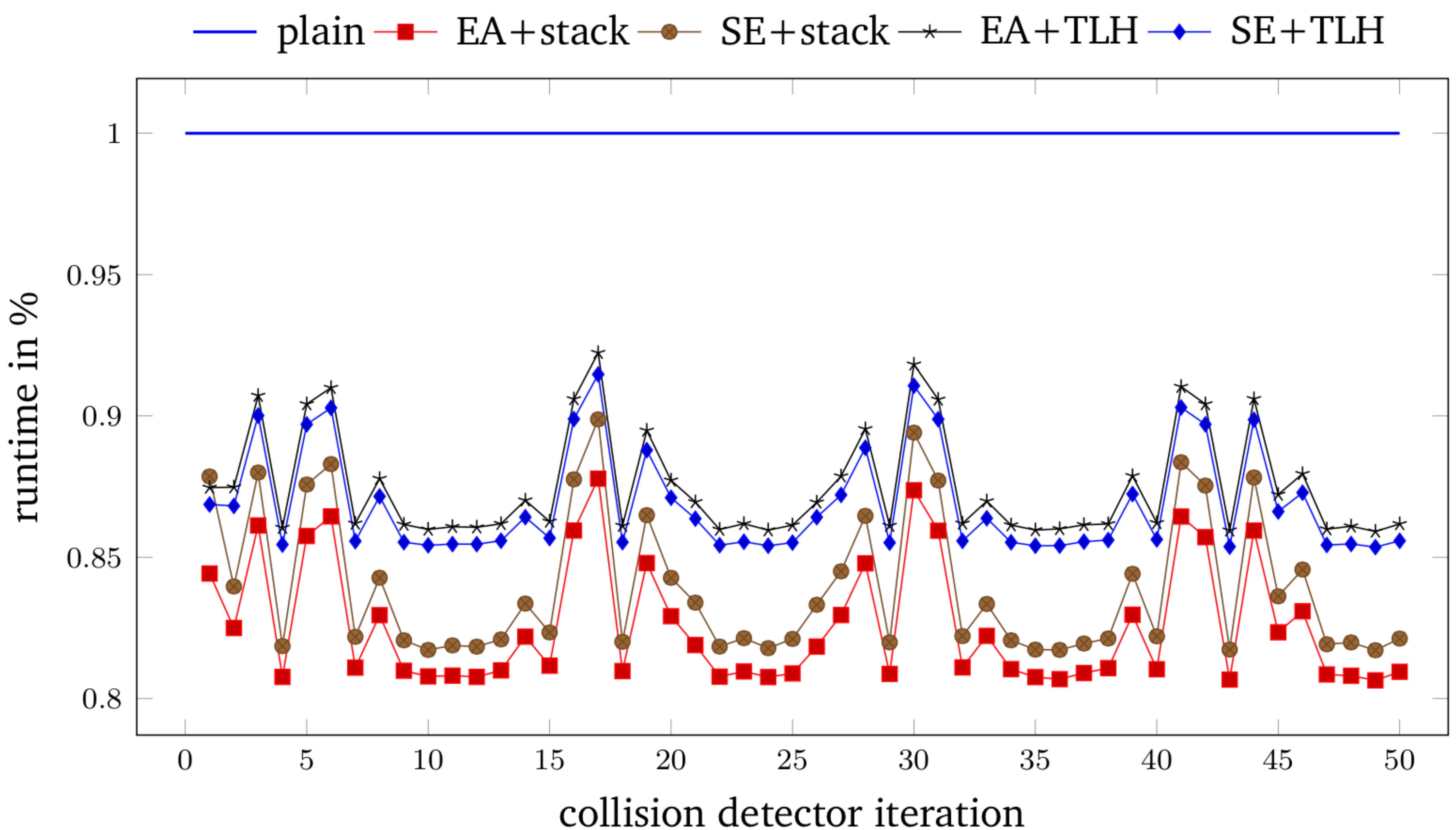
Remote-Procedure-Call Support

Call Type	Execution Time
portal call	3.76 μs
regular virtual method call	3.09 μs
AUTOSAR non-trusted function	32.91 μs
portal call (2 int params)	4.17 μs
portal call (3 int params)	4.70 μs
portal call (1 element linked list)	31.47 μs
portal call (2 element linked list)	56.08 μs
portal call (3 element linked list)	84.37 μs
portal call (escape analysis, 1 element linked list)	3.99 μs
portal call (escape analysis, 2 element linked list)	4.19 μs
portal call (escape analysis, 3 element linked list)	4.65 μs

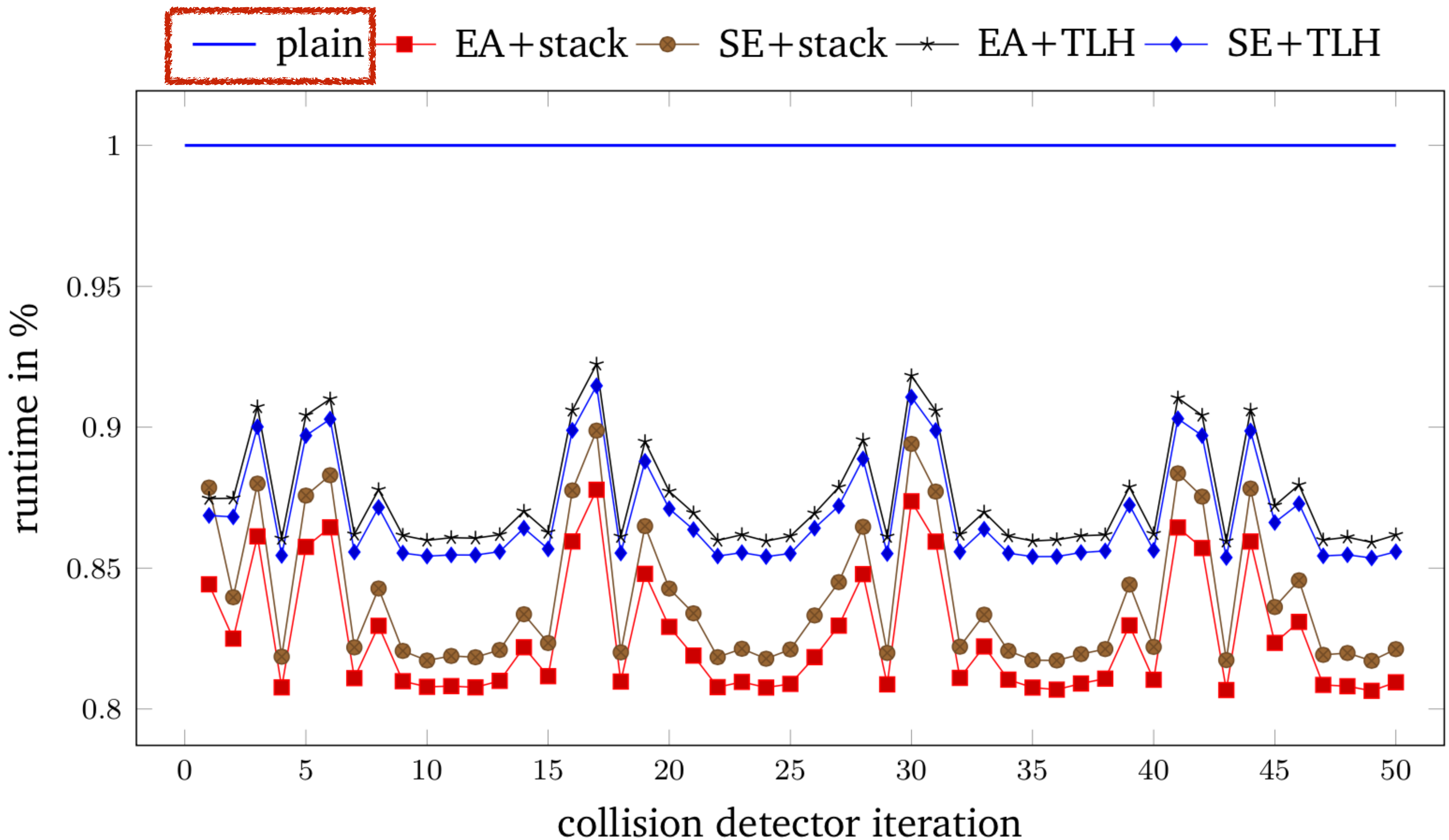
Omitted Copy

- In case parameters are not modified and do not escape, the execution time is reduced significantly
- Memory needed for deep copy is saved

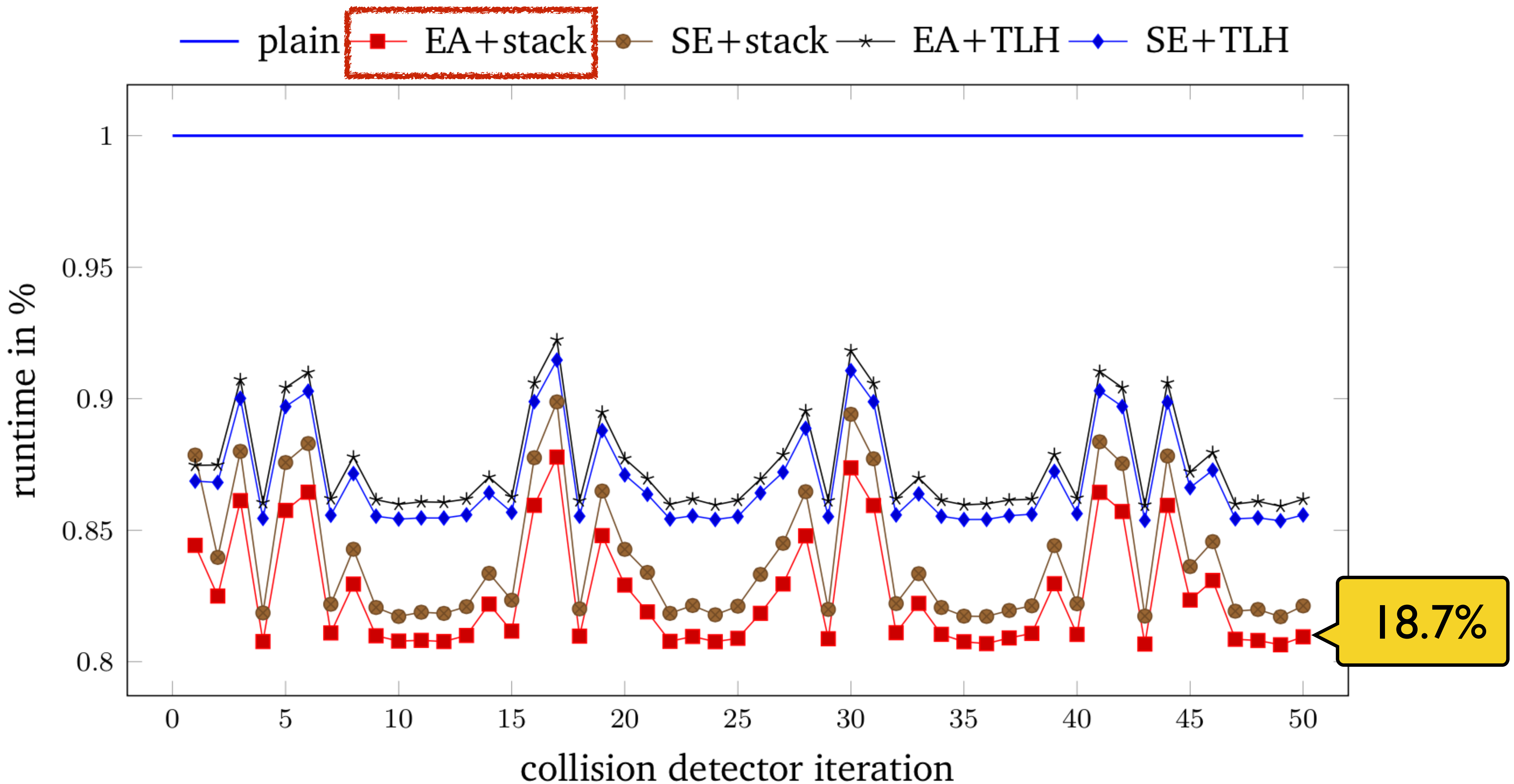
Runtime for CDj (SE / THL)



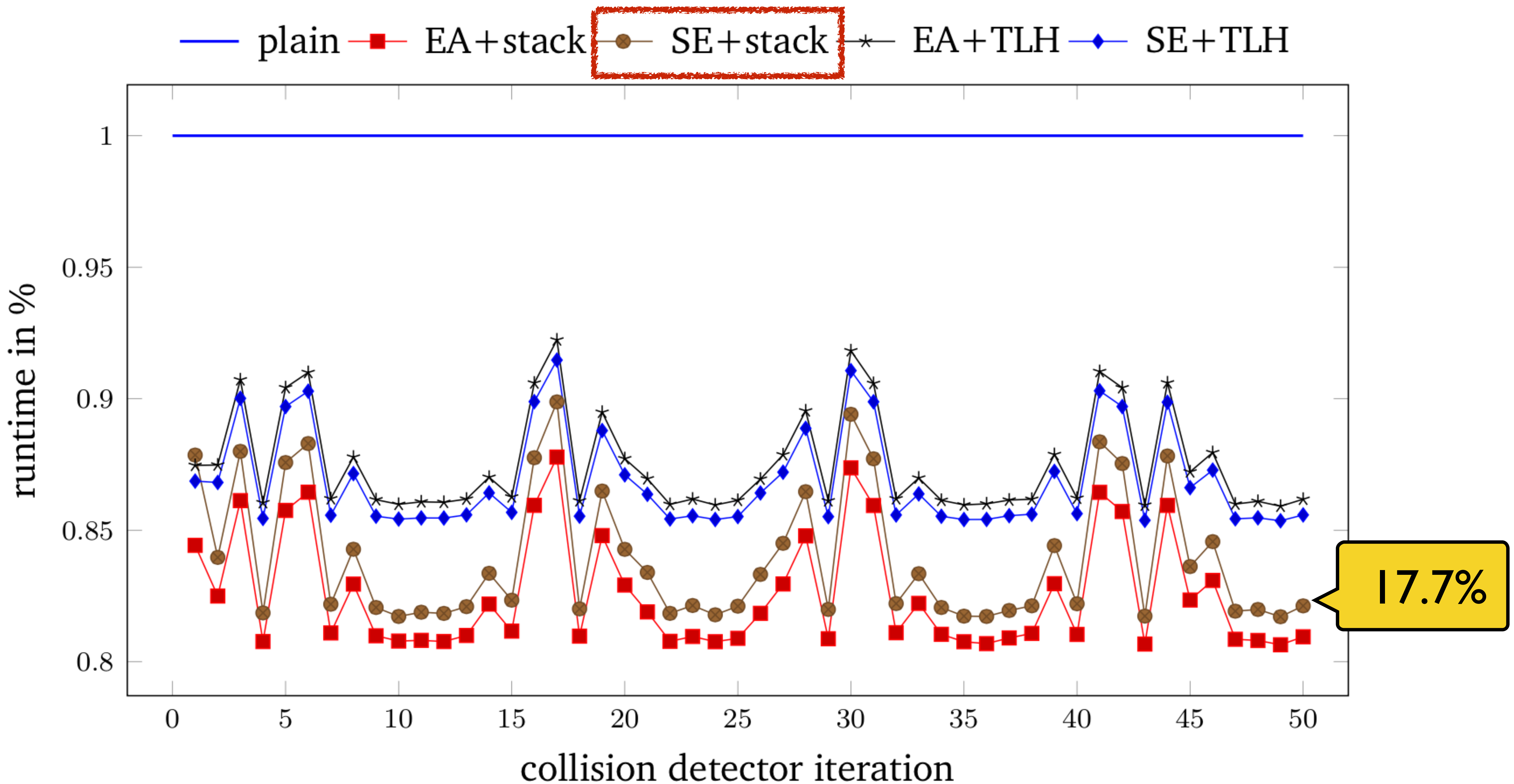
Runtime for CDj (SE / THL)



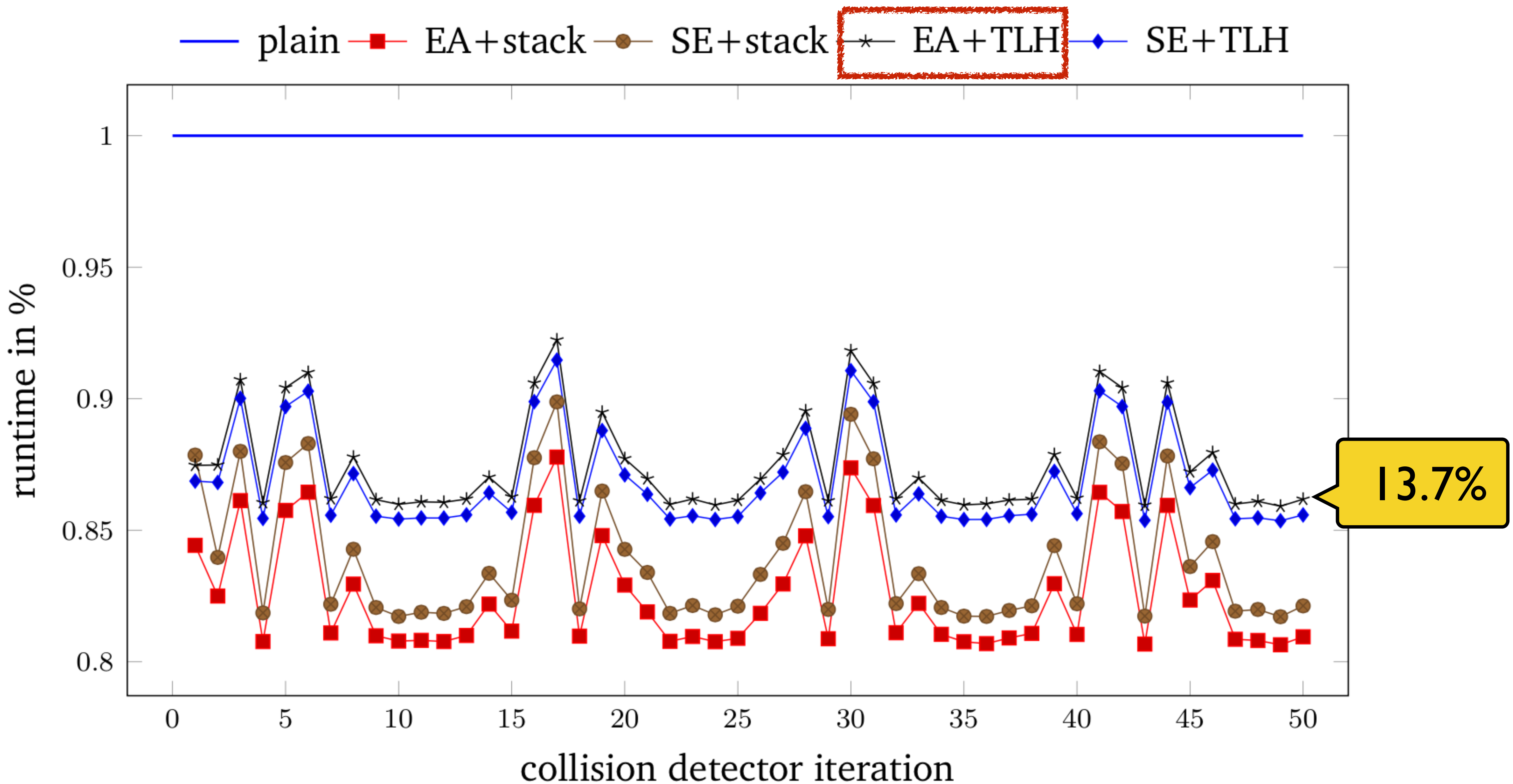
Runtime for CDj (SE / THL)



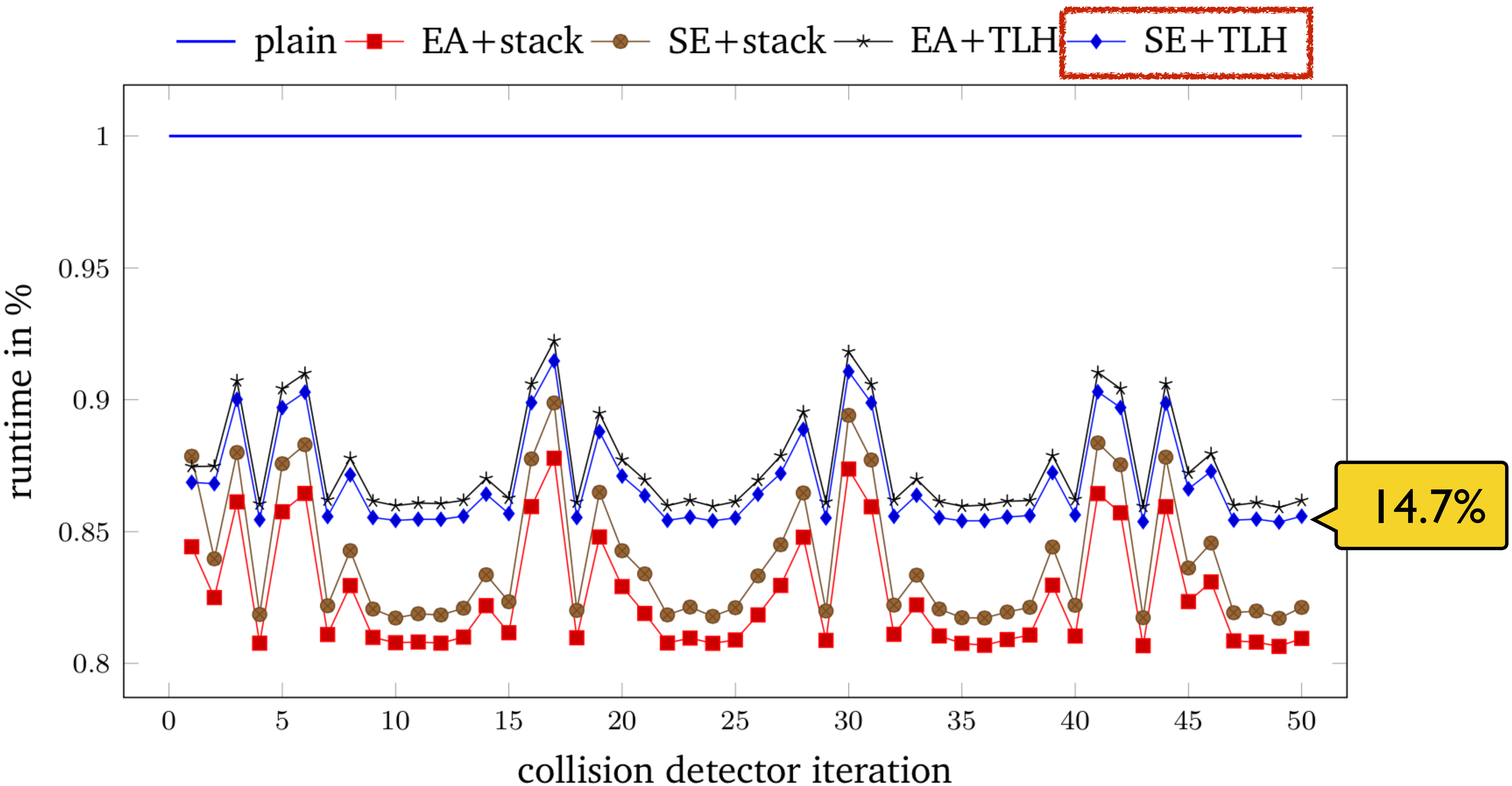
Runtime for CDj (SE / THL)



Runtime for CDj (SE / THL)

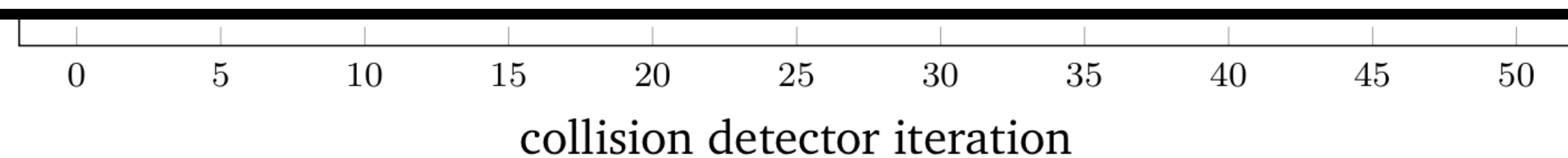


Runtime for CDj (SE / THL)



Runtime for CDj (SE / THL)

- Combining ordinary stack allocation and TLHs is beneficial
- Incremental GCs save synchronisation overhead
  - No barriers needed
  - No fragmentation caused by short-living objects
- Speed up for real-time garbage collection
  - Accesses to non-fragmented objects is faster
- 49-51% heap usage for all variants



Conclusion

How can the information collected by escape analysis help to support the use of Java in the domain of embedded systems?

- Global, whole-system escape analysis
  - Static, type-safe programs
  - Consideration of the system configuration

Conclusion

How can the information collected by escape analysis help to support the use of Java in the domain of embedded systems?

- Global, whole-system escape analysis
  - Static, type-safe programs
  - Consideration of the system configuration

Are the alternative applications of escape analysis beneficial for the non-functional properties of an application?

- Light-weight RPC support encourages application isolation
- EA for memory handling solves the same problem as region inference
  - The effects of extended stack scopes are application-specific
  - Task-local bump pointer heaps as alternative memory management strategy using scope extension (reduces allocator and GC effort)

Questions?

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