Compiler-Assisted Memory Management Using Escape Analysis in the KESO JVM

Vortrag zur Masterarbeit

Clemens Lang

Lehrstuhl für Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität Erlangen-Nürnberg

11. Juli 2014

Was ist Fluchtanalyse?

Was ist Fluchtanalyse?

- Statische Analyse zur Nachverfolgung von Objekten und Referenzen
- Benötigt Alias-Information
- Objekte, deren Lebenszeit durch die der allokierenden Methode beschränkt sind, können durch den Compiler verwaltet werden

Was ist Fluchtanalyse?

Was ist Fluchtanalyse?

- Statische Analyse zur Nachverfolgung von Objekten und Referenzen
- Benötigt Alias-Information
- Objekte, deren Lebenszeit durch die der allokierenden Methode beschränkt sind, können durch den Compiler verwaltet werden

Beispiel

```
1  public class Example {
2    public long distance(Point2D a, Point2D b) {
3      Vector2D vec = new Vector2D(b.x - a.x, b.y - a.y);
4      return vec.length();
5    }
6  }
```

Warum Fluchtanalyse?

Warum Fluchtanalyse?

- Reduzierung der Auslastung des Heaps

  $\Rightarrow$ Reduzierung der GC-Laufzeit, Größe der GC-Datenstrukturen

  $\Rightarrow$ Verbesserung der Vorhersagbarkeit des Allokationsaufwands

- Vermeidung von Fragmentierung
- Verbesserung der Laufzeit

Fluchtanalyse in KESO

- Basierend auf Choi *et al.*, TOPLAS '03: „Stack allocation and synchronization optimizations for Java using escape analysis“ [CGS+03]
- Initiale Implementierung in meiner Bachelorarbeit
  - Gute Ergebnisse, aber lange Übersetzungszeiten und keine Laufzeitverbesserung
  - Nur Optimierungen innerhalb einer Methode

Überblick

Einleitung

Verbesserungen der existierenden Fluchtanalyse
- Flusssensitivität
- Der Doppel-`return`-Bug
- Compilezeitverbesserungen
  - Ignorieren von Leseoperationen
  - Connection Graph-Kompression

Erweiterungen der Fluchtanalyse
- Scope-Erweiterung
- Tasklokale Heaps

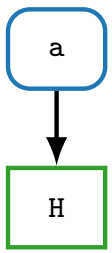
Evaluation

Zusammenfassung & Schluss

Überblick

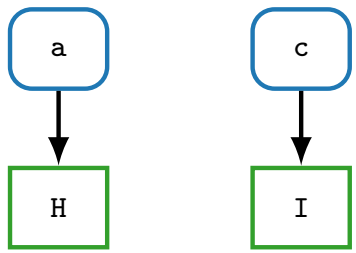
Flusssensitivität

```
1  public class FS {
2    static Obj global;
3
4    public void run() {
5      Obj a = new H();
6      Obj c = new I();
7
8      c = a;
9      FS.global = c;
10   }
11 }
```



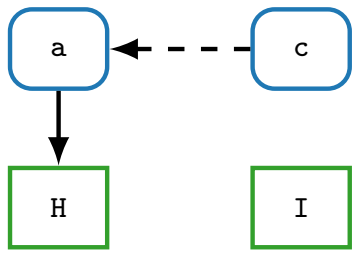
Flusssensitivität

```
1   public class FS {
2     static Obj global;
3
4     public void run() {
5       Obj a = new H();
6       Obj c = new I();
7
8       c = a;
9       FS.global = c;
10    }
11  }
```



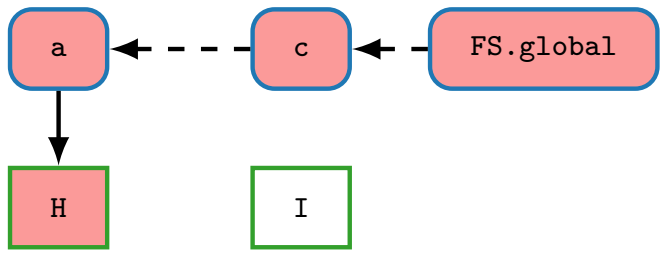
Flusssensitivität

```
1  public class FS {
2    static Obj global;
3
4    public void run() {
5      Obj a = new H();
6      Obj c = new I();
7
8      c = a;
9      FS.global = c;
10   }
11 }
```



Flusssensitivität

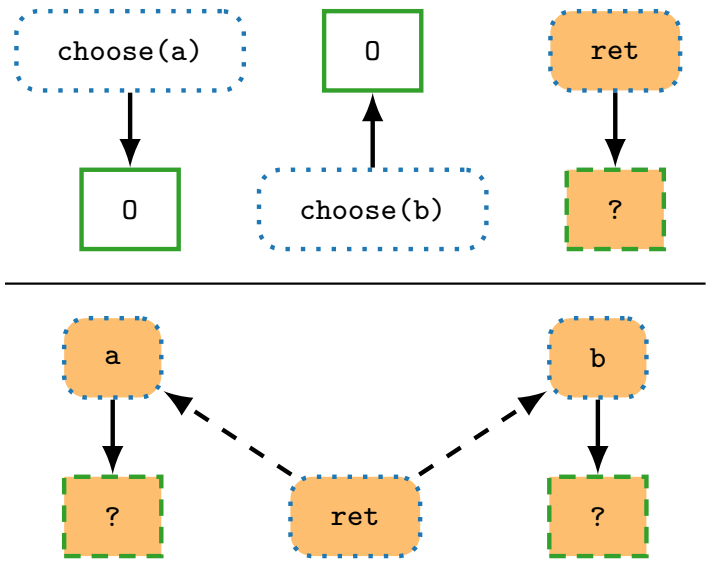
```
1  public class FS {
2    static Obj global;
3
4    public void run() {
5      Obj a = new H();
6      Obj c = new I();
7
8      c = a;
9      FS.global = c;
10   }
11 }
```



Der Doppel-`return`-Bug

- Konzeptioneller Fehler in der Arbeit von Choi et al.
- Bedingt durch einen der Beiträge der Publikation:
  Repräsentation des Effekts einer Methode unabhängig vom Aufrufkontext

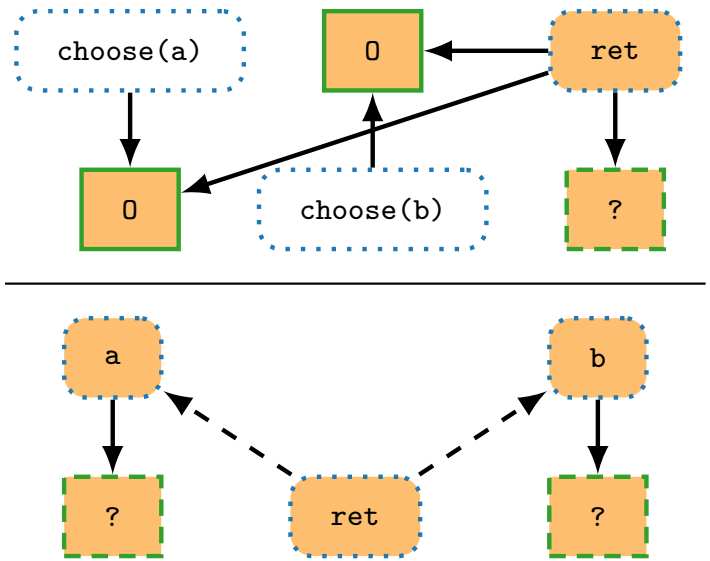
```
1   public class Choose {
2     public static O get() {
3       return choose(new O(), new O());
4     }
5
6     private static O choose(O a, O b) {
7       if (Math.random() < 0.5)
8         return a;
9       return b;
10    }
11  }
```



Der Doppel-`return`-Bug

- Konzeptioneller Fehler in der Arbeit von Choi et al.
- Bedingt durch einen der Beiträge der Publikation:
  Repräsentation des Effekts einer Methode unabhängig vom Aufrufkontext

```
1   public class Choose {
2     public static O get() {
3       return choose(new O(), new O());
4     }
5
6     private static O choose(O a, O b) {
7       if (Math.random() < 0.5)
8         return a;
9       return b;
10    }
11  }
```



- Behoben durch Identifikation von Referenz-Äquivalenz-Paaren zwischen Aufrufer und Aufgerufenem

Kampf der Compilezeit

- Lange Übersetzungszeiten nach Double-`return`-Fix: ~19 min. für 28 kSLOC

  $\Rightarrow$ Schneller analysieren, Präzision beibehalten. Aber wie?

Kampf der Compilezeit

- Lange Übersetzungszeiten nach Double-`return`-Fix: ~19 min. für 28 kSLOC

  ⇒ Schneller analysieren, Präzision beibehalten. Aber wie?
- Pathologisches Beispiel: `equals` in Kombination mit Java Collections API
  - Gegenseitige rekursive Aufrufe
  - Fixpunkt-Iteration in Analyse, Propagation von Knoten in CG der Aufrufer

    ⇒ Explosion der Verbindungsgraphen der Methoden die `equals` nutzen

Kampf der Compilezeit

- Lange Übersetzungszeiten nach Double-`return`-Fix: ~19 min. für 28 kSLOC

  ⇒ Schneller analysieren, Präzision beibehalten. Aber wie?
- Pathologisches Beispiel: `equals` in Kombination mit Java Collections API
  - Gegenseitige rekursive Aufrufe
  - Fixpunkt-Iteration in Analyse, Propagation von Knoten in CG der Aufrufer

    ⇒ Explosion der Verbindungsgraphen der Methoden die `equals` nutzen
- Idee: `equals` liest – nur durch Schreiboperationen können neue Alias-Beziehungen (und damit flüchtende Objekte) entstehen
  - Folgerung: In einem CG sind nur die Kanten relevant, die
    - durch eine Schreiboperation entstehen
    - auf einem zyklenfreien Pfad von einem Einsprungpunkt zu einer solchen Kante liegen

Kampf der Compilezeit

- Lange Übersetzungszeiten nach Double-`return`-Fix: ~19 min. für 28 kSLOC

  ⇒ Schneller analysieren, Präzision beibehalten. Aber wie?
- Pathologisches Beispiel: `equals` in Kombination mit Java Collections API
  - Gegenseitige rekursive Aufrufe
  - Fixpunkt-Iteration in Analyse, Propagation von Knoten in CG der Aufrufer

    ⇒ Explosion der Verbindungsgraphen der Methoden die `equals` nutzen
- Idee: `equals` liest – nur durch Schreiboperationen können neue Alias-Beziehungen (und damit flüchtende Objekte) entstehen
  - Folgerung: In einem CG sind nur die Kanten relevant, die
    - durch eine Schreiboperation entstehen
    - auf einem zyklenfreien Pfad von einem Einsprungpunkt zu einer solchen Kante liegen
- Manche Implementierungen von `equals` enthalten Schreiboperationen! :(
- Trotzdem deutliche Verbesserungen in Heap-Nutzung und Laufzeit in unseren Benchmarks :)

Kampf der Compilezeit (II)

- Beobachtungen
  - Verbindungsgraphen bestehen hauptsächlich aus *phantom nodes*
  - Geschwisterknoten repräsentieren häufig das selbe Objekt
- Idee: Knoten zusammenfassen, um Graphen zu vereinfachen

Kampf der Compilezeit (II)

- Beobachtungen
  - Verbindungsgraphen bestehen hauptsächlich aus *phantom nodes*
  - Geschwisterknoten repräsentieren häufig das selbe Objekt
- Idee: Knoten zusammenfassen, um Graphen zu vereinfachen
- Adaption von Steensgaards „Points-to analysis in almost linear time“ [Ste96]
  - Zwei Knoten werden zusammengeführt, wenn sie eingehende Kanten vom gleichen Knoten haben
  - Präzisionsverlust vermeiden: Nur Kompression von *phantom nodes* und innerhalb des gleichen Fluchtstatus

Kampf der Compilezeit (II)

- Beobachtungen
  - Verbindungsgraphen bestehen hauptsächlich aus *phantom nodes*
  - Geschwisterknoten repräsentieren häufig das selbe Objekt
- Idee: Knoten zusammenfassen, um Graphen zu vereinfachen
- Adaption von Steensgaards „Points-to analysis in almost linear time“ [Ste96]
  - Zwei Knoten werden zusammengeführt, wenn sie eingehende Kanten vom gleichen Knoten haben
  - Präzisionsverlust vermeiden: Nur Kompression von *phantom nodes* und innerhalb des gleichen Fluchtstatus
- Übersetzungszeit um eine Größenordnung besser :)

Überblick

Scope-Erweiterung

Wie können mehr Objekte durch den Compiler verwaltet werden?

- Aktueller Stand: Automatische Verwaltung methodenlokaler Objekte
- Idee: Erweiterung auf Objekte, die eine Ebene flüchten

Scope-Erweiterung

Wie können mehr Objekte durch den Compiler verwaltet werden?

- Aktueller Stand: Automatische Verwaltung methodenlokaler Objekte
- Idee: Erweiterung auf Objekte, die eine Ebene flüchten

Beispiel: Vorher

```
public class ScopeExtExample {
  public void run() {
    StringBuilder sb = buildString();
    System.out.println(sb.toString());
  }

  public StringBuilder buildString() {
    StringBuilder sb = new StringBuilder();
    sb.append("Ground control to Major Tom\n");
    sb.append("Ground control to Major Tom\n");
    sb.append("Commencing countdown, engines on\n");
    // ...

    return sb;
  }
}
```

Scope-Erweiterung

Wie können mehr Objekte durch den Compiler verwaltet werden?

- Aktueller Stand: Automatische Verwaltung methodenlokaler Objekte
- Idee: Erweiterung auf Objekte, die eine Ebene flüchten

Beispiel: Nachher

```
public class ScopeExtExample {
  public void run() {
    StringBuilder sb = buildString(new StringBuilder());
    System.out.println(sb.toString());
  }

  public StringBuilder buildString(StringBuilder sb) {

    sb.append("Ground control to Major Tom\n");
    sb.append("Ground control to Major Tom\n");
    sb.append("Commencing countdown, engines on\n");
    // ...

    return sb;
  }
}
```

Scope-Erweiterung (II)

Scope-Erweiterung kurz zusammengefasst

- Kopieren der Allokation in alle Aufrufer
- Übergeben einer Referenz beim Aufruf der Quell-Methode
- Ersetzen der Allokation durch Lesen eines Parameters
- Konstruktoraufruf unmodifiziert

Scope-Erweiterung (II)

Scope-Erweiterung kurz zusammengefasst

- Kopieren der Allokation in alle Aufrufer
- Übergeben einer Referenz beim Aufruf der Quell-Methode
- Ersetzen der Allokation durch Lesen eines Parameters
- Konstruktoraufruf unmodifiziert

Probleme und Beschränkungen von Scope-Erweiterung

- Anpassung von Methodensignaturen in virtuellen Aufrufen problematisch
- Allokation von Objekten aus wechselseitig exklusiven Kontrollflüssen
- Wachsende Codegröße bei vielen Aufrufern
- Zusatzaufwand durch Parameterübergabe

Scope-Erweiterung (II)

Scope-Erweiterung kurz zusammengefasst

- Kopieren der Allokation in alle Aufrufer
- Übergeben einer Referenz beim Aufruf der Quell-Methode
- Ersetzen der Allokation durch Lesen eines Parameters
- Konstruktoraufruf unmodifiziert

Probleme und Beschränkungen von Scope-Erweiterung

- Anpassung von Methodensignaturen in virtuellen Aufrufen problematisch
- Allokation von Objekten aus wechselseitig exklusiven Kontrollflüssen
- Wachsende Codegröße bei vielen Aufrufern
- Zusatzaufwand durch Parameterübergabe

$\Rightarrow$ **Ergebnisse durch Scope-Erweiterung nicht generell besser**

Tasklokale Heaps

- Stack-Allokation nicht notwendigerweise die beste Optimierung
  - KESO besitzt aktuell keine Stack-Überlauf-Prüfungen
  - Stack-Allokation kann zu schlechteren *worst-case*-Abschätzungen führen
  - Stack-Rahmen kann nicht zur Laufzeit vergrößert werden ($\Rightarrow$ kein `alloca(3)`)

Tasklokale Heaps

- Stack-Allokation nicht notwendigerweise die beste Optimierung
  - KESO besitzt aktuell keine Stack-Überlauf-Prüfungen
  - Stack-Allokation kann zu schlechteren *worst-case*-Abschätzungen führen
  - Stack-Rahmen kann nicht zur Laufzeit vergrößert werden ($\Rightarrow$ kein `alloca(3)`)
- Alternative: Separate Region für jeden Task
  - Explizites Sichern und Wiederherstellen des Kontexts
  - Präsize Überlaufprüfungen
  - Einfachere Vorhersag- und Messbarbarkeit
  - Rahmengröße dynamisch anpassbar

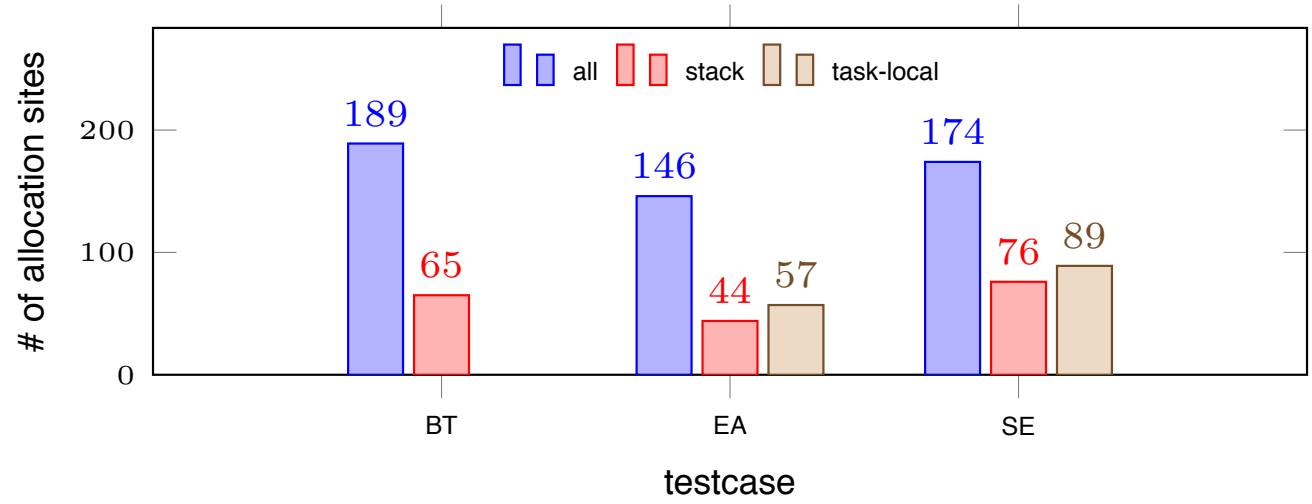
Überblick

Evaluation

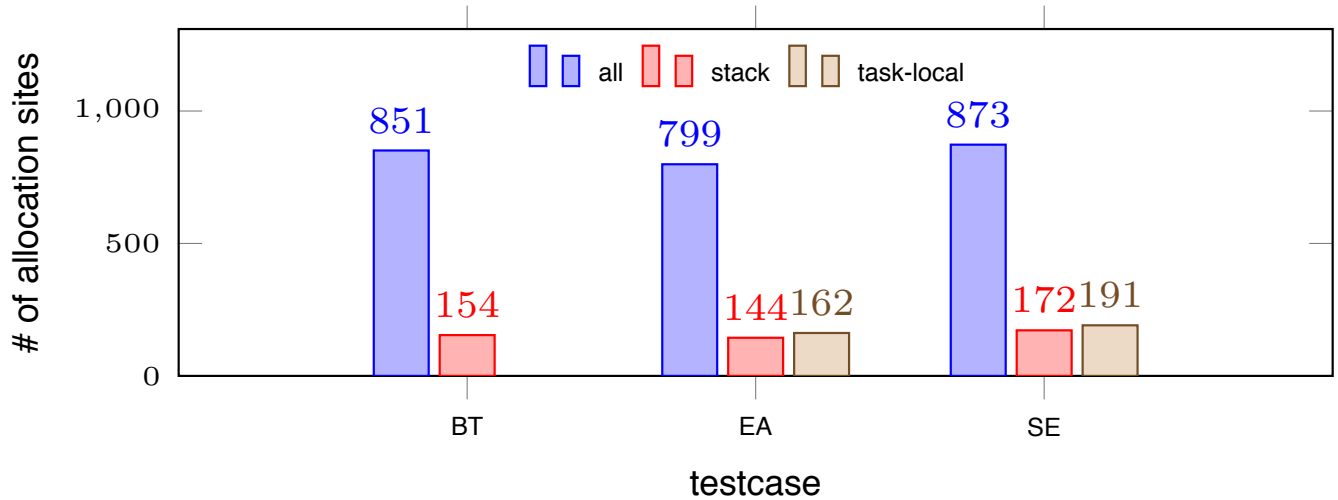
- CD_x Benchmark für Echtzeit-Java [KHP+09]
  - Statische Daten (Codegröße, Anzahl der optimierten Allokationen)
  - Laufzeitmessungen (Zeit, Heap-Nutzung, Instrumentierung der Speicherverwaltung)

Evaluation

- CD$_j$ *on-the-go*-Variante

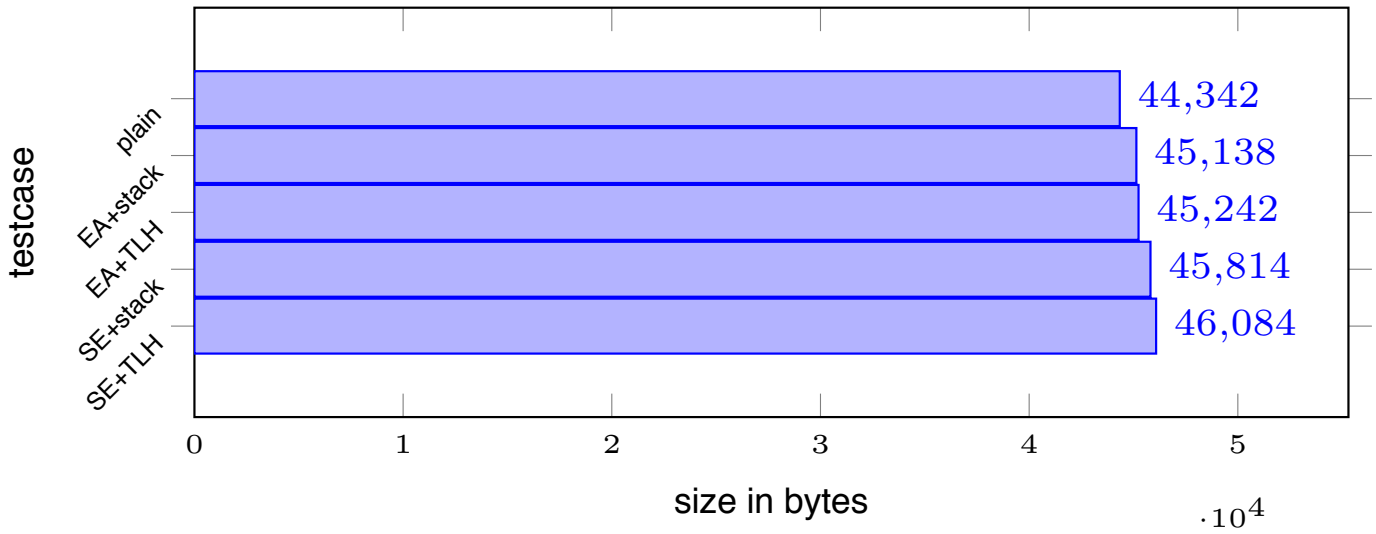


- CD$_j$ *simulated multidomain*-Variante

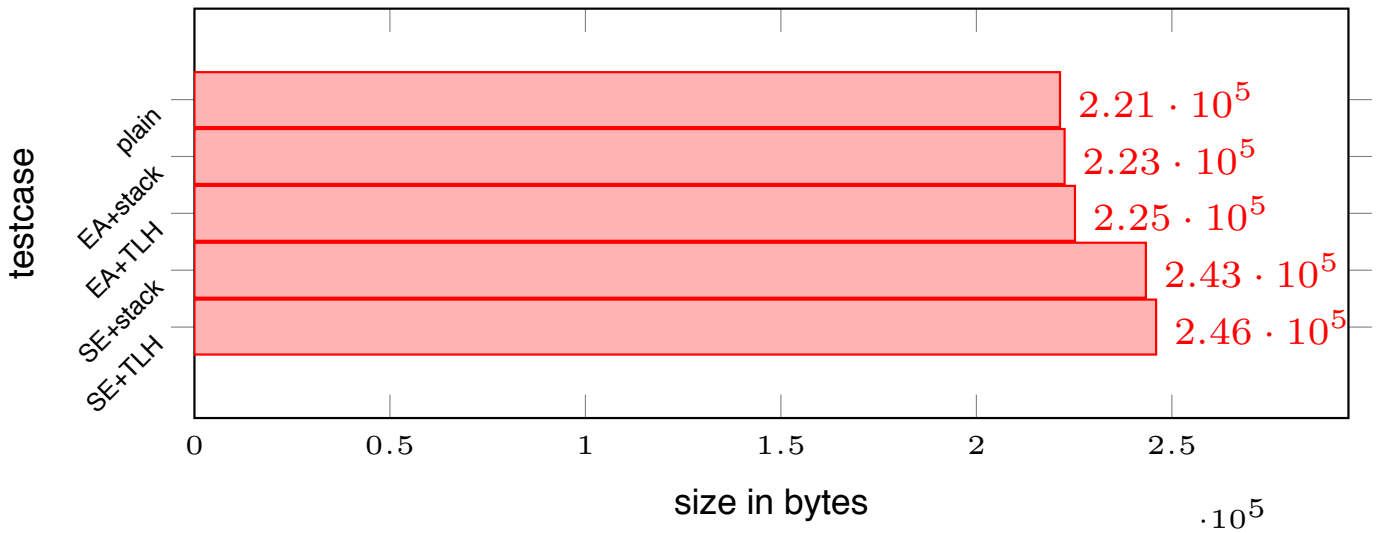


Evaluation

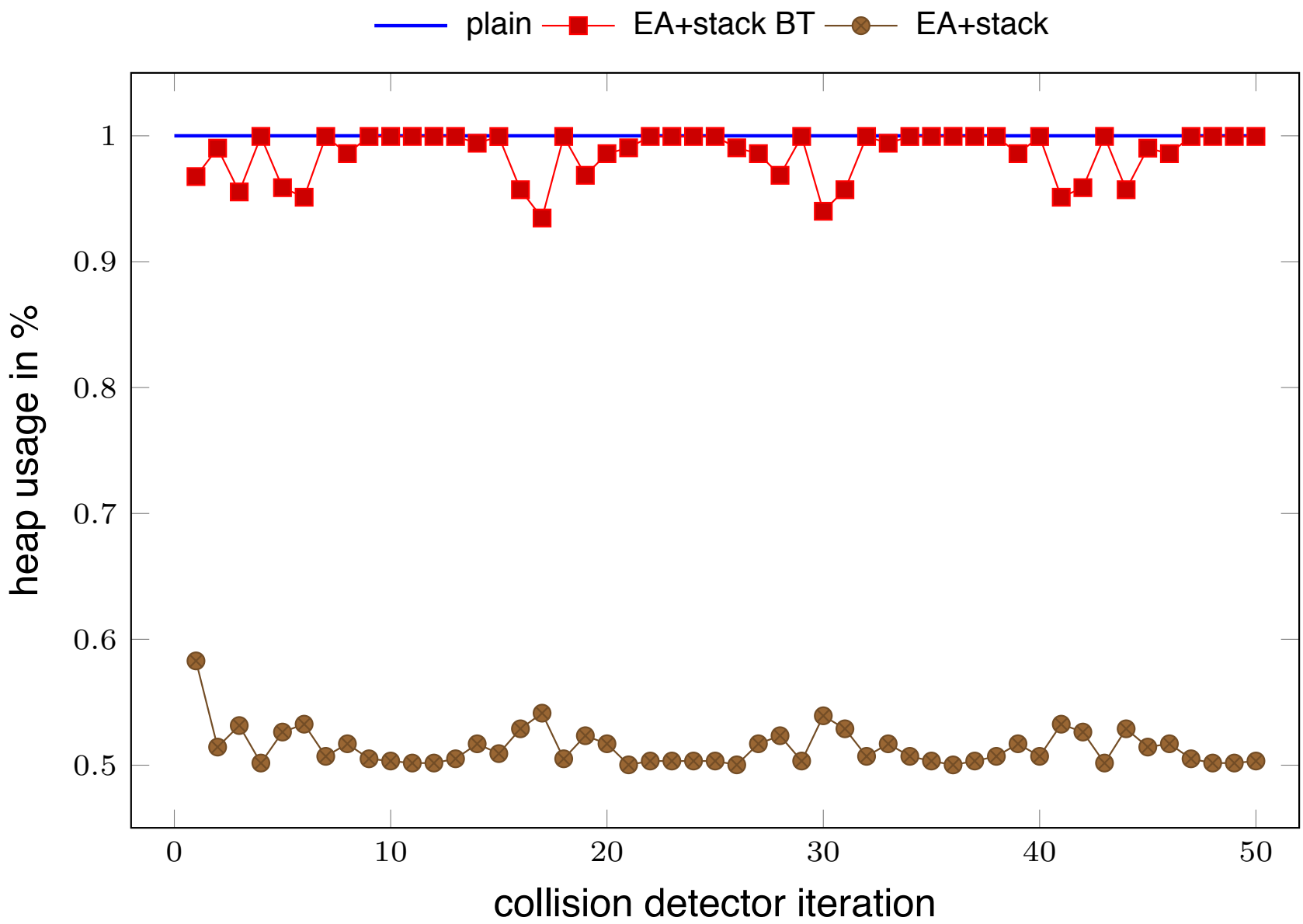
- CD$_j$ *on-the-go*-Variante



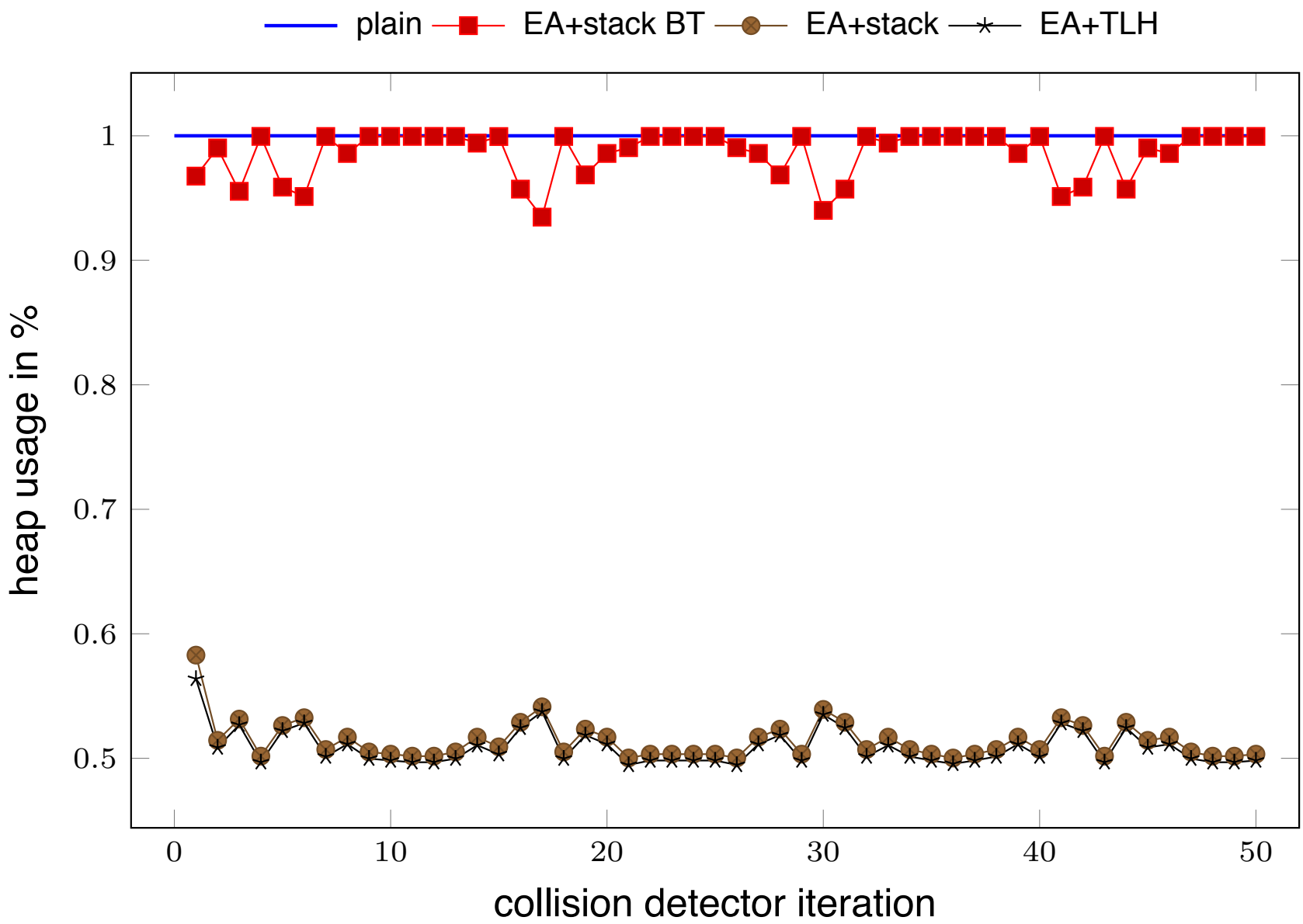
- CD$_j$ *simulated multidomain*-Variante



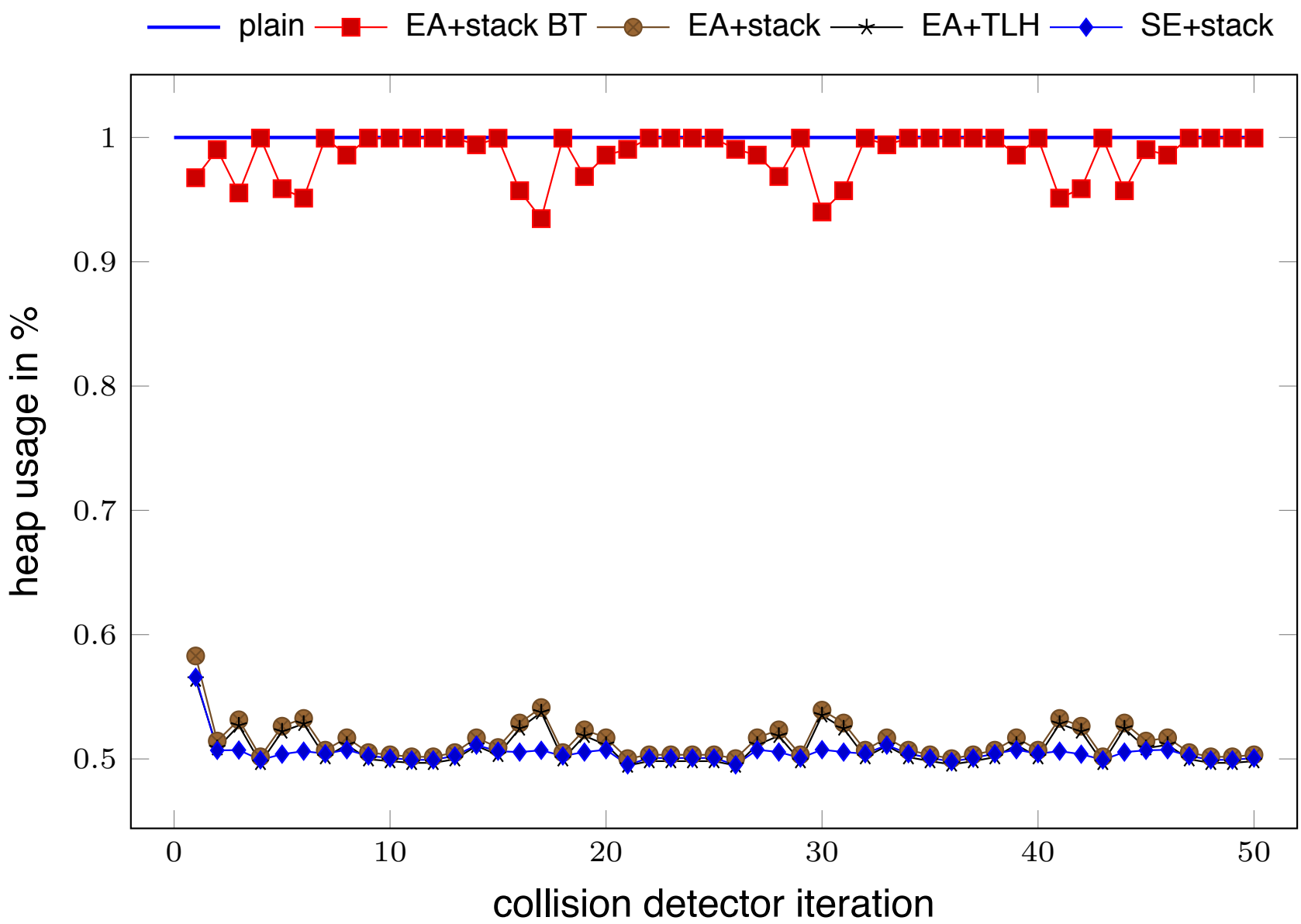
Evaluation


plain
EA+stack BT
EA+stack
heap usage in %
1
0.9
0.8
0.7
0.6
0.5
0
10
20
30
40
50
collision detector iteration

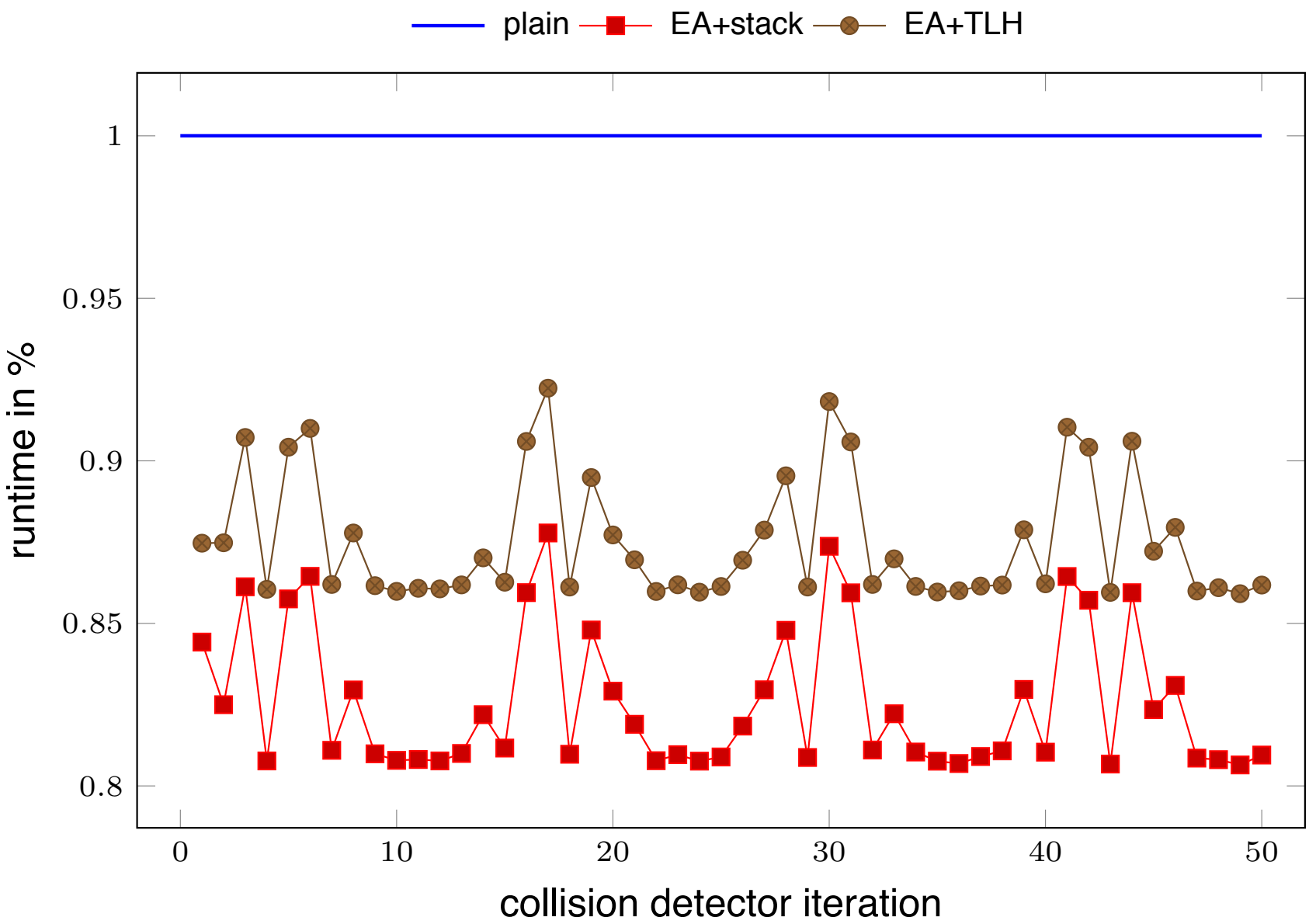
Evaluation


plain
EA+stack BT
EA+stack
EA+TLH
heap usage in %
1
0.9
0.8
0.7
0.6
0.5
0
10
20
30
40
50
collision detector iteration

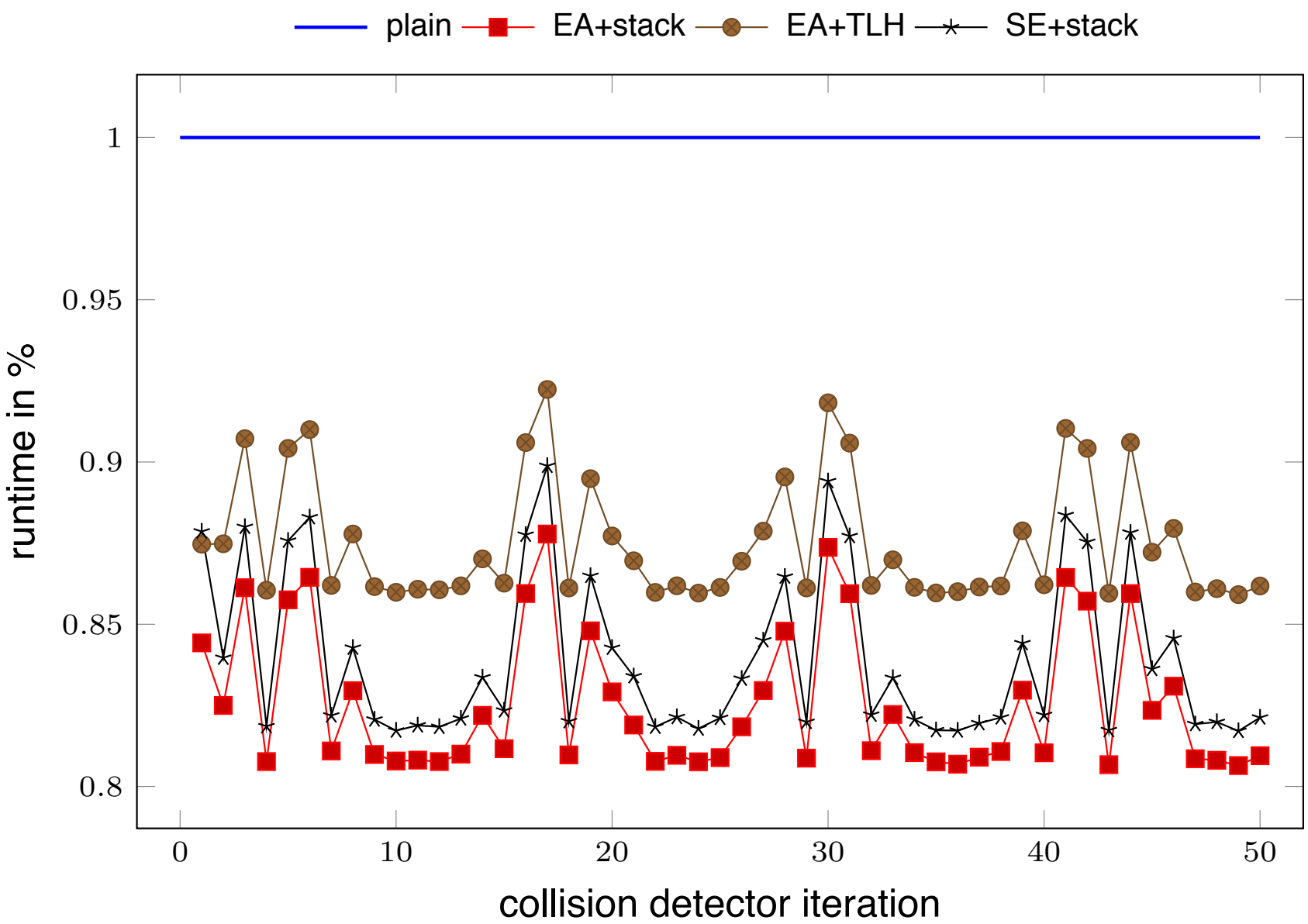
Evaluation


plain
EA+stack BT
EA+stack
EA+TLH
SE+stack
heap usage in %
1
0.9
0.8
0.7
0.6
0.5
0
10
20
30
40
50
collision detector iteration

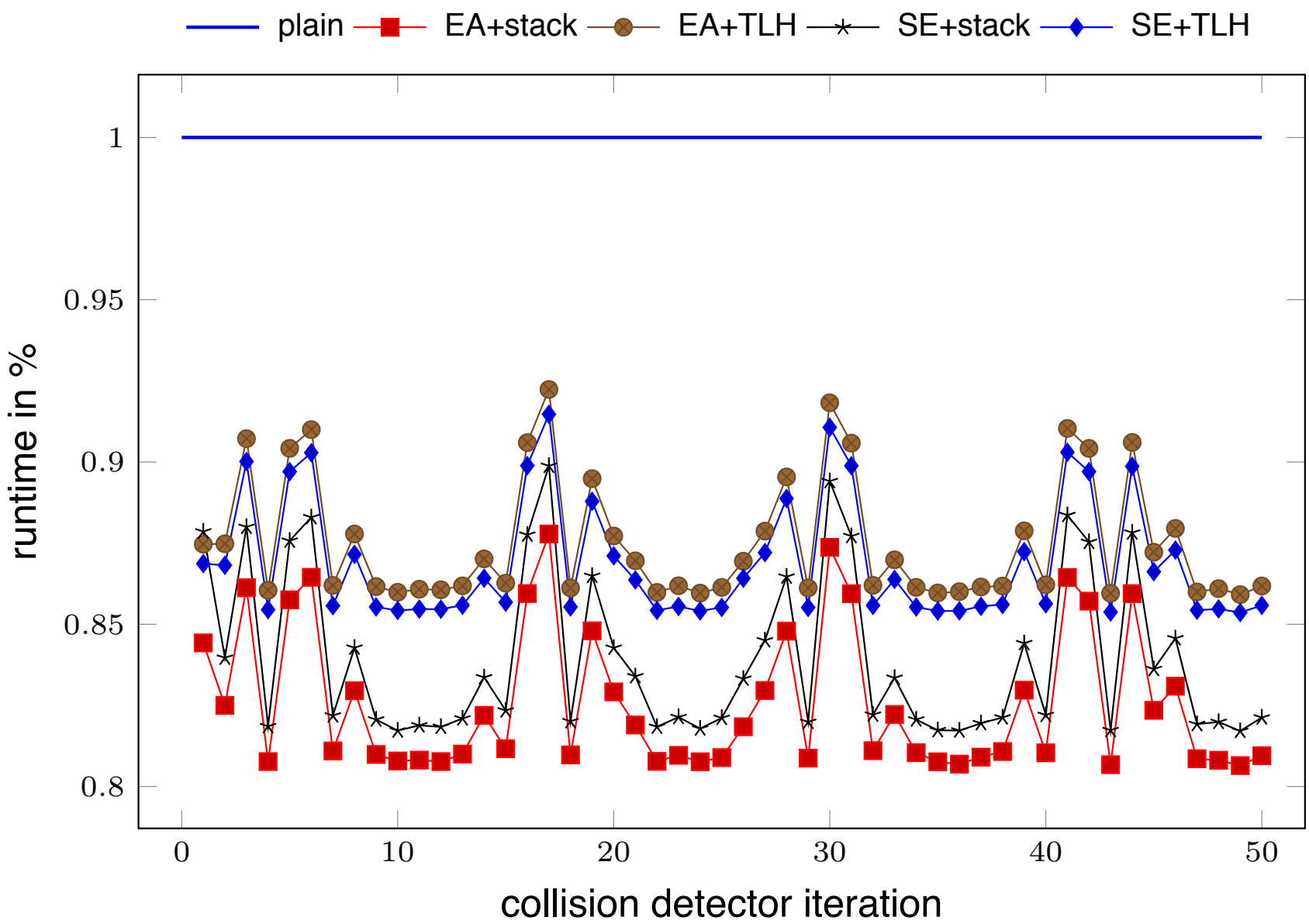
Evaluation


plain
EA+stack
EA+TLH
runtime in %
1
0.95
0.9
0.85
0.8
0
10
20
30
40
50
collision detector iteration

Evaluation


plain
EA+stack
EA+TLH
SE+stack
runtime in %
1
0.95
0.9
0.85
0.8
0
10
20
30
40
50
collision detector iteration

Evaluation


plain
EA+stack
EA+TLH
SE+stack
SE+TLH
runtime in %
1
0.95
0.9
0.85
0.8
0
10
20
30
40
50
collision detector iteration

Überblick

Zusammenfassung

- Fluchanalyse lohnt sich!
  - CD_j: Bis zu 43.7 % der Objekte automatisch verwaltet
  - CD_j: Weniger als 50 % Heap-Nutzung in gemessenen Abschnitten
  - Reduzierung der Fragmentierung (CD_j: von 21.2 % auf 9.4 %)
- Scope-Erweiterung nicht generell sinnvoll
  - Diverse Probleme, z. B. mit virtuellen Methodenaufrufen
  - Verbesserungen durch andere Optimierungen möglich

Ende

Fragen?

Danke für die Aufmerksamkeit

Referenzen

Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff.
Stack allocation and synchronization optimizations for Java using escape analysis.
ACM Trans. Program. Lang. Syst., 25(6):876–910, November 2003.

Tomas Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek, Ben Titzer, and Jan Vitek.
CD_x: A family of real-time java benchmarks.
In *JTRES '09: 7th*, page 41–50, 2009.

Bjarne Steensgaard.
Points-to analysis in almost linear time.
In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.