Partial Escape Analysis and Scalar Replacement

Мирошников Владислав
371

Escape analysis

- Техника анализа кода, не оптимизация сама по себе
- Определяет область достижимости для ссылки какого-то объекта
- Используется при определении того, нужно ли вообще создавать объект
- Проверяет, доступен ли объект вне метода/потока
- Bad case: объект выходит только в одной маловероятной ветви
- Используется в оптимизациях: Scalar Replacement, Stack Allocation and Lock Elision

Scalar Replacement

- Замена объекта на локальные переменные
- Как следствие – больше памяти в куче
- EA + SR – статический garbage collector, работает во время JIT компиляции

```
1  class Key {
2    int idx;
3    Object ref;
4    Key(int idx, Object ref) {
5      this.idx = idx;
6      this.ref = ref;
7    }
8    synchronized boolean equals(Key
         other) {
9      return idx == other.idx &&
10            ref == other.ref;
11   }
12 }
13 static CacheKey cacheKey;
14 static Object cacheValue;
15
16 Object getValue(int idx, Object ref) {
17   Key key = new Key(idx, ref);
18   if (key.equals(cacheKey)) {
19     return cacheValue;
20   } else {
21     return createValue(...);
22   }
23 }
```

Listing 1: Simple example.

```
1  Object getValue(int idx, Object ref) {
2    Key key = alloc Key;
3    key.idx = idx;
4    key.ref = ref;
5    Key tmp1 = cacheKey;
6    boolean tmp2;
7    synchronized (key) {
8      tmp2 = key.idx == tmp1.idx &&
9             key.ref == tmp1.ref;
10   }
11   if (tmp2) {
12     return cacheValue;
13   } else {
14     return createValue(...);
15   }
16 }
```

Listing 2: Example from Listing 1 after inlining.

```
1  Object getValue(int idx, Object ref) {
2    int idx1 = idx;
3    Object ref1 = ref;
4    Key tmp = cacheKey;
5    if (idx1 == tmp.idx && ref1 ==
         tmp.ref) {
6      return cacheValue;
7    } else {
8      return createValue(...);
9    }
10 }
```

Listing 3: Example from Listing 2 after Scalar Replacement and Lock Elision.

Partial Escape Analysis

- Идея: выполнить скаляризацию в ветвях, где объект не “убегает”
- Убедиться, что объект есть в куче в ветвях, где объект “убегает”
- Работает не с байт-кодом, а с IR
- Может быть применен, несколько раз, в любой момент во время компиляции

```
1  Object getValue(int idx, Object ref) {
2    Key key = new Key(idx, ref);
3    if (key.equals(cacheKey)) {
4      return cacheValue;
5    } else {
6      cacheKey = key;
7      cacheValue = createValue(...);
8      return cacheValue;
9    }
10 }
```

Before Partial Escape Analysis

```
1  Object getValue(int idx, Object ref) {
2    Key tmp = cacheKey;
3    if (idx == tmp.idx && ref ==
         tmp.ref) {
4      return cacheValue;
5    } else {
6      Key key = alloc Key;
7      key.idx = idx;
8      key.ref = ref;
9      cacheKey = key;
10     cacheValue = createValue(...);
11     return cacheValue;
12   }
13 }
```

After Partial Escape Analysis

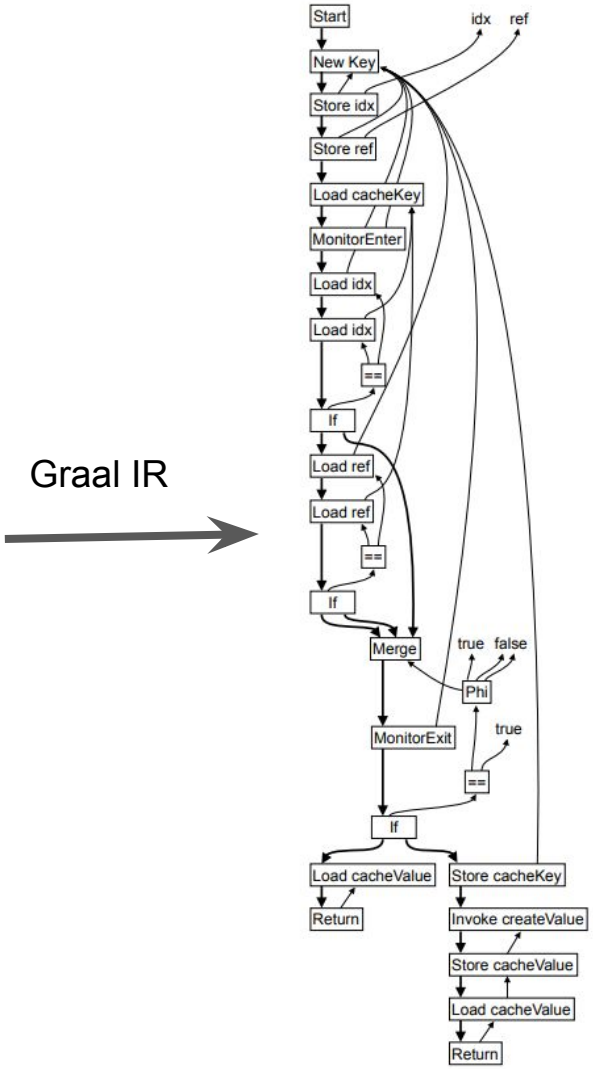
Graal

- JIT компилятор, работает поверх HotSpot VM
- Транслирует байт-код в Graal IR
- Graal IR – на основе SSA
- Оптимистичный компилятор, при неудаче делает деоптимизацию

Graal Partial Escape Analysis

```
1  Object getValue(int idx, Object ref) {
2    Key key = alloc Key;
3    key.idx = idx;
4    key.ref = ref;
5    Key tmp1 = cacheKey;
6    boolean tmp2;
7    synchronized (key) {
8      tmp2 = key.idx == tmp1.idx &&
9             key.ref == tmp1.ref;
10   }
11   if (tmp2) {
12     return cacheValue;
13   } else {
14     cacheKey = key;
15     cacheValue = createValue(...);
16     return cacheValue;
17   }
18 }
```

Graal IR



- Начинает анализ со Start узла
- Каждый узел обрабатывает, когда все его предшественники обработаны
- Итерация останавливается на поглотителях управления
- Если нет причин для реального создания объекта, он считается виртуальным объектом
- Если есть, то материализованным

Состояние аллокации

```
1  class Id extends Node {
2    Class<?> type;
3  }
4  class ObjectState {
5  }
6  class VirtualState extends ObjectState
       {
7    int lockCount;
8    Node[] fields;
9  }
10 class EscapedState extends ObjectState
       {
11   Node materializedValue;
12 }
13
14 class State {
15   Map<Id, ObjectState> states;
16   Map<Node, Id> aliases;
17 }
```

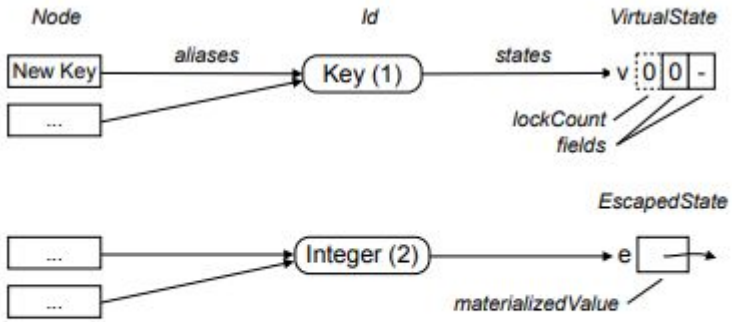
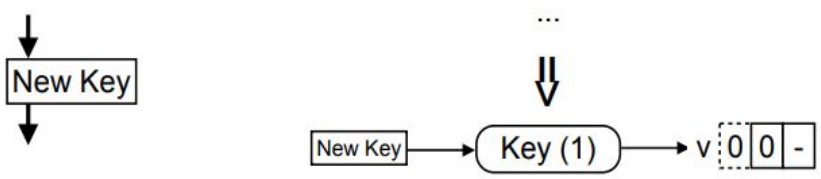


Figure 3: Visualization of the allocation state used in the rest of this paper.

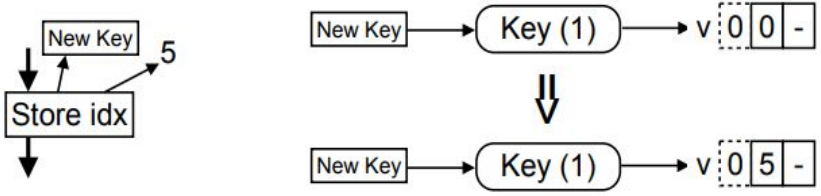
Влияние узлов на состояние аллокации

Три категории узлов:

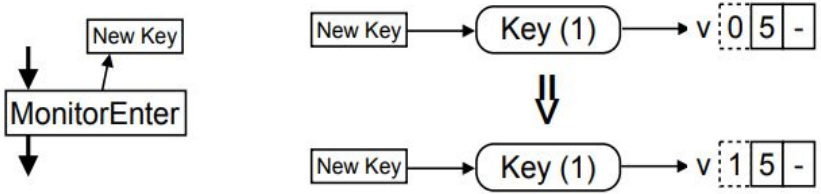
- Аллокации создают виртуальные объекты, поэтому они всегда изменяют состояние
- Если любой из входов узла является ключом в мапе алиасов, то узел нужно исследовать
- Узлы Merge и LoopBegin объединяют несколько состояний



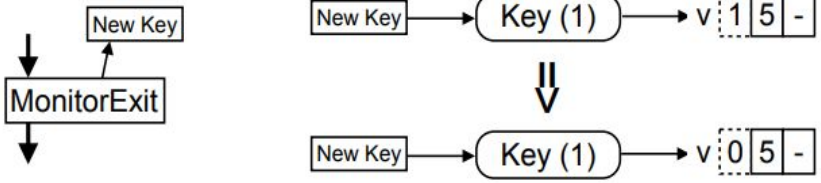
(a) New allocation.



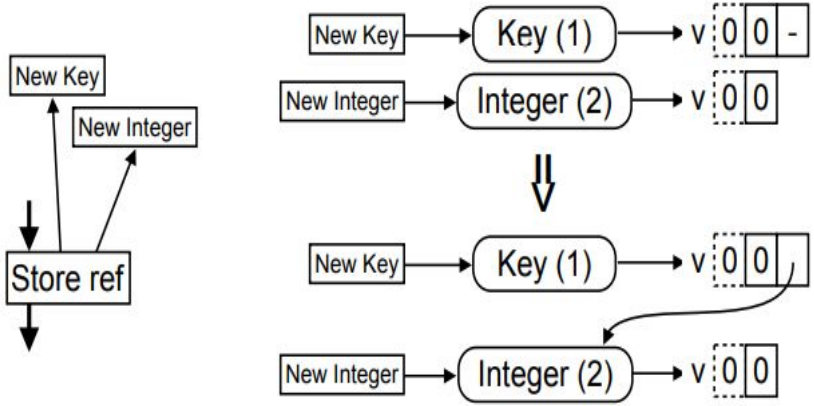
(b) Storing a value into a virtual object.



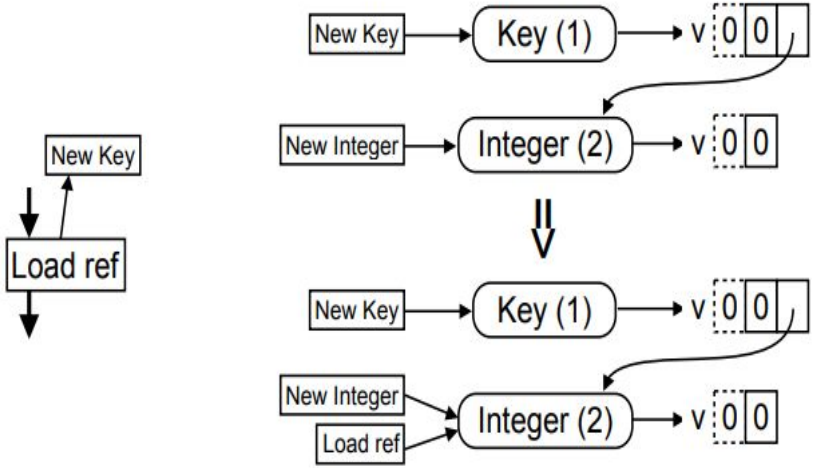
(c) Entering a synchronized region.



(d) Exiting a synchronized region.



(e) Storing a `virtual` object into another `virtual` object.



(f) Loading a `virtual` object from another `virtual` object.

- Входные данные, ссылающиеся на escaped объекты, заменяются на материализованные значения
- Любая операция, которая не обрабатывает явно, требует ссылки на реальный объект
- Любой виртуальный объект, на который ссылается такая операция, будет материализован

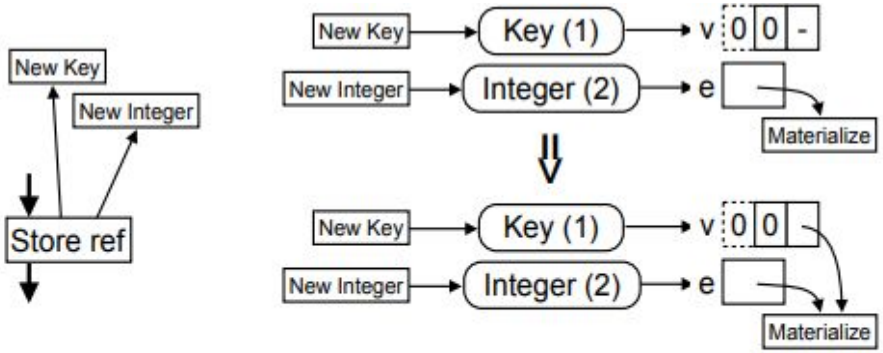
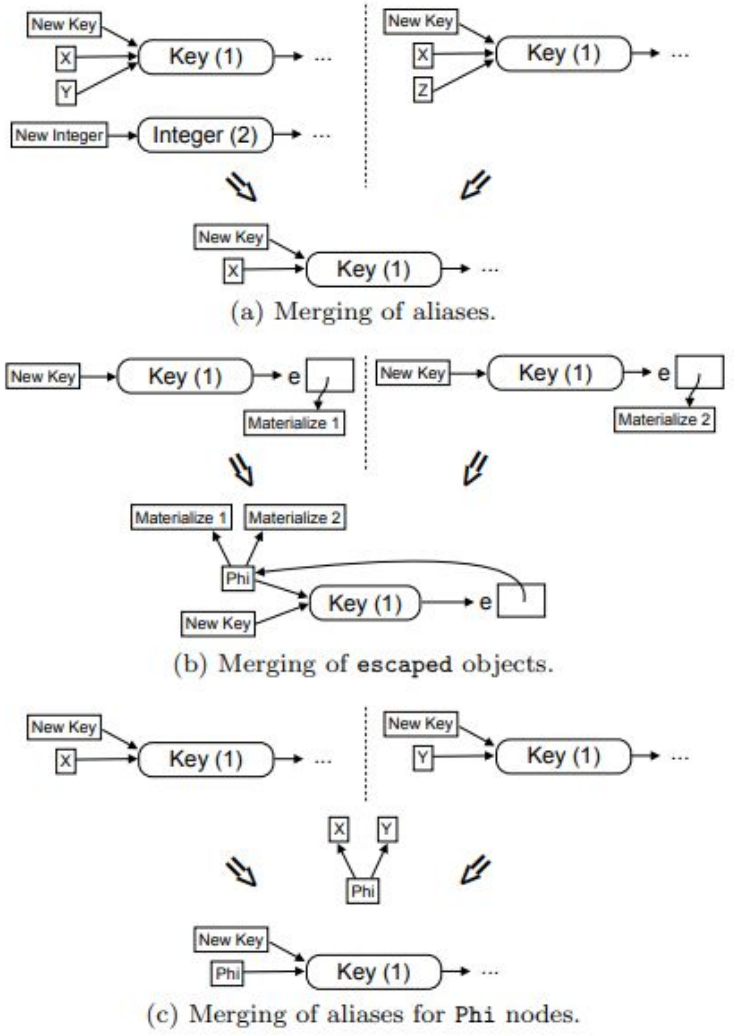


Figure 5: Store operation performed on an escaped object.

Merge узлы

- Делает MergeProcessor
- Просматривает различные варианты предшествующих узлов
- Процесс повторяет, пока не достигнуто стабильное состояние – нет новых материализаций



Loops узлы

- Обрабатывается итеративно
- На первой итерации обрабатывается со спекулятивным состоянием
- MergeProcessor используется для объединения спекулятивного и узлов LoopEnd
- Если результат merge совпадает со спекулятивным состоянием, то LoopExit, иначе заменяем спекулятивное и повторная итерация

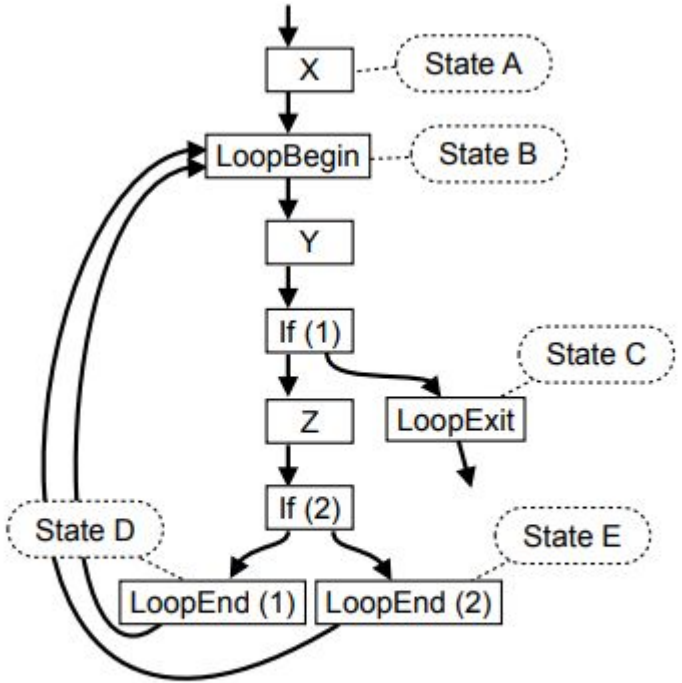


Figure 7: Example loop.

Что насчет деоптимизаций?

- Интерпретатор HotSpot не умеет работать с виртуальными объектами
- При деоптимизации все виртуальные объекты должны быть материализованы