Nested Trace Trees

Bailout Reduction & Optimization

Michael Bebenita, Andreas Gal, Michael Franz

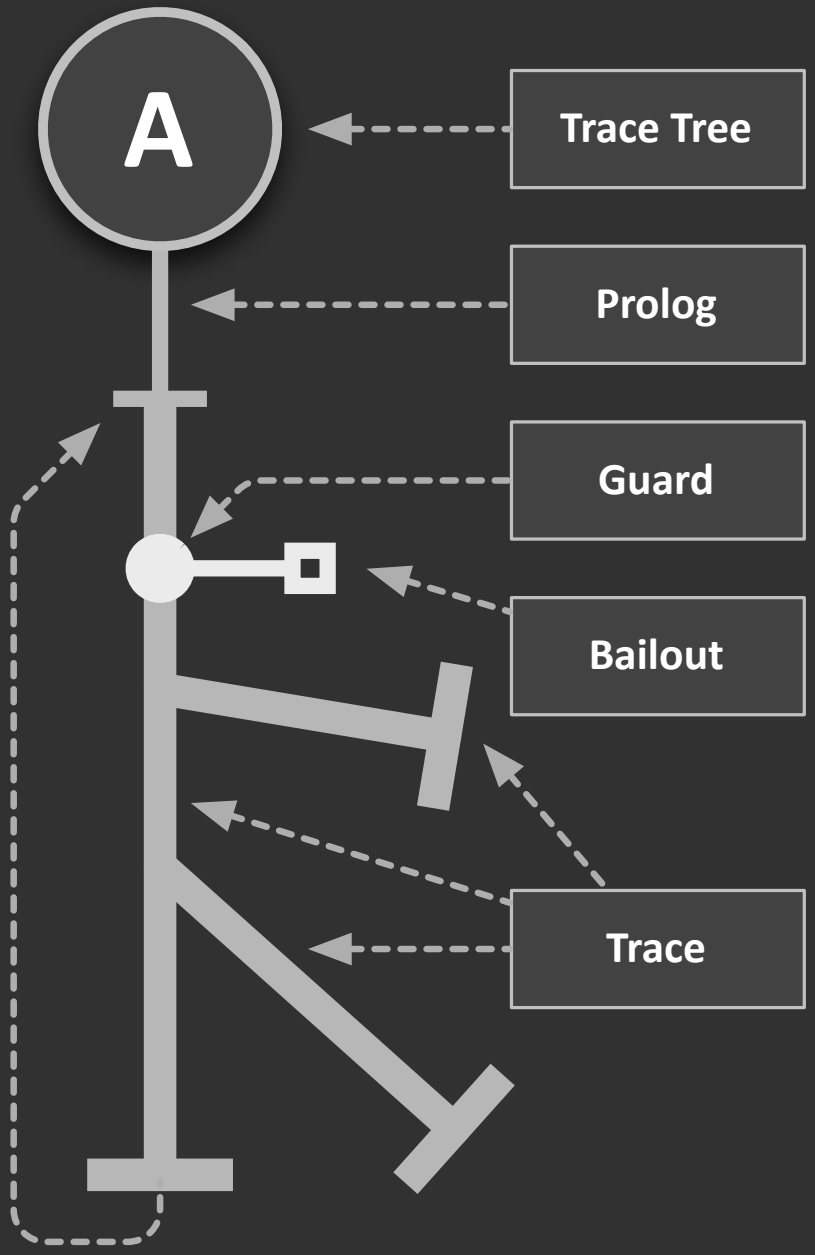
University of California, Irvine

Trace Based JIT Compilation

- Identifies and compiles relevant program regions using simple heuristics
- Uses a novel SSA based intermediate representation that is discovered and updated lazily during program execution

Anatomy of a Trace Tree

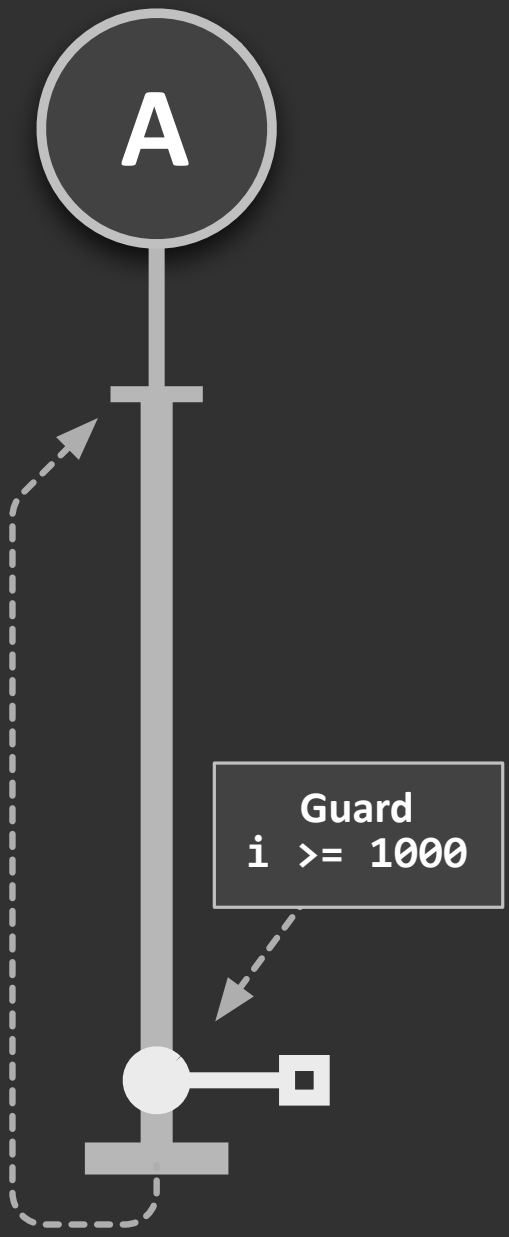
- Trace – A linear sequence of recorded instructions
- Trace Tree – A collection of traces through a hot loop
- Prolog – Loop invariant region of a Trace Tree
- Guard – Instruction that constrains control flow to the recorded trace
- Bailout – Code sequence that restores the interpreter state in case of a guard failure



Perfect Example

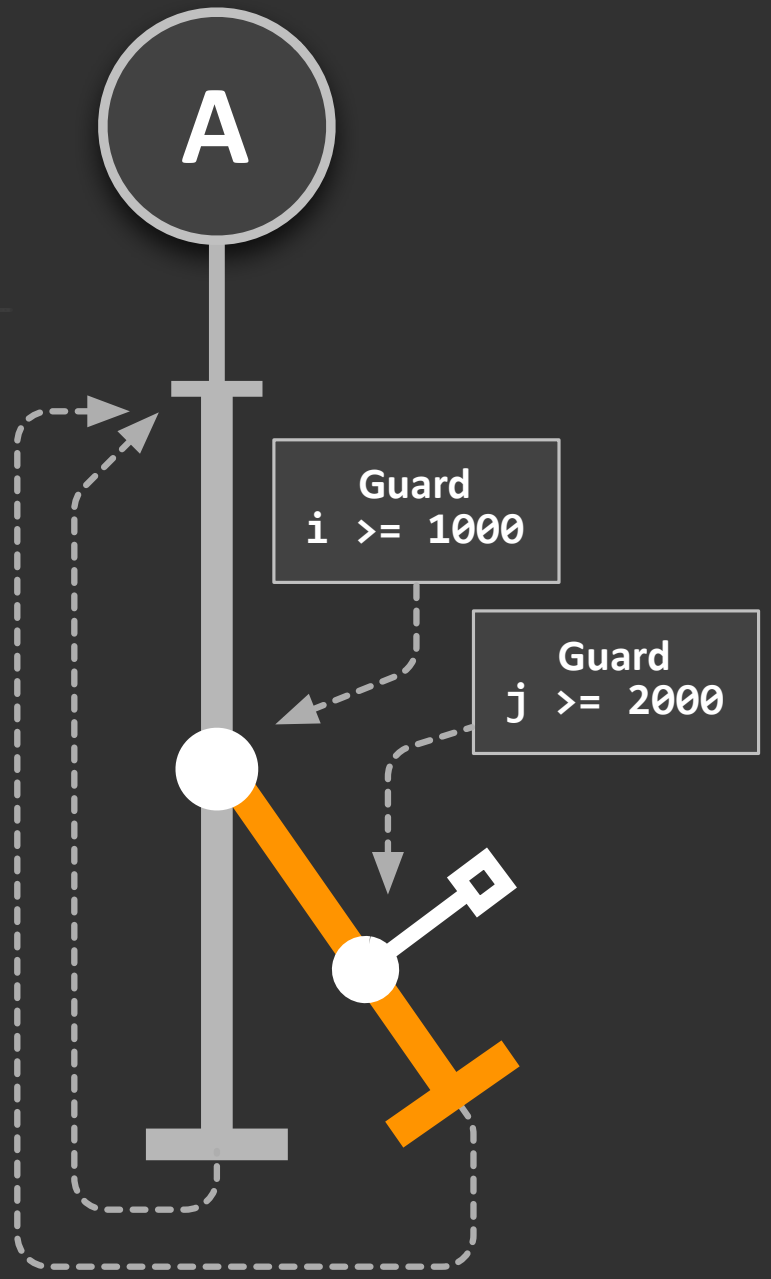
```
public static int bar() {
    int sum = 0;
    for (int i = 0; i < 1000; i++) {
        sum ++;
    }
    return sum;
}
```

```
0: CONTEXT    sum [write] [read]
1: CONTEXT    i   [write] [read]
HEAD:
2: CONSTANT   #1
3: ADD        (0) (2)
4: CONSTANT   #1
5: ADD        (1) (4)
6: CONSTANT   #1000
7: GUARD GE   (5) (6) [(3) (5) (5) (6)]
MOV           (0) (3)
MOV           (1) (5)
JUMP HEAD
PHIS          (0:3) (1:5)
```



Another Perfect Example

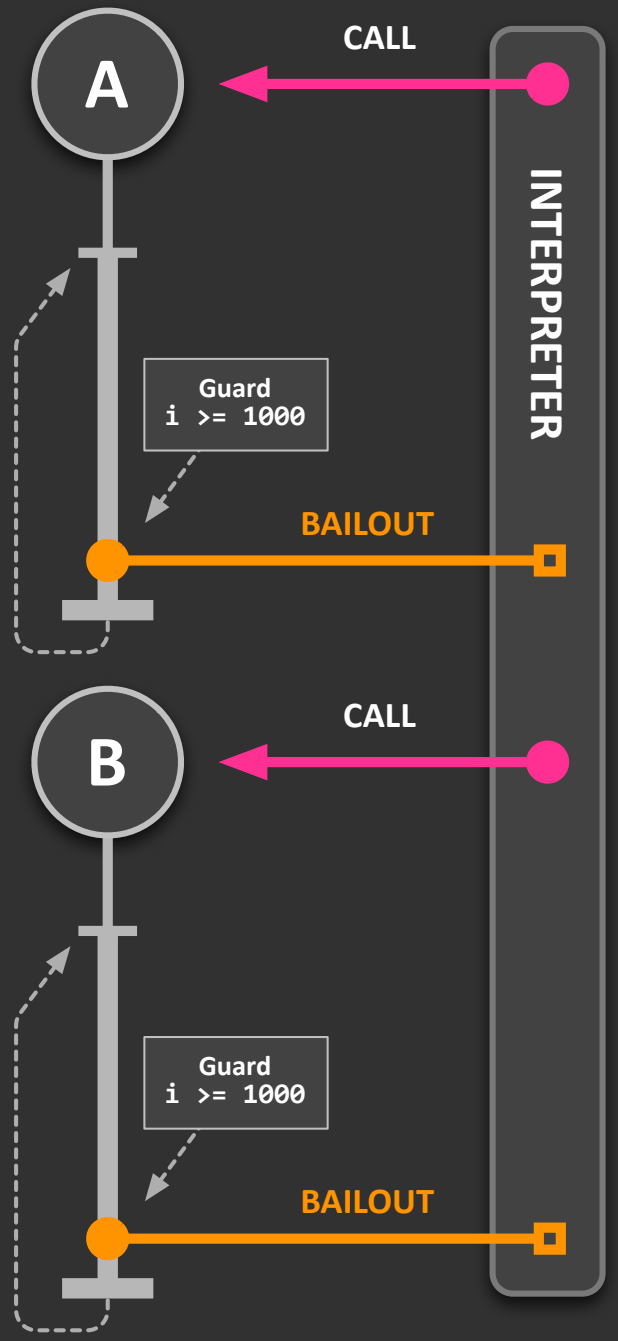
```
public static int bar() {
    int sum = 0;
    for (int j = 0; j < 2000; j++) {
        for (int i = 0; i < 1000; i++) {
    A       sum ++;
        }
    }
    return sum;
}
```



Hot Pairs

```
public static int bar() {
    int sum = 0;
    for (int j = 0; j < 2000; j++) {
        for (int i = 0; i < 1000; i++) {
    A       sum ++;
        }

        for (int i = 0; i < 1000; i++) {
    B       sum ++;
        }
    }
    return sum;
}
```

- Additional traces cannot be attached to either Trace Tree
- 2000 x 2 Expensive Calls and Bailouts
- Trace Trees compiled and optimized independently of each other



“Outerlining”

```
public static int bar() {
    int sum = 0;
    for (int i = 0; i < 1000; i++) {
        sum ++;
    }
    return sum;
}

public static int foo() {
    int sum = 0;
    for (int i = 0; i < 2000; j++) {
        sum += bar();
        sum += bar() + 1;
    }
}
```

A

- Inlining is easy, “Outerlining” is tougher
- Guards are needed for execution context

Recursion

```
public static int fac(int x) {
    if (x <= 1) {
        return 1;
    }
    return x * fac(x – 1) * fac(x – 2);
}
```

- Recursion is also problematic. Control flow does not loop back to a loop header
- Easy to identify, but impossible to trace through using the general Trace Tree approach

Performance Issues

- Slow Interpretation
  - Improve Interpretation Performance
    - Java in Java interpreter is roughly 5X slower than a hand assembled Interpreter
    - Solution: Outside the scope of our research for now. We're investigating two levels of Interpretation
  - Capture more control flow in Trace Trees
    - Trace Tree calls and bailouts are very expensive
    - Solution: Inlining, "outerlining", native method inlining, object allocation, object synchronization and finally **Nesting Trace Trees**
- Compilation Performance & Code Quality
  - Guard elimination and bailout stub compression
  - Trace Tree pruning or Tree Folding (Andreas' Presentation)

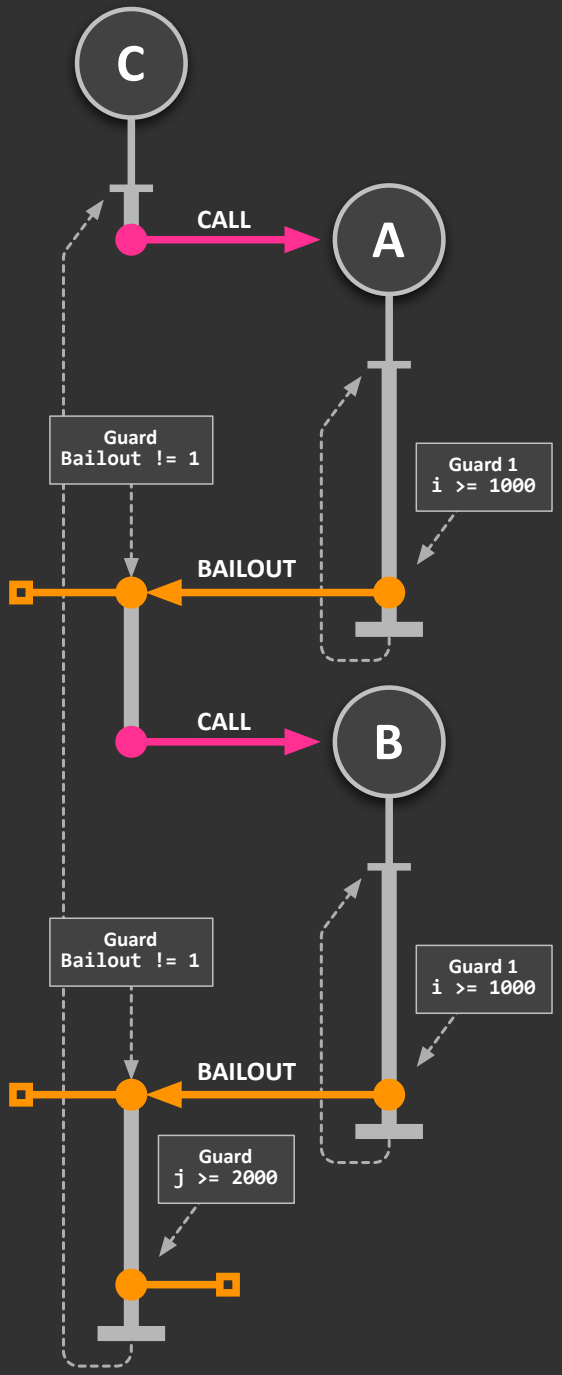
Nested Trace Trees

- Capture significantly more complicated control flow and therefore expose more optimizations
- Provide an elegant solution to each of the previously identified problem areas
- Preserve the SSA form of Trace Trees and add little overhead to the compiler pipeline
- Each Trace Tree can be compiled individually, or can be specialized based on nesting location
- Optimization results can be propagated up and down the nesting hierarchy
- Eliminate most switches between the Interpreter and compiled Trace Trees
- Improve register allocation. Hottest regions are in leaf nested trees, and are allocated separately

Revisiting Hot Pairs

```
public static int bar() {
    int sum = 0;
C   for (int j = 0; j < 2000; j++) {
        for (int i = 0; i < 1000; i++) {
A           sum ++;
        }

        for (int i = 0; i < 1000; i++) {
B           sum ++;
        }
    }
    return sum;
}
```

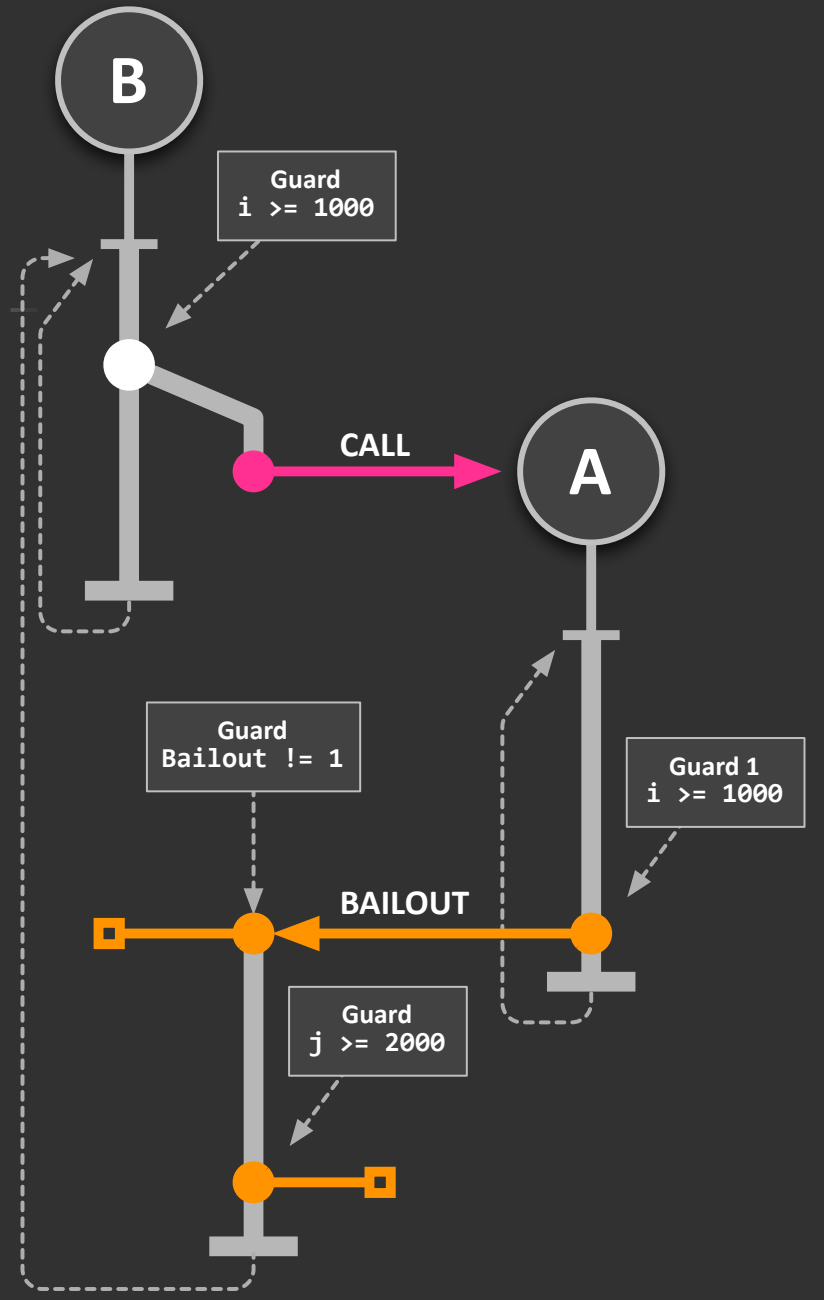
- A possible nesting of tree A and B into C
- Guards are inserted into the nesting tree to ensure that the nested tree exits at a captured exit



Revisiting Hot Pairs

```
public static int bar() {
    int sum = 0;
    for (int j = 0; j < 2000; j++) {
        for (int i = 0; i < 1000; i++) {
    A       sum ++;
        }

        for (int i = 0; i < 1000; i++) {
    B       sum ++;
        }
    }
    return sum;
}
```

- A much more likely nesting would be tree A into B
- Tree A is compiled before B. The trace recorded after a bailout in B nests the previously compiled tree A into B



Revisiting “Outerlining”

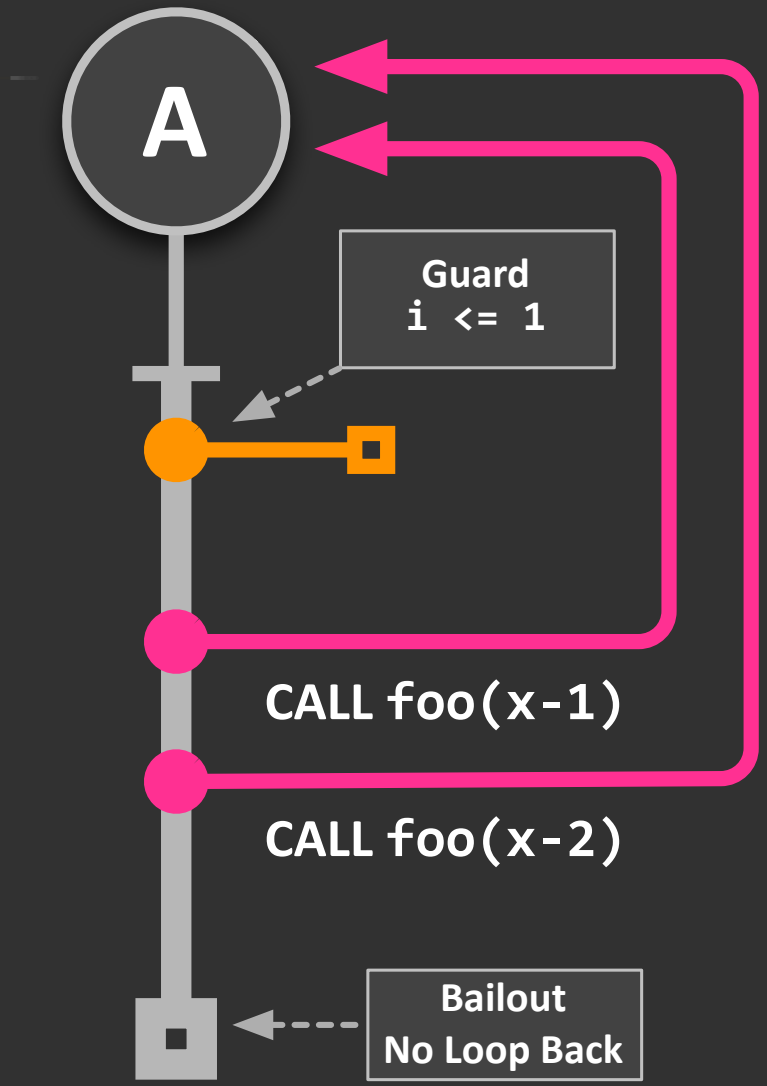
```
public static int bar() {
    int sum = 0;
    for (int i = 0; i < 1000; i++) {
A       sum ++;
    }
    return sum;
}

public static int foo() {
    int sum = 0;
B   for (int i = 0; i < 2000; j++) {
        sum += bar();
        sum += bar() + 1;
    }
}
```

- Tree A is compiled first. When tracing through tree B, we nest A twice. The resulting tree structure is identical to previous example
- “Outerlining” is transformed into Inlining + Tree Nesting

Revisiting Recursion

```
public static int fac(int x) {
    if (x <= 1) {
        return 1;
    }
    return x * fac(x - 1) * fac(x - 2);
}
```

- Detect recursive call by inspecting the method stack
- Record non cyclic traces through the recursive method
- Nest tree A into itself



Implementation

- Tree calling mechanism is the same in both the Interpreter and Nesting Trees
- Tree arguments are passed through the Marshal (frame). Return values are returned by referencing the previous Marshal
- The flow of SSA values is analyzed by comparing the execution states between nested tree entry and exits

Questions & Future Work

- Nested trace trees can be compiled independently of their nesting sites. In some cases, the expansion of a nested tree invalidates the nesting tree. We are looking at ways we can “generalize” the call site such that it will not become invalidated later
- Optimizations can be propagated upwards and downwards the nesting hierarchy. For example, invariant code in a nested tree may end up being invariant in the nesting tree as well. Also, information about invariant code in the nesting tree can be used to optimize the nested tree further
- Investigate the benefits of growing trees inside of trees. May cause code duplication but for certain cases this strong welding of trees may provide optimal optimization results