Tail Recursion Optimization in JVM

Akishige Yamamoto (Mathematical Systems Inc)
Taiichi Yuasa (Kyoto University)

What is tail recursion?

A method call is tail-recursive if the return value of the called method is returned as the value of the caller method.

```
static int foo(...) {
    …
    n = bar(…);
    …
    return bar(…);
}
```

Not tail recursive

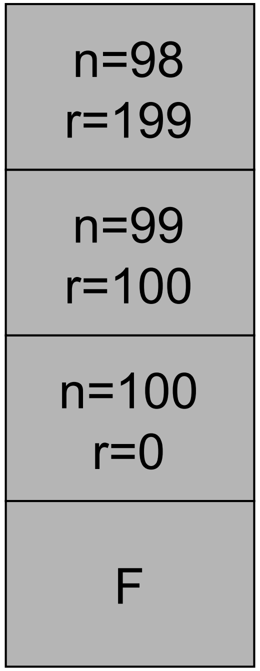
Tail recursive

Functional programming & Tail recursion

- In functional programming, loops are realized with tail recursion.

$$S(n)=T(n,0)$$
$$T(0,r)=r$$
$$T(n,r)=T(n-1,r+n) \quad (n>0)$$

```
static int sum(int n, int r) {
    if (n == 0) return r;
    else return sum(n-1, r+n);
}
```


n=98
r=199

n=99
r=100

n=100
r=0

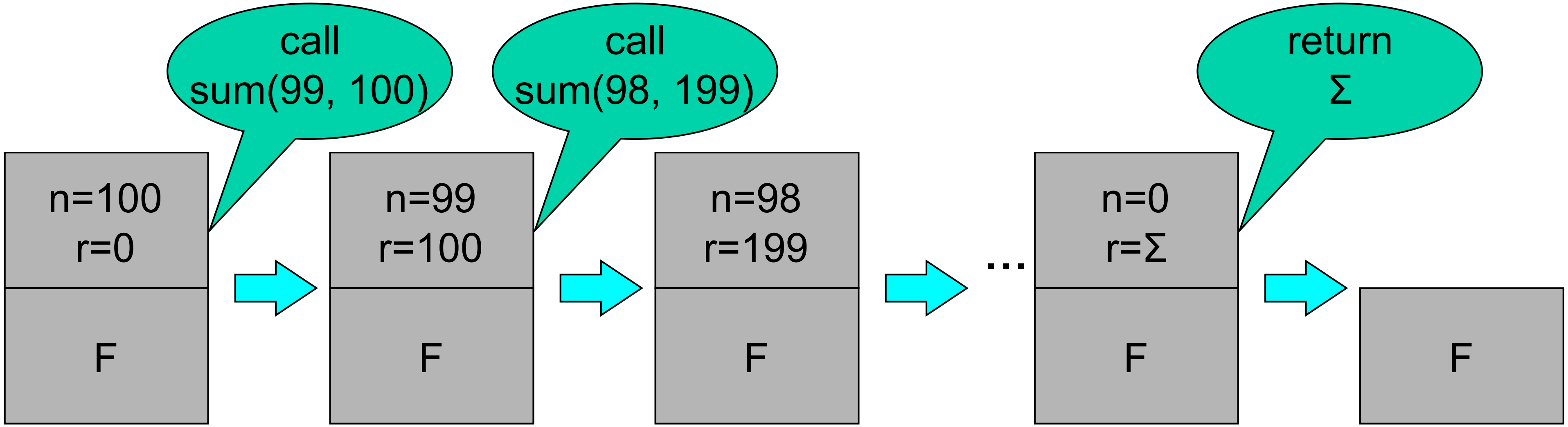
F

- Such programs cause stack overflow on Java VM because each call pushes a frame on the stack.

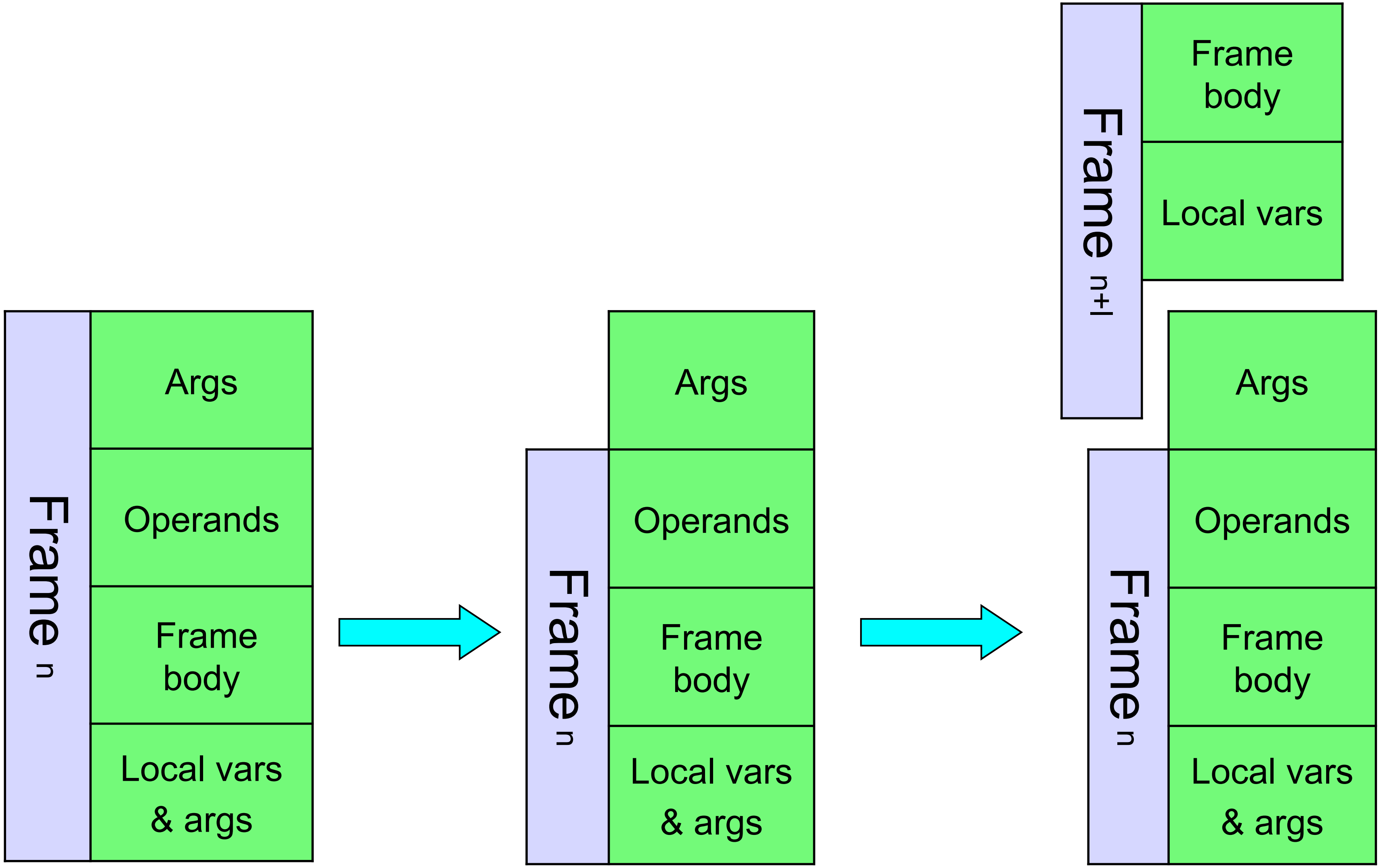
Functional programming & Tail recursion (cont.)

- Trampoline methods enable the realization of general tail-recursion optimization in JVM; however, its speed efficiency is poor.

⇒Improvement of JVM itself is necessary for efficient tail-recursion optimization



Frame operation for a method call

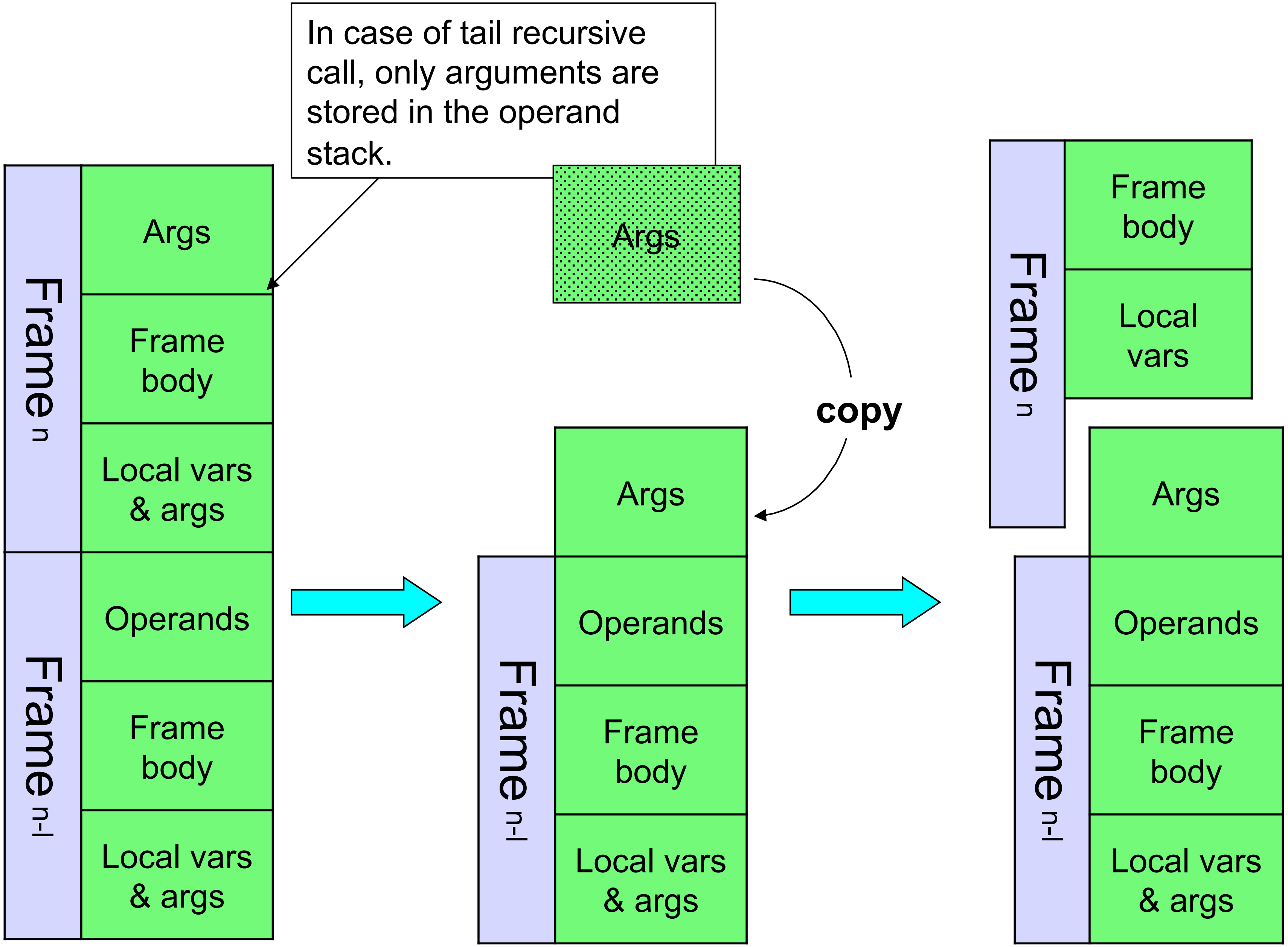


What is tail-recursion optimization?

- For a tail-recursive call, there are no codes to execute in the caller method after the called method returns.
- Therefore, the frame used under current execution can be discarded and its area can be reused as a new frame.
- This method is called tail-recursion optimization.

```
static int sum(int n, int r) {
    if (n == 0) return r;
    else return sum(n-1, r+n);
}
```

Frame operation in calling a method



In case of tail recursive call, only arguments are stored in the operand stack.

Implementation of tail-recursion optimization in JVM

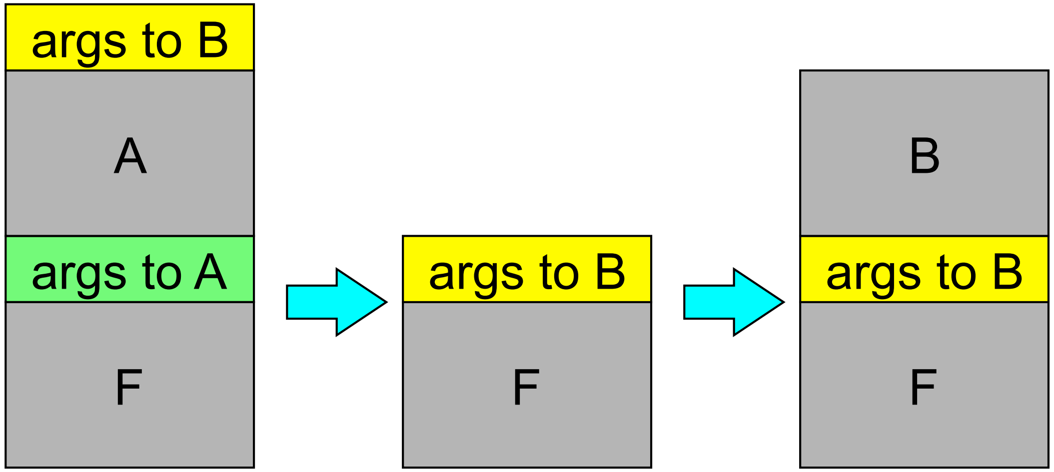
- The improved JVM automatically detects tail recursive calls while loading class files, and replaces them with tail recursion instructions
- The improved JVM is implemented so that tail recursive instructions are handled appropriately.

Extended tail recursion instructions

Standard method-call instructions	Extended tail-recursion instructions	Extended self-tail-recursion instructions
invokeStatic	tailInvokeStatic	selfTailInvokeStatic
invokeVirtual	tailInvokeVirtual	selfTailInvokeVirtual
invokeInterface	tailInvokeInterface	selfTailInvokeInterface
invokeSpecial	tailInvokeSpecial	selfTailInvokeSpecial

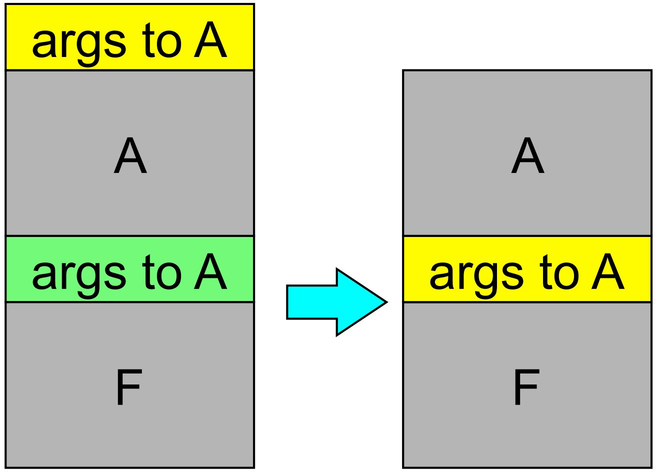
Operation of tail-recursion instructions

- When A calls B tail-recursively, ...
- Discard the current frame (for A), and overwrite the region with a new frame for the called method B
- Adjust the contents of the new frame so that B directly returns to the caller of A



Operation of self tail-recursion instructions

- Self tail recursion: a method calls itself tail recursively.
- The basic operation is the same as that of the tail-recursion instruction.
- However, more efficient implementation is expected owing to the similarity between the structure and size of the current frame and the frame for the called method



Automatic conversion into tail-recursion instructions

The improved JVM automatically detects tail recursion during class loading and replaces standard JVM instructions with corresponding tail-recursion instructions.

1. A method call instruction is followed by any number of instructions that change only pc (such as nop and goto), and then followed by a return instruction.
2. The type of the return value of the called method agrees with that of the return instructions (ireturn, lreturn, freturn, dreturn, areturn, return).
3. There is no exception handler between the method-call instruction and the return instruction.

When these conditions are satisfied, the method call instruction can be replaced with a corresponding tail-recursion instruction.

Detection of tail recursion from bytecode

Typical patterns when replacement with the tail-recursion instruction is possible.

1.
   ```
   Invokestatic foo // foo returns a reference
   areturn
   ```
2.
   ```
   Invokevirtual bar // bar returns void
   nop
   return
   ```
3.
   ```
   Invokeinterface baz 0 // baz returns double
   goto label
   …
   label:
   dreturn
   ```

Effects of tail-recursion optimization

Difference in processing time

$$\mathrm{Tnormal}(n,m)-\mathrm{Ttail}(n,m)$$
$$= n(\mathrm{Cret} + \mathrm{Ccall} - \mathrm{C'call} - m\mathrm{Ccopy}) - \mathrm{Cret}$$

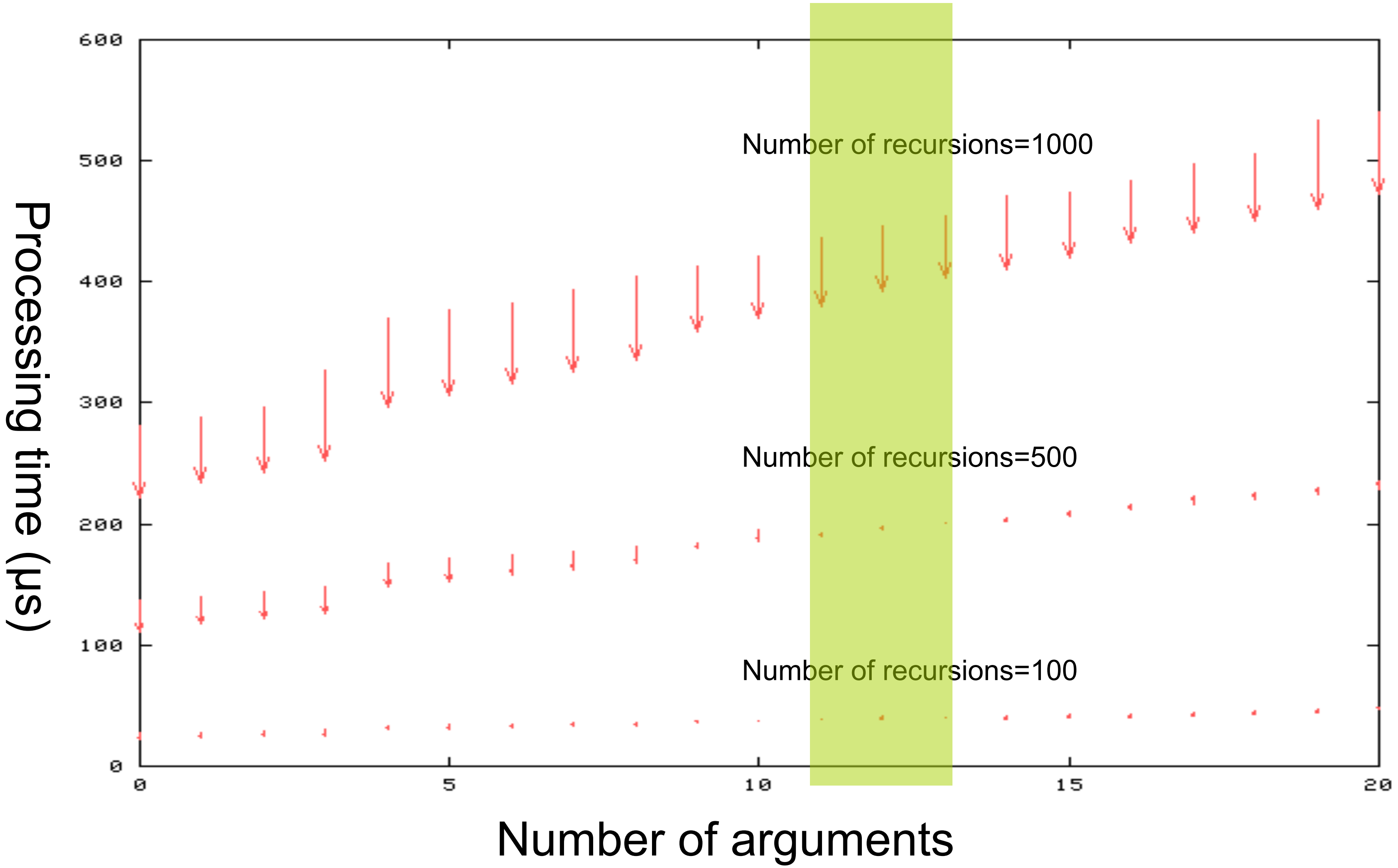
where

n: depth of recursion

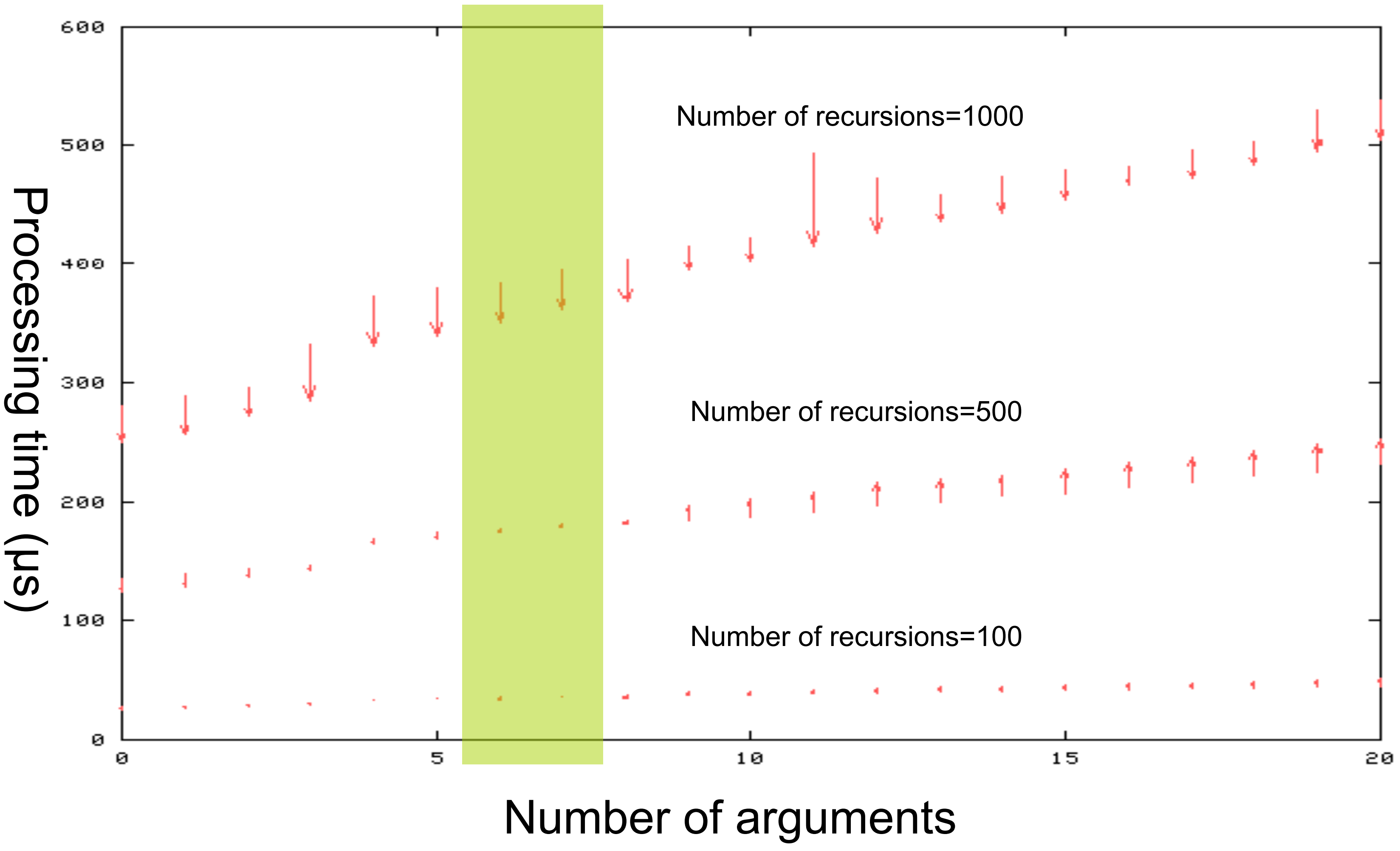
m: number of arguments

1. The effect of tail-recursion optimization on processing time is approximately proportional to the depth n. With increasing recursion depth, the effect (gain or loss in speed) is enhanced.
2. When the increase in speed as a result of reusing frames (Ccall-C'call) and the reduction in the number of returns surpasses the overhead (mCopy) due to memory transfer, execution efficiency improves; however, when the overhead surpasses the increase in speed, execution efficiency deteriorates.

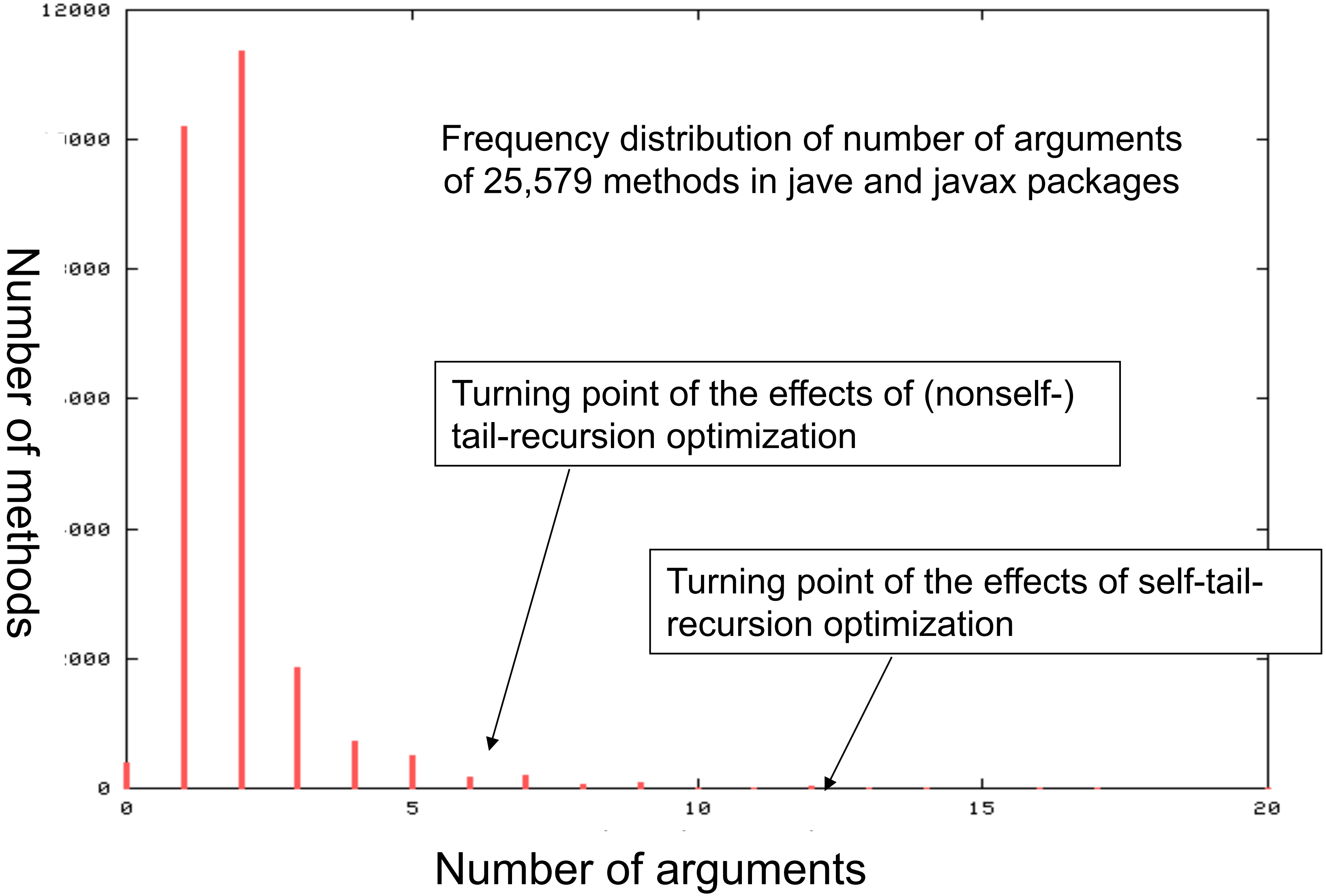
Effects of self-tail-recursion optimization (Numbers of recursions = 100, 500, 1000



Effects of (non-self)tail-recursion optimization
(Numbers of recursions = 100, 500, 1000)



Frequency distribution of number of arguments



Frequency distribution of number of arguments of 25,579 methods in jave and javax packages

Turning point of the effects of (nonself-) tail-recursion optimization

Turning point of the effects of self-tail-recursion optimization

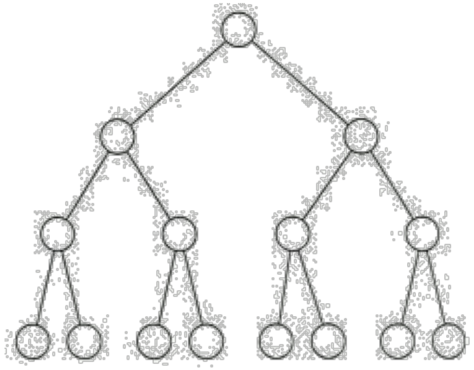
Effects of optimization

- The effects of self-tail-recursion optimization is significant.
- The effect of general (i.e., non-self) tail-recursion optimization is limited..

Side effects

- Programs that caused stack overflow might be executable with optimization.

Example in which optimization is effective (1)



- Trace a binary tree and double all the node values.

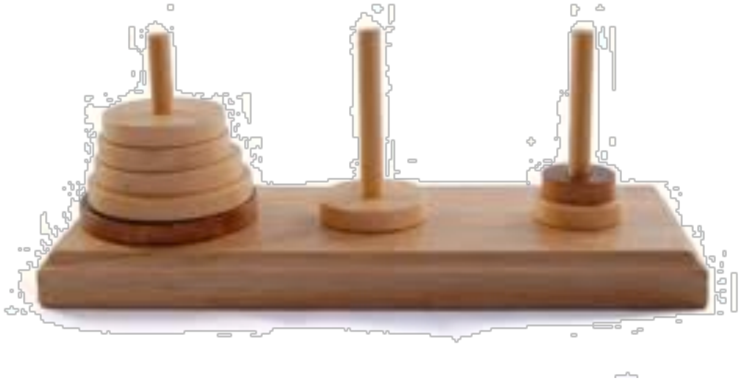
```
public static void twice(Tree tree) {
  if (tree != null) {
      tree.val *= 2;
      twice(tree.left);
      twice(tree.right);
  }
}
```

When applied to a binary tree with 2ˆ13-1=8191 nodes	
JVM	Time required (ms)
Conventional JVM	23
Improved JVM	20

Example in which optimization is effective (2)

- Tower of Hanoi

```
public static void solve(int n, int src, int dst) {
   if (n>1)
     solve(n-1, src, 3-src-dst);
   // Move disk n from src to dst
   if (n>1)
      solve(n-1, 3-src-dst, dst);
}
```



Execution time required when the number of disks is 23	
JVM	Time required (ms)
Conventional JVM	17.257
Improved JVM	16.312

Example of codes that are unexecutable on conventional JVM

```
public static boolean isEven(int n) {
    if (n==0) return true;
    else return isOdd(n-1);
}
public static boolean isOdd(int n) {
    if (n==0) return false;
    else return isEven(n-1);
}
```

- Stack overflow occurs on conventional JVM.
- Execution is completed regardless of *n*, on improved JVM.