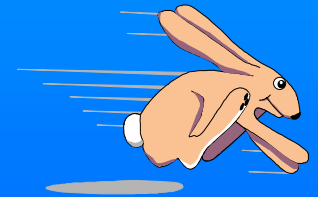


A Study of Devirtualization Techniques for a Java Just-In-Time Compiler



`<http://www.trl.ibm.co.jp/projects/jit/index_e.htm>`

Kazuaki Ishizaki
ishizaki@trl.ibm.co.jp

**IBM Research,
Tokyo Research Laboratory**

Table of Contents

- **Motivation**
- **Previous devirtualization techniques**
 - ▶ **Guarded devirtualization**
 - ▶ **Direct devirtualization**
- **Our direct devirtualization technique**
 - ▶ **Class hierarchy analysis (CHA) at runtime**
 - ▶ **Code patching mechanism**
 - ▶ **Impact on traditional compiler optimizations**
- **Experiments**
- **Summary**

Motivation of Direct Devirtualization

- **Method size is fairly small in Java programs**
 - ▶ **Narrow compiler optimization scope**
- **Virtual call is frequently used**
 - ▶ **Prevent a compiler from performing optimizations**
- **Guarded devirtualization is not so much effective like dynamically-typed language**
 - ▶ **Usually use dispatch table in statically-typed language**

Example of a dynamic method call with a dispatch table in Java

```
r0 = <receiver object>
load  r1, offset_class_in_object(r0) // load method table
load  r2, offset_method_in_class(r1) // load method block
load  r3, offset_code_in_method(r2)  // load native code address
call  r3
```

Previous Guarded Devirtualization

- Class Test [Calder94], Method Test [Detlefs99]
 - ▶ Simplify implementation :-)
 - ▶ Incur additional overhead by **a receiver test with memory accesses** :-)

Class Test

```
r0 = <receiver object>
load  r1, offset_class_in_object(r0)
if (r1 == #address_of_particular_class) {
    <inlined code>
} else {
    load  r2, offset_method_in_class(r1)
    load  r3, offset_code_in_method(r2)
    call  r3
}
```

Method Test

```
r0 = <receiver object>
load  r1, offset_class_in_object(r0)
load  r2, offset_method_in_class(r1)
if (r2 == #address_of_inlined_method) {
    <inlined code>
} else {
    load  r3, offset_code_in_method(r2)
    call  r3
}
```

Previous Direct Devirtualization (1)

- Use class hierarchy analysis and reoptimization [Self93, HotSpot99]
 - ▶ Difficult to implement on-stack replacement completely :-(
 - Replace methods while there is an active context on stack

```
void foo()  
{  
    a = this.o;  
    ...  
    a.x();  
}
```

Single implementation of receiver
of a.x() at a invocation.

→ r0 = <receiver object>
<inlined code>

Previous Direct Devirtualization (1)

- Use class hierarchy analysis and reoptimization [Self93, HotSpot99]
 - ▶ Difficult to implement on-stack replacement completely :-(
 - Replace methods while there is an active context on stack

Lots of works needed (Tue AM)

```
void foo()  
{  
    a = this.o;  
    ...  
    ...  
    a.x();  
}
```

r0 = <... object>
<inline ...>

If the assumption is invalid,
recompilation *always* occurs.

```
load  r1, offset_class_in_object(r0)  
load  r2, offset_method_in_class(r1)  
load  r3, offset_code_in_method(r2)  
call  r3
```

Previous Direct Devirtualization (2)

■ Type Analysis [Palsberg91]

- ▶ Type information helps other optimizations :-)
- ▶ Difficult to apply in a wide range :-)

```
void foo()  
{  
    a = new A();  
    ...  
    a.x();  
}
```



r0 = <receiver object>
<inlined code>

All class instantiations that reach receiver of a.x() have the same definition.

Previous Direct Devirtualization (3)

- Preexistence [Detlefs99]
 - ▶ Avoid on-stack replacement :-)
 - ▶ Difficult to apply in a wide range :-)

```
void foo(A a)
{
    ...
    ...
    ...
    a.x();
}
```

Single implementation of receiver of
a.x() during the invocation of foo().

→ r0 = <receiver object>
<inlined code>

Previous Direct Devirtualization (3)

- Preexistence [Detlefs99]
 - ▶ Avoid on-stack replacement :-)
 - ▶ Difficult to apply in a wide range :-)

```
void foo(A a)
{
    ...
    ...
    ...
    a.x();
}
```



```
r0 = <... object>
<inline ...>
```



If the assumption is invalid,
recompilation occurs *at next invocation*.

```
load r1, offset_class_in_object(r0)
load r2, offset_method_in_class(r1)
load r3, offset_code_in_method(r2)
call r3
```

Our Devirtualization Techniques

- Direct devirtualization with *the code patching mechanism*
 - ▶ Generate devirtualized code using CHA and the indirect call w/o conditional branch
 - No test overhead at method call :-)
 - Impact on some compiler optimizations :-(
 - ▶ Invalidate devirtualized code by patching an instruction when a method is overridden
 - No recompilation :-)

Devirtualization with the code patch

■ Method x in Class A is not overridden

► Generate devirtualized code and the indirect call w/o conditional branch

```
Class A {  
    void x() {this.o = 1;}  
}  
void foo(A a) {  
    a.x(); // x is not  
    overridden  
}
```

```
    store    offset_o(r0), 1
```

```
after_inline:
```

```
    ret
```

```
    ...
```

```
dynamic_call:
```

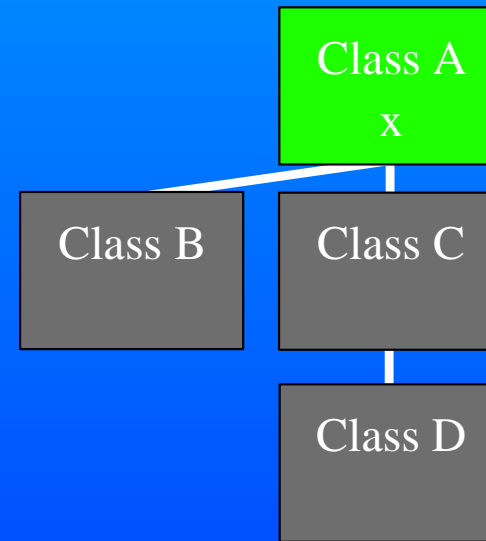
```
    load    r1, offset_class_in_object(r0)
```

```
    load    r2, offset_method_in_class(r1)
```

```
    load    r3, offset_code_in_method(r2)
```

```
    call   r3
```

```
    jmp    after_inline
```



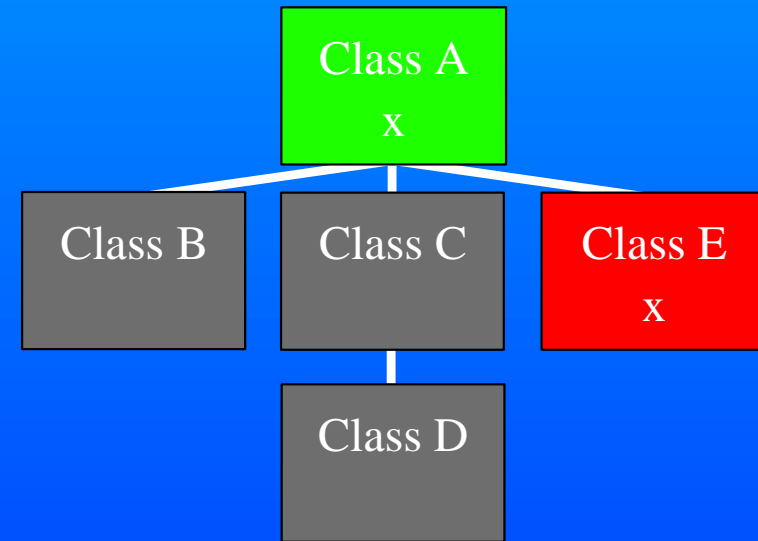
a backup path

Devirtualization with the code patch

- When method `x` in Class A is overridden
 - ▶ Apply code patching with care of cache coherence and atomicity

```
Class A {  
    void x() {this.o = 1;}  
}  
void foo(A a) {  
    a.x(); // x is overridden  
}
```

```
    jmp    dynamic_call  
after_inline:  
    ret  
    ...  
dynamic_call:  
    load   r1, offset_class_in_object(r0)  
    load   r2, offset_method_in_class(r1)  
    load   r3, offset_code_in_method(r2)  
    call   r3  
    jmp    after_inline
```



Invalidates devirtualized code

Code Patching on PowerPC

Write only one word (26bit relative jmp)

- **Atomic**

- ▶ **Write one word as an atomic memory operation**

- **Cache coherence**

- ▶ **Split cache for data and instruction**

- **Synchronize data and instruction cache explicitly**

- **Decode**

- ▶ **Flush instruction prefetch queue explicitly**

Code Patching on IA32

Write two or five byte (short or long relative jmp)

- **Atomic**

- ▶ **Write a single item as an atomic memory operation if aligned within a cache line**

- **Cache coherence**

- ▶ **Synchronize data and instruction cache automatically**

- **Decode**

- ▶ **Flush instruction prefetch queue automatically**
- ▶ **Adjust the lengths of old and new instructions**

Code Patching on IA32

■ Example (within a cache line)

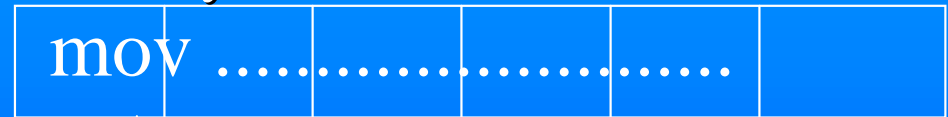
Patch 2bytes



atomic write (2byte)



Patch 5bytes



atomic write (2byte)



non-atomic write



atomic write (2byte)



Advantages of code patch mechanism

- It does not pay any overhead by explicit memory accesses, compares, and jumps
- It does not require recompilation (also on-stack replacement)

Impact on compiler optimizations

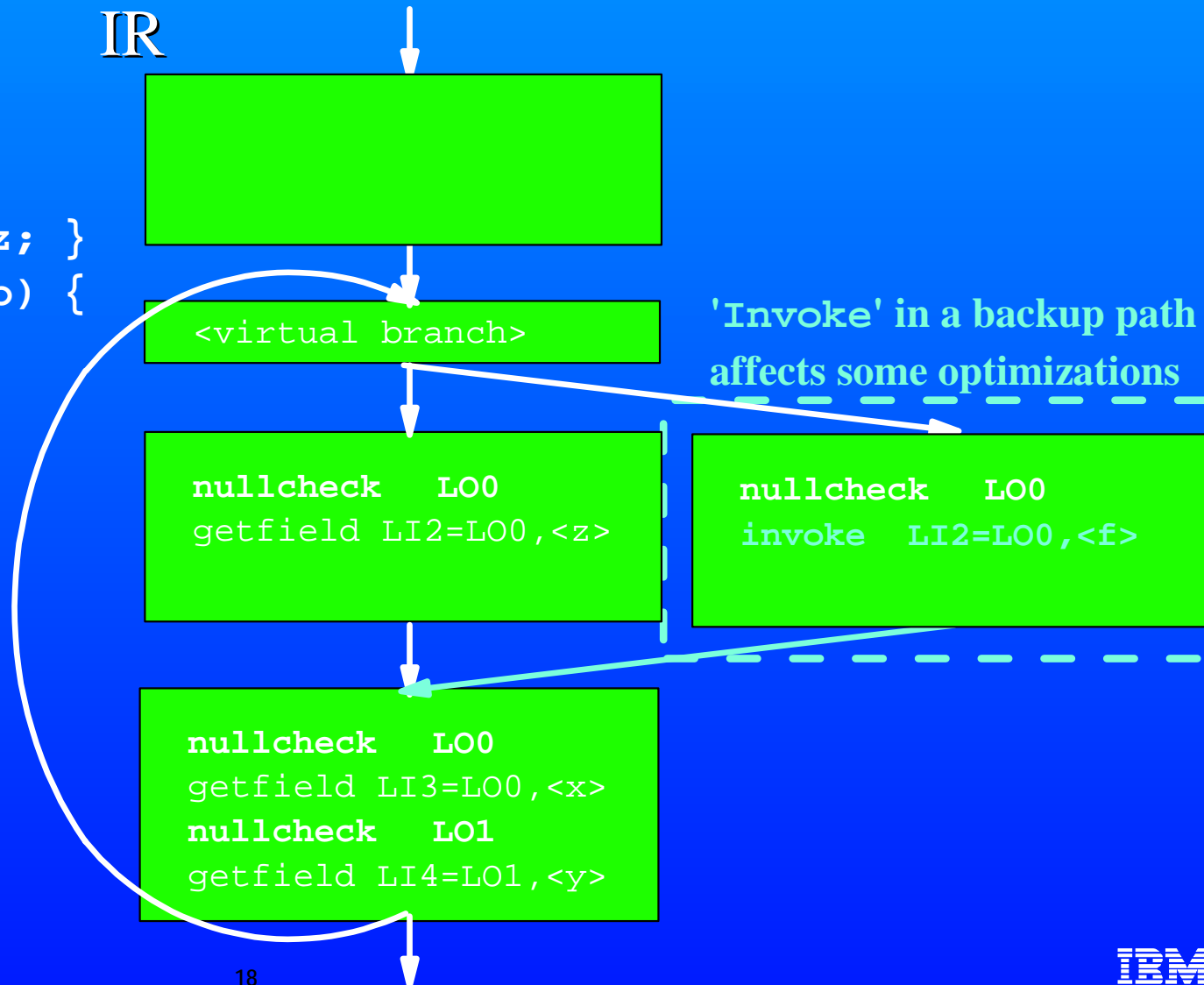
- It does not pay any overhead by explicit memory accesses, compares, and jump
- It does not require recompilation (also on-stack replacement)
- **It (and guarded devirtualization) requires to prepare two versions of the code**
 - ▶ Generate the merge point of the control flow. It prevents some compiler optimizations
 - Further, a backup path usually includes **indirect call**
 - Try to optimize devirtualized code with a backup path

An example program

Source

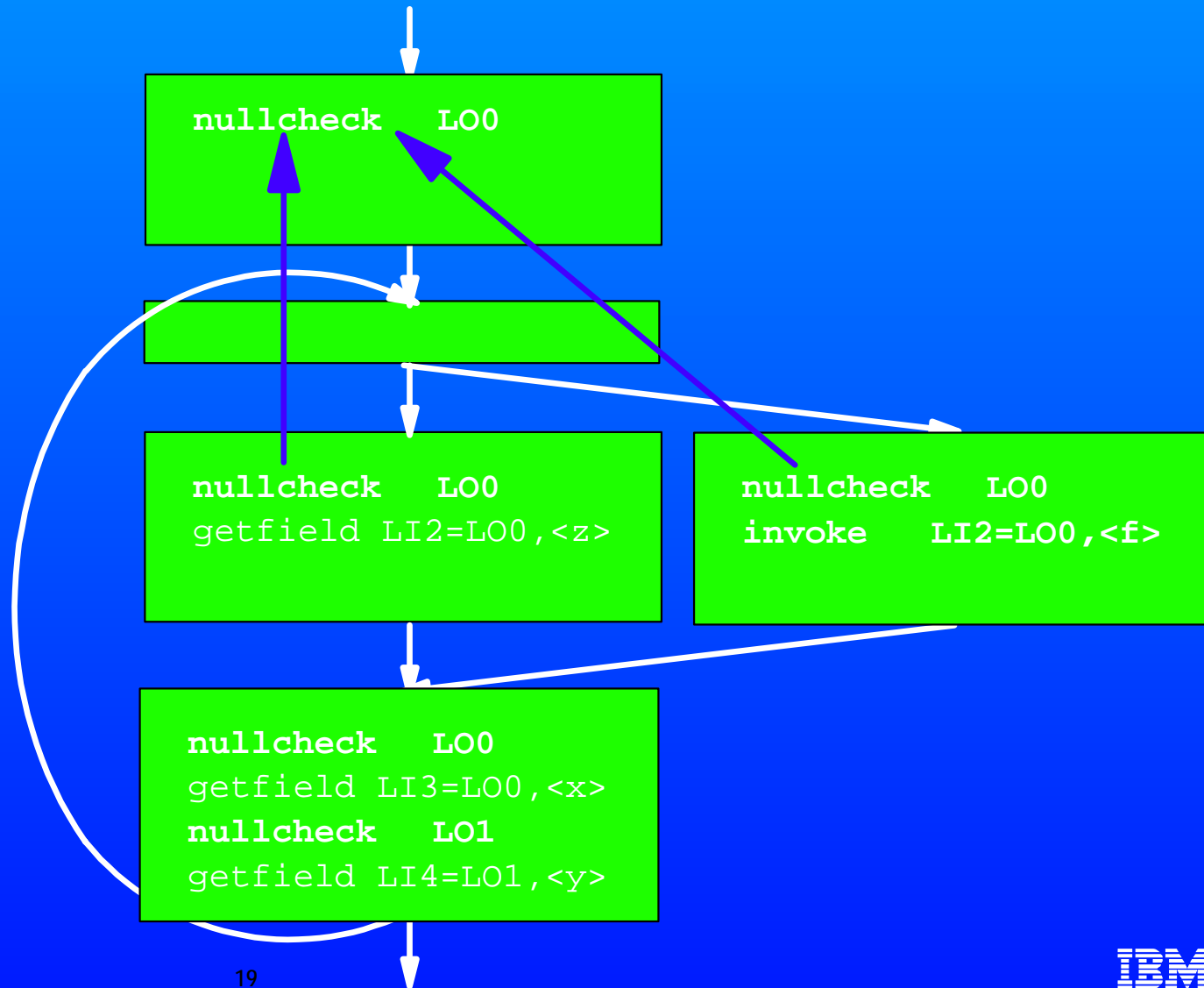
```
class Foo {  
  int x, y, z;  
  int f() { return this.z; }  
  int caller(Foo a, Foo b) {  
    do {  
      i = a.f();  
      j = a.x;  
      k = b.y;  
    } while (cond);  
    return i+j+k;  
  }  
}
```

IR



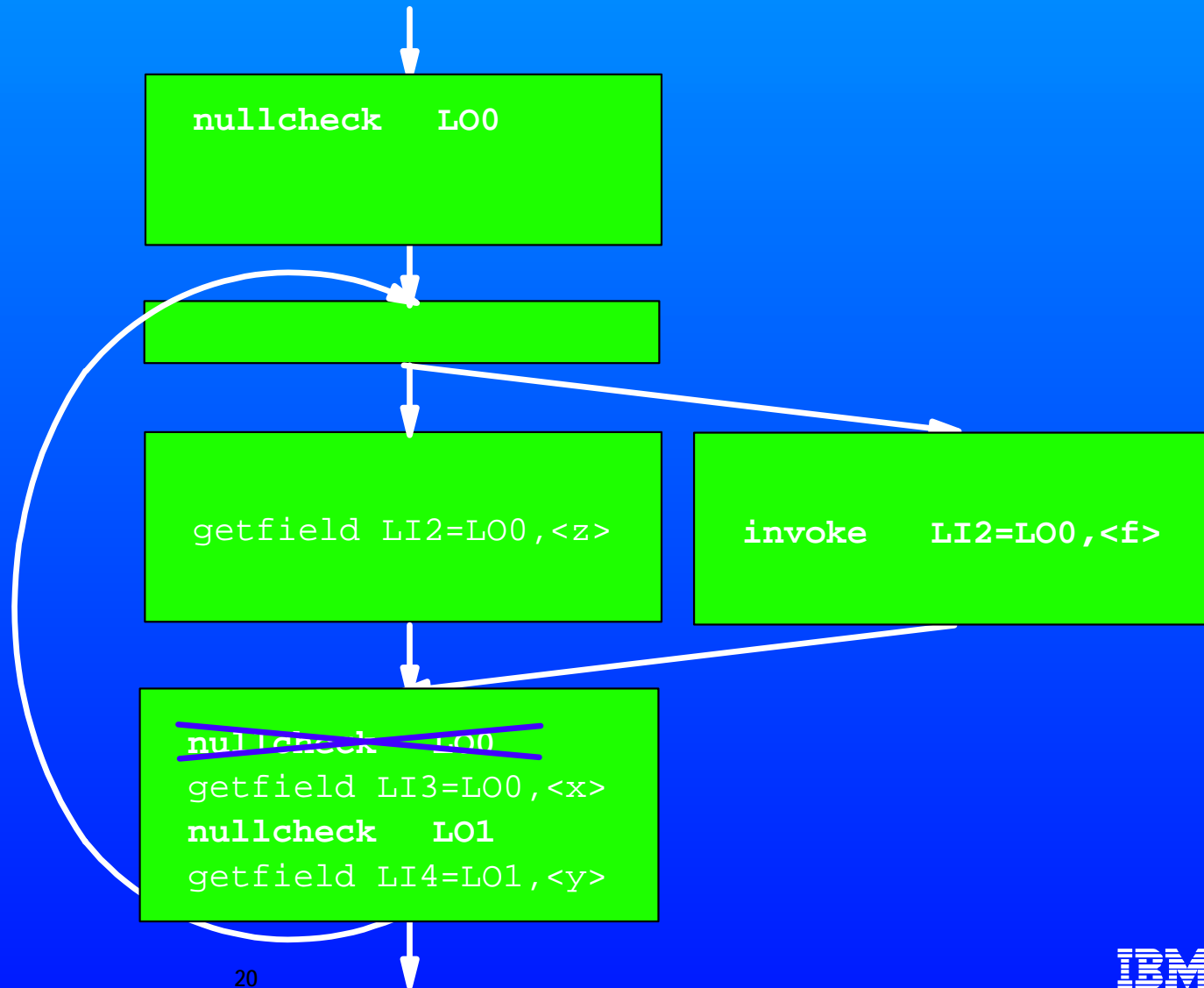
Nullpointer check optimization (1)

- Move 'nullcheck L00' out of a loop (backward)



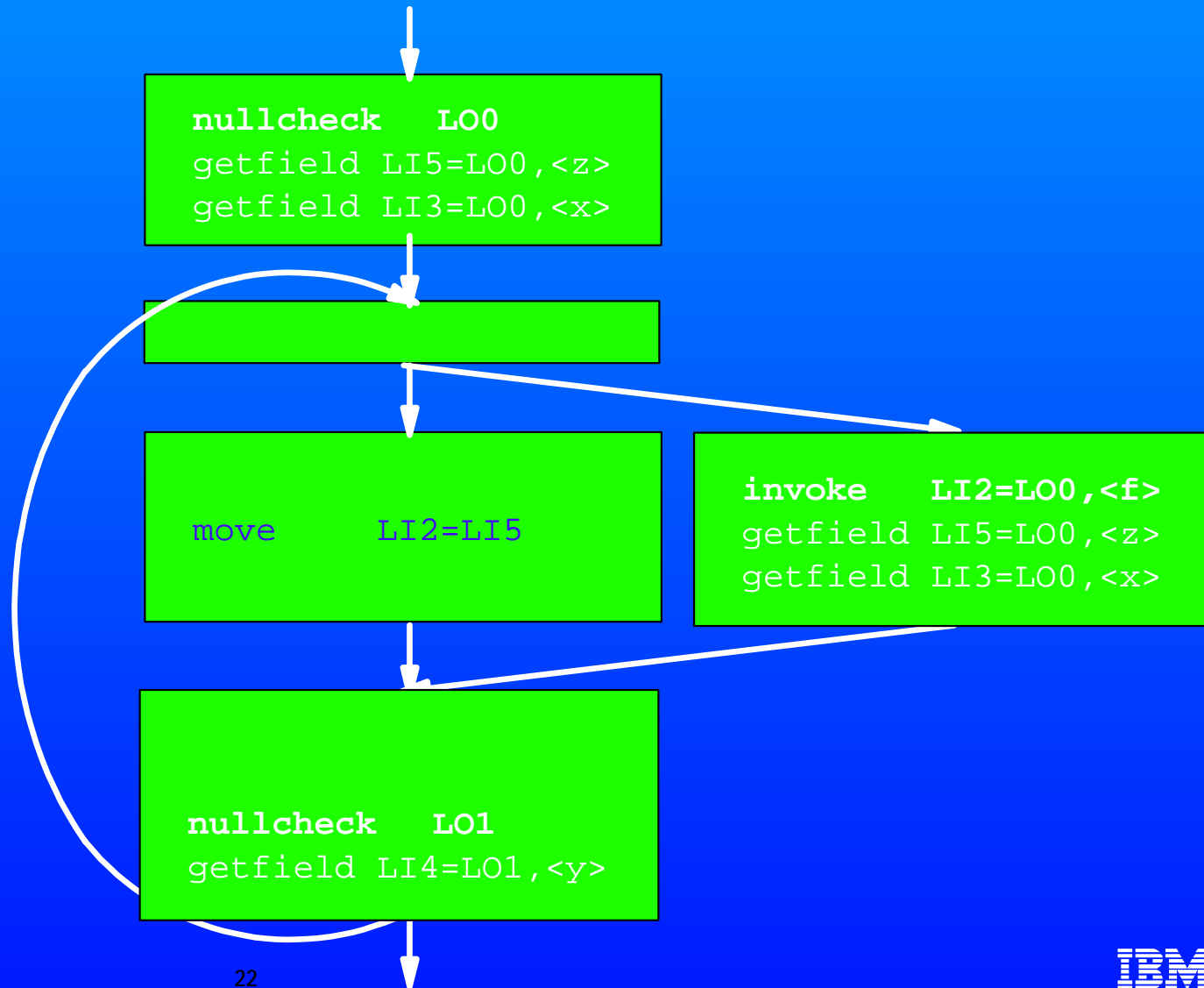
Nullpointer check optimization (2)

- Eliminate 'nullcheck L00' (forward)



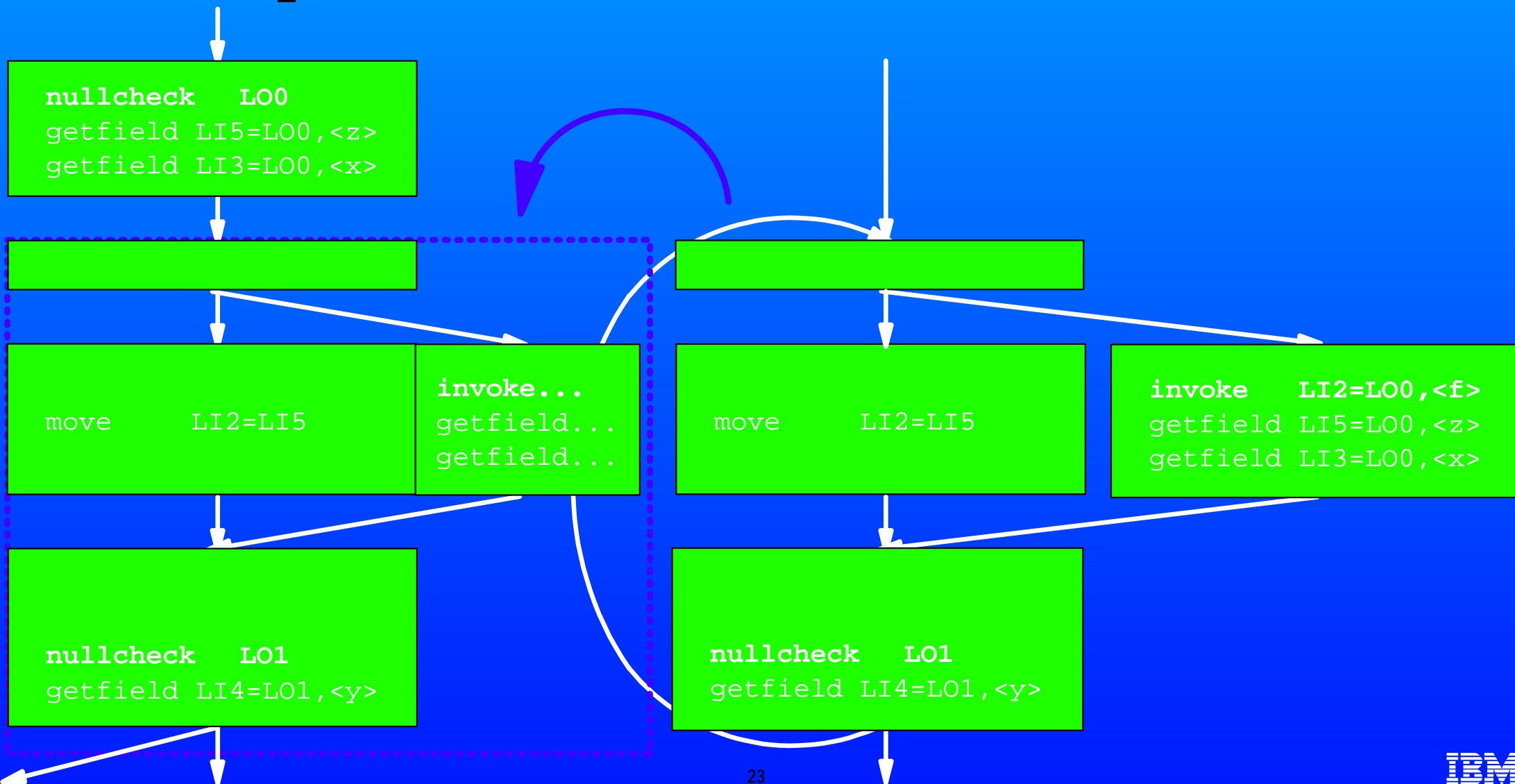
Partial redundancy elimination (2)

- Replace a reference with a temporary value



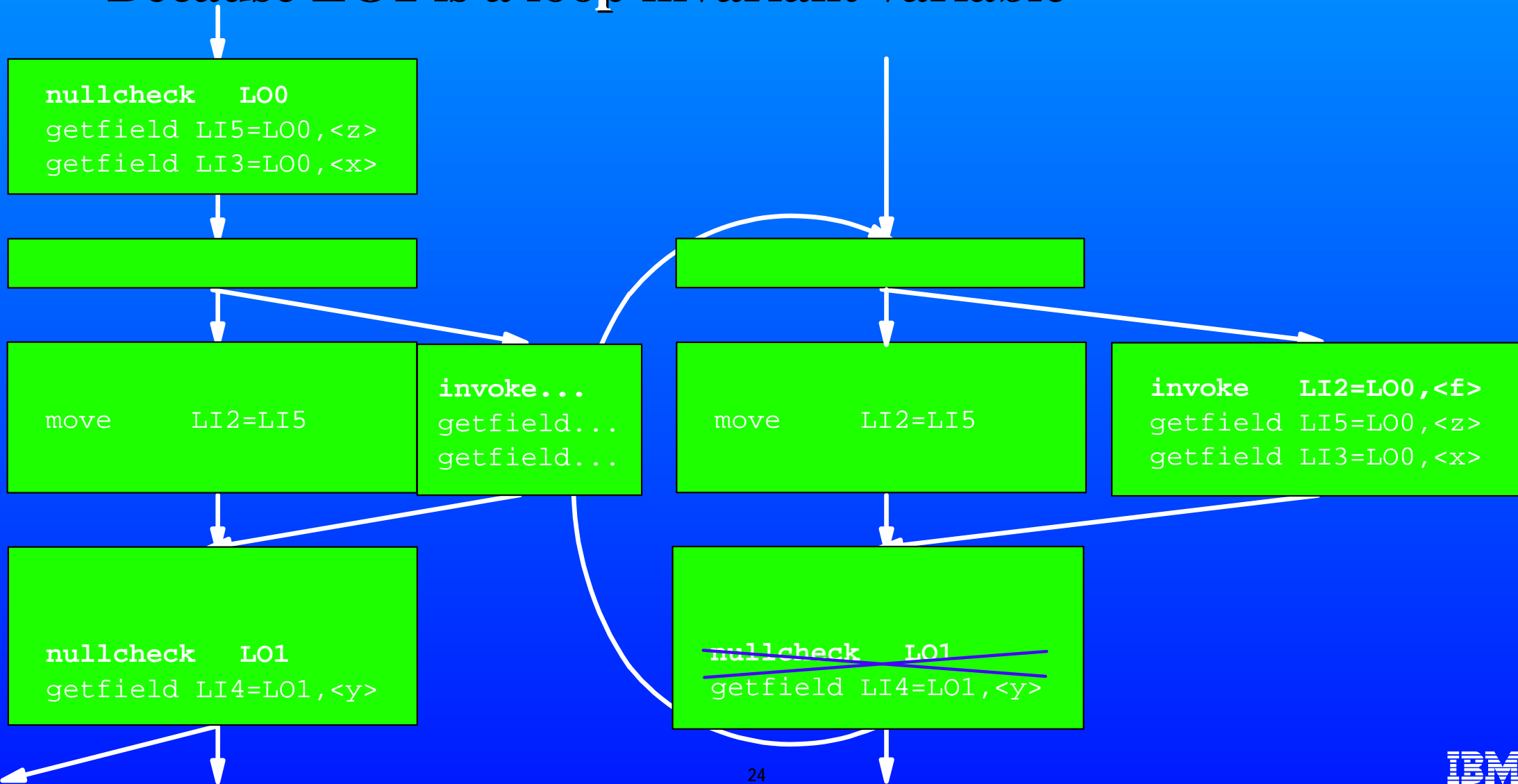
Loop peeling

- Insert a copy of a loop body before the beginning of a loop



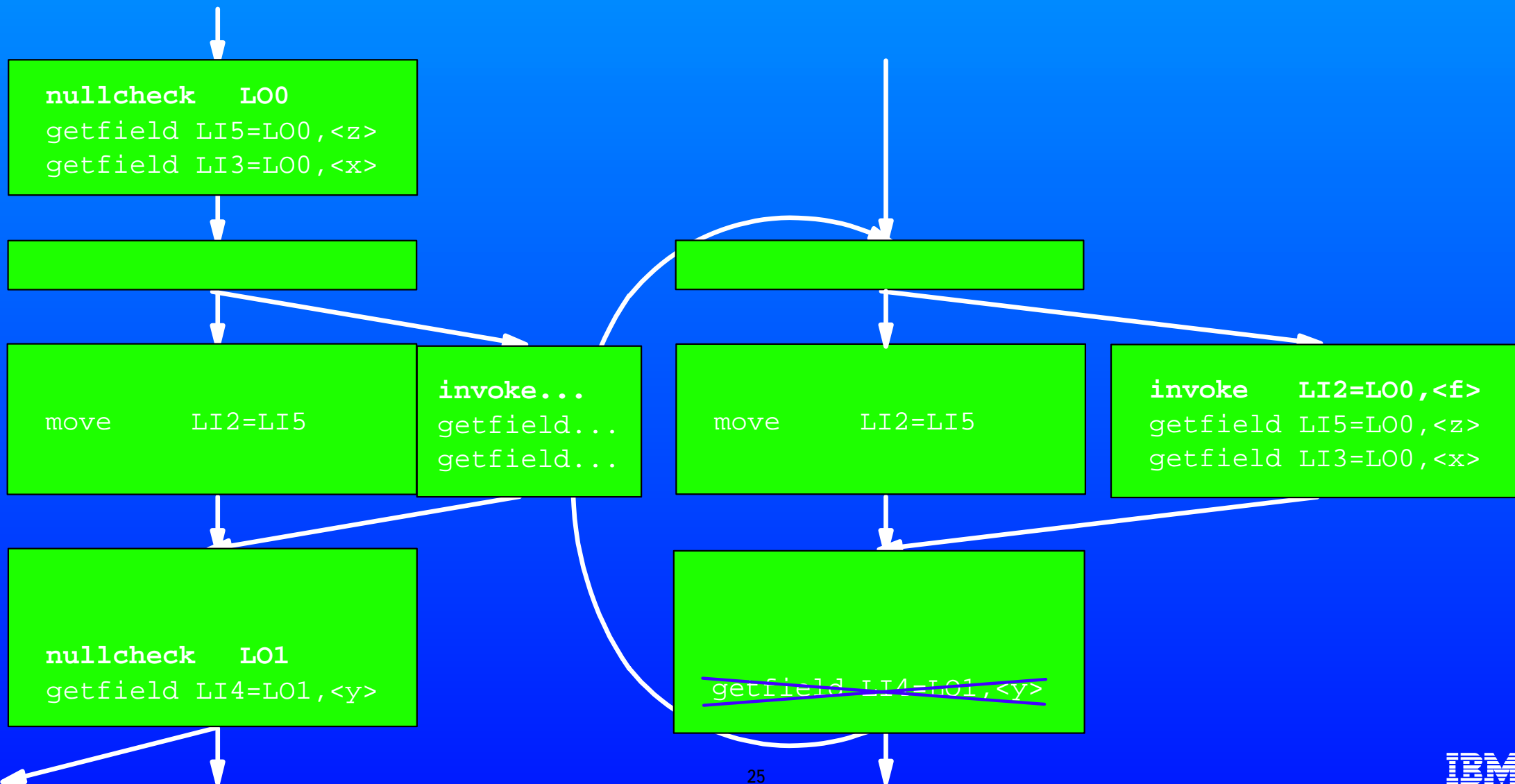
Nullpointer check optimization

- Eliminate 'nullcheck L01' in a loop (forward)
 - ▶ Because L01 is a loop invariant variable



Dead store elimination

- Eliminate 'getfield LI4=L01,<y>' in a loop



Apply devirtualization techniques

■ Guarded Devirtualization

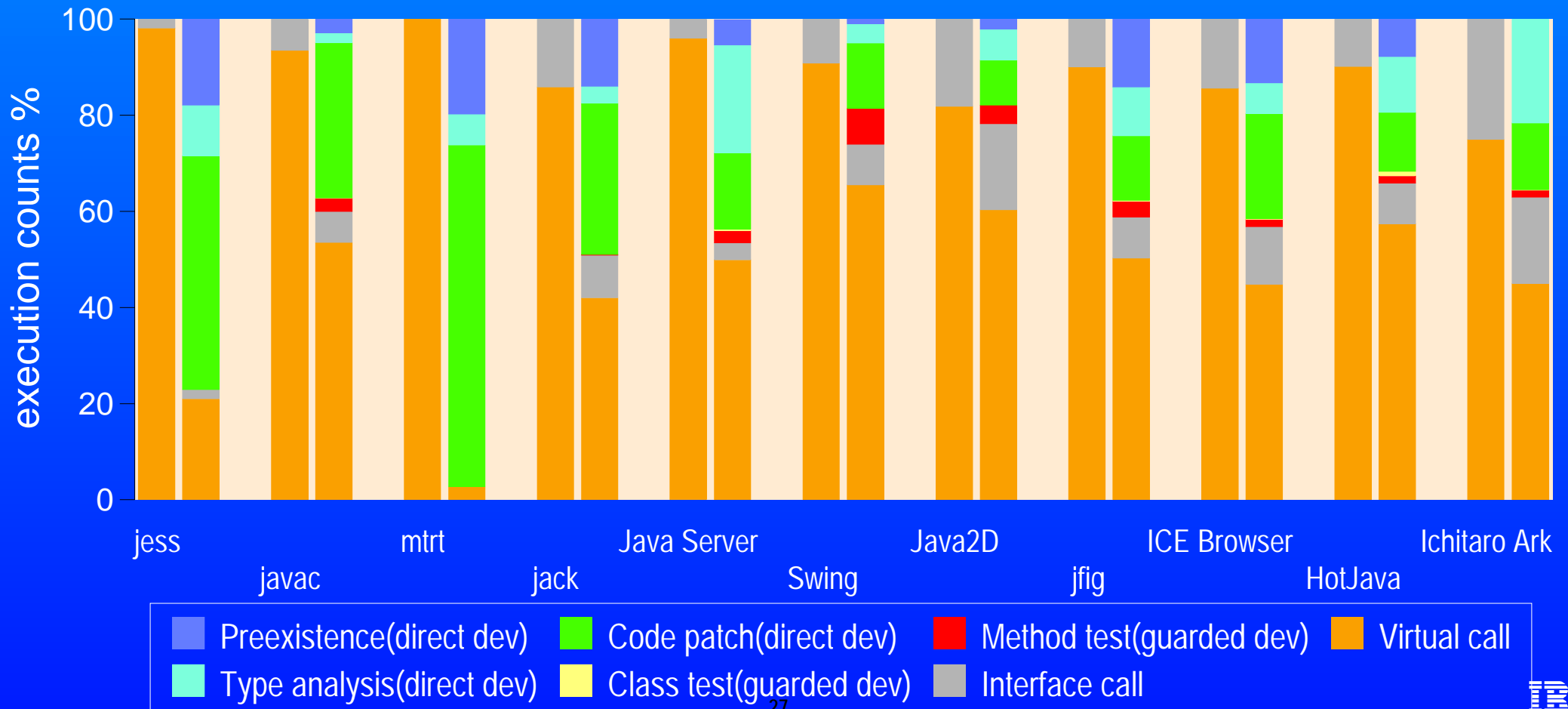
- ▶ **Class test [Calder94]**
 - For limited cases
- ▶ **Method test [Dettefs99]**
 - For polymorphic method calls

■ Direct Devirtualization

- ▶ **Code patch mechanism [Ishizaki00]**
 - Instead of recompilation with on-stack replacement
- ▶ **Type analysis [Palsberg91]**
 - Without backup paths
- ▶ **Preexistence [Dettefs99]**
 - Without backup paths

Statistics

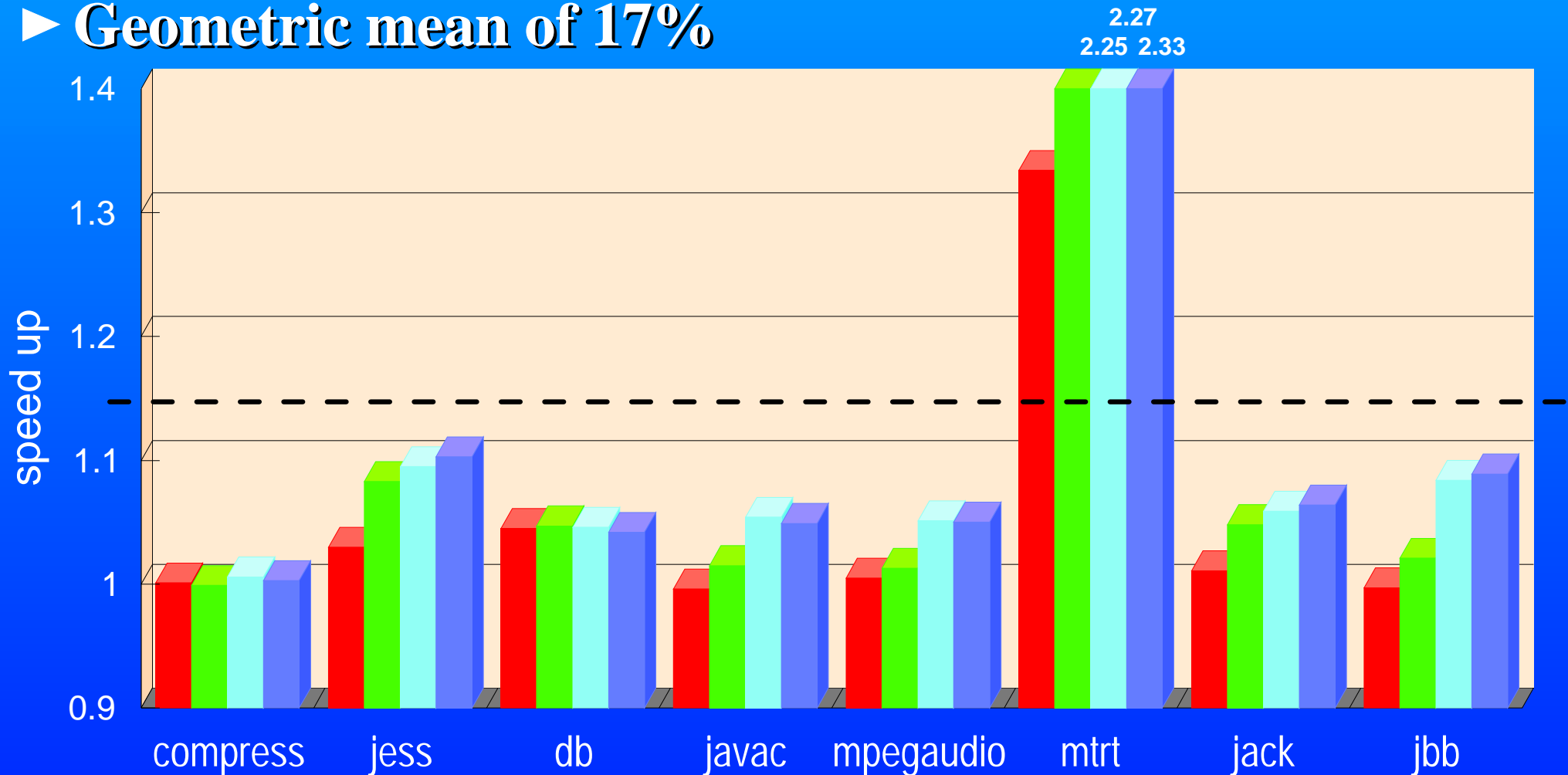
- **Dynamic method calls are devirtualized by:**
 - ▶ **Code patching (30%)**
 - ▶ **Type analysis and preexistence (24%)**



Performance

■ Performance improvement by devirtualizations

▶ Geometric mean of 17%



■ +Method&class tests(guarded dev) ■ +Code patch(direct dev) ■ +Type analysis(direct dev) ■ +Preexistence(direct dev)

● Base is not applied any devirtualization

Summary

- **Direct devirtualization with the code patching mechanism has two advantages**
 - ▶ **No overhead by guard statements**
 - ▶ **No recompilation**
- **Direct devirtualization with the code patching mechanism achieves fairly good performance though its implementation is simple**
- **Known techniques can optimize further in our technique**
 - ▶ **Generate compensation code and loop peeling**
 - ▶ **Type analysis and preexistence to eliminate backup paths**