



IBM Research, Tokyo Research Laboratory

Java Just-In-Time Compiler



Kazuaki Ishizaki

Tokyo Research Laboratory, IBM Japan, Ltd.

translation by Taiichi Yuasa

Summary

- These slides are a readily understandable tutorial material for beginners in compilers, as well as an interesting material for researchers with knowledge of compilers, and include the following.
 - Structure of Java just-in-time compiler
 - Conventionally known optimization
 - Optimization specific to Java
 - Optimization using runtime information

- Note
 - We introduce a wide range of optimizations, rather than details of algorithms for individual optimization
 - Please study on your own if lecture time is insufficient or to gain deeper understanding.

Contents

- Structure of Java system
- Conventionally known optimization

- Optimization specific to Java
- Optimization using runtime information

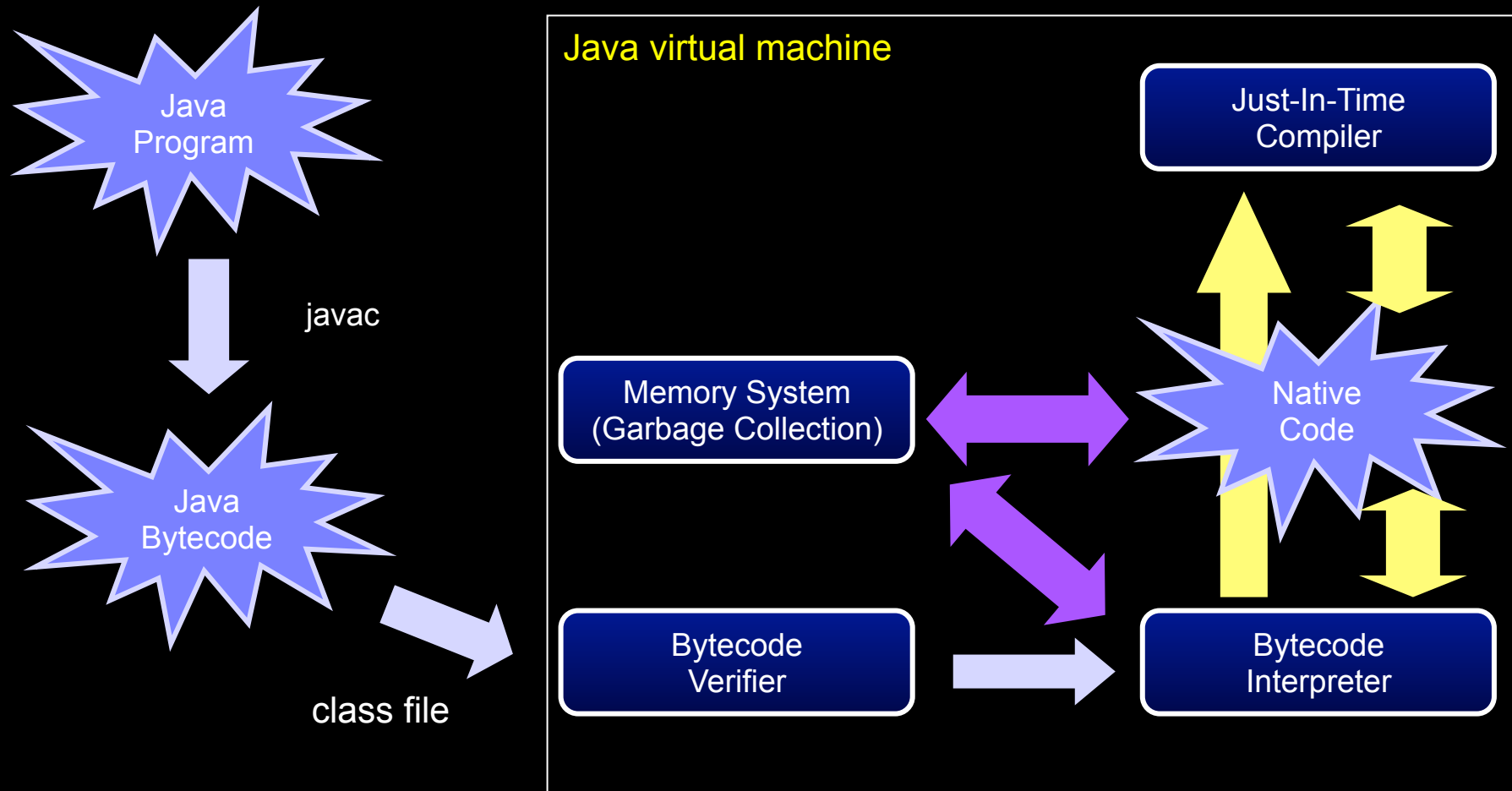
- References
 - for your self-study

Structure of Java System

- **Structure of Java system**
 - Structure of Java virtual machine
 - Structure of Java just-in-time (JIT) compiler
- Conventionally known optimization
- Optimization specific to Java
- Optimization using runtime information

Structure of Java virtual machine (JVM)

- JIT compiler is called selectively in the current JVM.



Conventionally known optimization

- Structure of Java system
- Conventionally known optimization
 - What is optimization?
 - Methods of optimization
- Optimization specific to Java
- Optimization using runtime information
- References

What is optimization?

■ Purpose

- To reduce execution time
 - The optimization of execution time is explained in this tutorial.
- To reduce required memory

■ Major premises

- The meaning of the program does not change after optimization.
 - The value observable from outside of a method does not change.
 - r is observable; a and b are not observable
 - Sequence of occurrence of exception does not change.
 - Division and remainder may generate exception.

```
int foo(int i, int j) {  
    int a, b, r;  
    a = i / j;  
    b = j % i;  
    r = a + b;  
    return r;  
}
```


Methods of optimization

- What to optimize?
 - Inside basic block
 - Basic block (BB) = sequence of instructions without branching and merging
 - Inter basic blocks
 - Carry out dataflow analysis

- How to optimize?
 - Reduce the number of executed instructions
 - Use faster instructions

Reduce the number of executed instructions (1)

- Preliminary computation during compilation
 - Constant folding
 - Constant propagation
- Elimination of redundant codes
 - Dead code elimination
 - Copy propagation
- Reuse of execution results
 - Common subexpression elimination
 - Partial redundancy elimination

Reduce the number of executed instructions (2)

- Deformation of method
 - Code hoisting
- Particularization and specialization of programs
 - Method inlining
 - Tail duplication

Constant folding

- Constant calculation is carried out during compilation.
 - Consideration should be given to the accuracy of floating point and exception condition (what is the result of $-32768/-1$?).

```
...  
x = 2 * 3;  
...  
→  
...  
x = 6;  
...
```

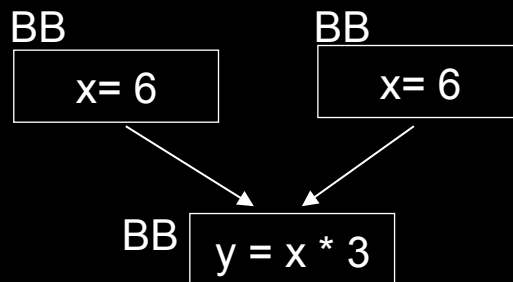
Constant propagation

- A constant value is propagated within methods.

```
x= 6;  
... // no definition of x between these statements.  
y = x * 3;
```




```
x = 6;  
...  
y = 18;
```



Dead code elimination

- Eliminate codes used to calculate values that are not used in the method.
- Eliminate unexecuted statements.

```
...  
x = a + b; // x is not used in later statements.  
...  
if (false) {  
    y = 1;  
}
```



```
...  
x = a + b;  
...  
if (false) {  
    y = 1;  
}
```

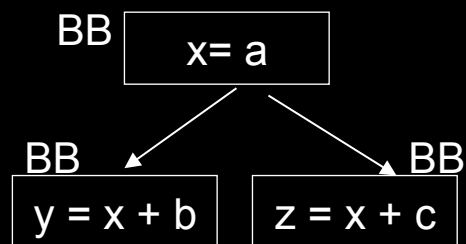
Copy propagation

- A simple substitution instruction is propagated to the right-hand side operand.

```
x = a; // x is defined only here.  
... // no definition of a between these statements.  
y = x + b;
```



```
x = a;  
...  
y = a + b;
```



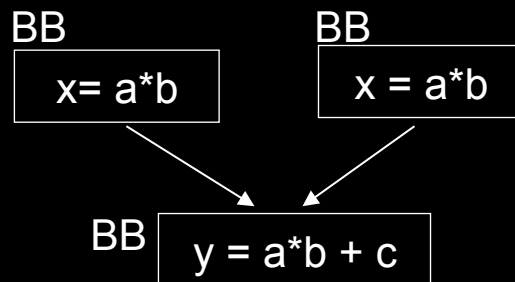
Common subexpression elimination

- Reuse the values once calculated.

```
x = a*b;  
... // no definition of a between these statements.  
y = a*b + c;
```

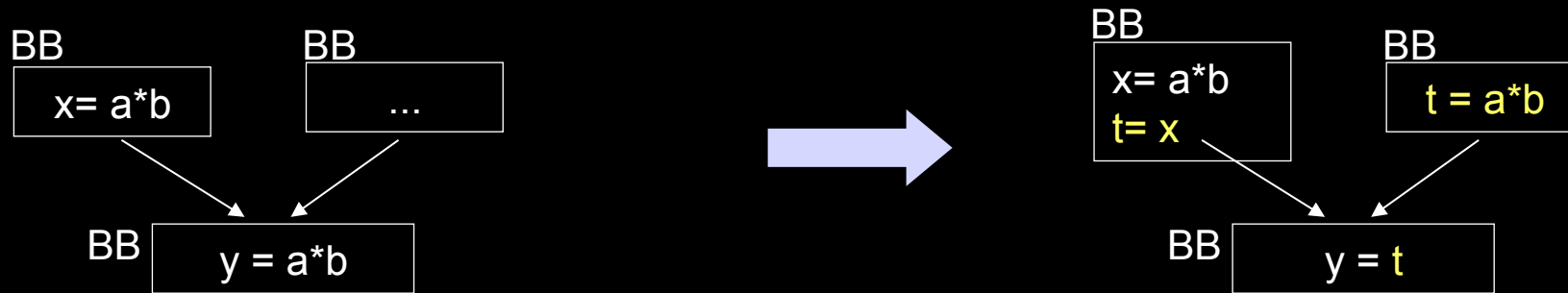


```
x = a*b;  
...  
y = x + c;
```



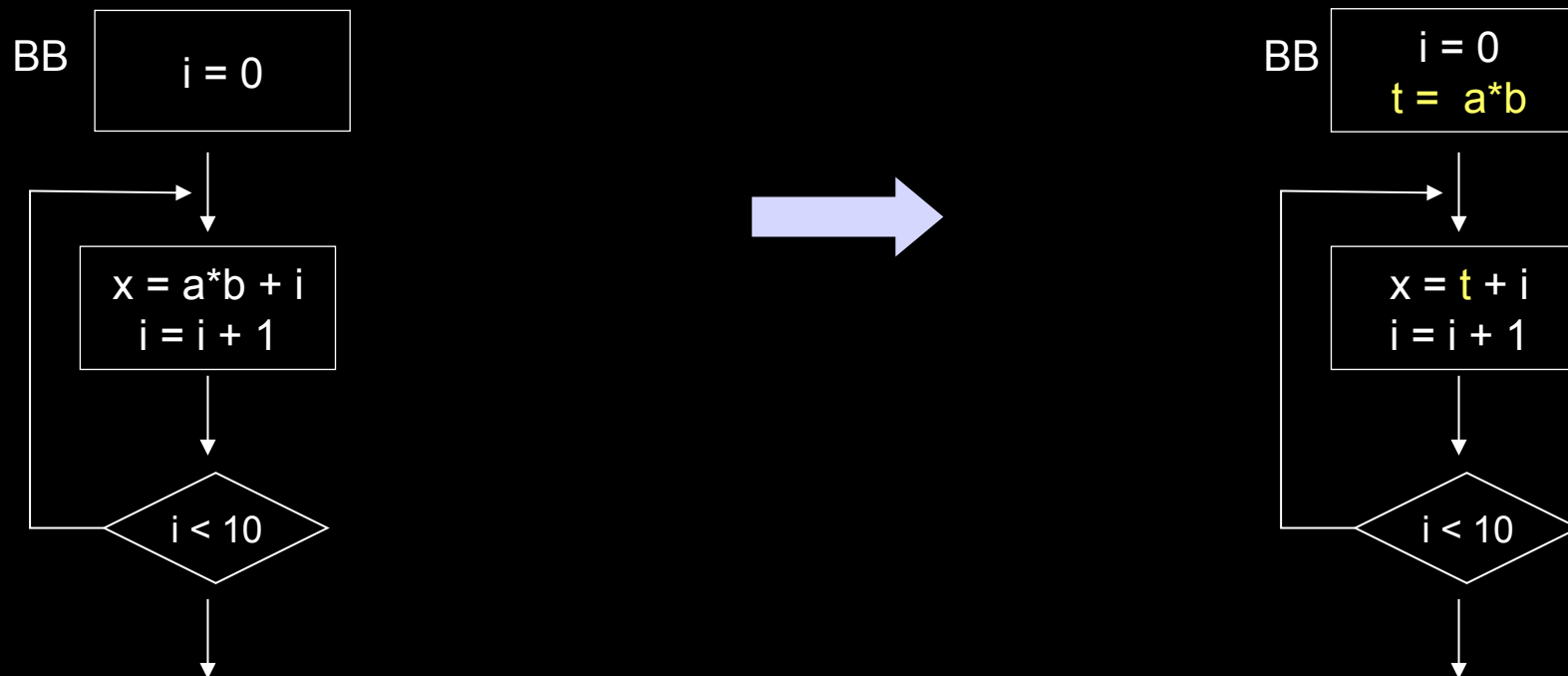
Partial redundancy elimination

- To reuse the values once calculated, move the expression to reuse the values in the execution path in accordance with the order of execution sequence.
 - Care should be taken in moving the expressions involving exceptions (division, remainder, etc.) in Java.



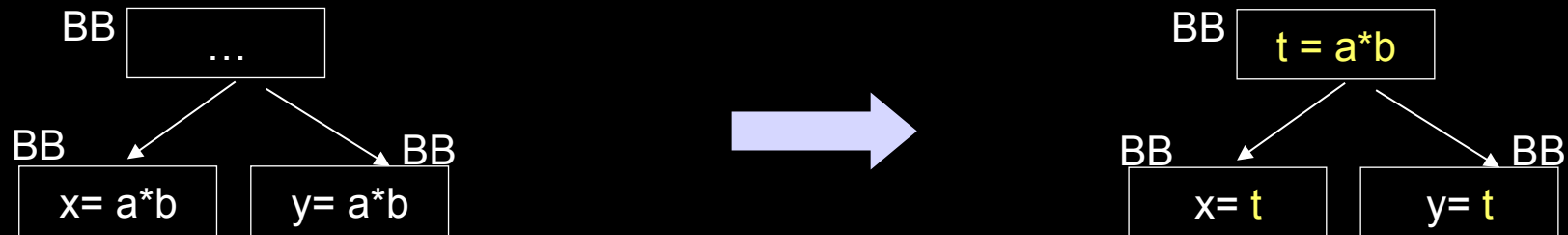
Loop invariant code motion

- Move loop-constants outside the loop.
 - Care should be taken in moving the expressions involving exceptions in Java.



Code hoisting

- Hoist the expression in accordance with the execution sequence.
 - The expression that is used in only one of the two paths can also be hoisted.
 - Care should be taken in moving the expressions involving exceptions in Java.



Method inlining

- The method body is expanded at call locations.
 - The range of applications of optimization expands.
 - The passing of argument and register save/recovery can be eliminated.

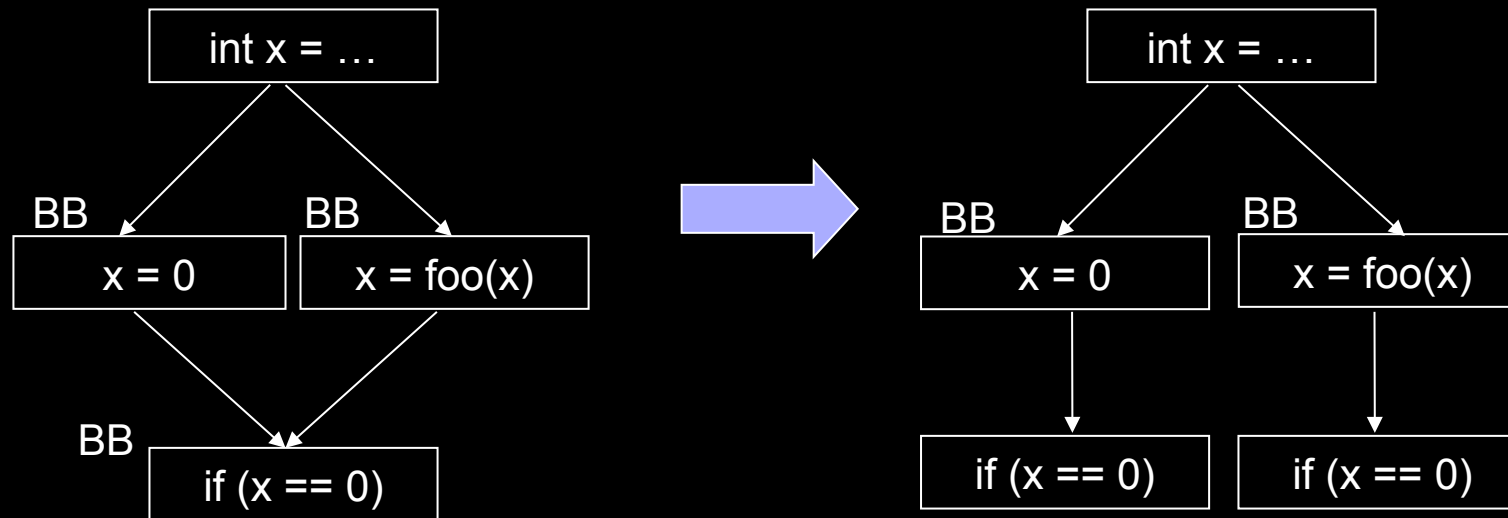
```
...  
x = foo(a)  
...  
  
int foo(int i) {  
    return i * i;  
}
```



```
...  
x = (a * a);  
...  
  
int foo(int i) {  
    return i * i;  
}
```

Tail duplication, splitting

- Duplicate codes after a merge point of control and make specialized codes.
 - The characteristic of the variable is determined uniquely.



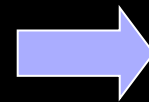
Use the instruction with a higher execution time.

- Strength reduction
- Scalar replacement
- Register allocation

Strength reduction

- Replace the operation with an equivalent instruction of the CPU with a shorter execution time.
 - For example, in many CPUs, shift instruction is faster than multiplication and division.

```
...  
x = a * 8; // Multiplication using power of two  
...
```

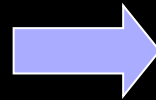


```
...  
x = a << 3;  
...
```

Scalar replacement

- Replace memory access instruction with instruction using simple variables
 - It is expected that simple variables will be allocated to the register during later register allocation.

```
class A {  
    int f;  
    int g;  
}  
A a = new A();  
a.f = 1;  
a.f = a.f * 2;
```

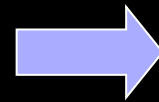


```
class A {  
    int f;  
    int g;  
}  
A a = new A();  
t = 1;  
t = t * 2;  
a.f = t;
```


Register allocation

- Allocate simple variables of a method to the registers of the CPU as much as possible.
 - Observing the whole method: Graph coloring register allocator
 - Sequentially from the beginning of the method: Linear scan register allocator

```
...  
a = 1;  
b = 2;  
c = a + 3;  
a = b  
...
```



There are two registers to which
a and c are allocated.

```
...  
R1 = 1;          // a: R1  
[mem_b] = 2;    // b: mem  
R2 = R1 + 3;    // c: R2  
R1 = [mem_b]  
...
```

Optimization specific to Java

- Structure of Java system
- Conventionally known optimization
- Optimization specific to Java
 - Performance issues specific to Java
 - Optimization issues specific to Java
- Optimization using runtime information
- References

Performance issues specific to Java

- Exception check to ensure program safety
- Type check to ensure program safety
- Virtual method call by introduction of polymorphism
- Reference to the field in an object by encapsulation
- Synchronization provided by language

Optimization issues specific to Java (1)

- Elimination of exception check
 - Elimination of simply redundant exception check
 - Elimination of partially redundant exception check
 - Inverse optimization
 - Loop versioning

- Elimination of type check and type conversion
 - Type analysis

Optimization issues specific to Java (2)

- Speed up of virtual-method call
 - Guarded devirtualization
 - Direct devirtualization

- Speed up of object reference
 - Stack allocation by escape analysis
 - Scalar replacement

- Speed up of synchronization
 - Speed up of synchronization operation
 - Elimination of synchronization by escape analysis

Elimination of exception check

- To ensure program safety,
 - nullcheck - to check if the object is not null
 - bndcheck - to check if an index is within the range of the array
 - Others, such as division by 0

- The instructions following an exception check cannot be executed before the exception check.

- Elimination of exception check
 - Eliminating comparison and branch instructions used for exception check
 - Increasing the chance of moving instructions

Elimination of simply redundant exception check

- Eliminate the same exception check for the same variable

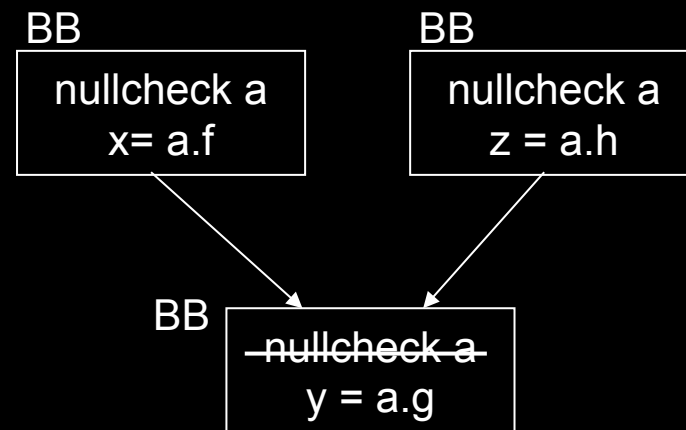
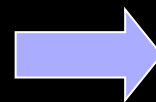
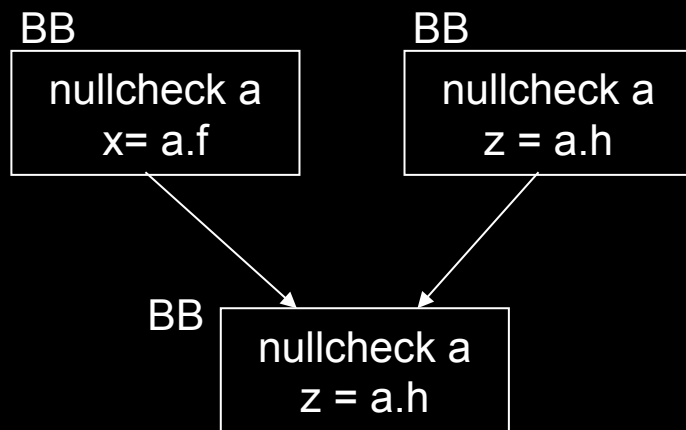
```
nullcheck a
x = a.f;
```

// no definition of a between these statements.

```
...
nullcheck a
y = a.g;
```

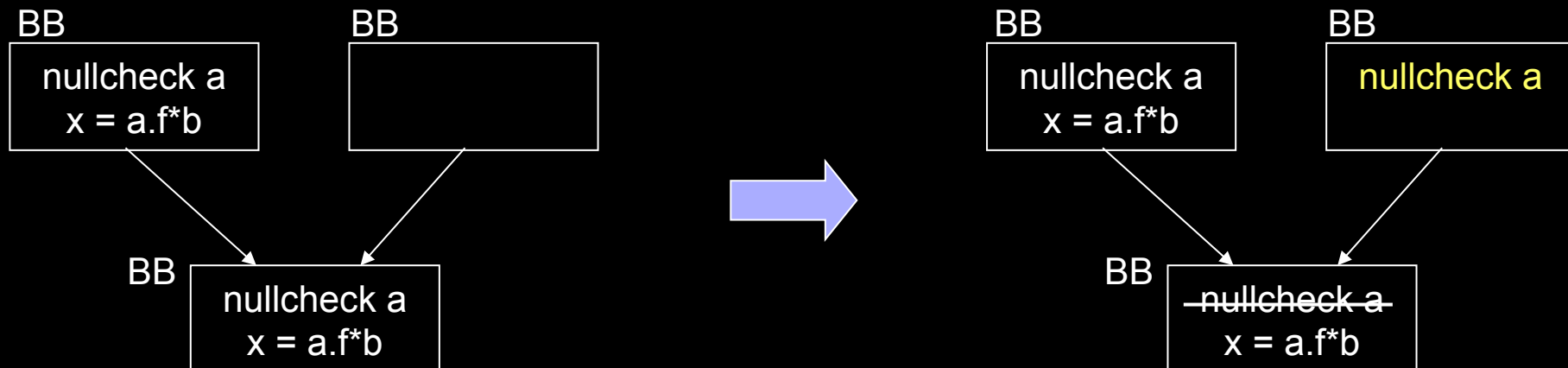
```
nullcheck a
x = a.f;
```

```
...
nullcheck a
y = a.g;
```



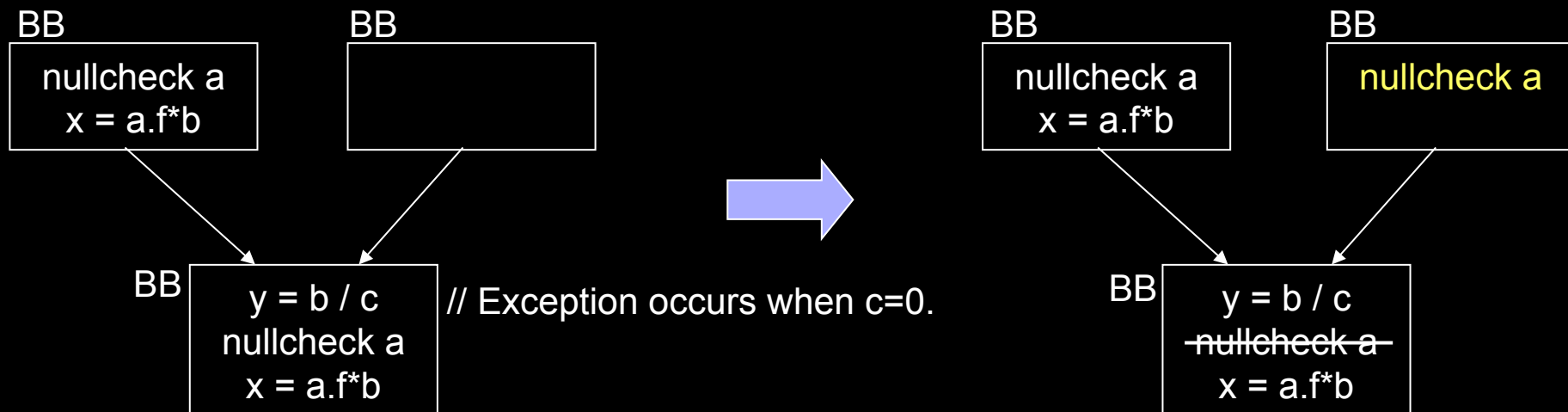
Elimination of partially redundant exception check

- Application of partial redundancy elimination
 - Exception check instruction cannot be moved beyond the instruction, which changes the condition observable from outside a method (i.e., instructions with side effect, such as exception check instruction and method call instruction).
 - By applying partial redundancy elimination several times, the chance of applying other optimizations involving code motion expands.



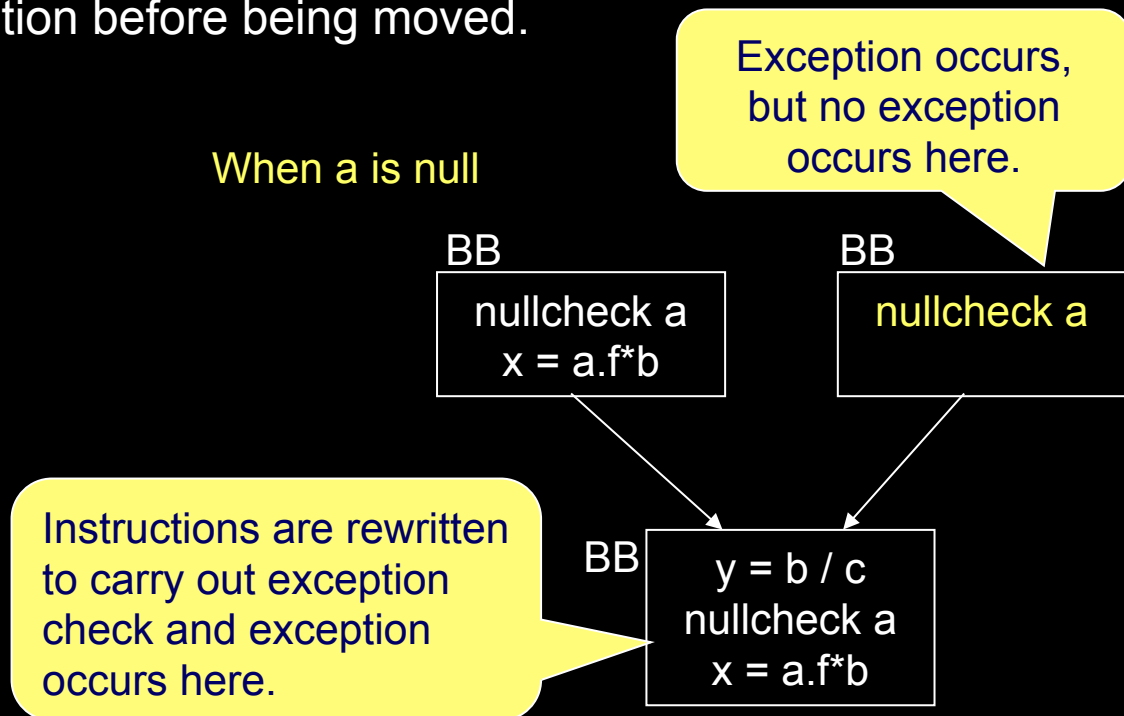
Elimination of partially redundant exception check and inverse optimization

- Ignore exception dependence and move the exception check instruction by partial redundancy elimination.



Elimination of partially redundant exception check and inverse optimization

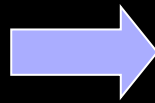
- Ignore exception dependence and move the exception check instruction by partial redundancy elimination.
- When exception is actually detected by the moved exception check instruction, exception occurs at the position before being moved.



Loop versioning

- Speculatively check if the range of suffixes accessed in the loop is within the array length immediately before the execution of the loop.
 - When the checking is successful, a loop without exception checks is executed.
 - When the checking is not successful, the original loop with exception checks is executed.

```
for (i = s; i < e; i++) {  
    a[i] = i; // require  
              nullcheck and  
              bndcheck  
}
```



```
if ((a != null) && (0 <= s) && (e <= a.length)) {  
    // optimized loop without exception checks  
    for (i = s; i < e; i++) {  
        a[i] = i;  
    }  
} else {  
    // original loop with exception checks  
    for (i = s; i < e; i++) {  
        nullcheck a;  
        bndcheck i, a.length;  
        a[i] = i;  
    }  
}
```

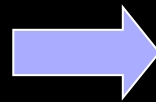
Elimination of type check and type conversion

- Type analysis

Type analysis

- Obtain the operand's type for object reference; when it is a class or subclass of type check/type conversion, the type check/type conversion can be eliminated.
 - new, instanceof, checkcast, argument, and return value are the sources of type inference.

```
if (x instanceof A) {  
    A y = (A) x;  
}
```



```
if (x instanceof A) {  
    A y = (A) x;  
}
```

Instanceof in the if statement ensures that x is A or a subclass of A.

Speed up of virtual method call

- Guarded devirtualization
 - Class test
 - Method test
 - Polymorphic Inline Cache

- Direct devirtualization
 - Type analysis
 - On stack replacement
 - Code patching
 - Preexistence

- Speed up of interface method call
 - Interface Method Table

Virtual method call

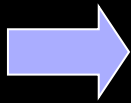
- Virtual method calls (invokevirtual, invokeinterface) are frequently used.
 - Optimization is inhibited by a method call whose target is determined only at runtime.
 - The size of one method is relatively small due to encapsulation.
 - The range of optimizations is narrowed.
- ➔ Optimization to determine the method call destination (devirtualization) is necessary.

Class test and Method test

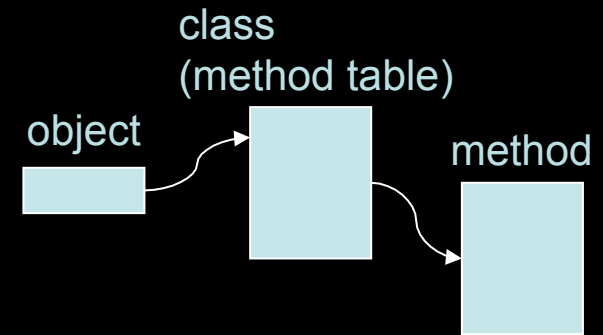
- **Guard: Compare** to determine whether a method or inlined code can be called directly

An example code for virtual method call in Java

```
foo(A x) {
...
x.m();
}
```



```
r0 = <receiver object>
load r1, offset_class_in_object(r0)
load r2, offset_method_in_class(r1)
load r3, offset_code_in_method(r2)
call r3
```



Class test

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

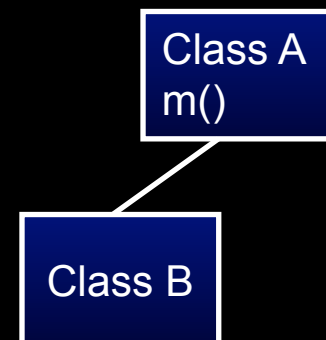
Method test

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


Class test and Method test

- The number of loads of a method test is larger by one than that of a class test; however, the former is more accurate.
 - When instance of B is passed to x
 - For the class test (assuming guard at A), $A \neq B$ holds and the virtual method is called.
 - For the method test, $\&A.m() == \&B.m()$ holds and the direct method is called.

```
foo(A x) {  
    ...  
    x.m();  
}
```

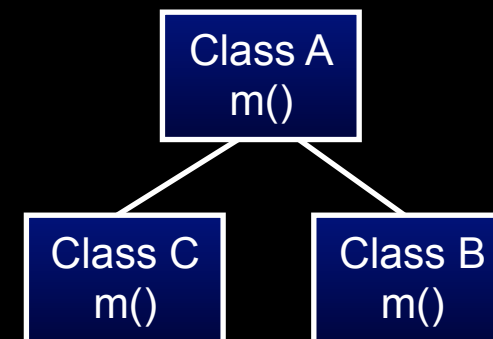


Polymorphic Inline Cache

- To speed up the call, which has multiple call destinations, multiple guards exist.
 - Virtual call in which multiple methods override
 - Interface call in which multiple classes implement

```
r0 = <receiver object>
load r1, offset_class_in_object(r0)
if (r1 == #address_of_paticular1_class) {
  call paticular1_method or inlined code
} else if (r1 == #address_of_paticular2_class) {
  call paticular2_method or inlined code
} else if (r1 == #address_of_paticular3_class) {
  call paticular3_method or inlined code
} else {
  load r2, offset_method_in_class(r1)
  load r3, offset_code_in_method(r2)
  call r3
}
```

An example in which multiple methods override A.m()



Directed devirtualization

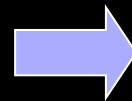
- In Java, the cost of the method call by guarded devirtualization is similar to that of the virtual method call.
 - Requirement for speed up by devirtualization without guard
- Method not requiring class hierarchy analysis
 - Type analysis
- Method requiring class hierarchy analysis
 - Code patching
 - On stack replacement
 - Preexistence

Type analysis

- Obtain the receiver's type for the method call; when it is determined uniquely, the virtual method call can be replaced with the direct method call or method inlining is possible.
 - "new", "instanceof", "checkcast", arguments, and return value are the sources of type inference.

```
A x = new A();
...
x.m();
```

The type reaching
x.m() is only A.



```
r0 = <receiver object>
call A.m() // Direct method call
```

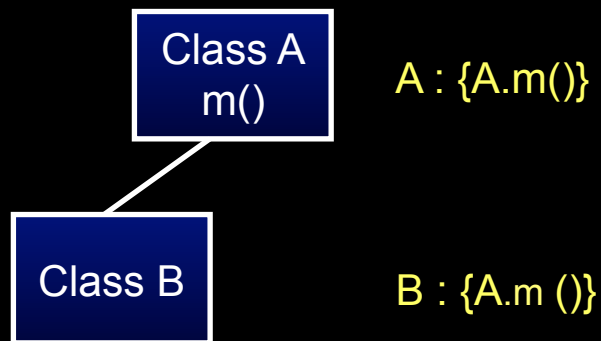
```
A x;
if (...) { x = new A(); }
else { x = new B(); } // B is subclass of A.
x.m();
```

Since both A and B are types reaching x.m(), the above replacement is not possible.

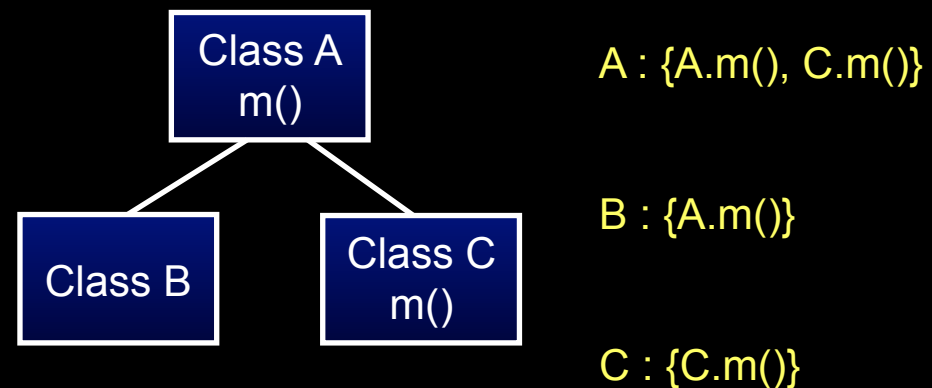
Class hierarchy analysis

- Examine what method is defined at the class hierarchy of the entire program.
- In Java, class load occurs during execution; periodical maintenance is required.

Possible set of methods



Possible set of methods

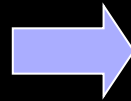


Code patching

- When the call destination method is not overridden in compilation, prepare both the direct method call and the indirect method call, and execute the direct method call.

```
A x;  
...  
x.m(); // Not overridden  
...
```

Class A
m()

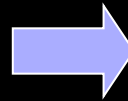
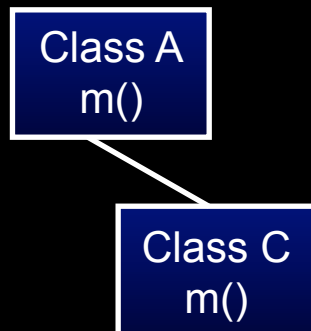


```
call A.m // Direct method call,  
after_call:  
...  
dynamic_call  
  load r1, offset_class_in_object(r0)  
  load r2, offset_method_in_class(r1)  
  load r3, offset_code_in_method(r2)  
  call r3 // Indirect method call  
  jmp after_call
```

Code patching

- When the method is overridden by dynamic class loading, rewrite the corresponding code and execute the indirect method call.

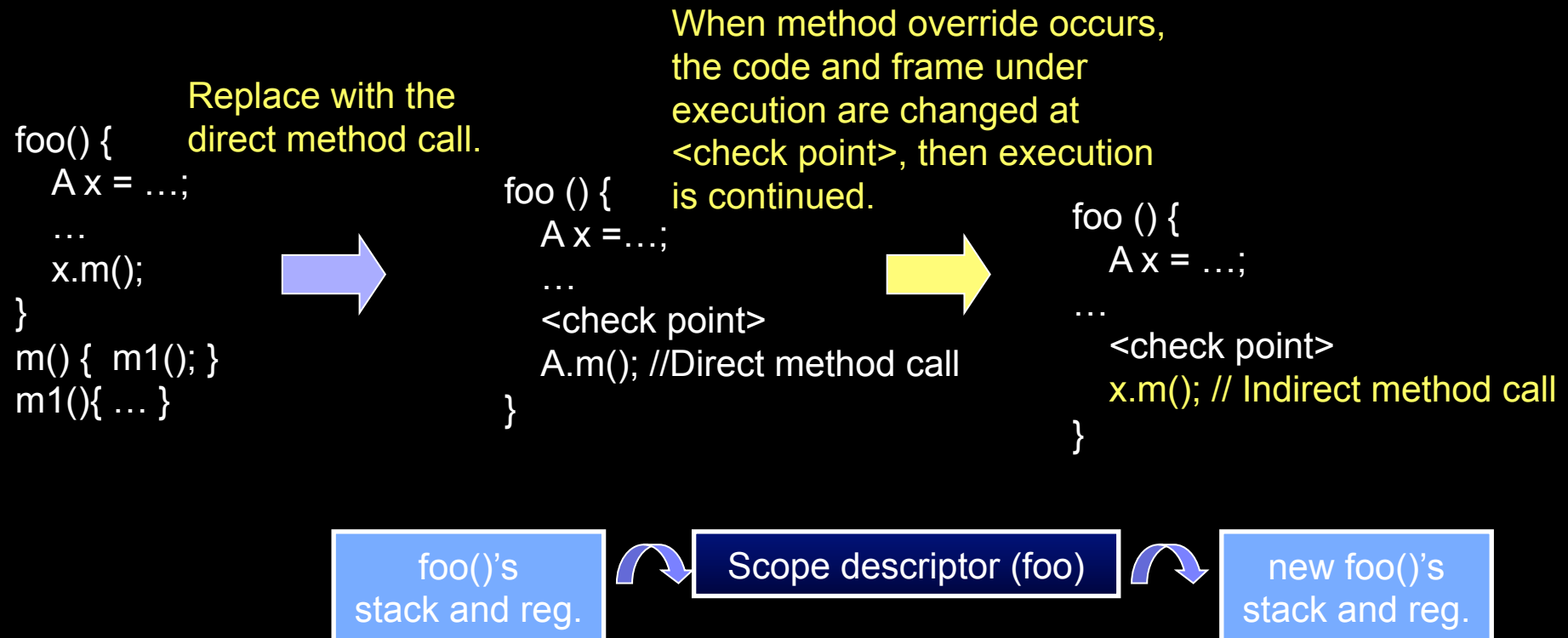
```
A x;  
...  
x.m(); // overridden  
...
```



```
jmp dynamic_call  
after_call:  
...  
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 // Indirect method call  
  jmp after_call
```

On stack replacement

- When method override occurs, the code under execution is changed during execution, then the content of the stack is replaced and execution is continued from the middle of the changed code.

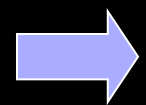


On stack replacement

- When method inlining is applied, the image before inlining should be recovered using the image stored in the stack, which is integrated into one; therefore, implementation is complicated.

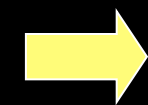
```
foo () {
  A x = ...;
  ...
  x.m();
}
m() { m1(); }
m1() { ... }
```

Apply method inlining for calling m() and m1().

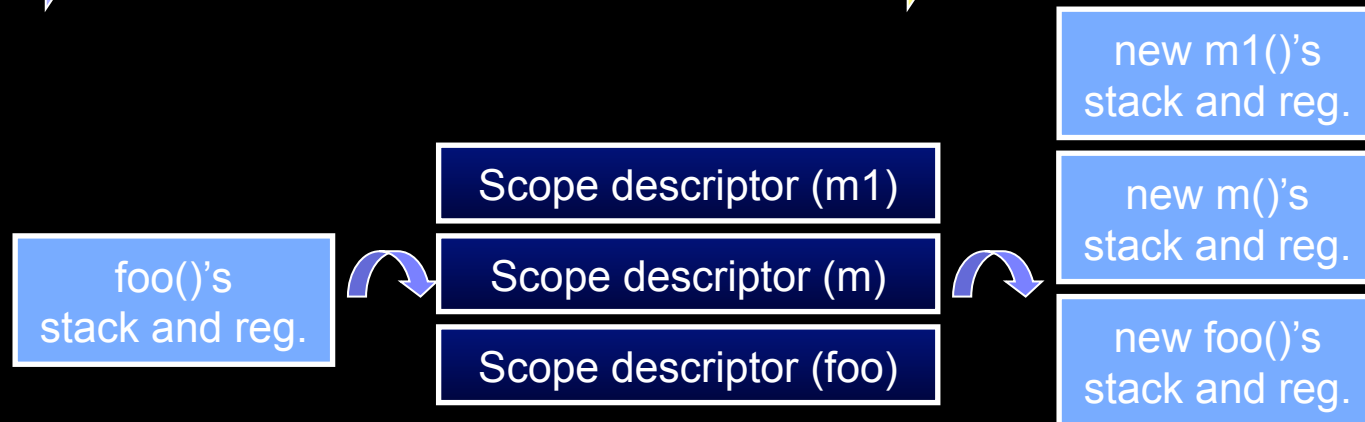


```
foo () {
  A x = ...;
  ...
  <check point>
  <inlined code m() and m1()>
}
```

When method override occurs, the code and frame under execution are changed at <check point>, then execution is continued.



```
foo () {
  A x = ...;
  ...
  x.m();
}
m() { m1(); }
m1() { ... }
```

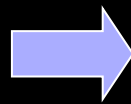


Preexistence

- If it is guaranteed that the target object of the method call remains the same as the object determined before the execution of the calling method, the virtual method call can be replaced with the direct method call or method inlining is possible.

```
foo(A x) {  
  ...  
  x.m();  
}
```

Class A
m()



r0 = <receiver object>
<inlined code of A.m()>

The definition of x reaching x.m() is that the arguments before and after the execution of foo have the same value.

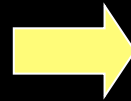
Preexistence

- When method override occurs, the replacement should be cancelled before the next method call.

```
foo(A x) {  
  ...  
  x.m();  
}
```

Class A
m()

Class C
m()

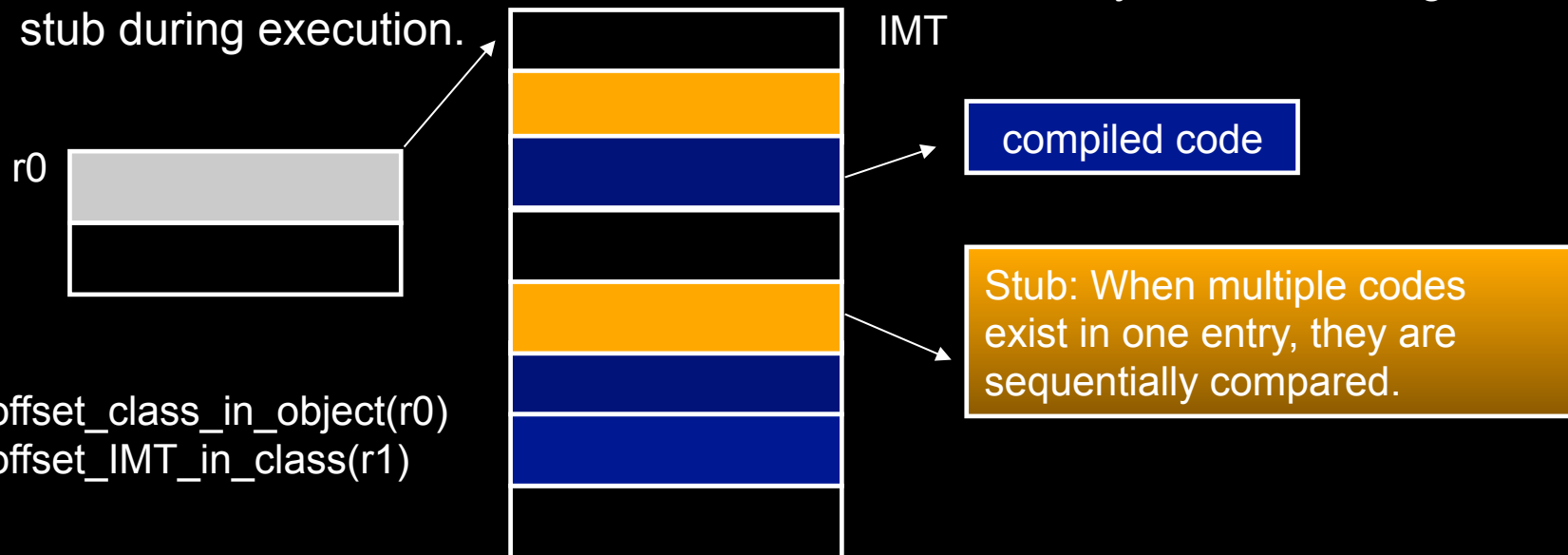


Replacement should
be cancelled before
the next foo() call.

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

Speed up of interface method call

- The search of the class that implements a call interface is replaced by referring to a hash table, i.e., the interface method table (IMT), in order to speed up the call.
 - Each class has an IMT, where a method to implement an interface class is registered.
 - For the interface method call, IMT is referred to in order to determine the call destination method.
 - In the case of hash collision, the collision is resolved by a search using stub during execution.



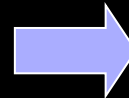
Speed up of object reference

- Stack allocation by escape analysis
- Scalar replacement

Stack allocation by escape analysis

- If the activation of an object is closed in a certain method, the method is allocable to the stack instead of the heap.

```
Class A { int f;}  
static int z;  
int foo() {  
    A x = new A();  
    z = x.f; // x is not substituted to an instance class variable.  
    bar(x.f); // x is not an argument of the method call.  
    return z; // x is not a return value of the method.  
}
```



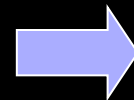
Object x is allocable to the stack.

Scalar replacement

- If the header of an object is out of use, the object allocated to the stack is replaceable with a simple variable.
 - Instructions to use object header virtual method call, type check, synchronization

```
Class A { int f;}
static int z;
int foo() {
    A x = new A();
    z = x.f;
    bar(x.f);
    return z;
}
```

Object x is allocable to a stack.



```
Class A { int f;}
static int z;
int foo() {
    t = 0;
    z = t;
    bar(t);
    return z;
}
```

Speed up of synchronization

- Speed up of synchronization operation
- Elimination of synchronization by escape analysis

Speed up of synchronization operation

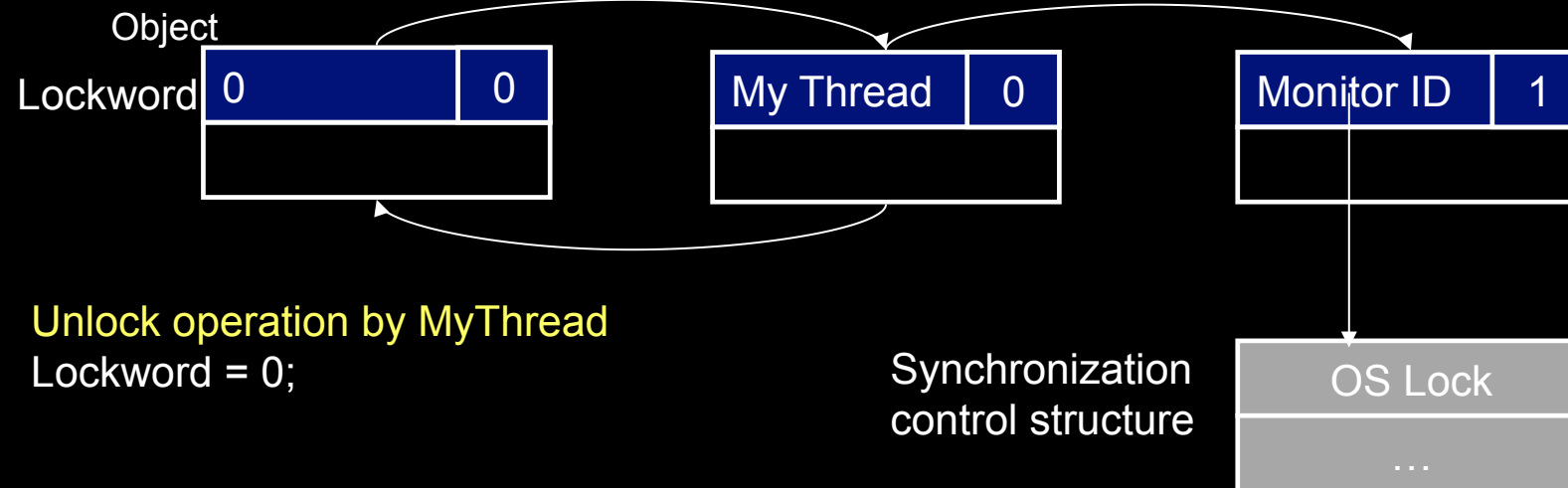
- When synchronization between threads does not result in collision, use the high-speed synchronization instruction (compare and swap) of CPU, instead of the synchronization of OS.

Lock operation by MyThread

CompareAndSwap(Lockword, 0, MyThread)

Lock operation by OtherThread

(Collision of Lock)



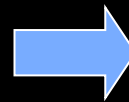
Elimination of synchronization by escape analysis

- When an object does not escape from a certain method, the synchronization related to the object can be eliminated.
 - For new Java memory model, refer to JSR 133.

```

Class A { int f;}
static int z;
int foo() {
    A x = new A();
    synchronized (x) {
        z = x.f; // x is not substituted to an instant class variable.
    }
    bar(z); // x is not an argument of the method call.
    return z; // x is not a return value of the method.
}

```



Synchronization
related to object x
can be eliminated.

```

Class A { int f;}
static int z;
int foo() {
    A x = new A();
    synchronized (x) {
        z = x.f;
    }
    bar(z);

    return z;
}

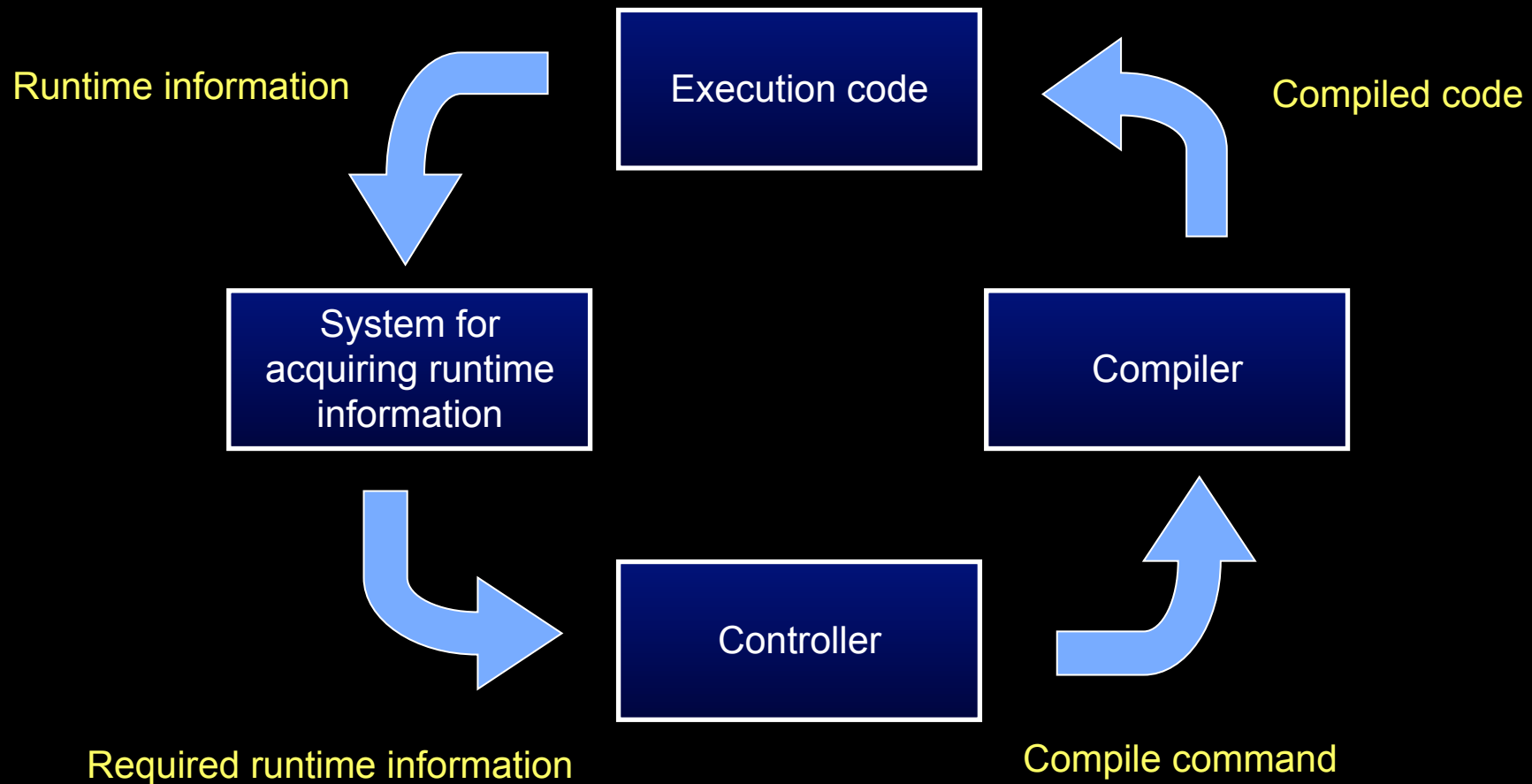
```

Optimization specific to Java

- Structure of Java system
- Conventionally known optimization
- Optimization specific to Java
- **Optimization using runtime information**
 - Framework for using runtime information
 - Method for acquiring runtime information
 - Usable runtime information
 - Method of optimization
- References

Framework for using runtime information

- The premise is that the system is recompilable.



Method for acquiring runtime information

- Sampling - detection of the method and code, which are frequently executed
 - Record log when the execution proceeds to predetermined points (such as entrance of method, at the timing of jump to the beginning of a loop) in the compiled code.
 - Obtain the execution address in the interruption handler using OS timer interruption.
- Instrumentation - detection of value under execution
 - Generate codes to record the value in the compiled codes.
- Hardware performance monitor counter - detection of cache miss
 - Read the value (such as cache miss) of the performance monitor provided by CPU.

Usable runtime information

- Sampling
 - Methods frequently executed
 - Execution path in the method frequently executed
 - Call stack with high rate of exception occurrence
- Instrumentation
 - Argument transferred to method
 - Number of loop iterations
- Hardware performance monitor counter
 - Load instruction in which cache miss occurs and the corresponding address are monitored.

Method of optimization

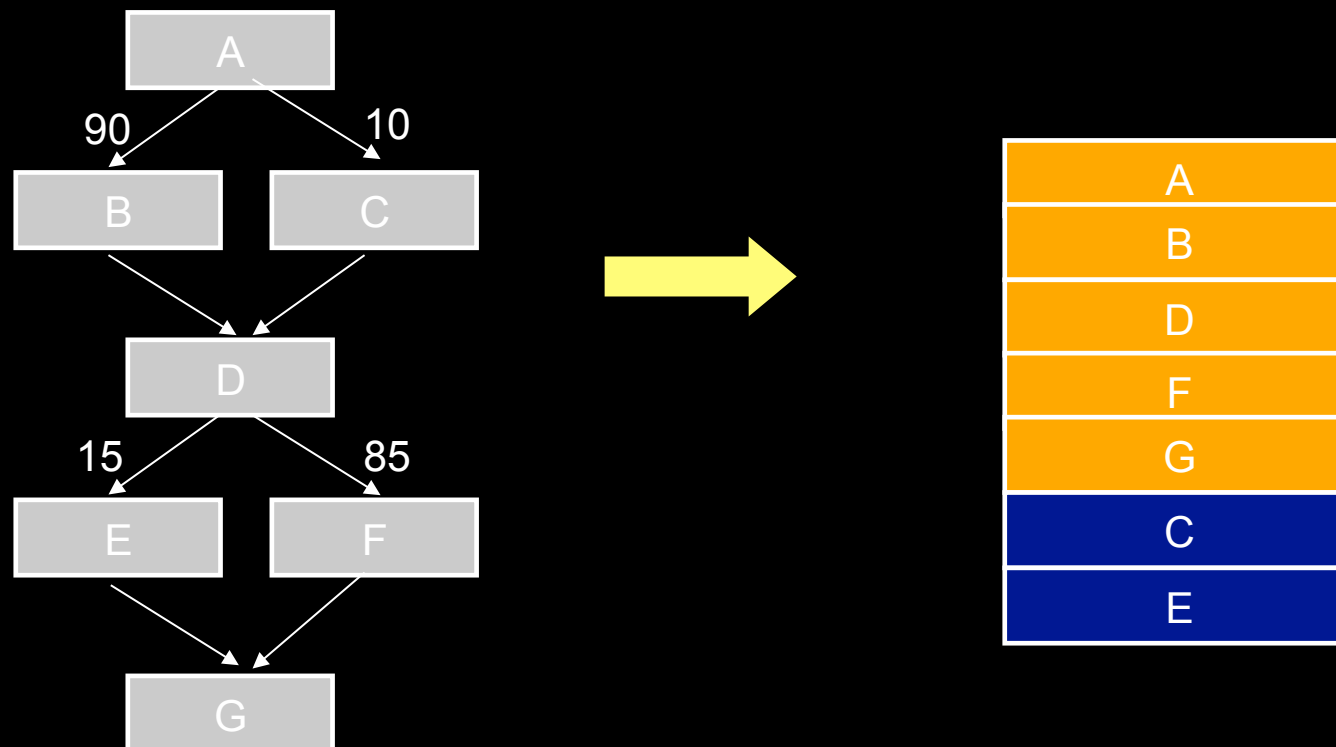
- Re-compile
- Use frequency information of path
 - Method inlining
 - Code reordering
- Specialization
- Prefetch
- Relocation of objects

Re-compile

- The method frequently executed is recompiled by increasing the level of optimization.
 - Apply time-consuming optimization.
 - Method inlining is carried out along the frequently executed call path.
 - Method inlining is carried out along the call path with frequent exception occurrence.
 - Reduction in overhead for exception processing

Code reordering

- BB codes frequently executed are stored in the neighborhood.
 - Reduce cache miss
 - Reduce branch estimation miss



Specialization, Customization

- Specialized codes are generated on the basis of a constant obtained by instrumentation.
 - Integer, real number
 - Constant propagation
 - Dead code elimination
 - Type of object
 - Elimination of type check
 - More accurate class hierarchy analysis by determining the receiver type
 - Length of array
 - Elimination of array index check
 - Simplification of loop and loop inlining
- (Slow) generic code should also be prepared.

Prefetch

- Prior to load instruction inducing data cache or TLB (translation lookaside buffer ; buffer used for conversion from logical address to physical address) miss, execute data load instruction to prevent the miss.

TokenLoop:

```
for (int i = 0; i < tv.ptr; i++) {  
    Token tmp = tv.v[i];  
    tmpNext = spec_load (&tv.v[i+1]); // tmp of next iteration  
    prefetch (tmpNext + offset(facts)); // prefetch of facts  
    prefetch (tmpNext + offset(facts) + c); // prefetch of facts array  
    for (int j = 0; j < t.size; j++)  
        if (!t.facts[j].equals(tmp.facts[j]))  
            continue TokenLoop;  
    return tmp;  
}
```

Slide from [Inagaki03]

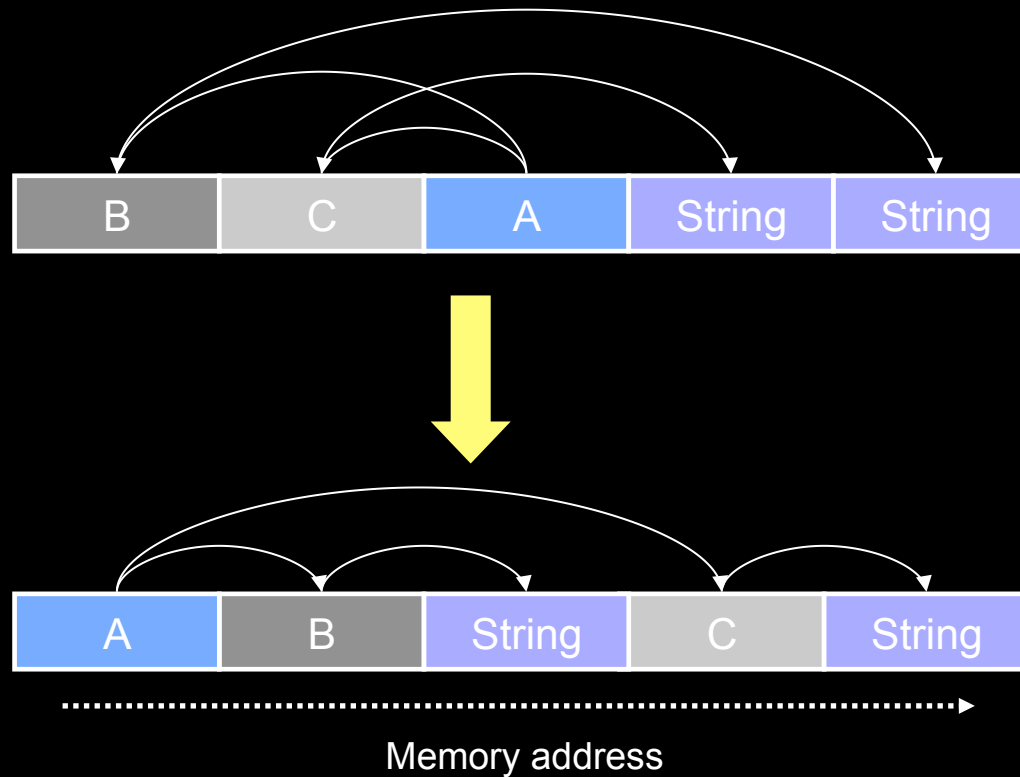
Relocation of objects

- Relocate related objects in the neighbourhood during GC to reduce cache miss.
 - The optimal location depends on the sequence of access.

```
Class A {  
  B y;  
  C z;  
}
```

```
Class B {  
  String s;  
}
```

```
Class C {  
  String s;  
}
```



Our research outcome

■ JIT compiler

- Method invocation optimization[OOPSLA00][JVM02]
- Exception optimization[ASPLOS00][OOPSLA01][PACT02]
- Profiling based optimization[JG00][PLDI03][PACT03]
- Float optimization[JVM02][ICS02]
- 64bit architecture optimization[PLDI02]
- Register allocation[PLDI02]
- Data prefetch[PLDI03]
- Instruction Scheduling[CGO03]
- Compiler overview[JG99][IBMSJ00][OOPSLA03][IBMSJ04]

■ Runtime systems

- Fast lock[OOPSLA99][OOPSLA02][ECOOP04][PACT04]
- Fast interpreter[ASPLOS02]

Please visit

http://www.research.ibm.com/trl/projects/jit/pub_int.htm

Special thanks to

- Toshio Nakatani
- Hodeaki Komatsu
- Tamiya Onodera
- Toshio Suganuma
- Kiyokuni Kawachiya
- Takeshi Ogasawara
- Motohiro Kawahito
- Toshiaki Yasue
- Kazunori Ogata
- Akira Koseki
- Tatsushi Inagaki
- Osamu Goda
- Mikio Takeuchi
- Kazuhiro Konno
- Mikio Tabata
- Hiroyuki Momose

Thank you very much

References

■ Structure of Java system

- IBM DK
 - Suganuma et al. Overview of the IBM Java Just-In-Time Compiler, IBM Systems Journals, 39(1), 2000.
 - Ishizaki et al. Effectiveness of Cross-Platform Optimizations for a Java Just-In-Time Compiler, OOPSLA, 2003.
 - Suganuma et al. Evolution of a Java just-in-time compiler for IA-32 platforms, IBM Systems Journals, 48(5), 20
- Sun HotSpot
 - Paleczny et al. The Java HotSpot Server Compiler, JVM, 2001.
- Jikes RVM
 - Alpern et al. The Jalapeno Virtual Machine, IBM Systems Journals, 39(1), 2000.
 - Fink et al. The Design and Implementation of the Jikes RVM Optimizing Compiler, 2002, available at <http://www124.ibm.com/developerworks/oss/jikesrvm/info/course-info.shtml>.
- Intel ORP
 - A-Tabatabai et al. Fast, Effective Code Generation in a Just-In-Time Java Compiler, PLDI, 1998.
 - Cierniak et al. Practicing judo: Java under dynamic optimizations, PLDI, 2000.
 - A-Tabatabai et al. The StarJIT Compiler: A Dynamic Compiler for Managed Runtime Environments, Intel Techn <http://developer.intel.com/technology/itj/2003/volume07issue01/>.

■ Conventionally known optimization

- Intel ORP
 - Aho et al. Compilers: Principles, Techniques, and Tools, ISBN 0201101947.
 - (和訳) コンパイラ・II, ISBN4-7819-0585-4, ISBN4-7819-0586-2.
 - Muchnick. Advanced compiler design and implementation, ISBN 1-55860-320-4, 1997.
 - 中田. コンパイラの構成と最適化, ISBN 4-254-12139-3, 1999.
- Partial redundancy elimination
 - J. Knoop, O. Ruthing, and B. Steffen. Optimal code motion, TOPLAS, 1995.
- Method inlining
 - Suganuma et al. An Empirical Study of Method Inlining for a Java Just-In-Time Compiler, JVM, 2002.
- Register allocation
 - Chaitin. Register allocation and spilling via graph coloring, Compiler Construction, 1982.
 - Poletto et al. Linear scan register allocation, TOPLAS, 1999.

References

■ Optimization specific to Java

- Elimination of exception check
 - Kawahito et al. Effective Null Pointer Check Elimination Utilizing Hardware Trap, ASPLOS, 2000.
 - Odaïra et al. Partial redundancy elimination beyond exceptional dependency, Information Processing Society of Japan, 2004.
 - Midkiff et al. Optimizing array reference checking in Java programs. IBM Systems Journal, 37(3), 1998.
 - Gupta et al. Optimizing array bound checks using flow analysis, LOPLAS, 1993.
 - Kawahito et al. Eliminating Exception Checks and Partial Redundancies for Java Just-in-Time Compilers. IBM Research Report RT0350, 2000.
 - Bodik et al. ABCD: Eliminating Array-Bounds Checks on Demand, PLDI, 2000.
- Type analysis
 - Palsberg et al. Object-Oriented Type Inference, OOPSLA, 1991.
 - Gagnon et al. Efficient Inference of Static Types for Java Bytecode, SAS, 2000.
- Guarded devirtualization
 - Calder et al. Reducing Indirect Function Call Overhead In C++ Programs, POPL, 1994.
 - Detlefs et al. Inlining of virtual methods, ECOOP, 1999.
 - Arnold et al. Thin Guards: A Simple and Effective Technique for Reducing the Penalty of Dynamic Class Loading, ECOOP, 2002.
- Type analysis
 - Chambers et al. Interactive type analysis and extended message splitting; optimizing dynamically-typed object-oriented programs, PLDI, 1990.
- Class hierarchy analysis
 - Dean et al. Optimization of object-oriented programs using static class hierarchy, ECOOP, 1995.
 - Bacon et al. Fast Static Analysis of C++ Virtual Function Calls, OOPSLA, 1996.
 - Sundaresan et al. Practical Virtual Method Call Resolution for Java, OOSPLA, 2000.
- Code patching
 - Ishizaki et al. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler, OOPSLA, 2000.
- On stack replacement
 - Holzle et al. Debugging Optimized Code with Dynamic Deoptimization, PLDI, 1992.
 - Fink et al. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement, CGO, 2003.
- Interface method call
 - Alpern et al. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless, OOPSLA, 2001.

References

■ Optimization specific to Java

- Speed up of synchronization
 - Bacon et al. Thin Locks: Featherweight Synchronization for Java, PLDI, 1998.
 - Onodera et al. A Study of Locking Objects with Bimodal Fields, OOPSLA 1999.
- Escape analysis
 - Choi et al. Escape Analysis for Stack Allocation and Synchronization Elimination in Java, TOPLAS, 2003.
 - Whaley et al. Compositional Pointer and Escape Analysis for Java Programs, OOPSLA 1999.
 - Lea. The JSR-133 Cookbook for Compiler Writers, 2004, available at <http://gee.cs.oswego.edu/dl/jmm/cookbook.html>.
- Other optimization
 - Pechtchanski et al. Dynamic Optimistic Interprocedural Analysis: A Framework and an Application, OOPSLA, 2001.
 - Hirzel et al. Pointer Analysis in the Presence of Dynamic Class Loading, ECOOP, 2004.

■ Optimization using runtime information

- Arnold et al. Adaptive Optimization in the Jalapeno JVM, OOPSLA, 2000.
- Arnold et al. A Framework for Reducing the Cost of Instrumented Code, PLDI, 2001.
- Arnold et al. Online Feedback-Directed Optimization of Java, OOPSLA, 2002.
- Arnold et al. A Survey of Adaptive Optimization in Virtual Machines. IBM Research Report RC23143, 2003.
- Re-compile
 - Ogasawara et al. A Study of Exception Handling and Its Dynamic Optimization in Java, OOPSLA, 2001.
- Code reordering
 - Pettis et al. Profile-Guided Code Positioning, PLDI, 1990.
- Specialization
 - Chambers et al. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language, PLDI, 1989.
 - Dean et al. Selective specialization for object-oriented languages, PLDI, 1995.
 - Suganuma et al. A Dynamic Optimization Framework for a Java Just-In-Time Compiler, OOPSLA, 2001.
- Prefetch
 - Inagaki et al. Stride Prefetching by Dynamically Inspecting Objects, PLDI, 2003.
- Relocation of objects
 - A-Tabatabai et al. Prefetch Injection Based on Hardware Monitoring and Object Metadata, PLDI, 2004.