IBM Research, Tokyo Research Laboratory

Java Just-In-Time コンパイラ

石崎　一明　Kazuaki Ishizaki
日本アイ･ビー･エム　東京基礎研究所

要旨

- 以下の内容について、コンパイラの初心者が理解しやすい内容としながら、コンパイラの知識がある研究者にも興味が持てるチュートリアルに。
  - Java Just-In-Timeコンパイラの構成
  - 従来から知られている最適化
  - Java言語特有の最適化
  - 実行時情報を利用した最適化

- 最初に
  - 幅広く最適化を紹介したいので、個々の最適化のアルゴリズムは省きます。
  - 時間が足りなかったら or 深い理解は、後ろの文献で勉強して下さい。

目次

- Java処理系の構成
- 従来から知られている最適化

- Java言語特有の最適化
- 実行時情報を利用した最適化

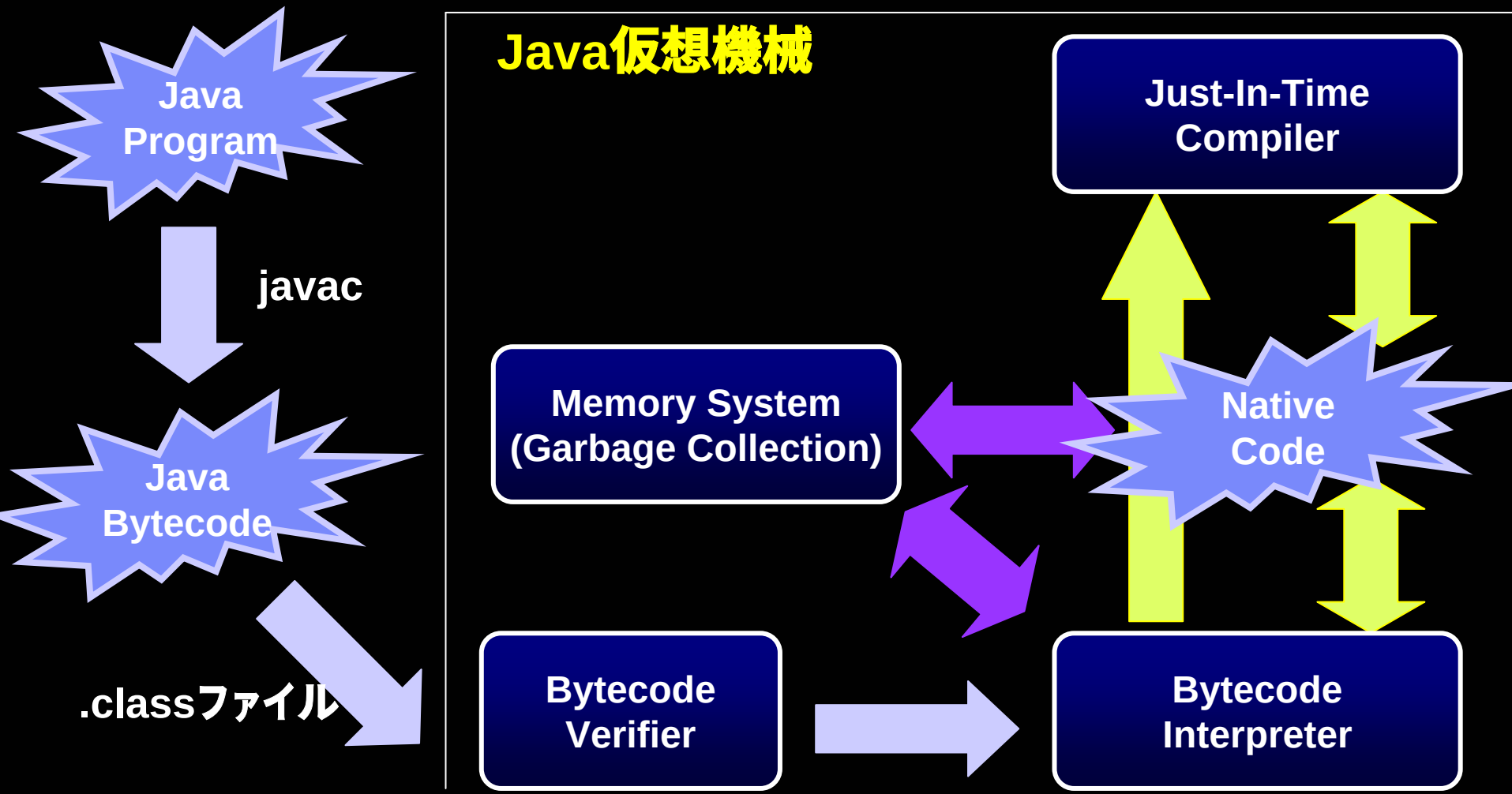
- 文献リスト
  - 自習用のおまけ

Java処理系の構成

- **Java処理系の構成**
  - Java仮想機械の構成
  - Java Just-In-Time(JIT) コンパイラの構成
- 従来から知られている最適化
- Java言語特有の最適化
- 実行時情報を利用した最適化

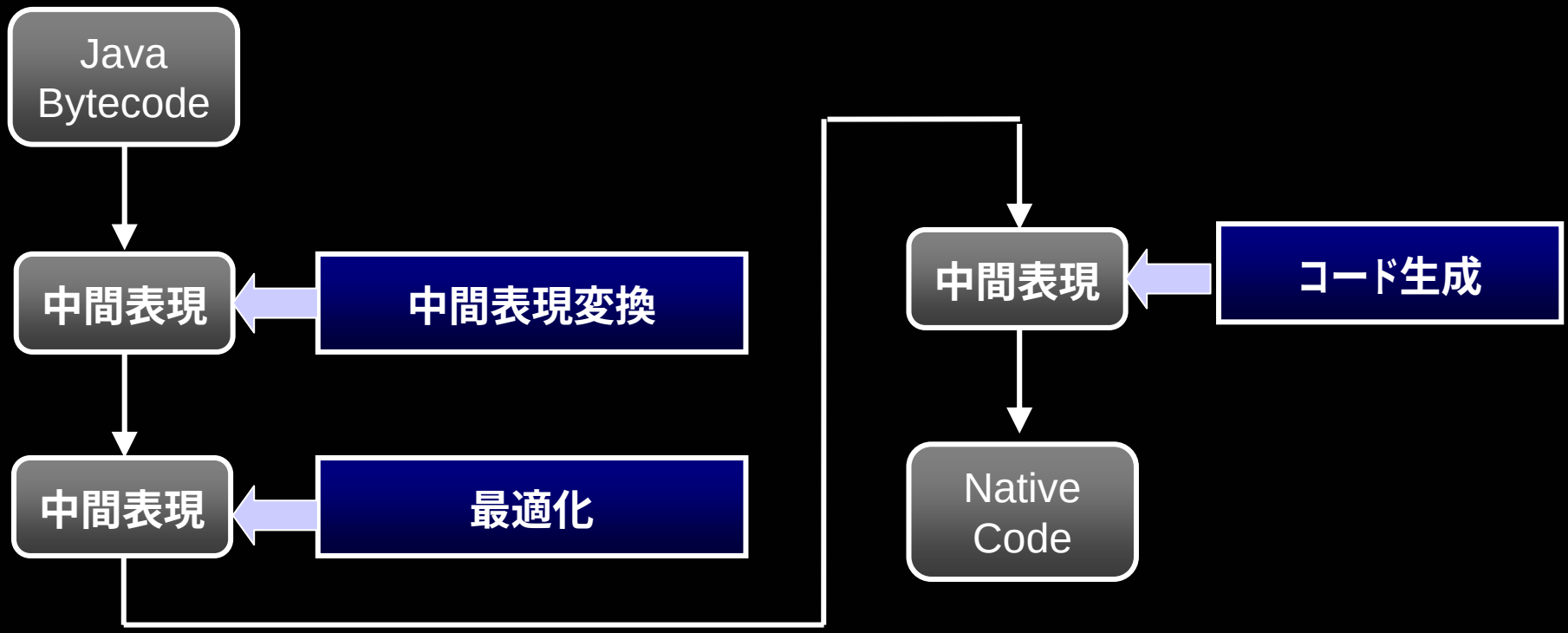
Java仮想機械(Java Virtual Machine)の構成

- 現在のJVMではJITコンパイラは選択的に呼び出される



Java Just-In-Timeコンパイラの構成

- 中間表現変換
- 最適化
- コード生成



従来から知られている最適化

- Java処理系の構成
- 従来から知られている最適化
  - 最適化とは？
  - 最適化の方法
- Java言語特有の最適化
- 実行時情報を利用した最適化
- 文献リスト

最適化(Optimization)とは？

- 目的
  - 実行時間を短縮する
    - このチュートリアルでは実行時間の最適化について
  - メモリ使用量を小さくする

- 大前提
  - 最適化前と最適化後のコードで、プログラムの意味を変えないこと
    - メソッド外部から観測可能な値を変えない
      rは観測可能、a･bは観測不能
    - 例外の発生順序を変えない
      割り算も、余りも例外を発生する可能性がある

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

最適化の方法

- どこを？
  - 基本ブロック内
    - 基本ブロック (Basic Block, BB) ＝制御の分岐も合流もない命令列
  - 基本ブロック間
    - データフロー解析 (Dataflow Analysis) を使う
- どうやって？
  - 命令の実行回数を減らす
  - 速い命令を使う

命令の実行回数を減らす (1)

- コンパイル時に前もって計算
  - 定数の畳み込み
  - 定数伝搬
- 冗長な命令の除去
  - 不要命令除去
  - 複写伝搬
- 計算結果の再利用
  - 共通部分式の削除
  - 部分冗長性の削除

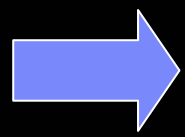
命令の実行回数を減らす (2)

- メソッドの変形
  - コード上昇
- プログラムの特殊化・専用化
  - メソッド展開
  - 末尾複製

定数の畳み込み (constant folding)

- 定数計算をコンパイル時に行う。
  - Javaでは浮動小数点の精度や例外条件 (-32768/-1の答えは？) に考慮することが必要

```
…
x = 2 * 3;
…
```

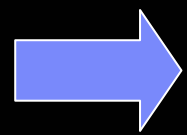


```
…
x = 6;
…
```

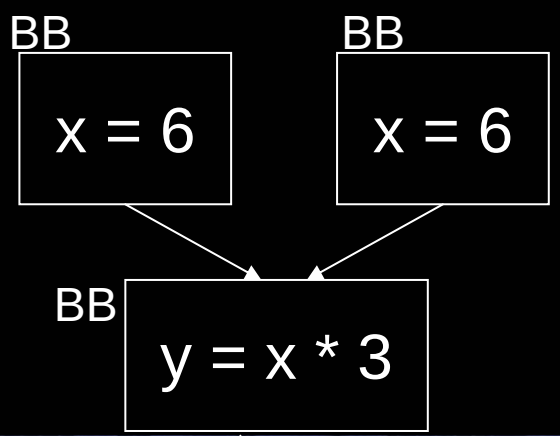
定数伝搬 (constant propagation)

- 定数値を、メソッド中に伝搬させる。

```
x = 6;
…        // この間にxの定義がない
y = x * 3;
```



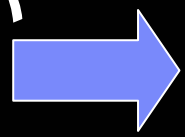
```
x = 6;
…
y = 18;
```



不要命令除去 (dead code elimination)

- メソッド中で使われない値を計算する命令を削除する。
- 実行されない文を削除する。

```
…
x = a + b;    // xはこの後使われない
…
if (false) {
    y = 1;
}
```

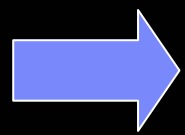


```
…
~~x = a + b;~~
…
~~if (false) {~~
~~    y = 1;~~
~~}~~
```

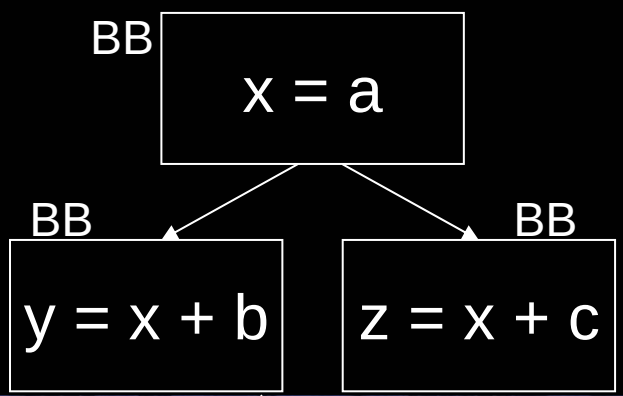
複写伝搬 (copy propagation)

- 単純な代入命令を、右辺のオペランドに伝搬させる。

```
x = a;   // xの定義はここだけ
…        // この間にaの定義がない
y = x + b;
```



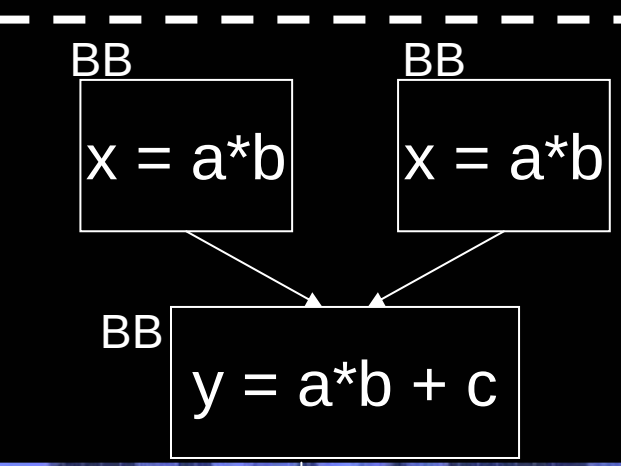
```
x = a;
…
y = a + b;
```



共通部分式の削除 (common subexpression elimination)

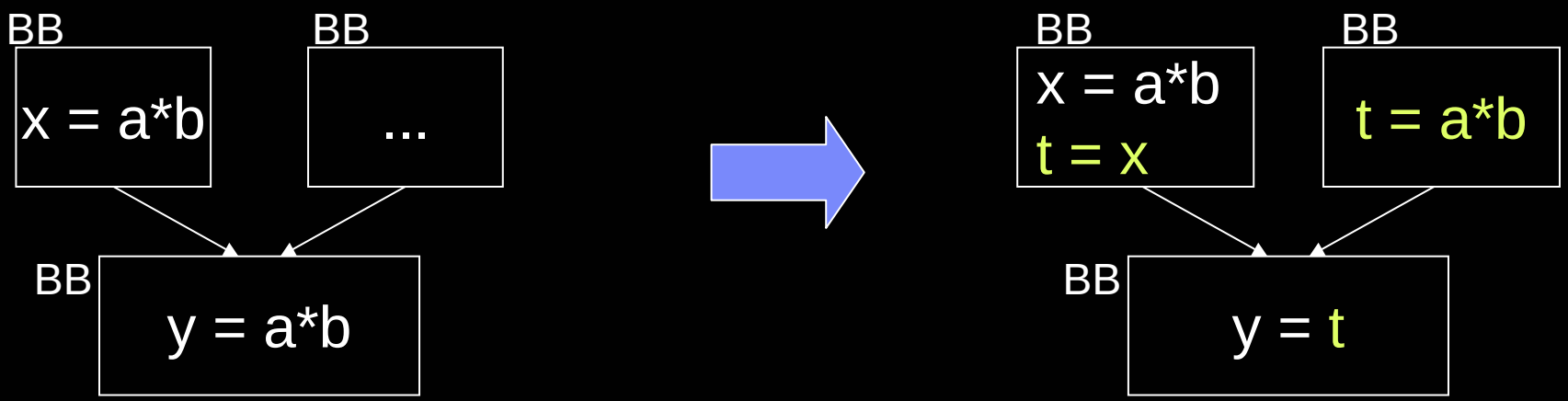
- 一度計算された式の値を再利用する

```
x = a*b;                                    x = a*b;
…          // この間にx,a,bの定義がない      …
y = a*b + c;                                y = x + c;
```



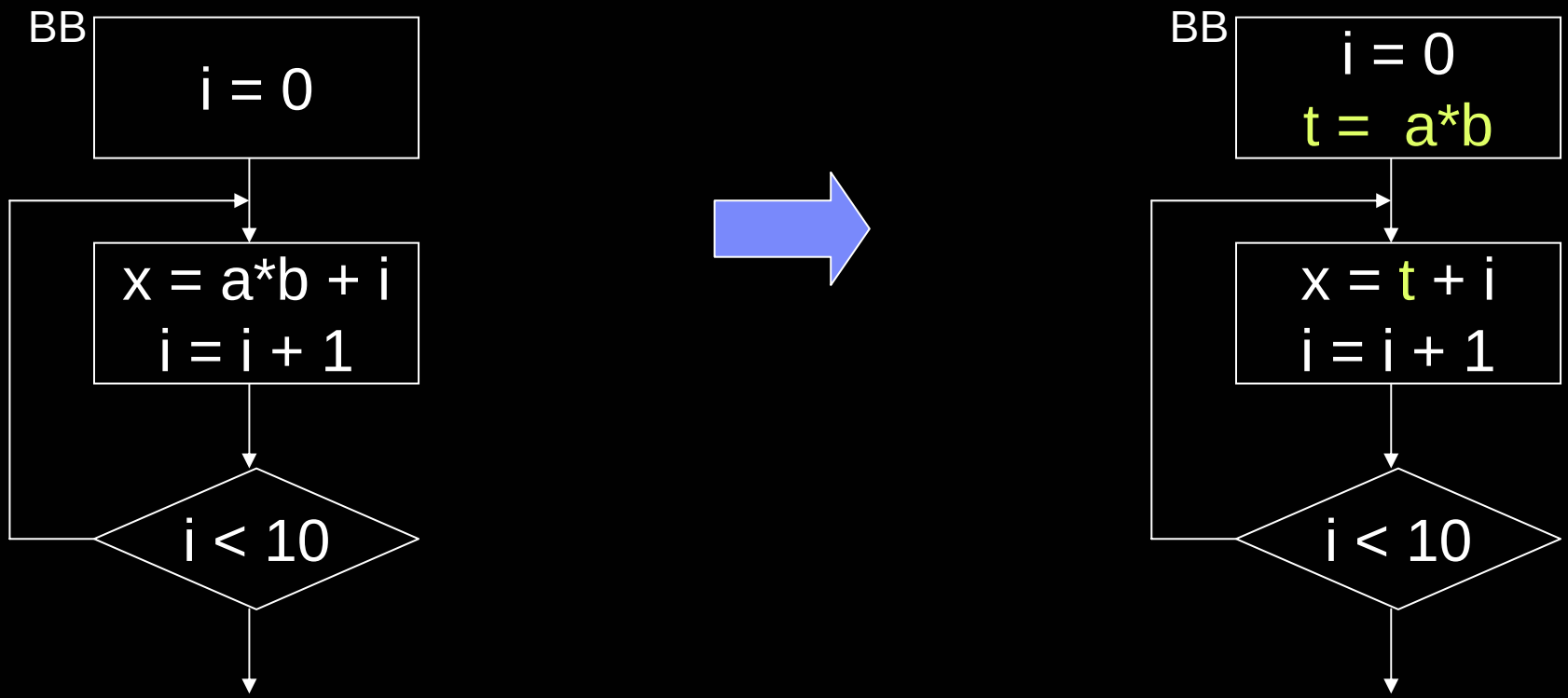
部分冗長性の削除 (partial redundancy elimination)

- 一度計算された式の値を再利用するために、式が存在しなかった実行経路へ、式を実行順で前へ移動する。
  - Javaでは、例外を伴う式 (割り算、余り、その他) の移動に注意



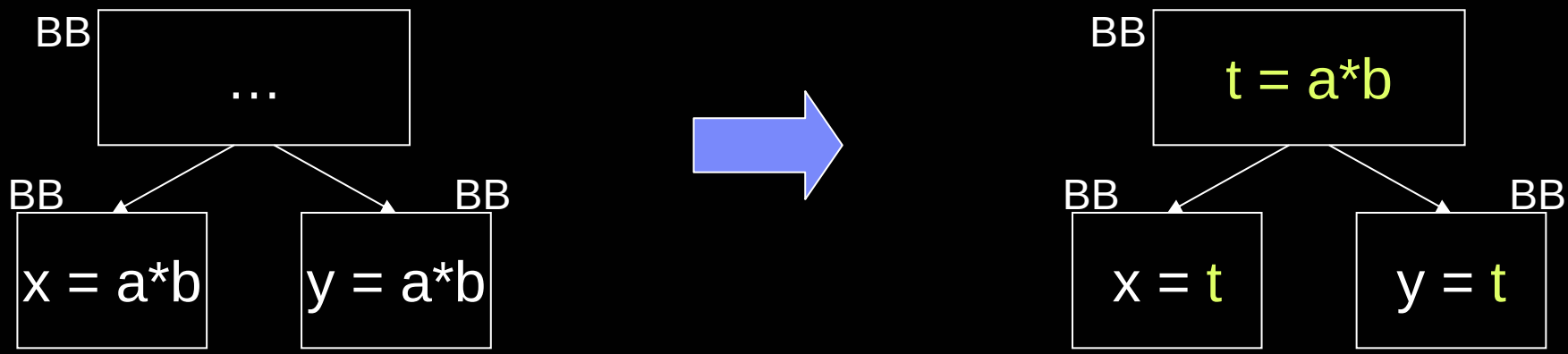
ループ内不変値移動 (loop invariant code motion)

- ループ内で定数の値をループ外へ移動する。
  - Javaでは、例外を伴う式の移動に注意



コード上昇 (code hoisting)

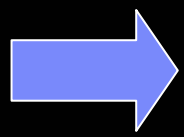
- 式を実行順で前へ移動する。
  - 片方の経路でしか実行されない式でも移動可能
  - Javaでは、例外を伴う式の移動に注意



メソッド展開 (method inlining)

- メソッド呼び出しの場所に、メソッド本体を展開する。
  - 最適化の適用範囲が大きくなる。
  - 引数の受け渡しや、レジスタ待避・回復処理を削除出来る。

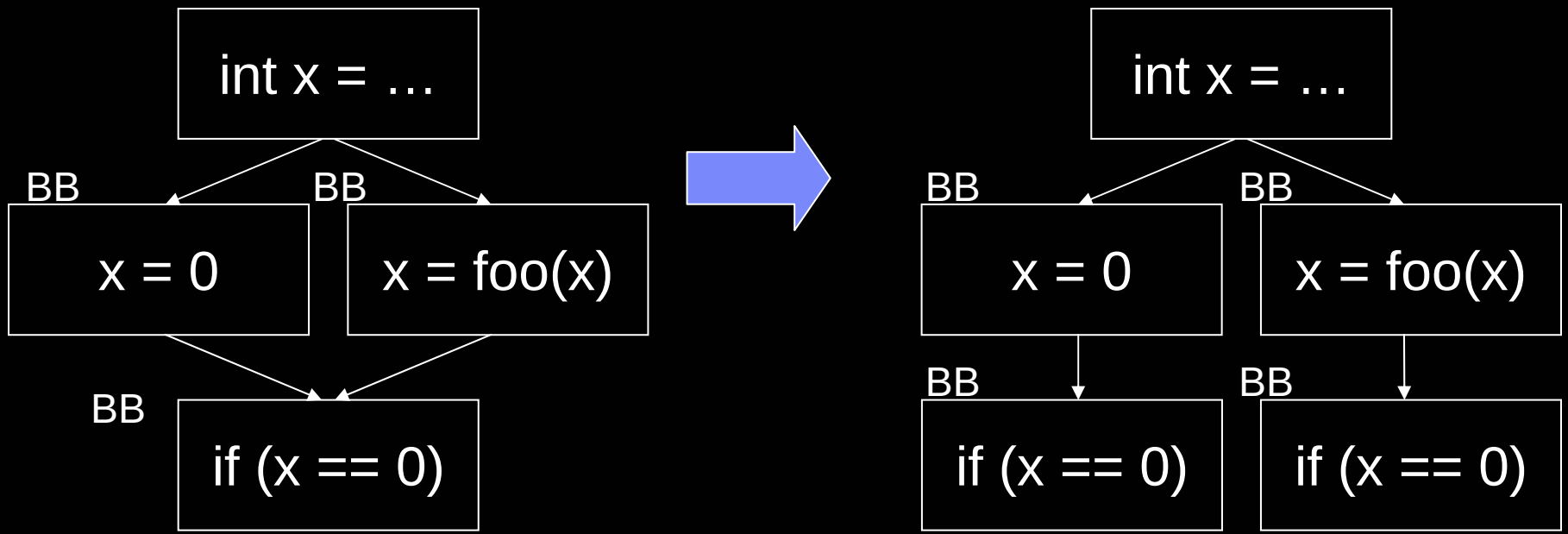
```
...
x = foo(a)
...

int foo(int i) {
   return i *i;
}
```



```
...
x = (a * a);
...

int foo(int i) {
   return i *i;
}
```

末尾複製 (tail duplication, splitting)

- 制御の合流部分以降を複製して、専用コードを作る。
  - 変数の性質が一意に決定する。



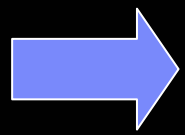
速い命令を使う

- 命令強度軽減
- スカラ置き換え
- レジスタ割付

命令強度軽減 (strength reduction)

- 同じ演算を、CPUが持つ速い命令で置き換える。
  - 例えば、多くのCPUでは、乗算・除算よりシフト命令の方が高速である。

```
…
x = a * 8;  // 2のべき乗による乗算
…
```

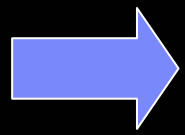


```
…
x = a << 3;
…
```

スカラ置き換え (scalar replacement)

- メモリアクセス命令を、単純変数を使った命令に置き換える。
  - 後のレジスタ割付で、単純変数がレジスタに割り付けられることを期待

```
class A {
  int f;
  int g;
}

A a = new A();
a.f = 1;
a.f = a.f * 2;
```

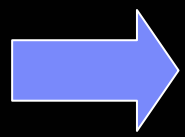


```
class A {
  int f;
  int g;
}

A a = new A();
t = 1;
t = t * 2;
a.f = t;
```

レジスタ割付 (register allocation)

- メソッド中の単純変数を、プロセッサが持つレジスタ資源にできるだけ割り付ける。
  - メソッド全体を見て　Graph coloring register allocator
  - メソッドの前から順に　Linear scan register allocator

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



レジスタは2個
aとcに割り当てる

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

Java言語特有の最適化

- Java処理系の構成
- 従来から知られている最適化
- Java言語特有の最適化
  - Java言語特有の性能上の問題
  - Java言語特有の最適化
- 実行時情報を利用した最適化
- 文献リスト

Java言語特有の性能上の問題

- プログラムの安全性を保証するための例外検査
- プログラムの安全性を保証するための型検査
- 多様性 (polymorphism) の導入による、仮想メソッド呼び出し
- カプセル化 (encapsulation) による、オブジェクト内のフィールドへの参照
- 言語が提供する同期操作

Java言語特有の最適化 (1)

- 例外検査の削除
  - 単純に冗長な例外検査削除
  - 部分的に冗長な例外検査削除
  - 逆最適化
  - ループバージョニング
- 型検査・型変換の削除
  - 形解析

Java言語特有の最適化 (2)

- 仮想メソッド (virtual method) 呼び出しの高速化
  - Guarded devirtualization
  - Direct devirtualization
- オブジェクト参照の高速化
  - 脱出解析によるスタック割付
  - スカラ置き換え
- 同期の高速化
  - 同期操作の高速化
  - 脱出解析による同期操作除去

例外検査の削除

- プログラムの安全性を保証する
  - nullcheck – オブジェクトがnullでない
  - bndcheck – 配列のインデックスが配列の範囲内である
  - その他、0での割り算チェックなど
- 例外検査に依存する後続の命令は、例外検査より前に移動出来ない

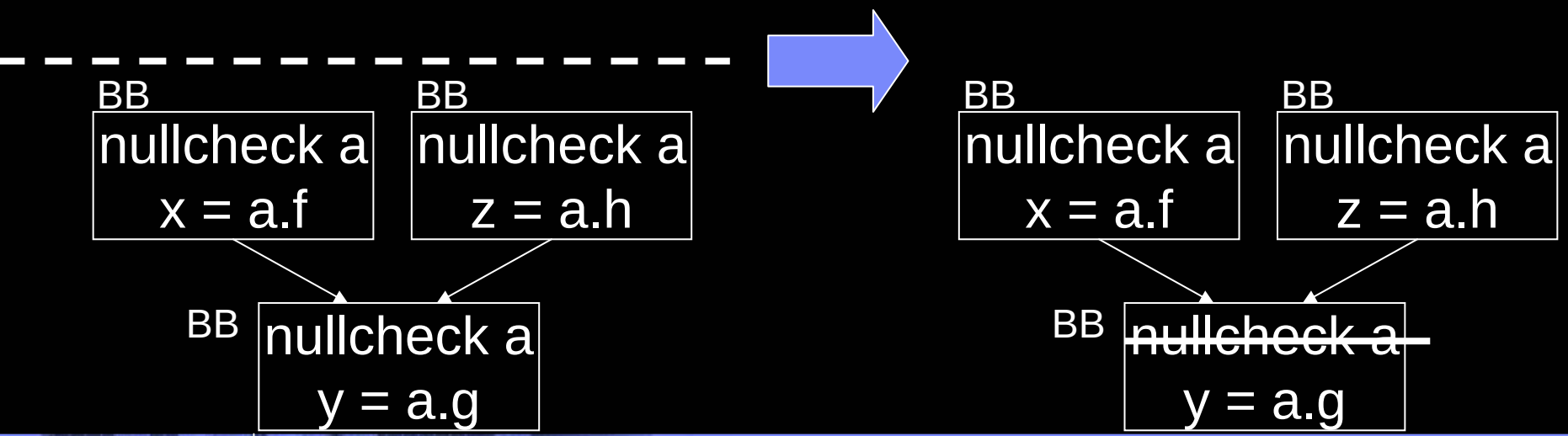
- 例外検査の削除
  - 例外検査のための、比較命令、分岐命令の削除
  - 命令の移動機会を増やす

単純に冗長な例外検査削除

- 同じ変数に対する、同一の例外検査を削除する。

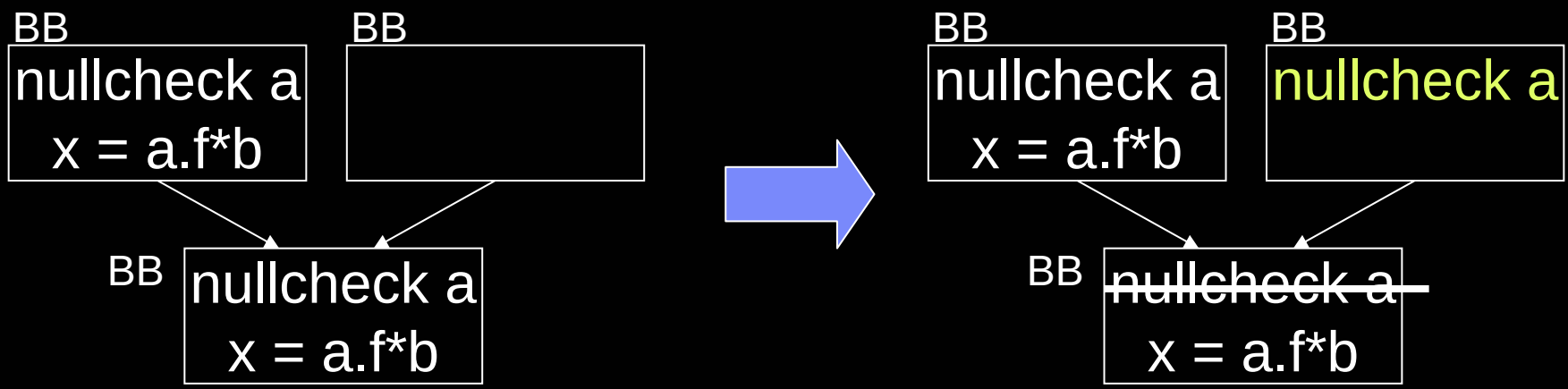
```
nullcheck a
x = a.f;

…          // この間にaの定義がない
nullcheck a
y = a.g;
```

```
nullcheck a
x = a.f;

…
~~nullcheck a~~
y = a.g;
```



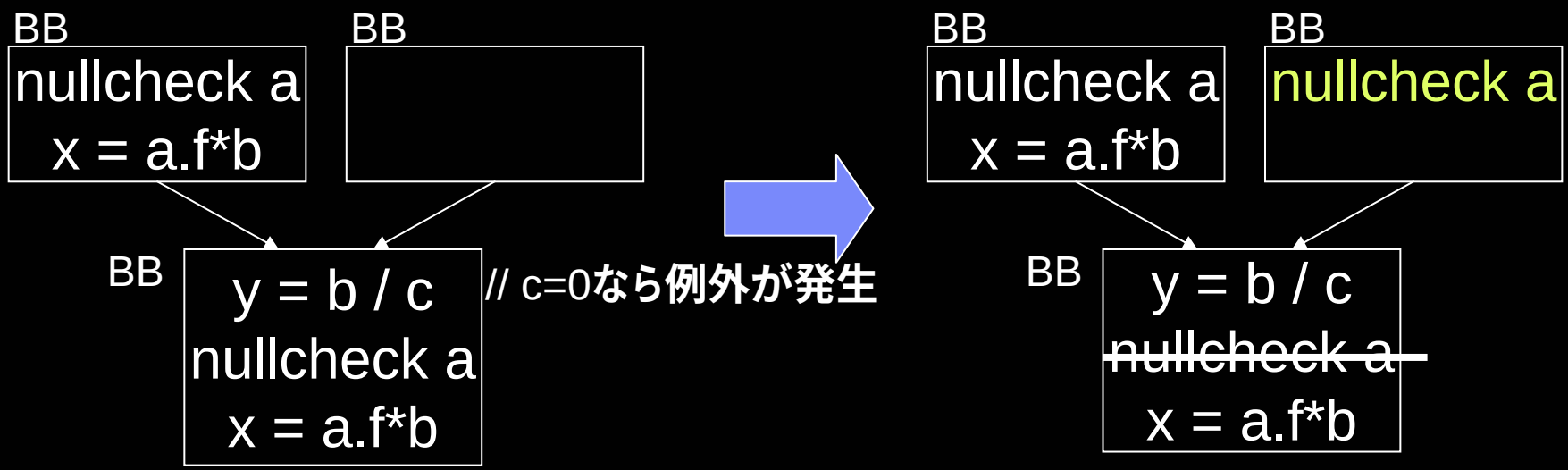
部分的に冗長な例外検査削除

- 部分冗長削除の応用。
  - メソッド外部から観測可能な状態を変更する（副作用を持つ）命令（例外検査命令・メソッド呼び出し命令など）越えて、例外検査命令を移動出来ない。
  - 複数回適用することで、命令移動を伴う他の最適化を適用する機会を広げる。



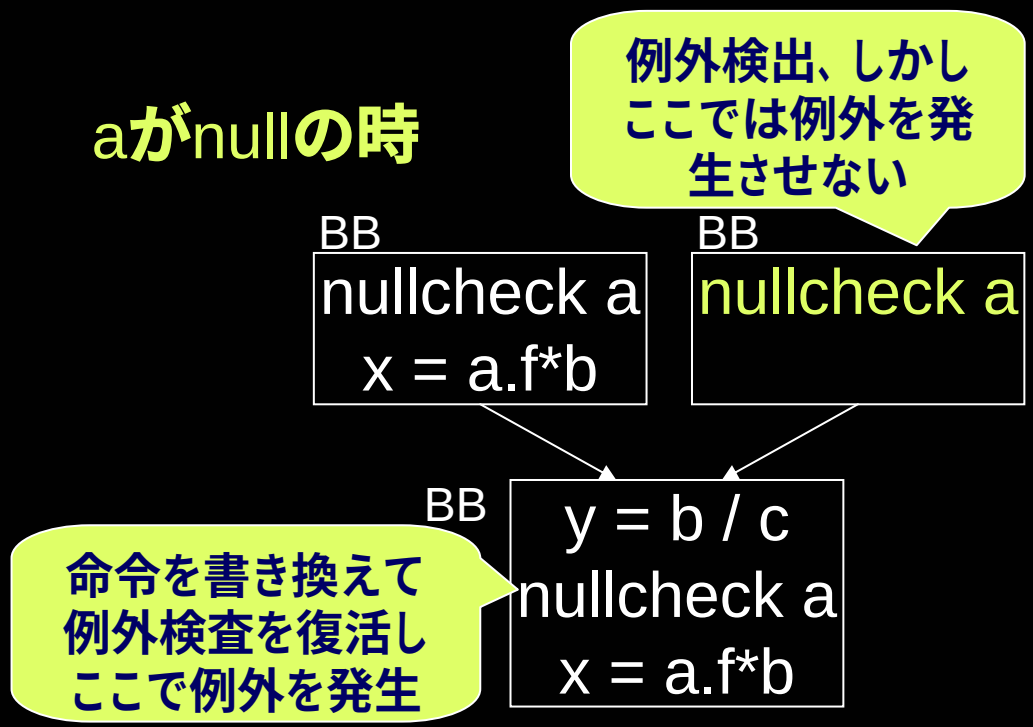
部分的に冗長な例外検査削除＋逆最適化

- 例外依存関係を無視して、部分冗長削除で例外検査命令を移動する。



部分的に冗長な例外検査削除＋逆最適化

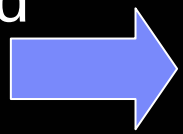
- 例外依存関係を無視して、部分冗長削除で例外検査命令を移動する。
- 移動した例外検査命令で、実際に例外を検出したときには、移動前の位置で例外を発生させる。



ループバージョニング

- ループ内でアクセスされる添え字の範囲が配列の長さの範囲内にあるか検査をループ実行直前に投機的に行い、
  - 検査が成功した場合は例外検査が除去されたループを実行
  - 検査が成功しない場合は例外検査があるループを実行

```
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;
  }
}
```

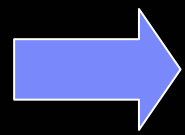
型検査・型変換の削除

- 型解析

型解析 (type analysis)

- オブジェクト参照を行うオペランドの型を計算して、肩検査や型変換のクラスもしくはサブクラスである場合には、型検査・型変換を削除可能。
  - new, instanceof, checkcast, 引数、戻り値、等を型の種にする

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



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

上のif文内のinstanceofで、
xはAもしくはAのサブクラスであることが保証される。

仮想メソッド呼び出しの高速化

- Guarded devirtualization
  - Class test
  - Method test
  - Polymorphic Inline Cache
- Direct devirtualization
  - 型解析
  - On stack replacement
  - Code patching
  - Preexistence
- インタフェースメソッド呼び出しの高速化
  - Interface Method Table

仮想メソッド呼び出し

- 仮想メソッド呼び出し (invokevirtual, invokeinterface) が多用される。
  - 呼び出し先が決定出来ないメソッド呼び出しによって、最適化が妨げられる。
- カプセル化により、1つのメソッドの大きさが比較的小さい。
  - 最適化の範囲が狭まる

➔メソッド呼び出し先を決定する最適化 (devirtualization) が必要。

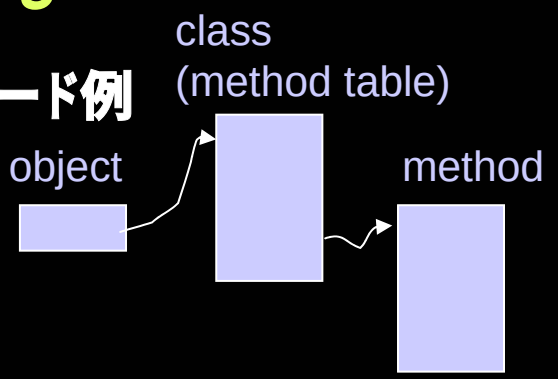
Class test and Method test

- 直接メソッド呼び出しもしくはメソッド展開されているコードを呼び出してよいか、比較を行う（guard）。

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

Javaでの仮想メソッド呼び出しのコード例

```
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
}
```

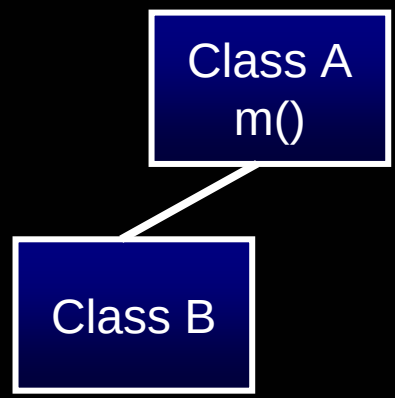
Method test

```
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

- Method testのほうがロードが1つ多いが、精度が高い。
  - 下記のような場合、xにBのインスタンスが渡されると、
    - Class test (Aでguardされていると仮定) では、A != Bということで、仮想メソッド呼び出しが実行される。
    - Method testでは、&A.m() == &B.m()ということで、直接メソッド呼び出しが実行される。

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

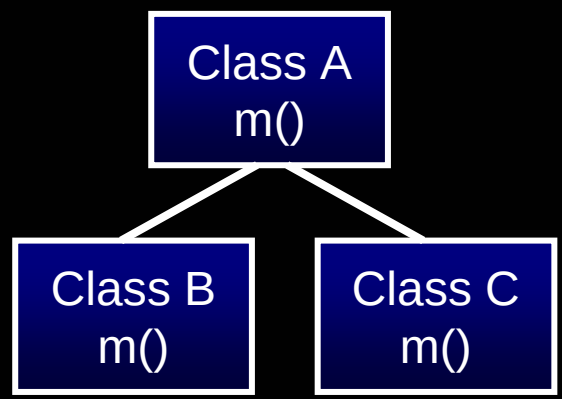


Polymorphic Inline Cache

- 複数の呼び出し先を持つcallの高速化のために、複数のguardが存在する。
  - 複数のメソッドがオーバライドしているvirtual呼び出し
  - 複数のクラスがimplementしているinterface呼び出し

```
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
}
```

A.m()を、複数のメソッドがオーバライドしている例



Directed devirtualization

- Javaでは、guarded devirtualizationによるメソッド呼び出しに関するコスト(guard)は、仮想メソッド呼び出しのコストと比べて、あまり差がない。
  - Guard無しのdevirtualizationによる高速化の要求
- クラス階層解析を必要としない方式
  - 型解析
- クラス階層解析を必要とする方式
  - Code patching
  - On stack replacement
  - Preexistence

型解析 (type analysis)

- メソッド呼び出しのレシーバーの型を計算して、一意に決まる場合には、仮想メソッド呼び出しを直接メソッド呼び出しに変換、もしくはメソッド展開、が可能。
  - new, instanceof, checkcast, 引数、戻り値、等を型の種にする

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

x.m()に到達する型は、Aだけである。

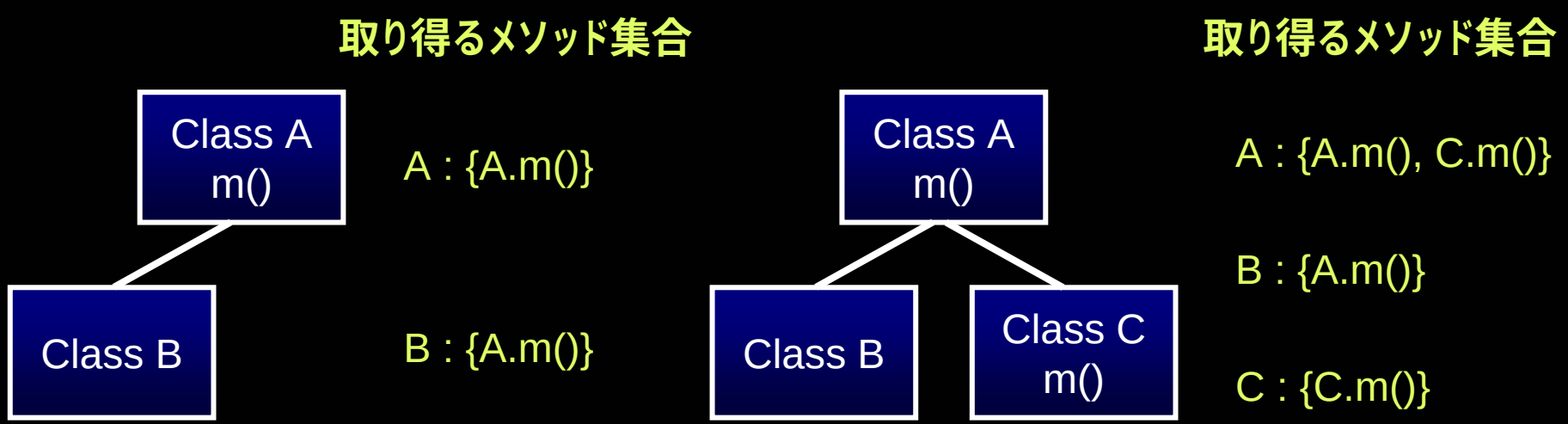
```
r0 = <receiver object>
call A.m()  // 直接メソッド呼び出し
```

```
A x;
if (…) { x = new A();}
else { x = new B();}    // BはAのサブクラス。
x.m();
```

x.m()に到達する型はAとBで、一意に決定しないので、変換不可能。

クラス階層解析 (class hierarchy analysis)

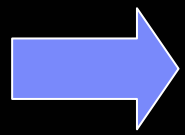
- プログラム全体のクラス階層においてどのメソッドが定義されているかを調べる。
- Javaの場合、実行中にクラスロードが発生するので、常にメンテナンスする必要がある。



Code patching

- コンパイル時に呼び出し先メソッドがオーバライドされていなかったら、直接メソッド呼び出しと間接メソッド呼び出しの両方を用意し、直接メソッド呼び出しを実行する。

```
A x;
…
x.m(); // オーバライドされていない
…
```



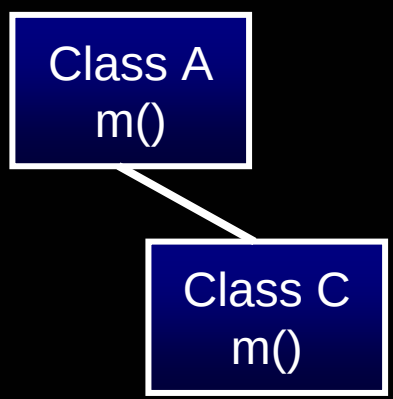
Class A
m()

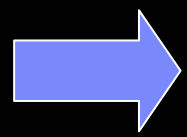
```
 call A.m // 直接メソッド呼び出し
after_call:
…

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   // 間接メソッド呼び出し
 jmp after_call
```

Code patching

- 動的クラスローディングによってメソッドがオーバライドされた時は、コードを書き換えて間接メソッド呼び出しを実行する。

```
A x;
…
x.m(); // オーバライドされた
…
```





```
 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   // 間接メソッド呼び出し
 jmp after_call
```

On stack replacement

- メソッドオーバライドが起きたら、メソッドの実行中に、実行中のコードを変更するともに、スタックの状態を置き換え、変更後のコードの途中から実行を継続する。

```
foo () {
 A x = …;
 …
 x.m();
}

m() {  m1(); }

m1(){ … }
```

直接呼び出しへ置き換え

```
foo () {
 A x =…;
 …
 <check point>
 A.m(); // 直接呼び出し
}
```

オーバライドが起きたら、<check point>で実行中のコードとframeを変更し、実行を再開する。

```
foo () {
 A x = …;
 …
 <check point>
 x.m();  // 間接呼び出し
}
```

On stack replacement

- メソッド展開が適用された場合は、1つに統合されたスタックのイメージから、インライン展開前のイメージを戻す必要があり、実装が複雑。

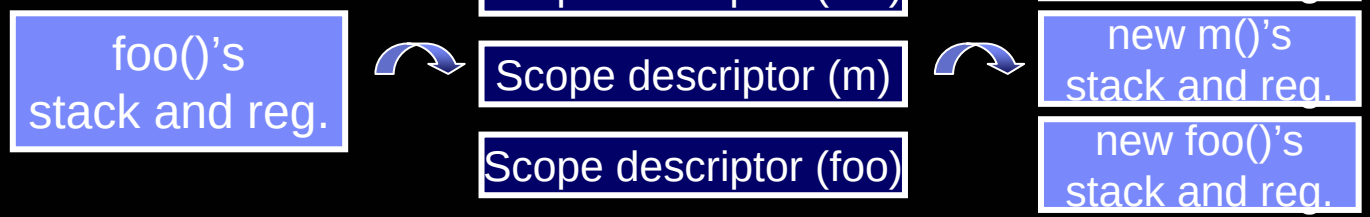
```
foo () {
 A x = …;
 …
 x.m();
}

m() {  m1(); }

m1() { … }
```

m()とm1()の呼び出しにインライン展開を適用

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

オーバライドが起きたら、<check point>で実行中のコードとframeを変更し、実行を再開する。

```
foo () {
 A x = …;
 …
 x.m();
}

m() {  m1(); }

m1() { … }
```

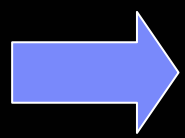


Preexistence

- メソッド呼び出しのレシーバーが、呼び出し側のメソッドを実行する前に決定したオブジェクトのまま変わらないことが保証出来れば、仮想メソッド呼び出しを直接メソッド呼び出しに変換、もしくはメソッド展開、が可能。

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

Class A
m()



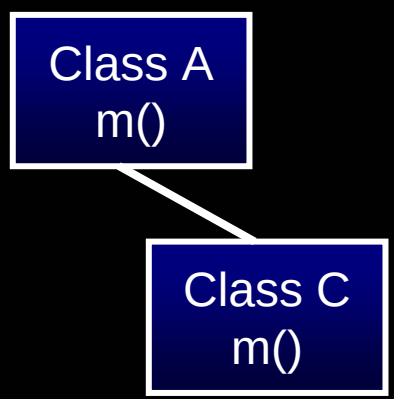
```
r0 = <receiver object>
<inlined code of A.m()>
```

x.m()に到達するxの定義は、(fooを実行する前の値で変わらない) 引数だけである。

Preexistence

- メソッドのオーバライドが起きたときは、次回のメソッド呼び出し時までに、変換を取り消せばよい。

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

}
```

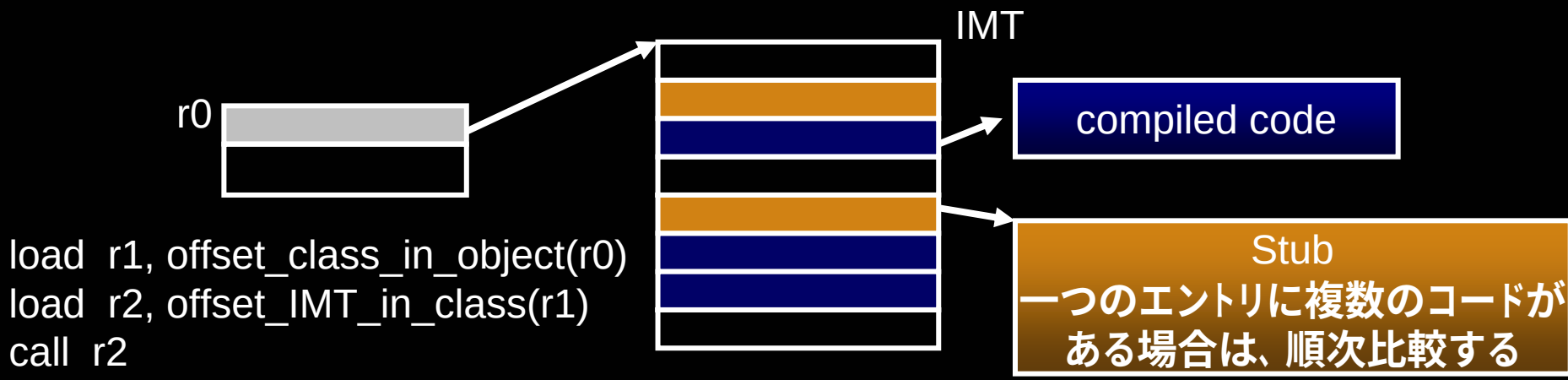


次のfoo()の呼び出しまでに変換すればよい

```
load  r1, offset_class_in_object(r0)
load  r2, offset_method_in_class(r1)
load  r3, offset_code_in_method(r2)
call  r3   // 間接メソッド呼び出し
```

インタフェースメソッド呼び出しの高速化

- 呼び出し時のインタフェースを実装するクラスのサーチを、ハッシュテーブルを引くことで置き換え、高速化する。
  - クラス毎に、ハッシュテーブル (Interface Method Table) を持ち、インタフェースクラスを実装するメソッドを登録する。
  - インタフェースメソッド呼び出しの際は、IMTを引いて、呼び出し先メソッドを決定する。
  - ハッシュが衝突した場合は、stubで実行時にサーチして解決。



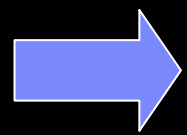
オブジェクト参照の高速化

- 脱出解析によるスタック割付
- スカラ置き換え

脱出解析によるスタック割付

- オブジェクトの生存区間が、あるメソッドで閉じているならば、そのオブジェクトはヒープではなくスタックに割り付け可能。

```
Class A { int f;}
static int z;
int foo() {
  A x =  new A();
  z = x.f;   // xはインスタンス・クラス変数に
             代入されない
  bar(x.f);  // xはメソッド呼び出しの
             引数でない
  return z; // xはメソッドの返値でない
}
```



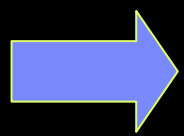
オブジェクトxは、スタックに割り付け可能。

スカラ置き換え

- オブジェクトのヘッダが使われていなければ、スタックに割り付けられたオブジェクトは単純変数に置き換え可能。
  - オブジェクトのヘッダを使う命令 - 仮想メソッド呼び出し、型検査、同期

オブジェクトxは、スタックに割り付け可能。

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



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

同期の高速化

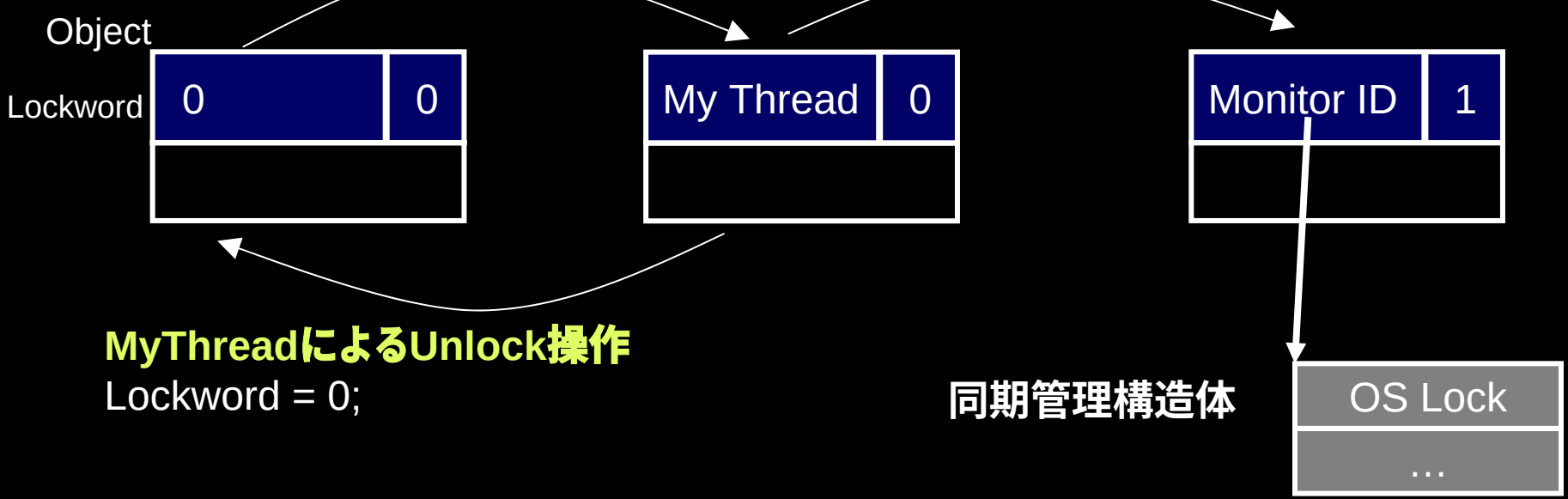
- 同期操作の高速化
- 脱出解析による同期操作除去

同期操作の高速化

- 同期がスレッド間で衝突しない場合には、OSの同期ではなく、CPUが持つ高速な同期命令（Compare and swap）を利用する。

MyThreadによるLock操作
CompareAndSwap(Lockword, 0, MyThread)

OtherThreadによるLock操作
（Lockの衝突）



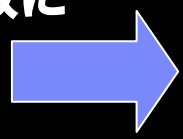
MyThreadによるUnlock操作
Lockword = 0;

同期管理構造体

脱出解析による同期操作除去

- オブジェクトが、あるメソッドから脱出していないならば、そのオブジェクトに関する同期操作は除去可能。
  - 新しいJavaのメモリモデルに関しては、JSR 133参照

```
Class A { int f;}
static int z;
int foo() {
  A x =  new A();
  synchronized (x) {
    z = x.f;   // xはインスタンス・クラス変数に
               代入されない
  }
  bar(z);    // xはメソッド呼び出しの
               引数でない
  return z; // xはメソッドの返値でない
}
```



オブジェクトxに関する同期は除去可能。

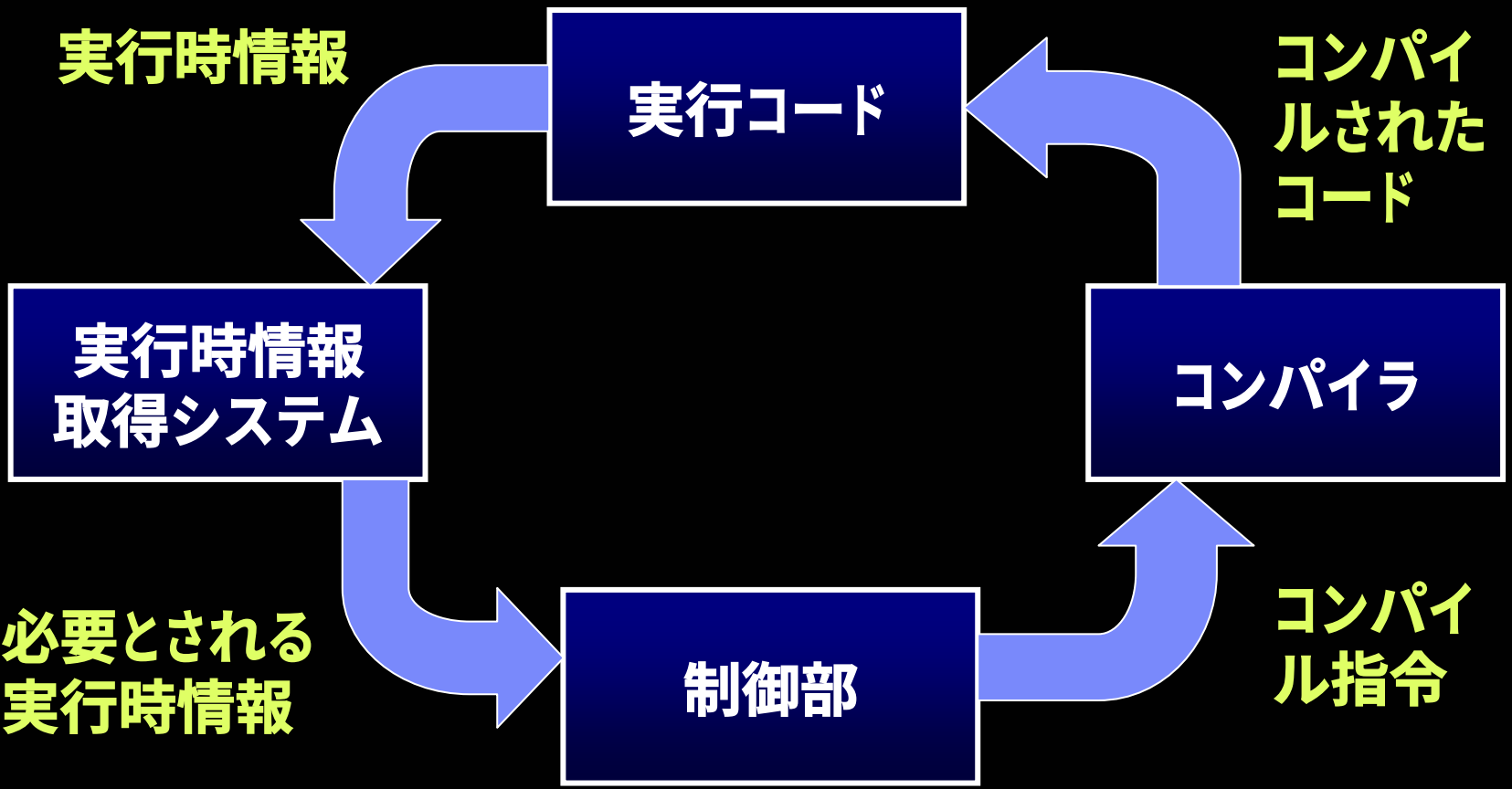
```
Class A { int f;}
static int z;
int foo() {
  A x =  new A();
 synchronized (x) {
    z = x.f;

  }
  bar(z);

  return z;
}
```

Java言語特有の最適化

- Java処理系の構成
- 従来から知られている最適化
- Java言語特有の最適化
- 実行時情報を利用した最適化
  - 実行時情報を利用するためのフレームワーク
  - 実行時情報の取得方法
  - 利用出来る実行時情報
  - 最適化の方法
- 文献リスト

実行時情報を利用するためのフレームワーク

- 再コンパイルできるシステムであることが前提



実行時情報の取得方法

- Sampling – 頻繁に実行されるメソッド、コードの特定
  - コンパイルされたコード内の適当な地点 (メソッドの入り口、ループの先頭へのjump等) を通過したときに、ログを取る。
  - OSのタイマー割り込みを使って、割り込みハンドラの中で実行されているアドレスを取得する。
- Instrumentation – 実行中の値の特定
  - コンパイルされたコード内に値を記録するコードを生成する。
- Hardware Performance Monitor Counter – キャッシュミス等の特定
  - CPUが提供するパフォーマンス・モニタの値 (キャッシュミス等) を読み取る。

利用出来る実行時情報

- Sampling
  - 実行頻度の高いメソッド
  - 実行頻度の高いメソッド内の実行経路
  - 例外発生頻度の高いコールスタック
- Instrumentation
  - メソッドに渡される引数
  - ループの実行回数
- Hardware Performance Monitor Counter
  - キャッシュミスが起きるロード命令とミスが起きるアドレス

最適化の方法

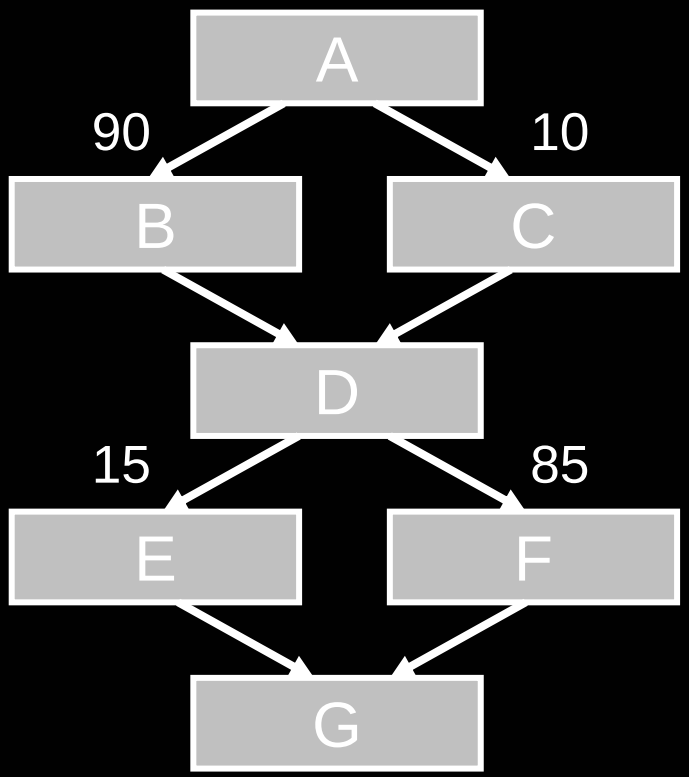
- 再コンパイル
- 経路の頻度情報を使う
  - メソッド展開
  - コードの並べ直し
- 特殊化
- プリフェッチ
- オブジェクトの再配置

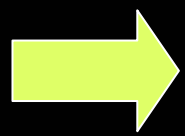
再コンパイル

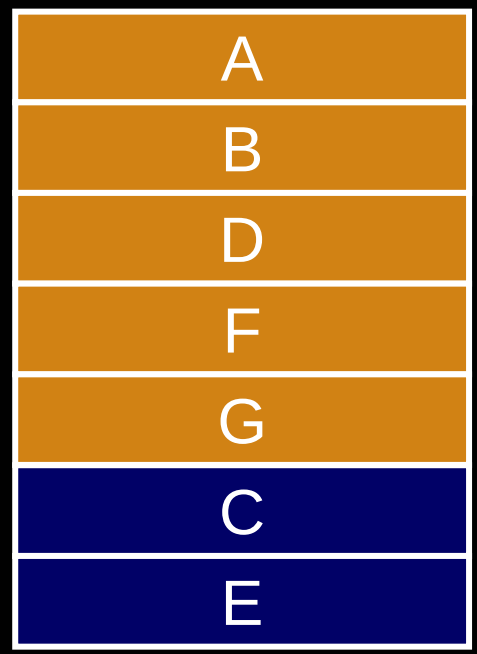
- 頻繁に実行されるメソッドは、最適化のレベルを上げて再コンパイルする。
  - 時間がかかる最適化を適用する
  - 頻繁に実行されるcall pathに沿って深く、メソッド展開する
  - 例外が頻繁に起こるcall pathに沿って深く、メソッド展開する
    - 例外処理のオーバヘッドの削減

コードの並べ直し(code reordering)

- 頻繁に実行されるＢＢのコードを、近い位置に並べる。
  - キャッシュミスを減らす
  - 分岐予測ミスを減らす







特殊化(specialization, customization)

- Instrumentationで得られた定数を元に、専用コードを生成する。
  - 整数・実数
    - 定数伝搬
    - 不要命令除去
  - オブジェクトの型
    - 型検査の除去
    - レシーバの型特定による、より正確なクラス階層解析
  - 配列の長さ
    - 配列インデックスチェックの除去
    - ループ単純化・ループ展開
- (遅い) 汎用コードも用意しておくことが必要

プリフェッチ(prefetch)

- データキャッシュ・ＴＬＢ（Translation Look-aside Buffer: 論理アドレス->物理アドレス変換で用いられるバッファ）ミスが起きるロード命令の前に、あらかじめデータをロードする命令を実行し、ミスをなくす。

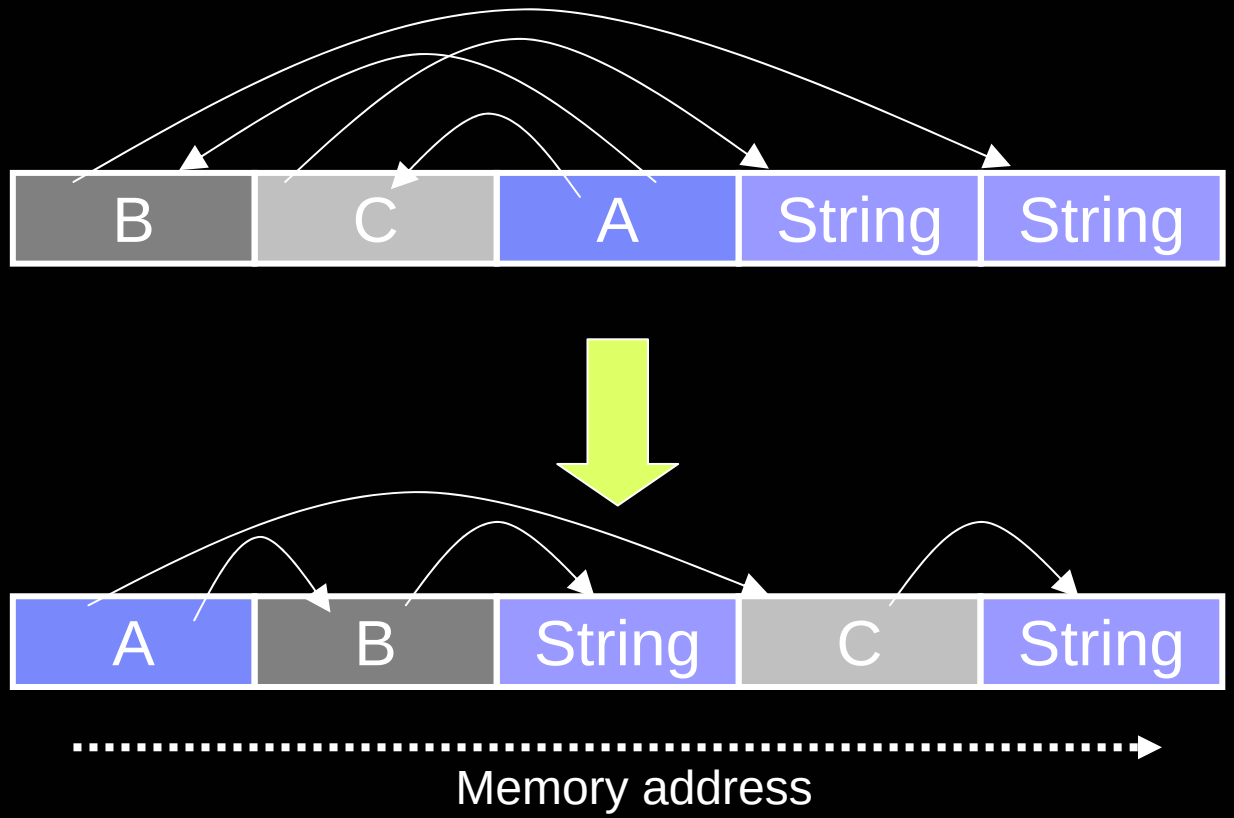
```
TokenLoop:
  for (int i = 0; i < tv.ptr; i++) {
    Token tmp = tv.v[i];
    tmpNext = spec_load (&tv.v[i+1]);    // 次イタレーションのtmp
    prefetch (tmpNext + offset(facts));   // factsのprefetch
    prefetch (tmpNext + offset(facts) + c); // facts arrayのprefetch
    for (int j = 0; j < t.size; j++)
      if (!t.facts[j].equals (tmp.facts[j]))
        continue TokenLoop;
    return tmp;
  }
```

Slide from [Inagaki03]

オブジェクトの再配置

- 関連があるオブジェクトを、GCの際に近くに並べて、キャッシュミスを減らす。
  - アクセス順序によって、最適な配置が異なる

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



Our research outcome

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

- 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]

Special thanks to

- 中谷登志男
- 小松秀昭
- 小野寺民也
- 菅沼俊夫
- 河内谷清久仁
- 小笠原武史
- 川人基弘
- 安江俊明
- 緒方一則
- 古関聰
- 稲垣達氏
- 郷田修
- 竹内幹雄
- 今野和浩
- 田端幹男
- 百瀬浩之

ありがとうございました

文献リスト

- Java**処理系の構成**
  - 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), 2004.
  - 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://www-124.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 optimzations, PLDI, 2000.
    - A-Tabatabai et al. The StarJIT Compiler: A Dynamic Compiler for Managed Runtime Environments, Intel Technology Journal, 2003. available at http://developer.intel.com/technology/itj/2003/volume07issue01/.
- **従来から知られている最適化**
    - Aho et al. Compilers: Principles, Techniques, and Tools, ISBN 0201101947.
    - **（和訳）コンパイラI・**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.
  - **部分冗長性除去**
    - J. Knoop, O. Ruthing, and B. Steffen. Optimal code motion, TOPLAS, 1995.
  - **メソッド展開**
    - Suganuma et al. An Empirical Study of Method Inlining for a Java Just-In-Time Compiler, JVM, 2002.
  - **レジスタ割り付け**
    - Chaitin. Register allocation and spilling via graph coloring, Compiler Construction, 1982.
    - Poletto et al. Linear scan register allocation, TOPLAS, 1999.

文献リスト

- Java言語特有の最適化
  - **例外検査の削除**
    - Kawahito et al. Effective Null Pointer Check Elimination Utilizing Hardware Trap, ASPLOS, 2000.
    - **大平他. 例外依存関係を越える部分冗長性除去, 情報処理学会・プログラミング研究会**, 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.
  - **型解析**
    - 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.
  - **型解析**
    - Chambers et al. Interactive type analysis and extended message splitting; optimizing dynamically-typed object-oriented programs, PLDI, 1990.
  - **クラス階層解析**
    - 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.
  - **インタフェースメソッド呼び出し**
    - Alpern et al. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless, OOPSLA, 2001.

文献リスト

- Java言語特有の最適化
  - 同期操作の高速化
    - Bacon et al. Thin Locks: Featherweight Synchronization for Java, PLDI, 1998.
    - Onodera et al. A Study of Locking Objects with Bimodal Fields, OOPSLA 1999.
  - 脱出解析
    - 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.
  - その他の最適化
    - 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.
- 実行時情報を利用した最適化
    - 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.
  - 再コンパイル
    - Ogasawara et al. A Study of Exception Handling and Its Dynamic Optimization in Java, OOPSLA, 2001.
  - コードの並べ直し
    - Pettis et al. Profile-Guided Code Positioning, PLDI, 1990.
  - 特殊化
    - 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.
  - プリフェッチ
    - Inagaki et al. Stride Prefetching by Dynamically Inspecting Objects, PLDI, 2003.
  - オブジェクトの再配置
    - A-Tabatabai et al. Prefetch Injection Based on Hardware Monitoring and Object Metadata, PLDI, 2004.