コンパイラ演習
第 11 回
(2011/12/22)

中村 晃一　野瀬 貴史　前田 俊行
秋山 茂樹　池尻 拓朗
鈴木 友博　渡邊 裕貴
潮田 資秀
小酒井 隆広
山下 諒蔵　佐藤 春旗
大山 恵弘　佐藤 秀明
住井 英二郎

今日の内容

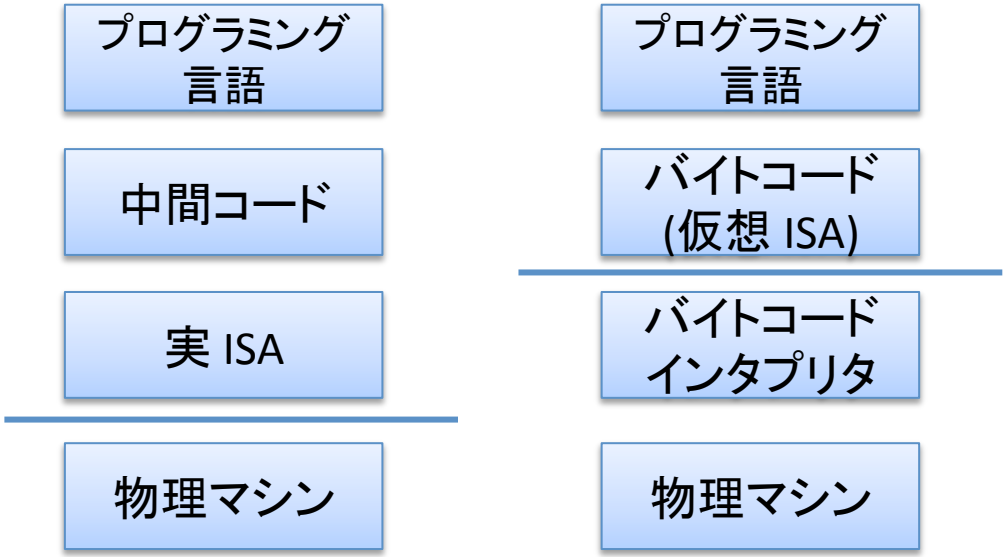
- 仮想マシンとは
  - バイトコードインタプリタとは
- Just-in-Time コンパイル
  - ナイーブな JIT
  - より良い JIT
  - 動的な情報に基づく最適化

仮想マシン (VM) とは

- 仮想的に構築されたプログラム実行環境
- 大きく 2 種類に分類できる
  - System VM
    - 物理的なコンピュータを模擬した仮想環境
      - 例: VirtualBox, VMware など
  - Process VM
    - アプリケーションを動かすための仮想環境
      - 例: バイナリトランスレータ など
        - » OS が資源を仮想記憶を通してプロセスに対し見せているのも広義には Process VM と言える

今回対象にする VM: バイトコードインタプリタ

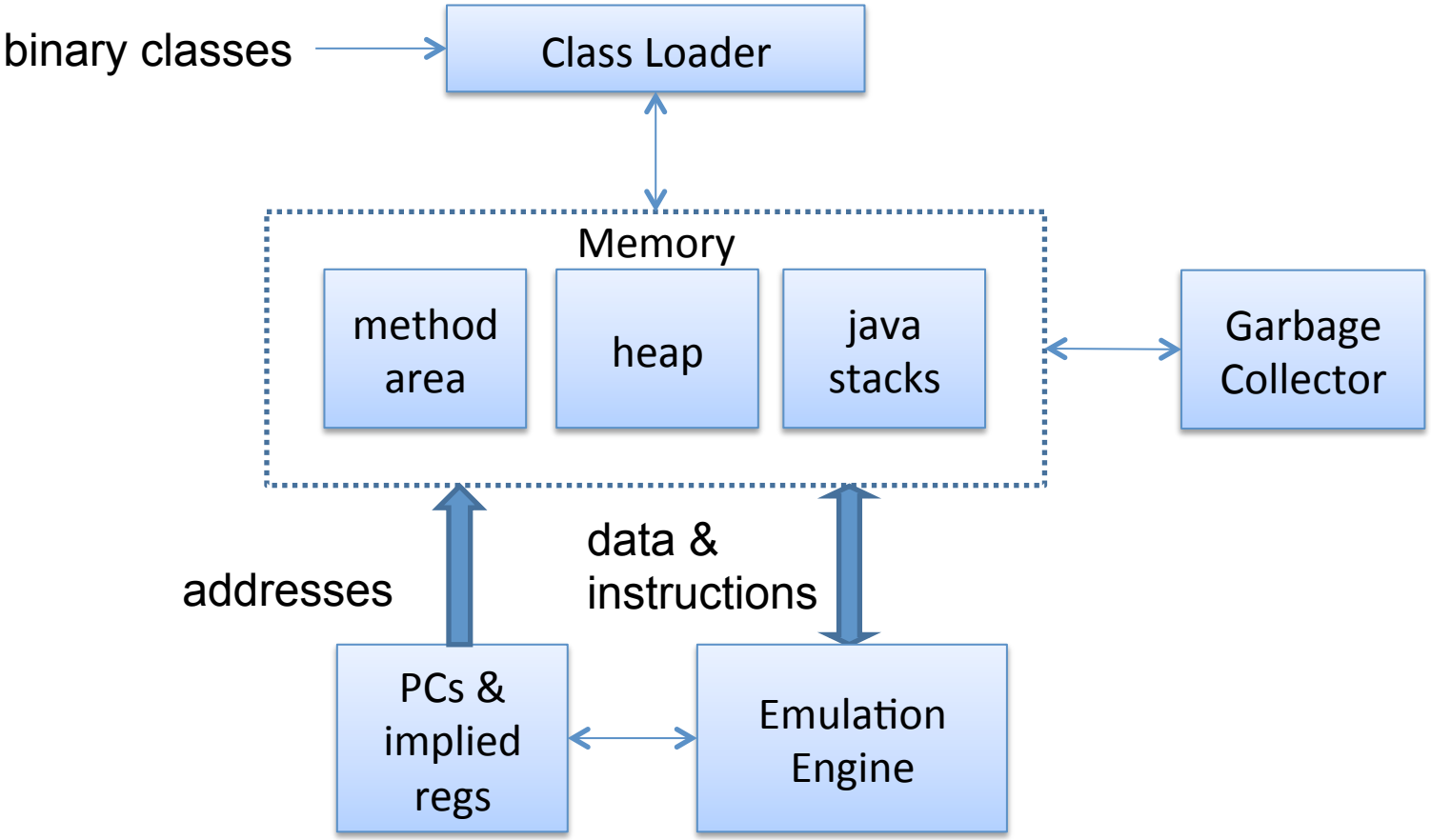
- プログラミング言語を実行するために仮想的に定義したマシンアーキテクチャおよびその実行系
  - Process VM の一種



なぜわざわざプログラムを バイトコードインタプリタで実行するのか

- ポータビリティを確保しやすい
  - 参考: Java の標語: Write Once, Run Anywhere
- 構文木をたどって評価するインタプリタに比べ速度面で有利
- 機械語に近づけることで最適化アルゴリズムを適用しやすくなる

バイトコードインタプリタの構成の例: Java Virtual Machine (JVM)



バイトコードインタプリタの仮想 ISA (Instruction Set Architecture)

- 大きく分けて 2 種類ある

	レジスタマシン	スタックマシン
命令長	長い	短い
機械語に	似ている	(比較的) 似ていない
例	Dalvik, Parrot, LLVM, SquirrelFish など	Java VM, SpiderMonkey, CLR など

性能面で有利

コードサイズが小さい

比較論文: Yunhe Shi, et al., Virtual machine showdown: Stack versus registers, In Proc. of VEE'05, 2005.

例: Java バイトコード

- 以降、以下の Java プログラムを
  バイトコードにコンパイルしたものを見てみる
  - javap –c ClassName で逆アセンブルできる

```
public class HelloJava{
  public static void main(String [] args) {
    int a,b,c;
    a = 1;
    b = 100000;
    c = a + b;
    System.out.print(c);
  }
}
```

逆アセンブル結果

```
0: iconst_1
1: istore_1
2: ldc           #2                  // int 100000
4: istore_2
5: iload_1
6: iload_2
7: iadd
8: istore_3
9: getstatic     #3                  // Field java/
lang/System.out:Ljava/io/PrintStream;
12: iload_3
13: invokevirtual #4                  // Method
java/io/PrintStream.print:(I)V
16: return
```

JVM バイトコード実行の様子

スタック
1

変数テーブル	
0	
1	
2	
3	

```
0: iconst_1
1: istore_1
2: ldc                                    //
int 100000
4: istore_2
5: iload_1
6: iload_2
7: iadd
8: istore_3
9: getstatic     #3                       //
Field java/lang/System.out:Ljava/io/
PrintStream;
12: iload_3
13: invokevirtual #4                      //
Method java/io/PrintStream.print:(I)V
16: return
```

定数 1 をスタックに積む

※ -1から5まではよく使う値なのでそれぞれ専用の命令
iconst_m1, iconst_0 ～ iconst_5 で定数をロードできる

JVM バイトコード実行の様子

スタック

変数テーブル	
0	
1	1
2	
3	

```
0: iconst_1
1: istore_1
2: ldc           #2                  //
int 100000
4: istore_2
5: iload_1
6: iload_2
7: iadd
8: istore_3
9: getstatic     #3                  //
Field java/lang/System.out:Ljava/io/
PrintStream;
12: iload_3
13: invokevirtual #4                  //
Method java/io/PrintStream.print:(I)V
16: return
```

スタックから変数 1 に取り出す

※ ローカル変数は0～3まである

JVM バイトコード実行の様子

スタック
100000

変数テーブル	
0	
1	1
2	
3	

```
0: iconst_1
1: istore_1
2: ldc           #2                  //
int 100000
4: istore_2
5: iload_1
6: iload_2
7: iadd
8: istore_3
9: getstatic     #3                  //
Field java/lang/System.out:Ljava/io/
PrintStream;
12: iload_3
13: invokevirtual #4                  //
Method java/io/PrintStream.print:(I)V
16: return
```

コンスタントプールから
定数を読みだしてスタックに積む

JVM バイトコード実行の様子

スタック

変数テーブル	
0	
1	1
2	100000
3	

```
0: iconst_1
1: istore_1
2: ldc           #2                  //
int 100000
4: istore_2
5: iload_1
6: iload_2
7: iadd
8: istore_3
9: getstatic     #3                  //
Field java/lang/System.out:Ljava/io/
PrintStream;
12: iload_3
13: invokevirtual #4                  //
Method java/io/PrintStream.print:(I)V
16: return
```

スタックから変数 2 に取り出す

JVM バイトコード実行の様子

スタック
1

変数テーブル	
0	
1	1
2	100000
3	

```
0: iconst_1
1: istore_1
2: ldc           #2                  //
int 100000
4: istore_2
5: iload_1
6: iload_2
7: iadd
8: istore_3
9: getstatic     #3                  //
Field java/lang/System.out:Ljava/io/
PrintStream;
12: iload_3
13: invokevirtual #4                  //
Method java/io/PrintStream.print:(I)V
16: return
```

変数 1 からコピーして
スタックに積む

JVM バイトコード実行の様子

スタック
100000
1

変数テーブル	
0	
1	1
2	100000
3	

```
0: iconst_1
1: istore_1
2: ldc           #2                  //
int 100000
4: istore_2
5: iload_1
6: iload_2
7: iadd
8: istore_3
9: getstatic     #3                  //
Field java/lang/System.out:Ljava/io/
PrintStream;
12: iload_3
13: invokevirtual #4                  //
Method java/io/PrintStream.print:(I)V
16: return
```

変数 2 からコピーして
スタックに積む

JVM バイトコード実行の様子

スタック
100001

変数テーブル	
0	
1	1
2	100000
3	

```
0: iconst_1
1: istore_1
2: ldc           #2                  //
int 100000
4: istore_2
5: iload_1
6: iload_2
7: iadd
8: istore_3
9: getstatic     #3                  //
Field java/lang/System.out:Ljava/io/
PrintStream;
12: iload_3
13: invokevirtual #4                  //
Method java/io/PrintStream.print:(I)V
16: return
```

スタックから値を 2 つ取り出して
足したあとスタックに積む

JVM バイトコード実行の様子

スタック

変数テーブル	
0	
1	1
2	100000
3	100001

```
0: iconst_1
1: istore_1
2: ldc           #2                  //
int 100000
4: istore_2
5: iload_1
6: iload_2
7: iadd
8: istore_3
9: getstatic     #3                  //
Field java/lang/System.out:Ljava/io/
PrintStream;
12: iload_3
13: invokevirtual #4                  //
Method java/io/PrintStream.print:(I)V
16: return
```

スタックから変数 3 に取り出す

JVM バイトコード実行の様子

スタック
PrintStream

変数テーブル	
0	
1	1
2	100000
3	100001

```
0: iconst_1
1: istore_1
2: ldc           #2                  //
int 100000
4: istore_2
5: iload_1
6: iload_2
7: iadd
8: istore_3
9: getstatic     #3                  //
Field java/lang/System.out:Ljava/io/
PrintStream;
12: iload_3
13: invokevirtual #4                  //
Method java/io/PrintStream.print:(I)V
16: return
```

java.lang.System.out への参照をスタックに積む

JVM バイトコード実行の様子

スタック
100001
PrintStream

変数テーブル	
0	
1	1
2	100000
3	100001

```
0: iconst_1
1: istore_1
2: ldc           #2                  //
int 100000
4: istore_2
5: iload_1
6: iload_2
7: iadd
8: istore_3
9: getstatic                         //
Field java/lang/System.out:Ljava/io/
PrintStream;
12: iload_3
13: invokevirtual #4                 //
Method java/io/PrintStream.print:(I)V
16: return
```

変数 3 からコピーして
スタックに積む

JVM バイトコード実行の様子

スタック

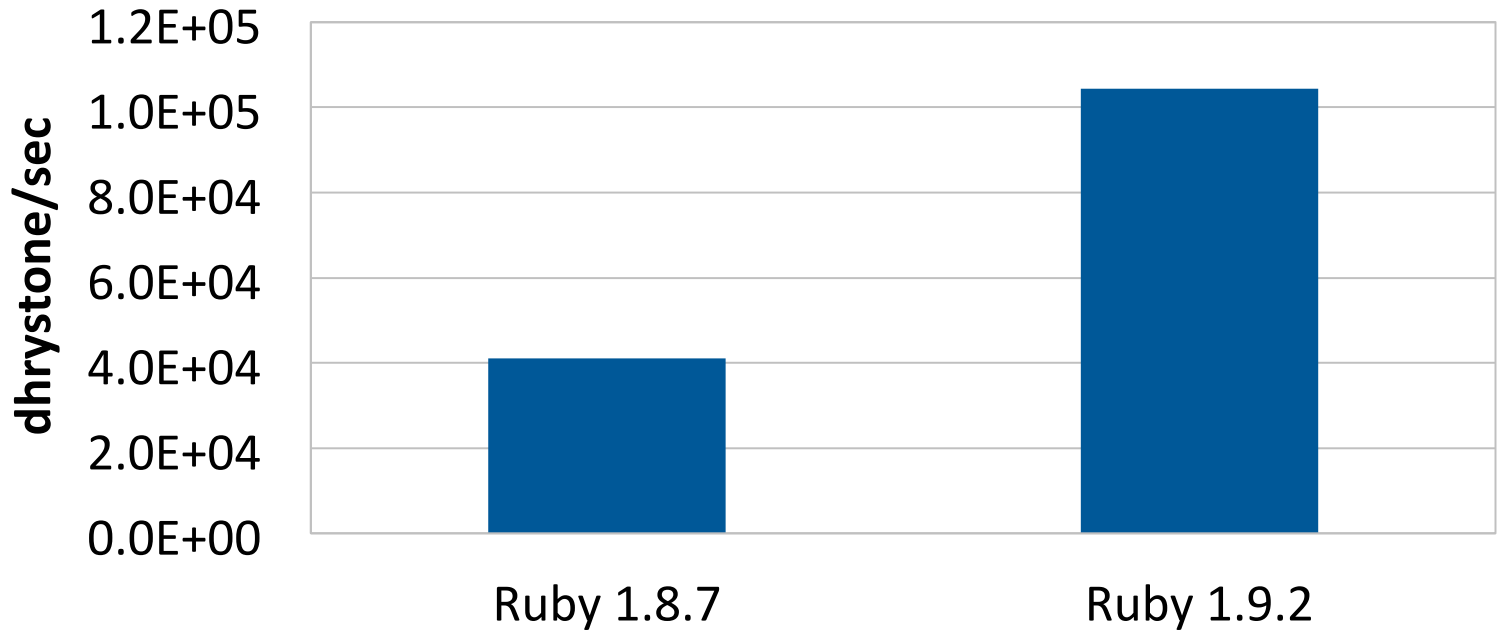
変数テーブル	
0	
1	1
2	100000
3	100001

```
0: iconst_1
1: istore_1
2: ldc           #2                  //
int 100000
4: istore_2
5: iload_1
6: iload_2
7: iadd
8: istore_3
9: getstatic     #                   //
Field java/lang/Sy    .out:Ljava/io/
PrintStream;
12: iload_3
13: invokevirtual #4                  //
Method java/io/PrintStream.print:(I)V
16: return
```

print を呼び出す

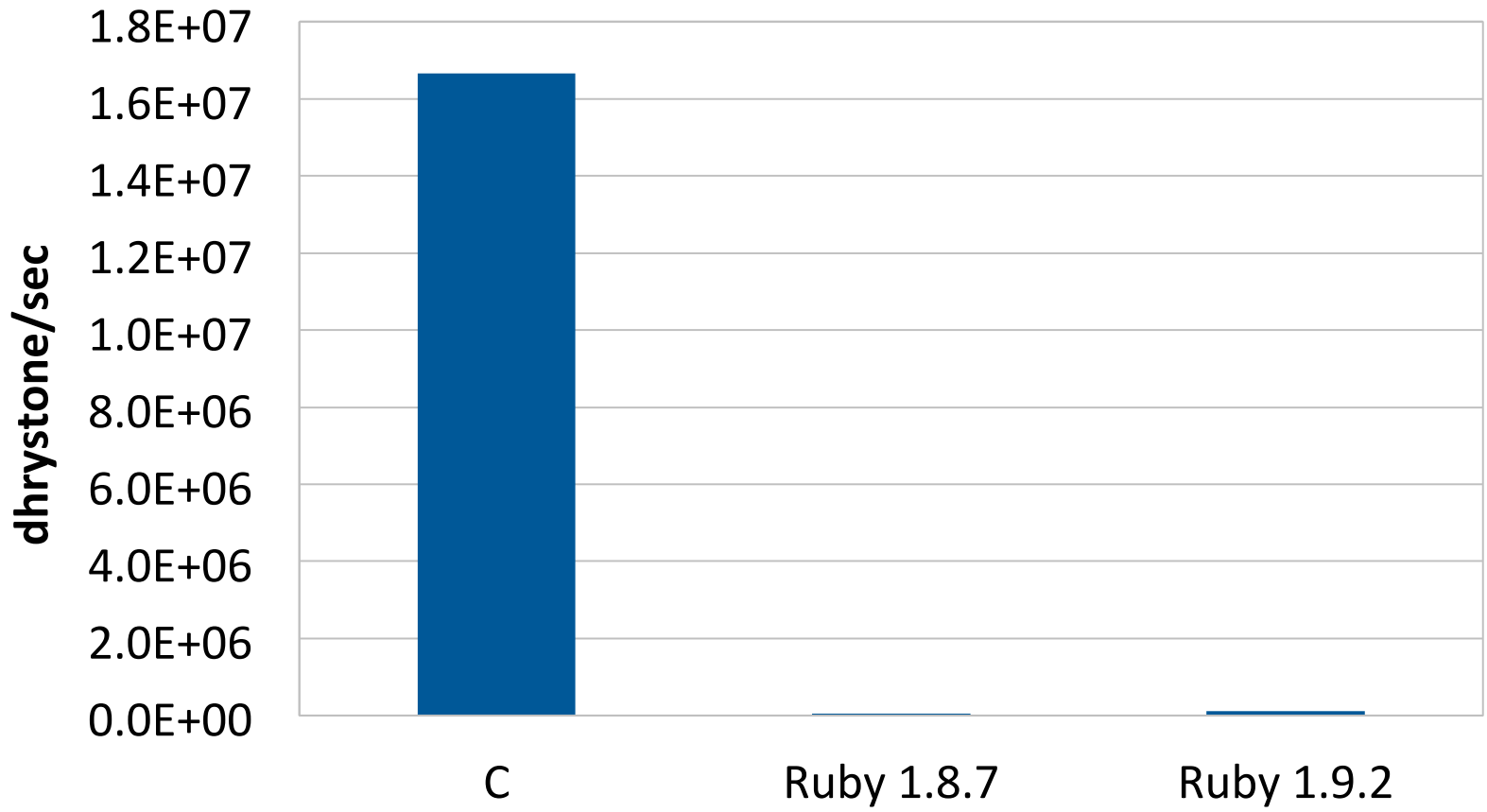
インタプリタとバイトコードインタプリタの性能比較

- 例えば Dhrystone ベンチマークでは Ruby1.9 (バイトコードインタプリタ) は Ruby1.8 (インタプリタ) より2倍以上速い



でもまだまだ遅い

- C に比べると 100 倍以上遅い



Just-in-Time (JIT) コンパイルによる性能改善

- バイトコードを動的に (つまり実行時に) 機械語にコンパイルし実行する

ナイーブな JIT コンパイラの実装方法

1. ローダがバイトコードを読み込む
2. バイトコードを機械語へコンパイル
   – レジスタ割り付けと命令同士の対応付けなど
3. 変換されたコードを実行する

バイトコードの変換

- VM ↔ 機械語間の命令同士の対応付け
  - 基本的には綺麗に対応付くが
    仮想メソッド呼出 (invokevirtual) 等
    実マシンには存在しない命令の扱いに注意
- レジスタ割り付け
  - 普通に割り付けてよい
  - ただし実行時なのであまり複雑なことをすると
    かえって性能に悪影響が出ることに注意

JVM でのレジスタ割り付け

- JVM ではレジスタ割付けは比較的簡単
  - メソッド呼び出しの際に
    必要な分だけスタックが確保されるので
  - 詳しい方法については
    「コンパイラの構成と最適化 第2版
    (中田育男著，朝倉書店)」の
    P. 236以降等を参照

ナイーブな JIT の問題点

- JIT コンパイルしてもコンパイルのコストが実行のコストを上回ってしまったら意味が無い
- 実行時間よりもプログラムの応答性の方が重要なケースもある
  - 例えば GUI アプリケーション

解決法

- インタプリタの実行を行いながら
  効果の高い場所・タイミングに絞って
  コンパイルを行えばよい
  - プログラムの局所性を有効利用する

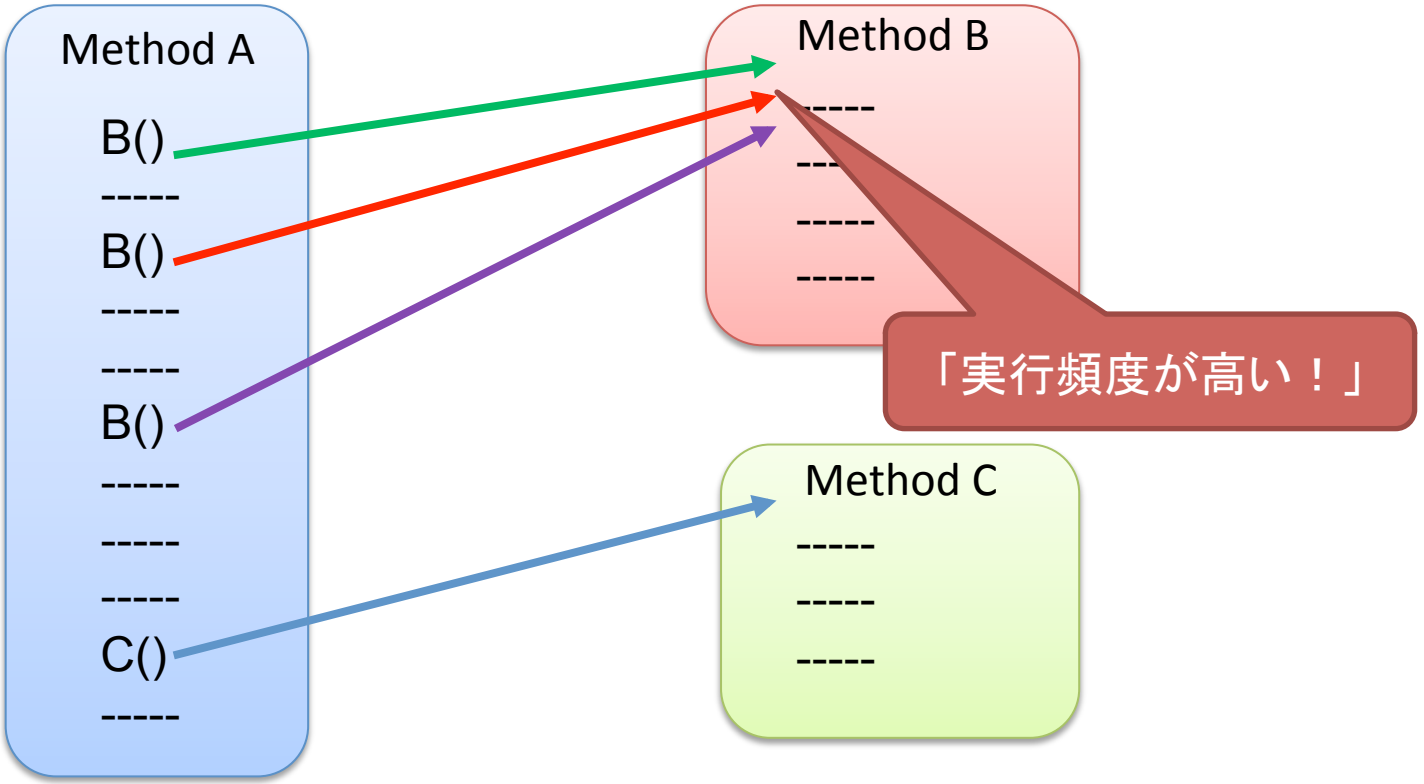
- 上記を実現するためには実行時に
  プロファイリングを行う必要がある
  - プロファイリング = プログラムの実行情報を取ること

より良い JIT コンパイル

1. ローダがバイトコードを読み込む
2. バイトコードをインタプリタで実行しながら
   実行情報を集める (実行時プロファイリング)
3. JIT コンパイルを行う条件を満たす部分を探す
4. バイトコードの一部分を JIT コンパイル
   – レジスタ割り付けと命令同士の対応付け
5. 変換されたコードを実行する

方法 1: メソッドごとにコンパイルする

- メソッドの実行頻度を見て一定回数を超えたらそのメソッドをコンパイル済みのものに置き換える

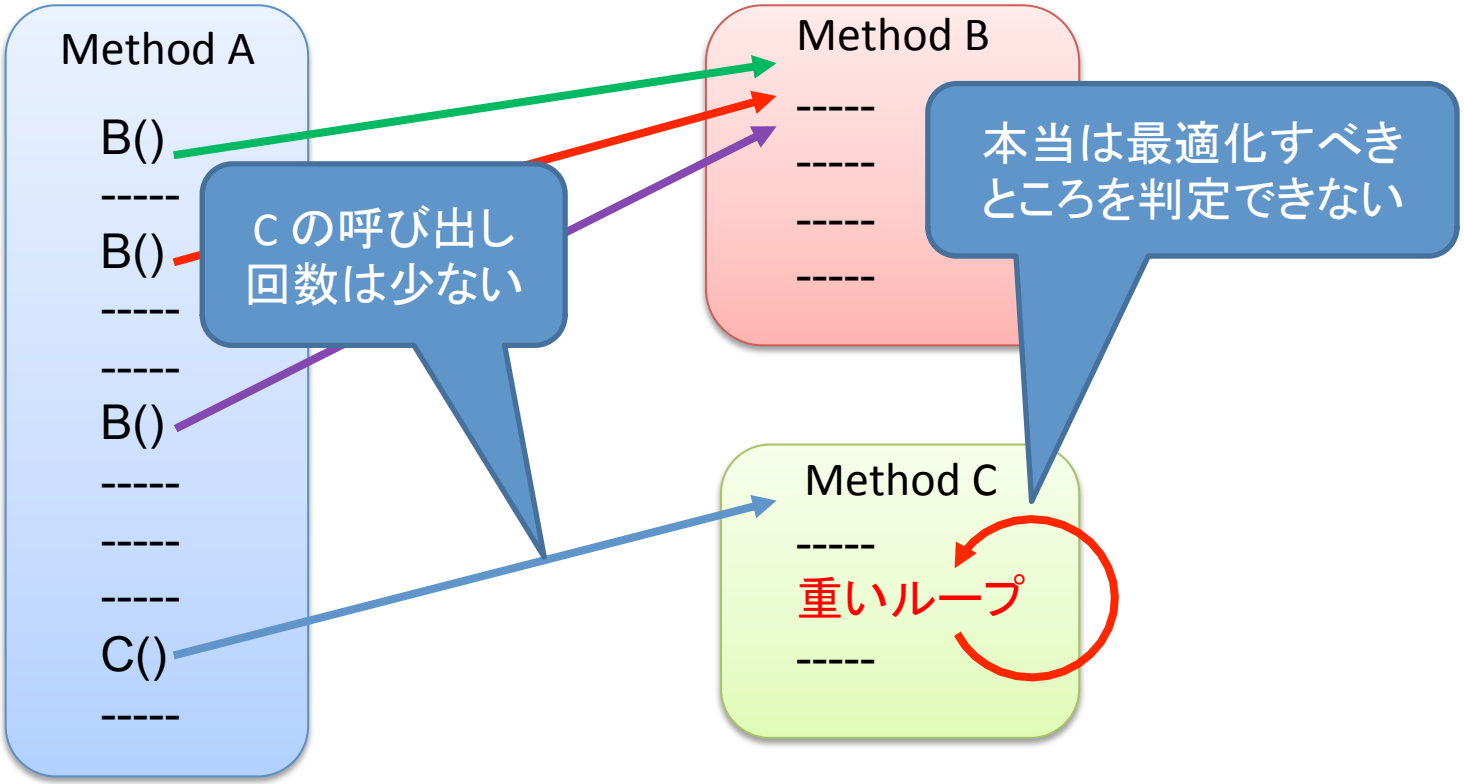


どうやってメソッドの実行頻度を取るか？

- 方法1:
  カウンタをメソッド呼び出しごとにインクリメント
- 方法2:
  タイマ割り込みでサンプリング
  – スタックのトップにあるメソッドを計測する

メソッドの頻度情報を取る手法の問題点

- メソッドの呼び出し回数が少ないが
  中のループが重いケースに対応できない



方法 1’: もう少し賢く

- メソッドそのものの呼び出し回数に加え
  メソッド内のループの実行回数が
  一定回数を超えたらメソッドを hot と判定する
  - Java の HotSpot はこの方式
- 具体的には後方ジャンプの回数を見る

```
Method void foo()
 0 hoge
 1 fuga
 4 piyo
 5 goto 1
```

後方ジャンプ

On-Stack Replacement (OSR)

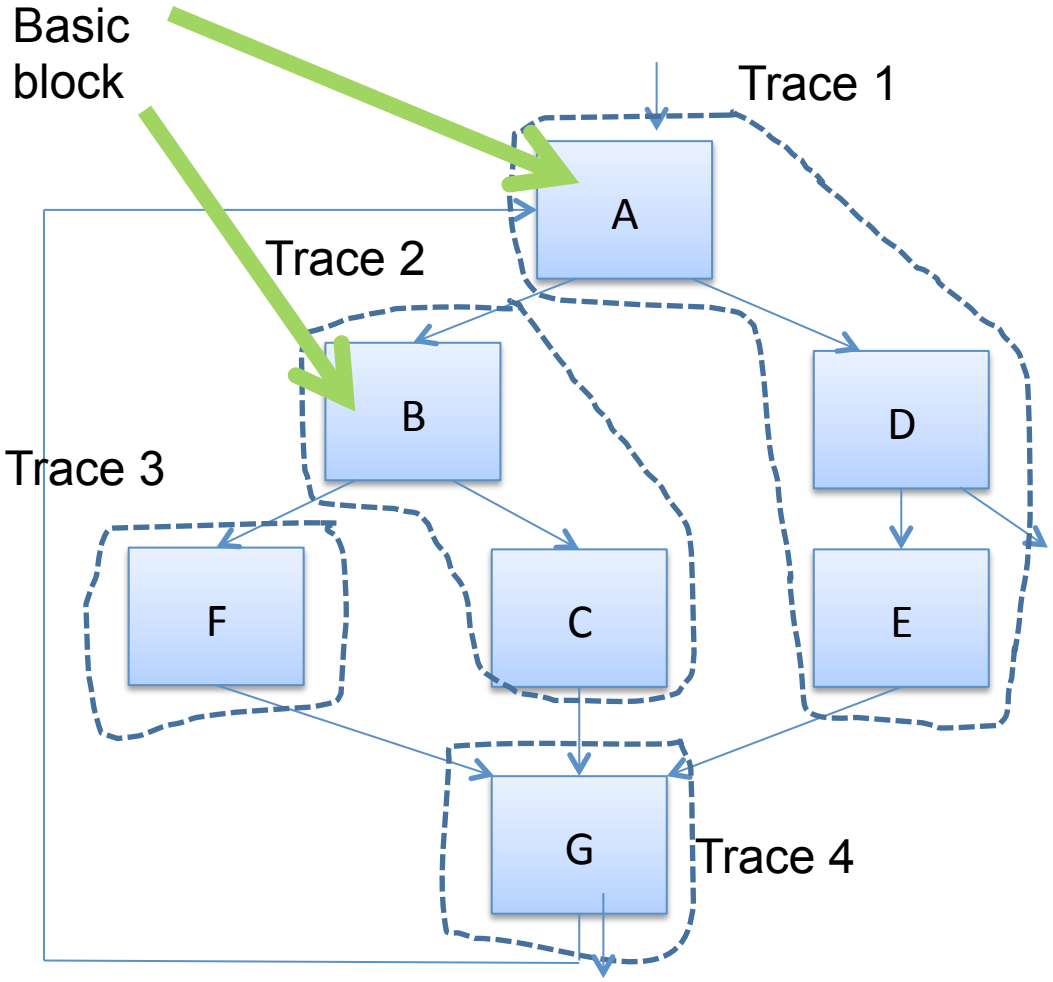
- ループの実行途中でバイトコード実行を
  コンパイル後の機械語実行に差し替える手法
- 手順
  1. バイトコードの実行を一旦停止
  2. 変数の格納位置をコンパイル後の機械語に
     合わせたものに変更する
  3. 実行していたバイトコードに対応する機械語に
     ジャンプし再開する

参考:
OSR: On-Stack replacement of interpreted methods by JIT-ted methods
(https://bugzilla.mozilla.org/show_bug.cgi?id=539094)
に投稿されているパッチ(Firefox本体には反映されていないことに注意)

メソッド単位の JIT コンパイルの問題点

- 本当に hot な部分は
  メソッド全域ではないかもしれない
- メソッドをまたがって最適化した方が
  良い場合もあるかもしれない

方法 2: Trace 単位でのコンパイル



- Trace の定義:
  - トップに 1 つだけ入口を持つような制御フローグラフ上のパス
    - 出口は複数許す

Trace の構築におけるポイント

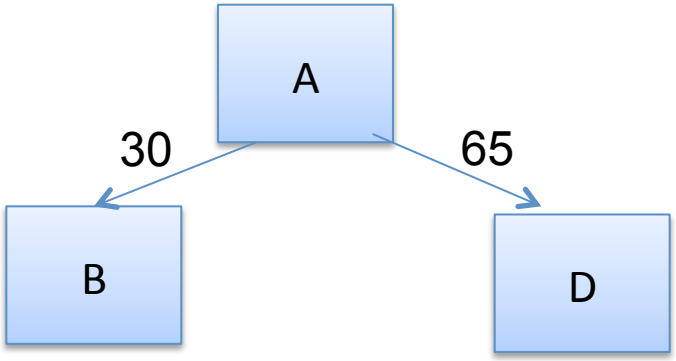
- 実行時に漸進的に構築したい
- ポイントは 3 つ
  1. どの基本ブロックを trace の開始点にするか
  2. 分岐がある場合、複数の後続のブロックのうち次に選ぶブロックはどれにすべきか
  3. どの基本ブロックを traceの終わりとするか

どのブロックを trace の開始点とするか

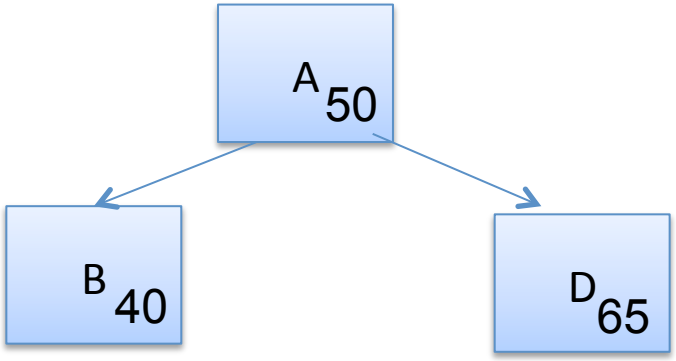
- 実行頻度の高いブロックを開始点とすべき
- 幾つか方法がある
  - 全ての基本ブロックの実行頻度を計測して決める
    - 一番確実
  - 後方ジャンプの飛び先を候補とする
    - ループらしきものを検出
  - 既存の trace の直後を候補とする
    - Traceは hot なのだから
      それから脱出した直後も hot になる可能性が高い

どの後続ブロックを選ぶか

- エッジもしくはブロックに重みをつけて選択する
- 重み付けに使う指標
  - Most frequently used
  - Most recently used
  - 上記 2 つの組み合わせ
  - etc.



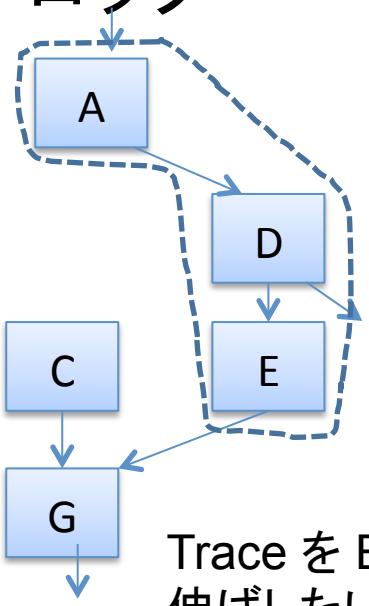
エッジについて計測



ブロックについて計測

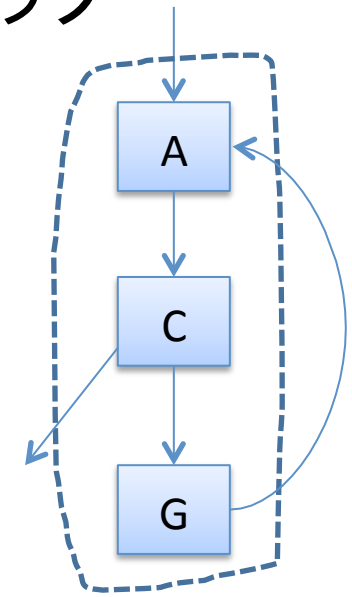
どのブロックを traceの終わりとするか (1 of 2)

- 脇からの入り口がある ブロック



Trace を Eから G まで
伸ばしたいが G には
C からの合流もあるのでダメ

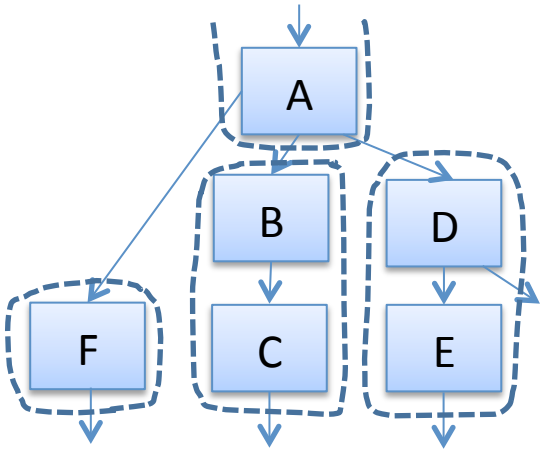
- 同じ trace に到達する ブロック



ループの最後からの
後方ジャンプが該当

どのブロックを traceの終わりとするか (2 of 2)

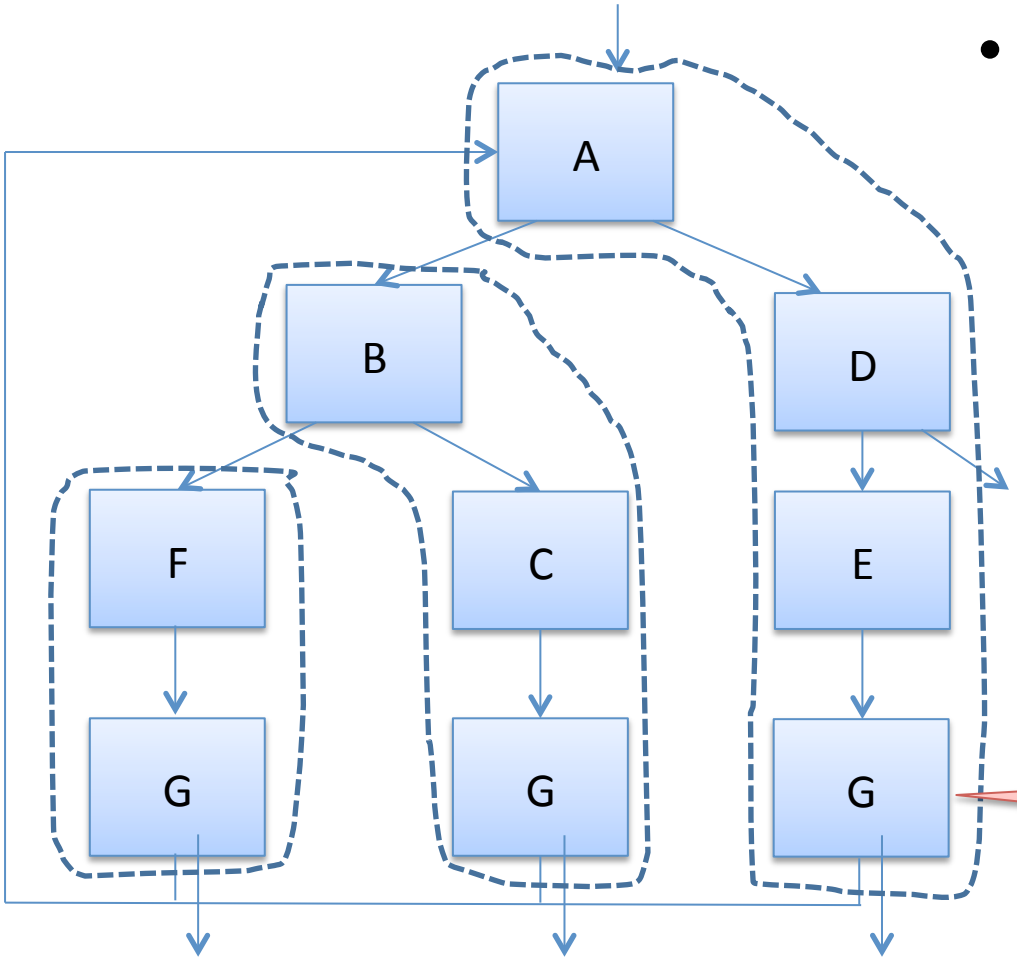
- どの後続ブロックも違う trace に含まれるブロック



A からは F, B, D の
3 つのブロックに分岐するが
どれも既存の Trace の中なので
ここで打ち切る

- その他
  - 間接ジャンプまたは関数呼出に到達
  - ある長さで打ち切る
  - など何らかの heuristics

Tail duplication



- 分岐の合流地点のブロックを複製することでtrace をより大きくできることがある

Gを複製した

動的な最適化

- 実行時に初めてわかる情報を用いて
  最適化を行う
  - 前回までに解説した各種最適化が適用できる
    - 例えば、実行時にユーザが入力する値を
      定数的に使っている (入力後は変化しない) プログラムなら
      定数畳みこみが使えるようになるかもしれない
    - など
  - 実行時にはオブジェクトの型が分かっているので
    仮想関数呼出しについて最適化ができる

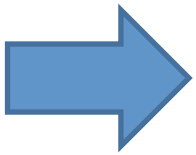
仮想関数呼び出しのオーバーヘッド

```
class A { int getNum() { return 0; }};
class B extends A {  int getNum() { return 1; } };
class C extends A {  int getNum() { return 2; } };
```

となっているとき

```
void someMethod(A obj){
  System.out.print(obj.getNum());
}
```

で呼ばれている getNum が A, B, C のどれのものなのかは実行時に探索して決定する必要がある



(関数探索 + 間接ジャンプ) になってしまうので遅い

オーバーヘッドの削減方法: 直接呼出しに置き換える

- 呼ばれる関数が一意に定まるならば
  仮想関数呼出しを直接呼出しに置き換えられる
  - オブジェクトが生成された直後の仮想関数呼出しでは
    呼ばれる関数は一意に定まる

```
A obj = new A();
r = obj.getNum(); //定義の直後なので呼ばれる関数は一意
```

  - A で定義された A.getNum をオーバーライドしている
    B, C のインスタンスが実行時に一つも存在しなければ
    呼ばれる関数は一意に定まる
    - もし新たに B,C のインスタンスが生まれたら
      最適化をやめないといけないことに注意
  - etc.

インラインキャッシング

- 本来仮想関数を探索するところを
  型チェックのみを行うガード条件に置き換える

```
r = obj.getNum();
```

一番最近に呼出した関数や
統計的に多く呼び出される関数をキャッシュ

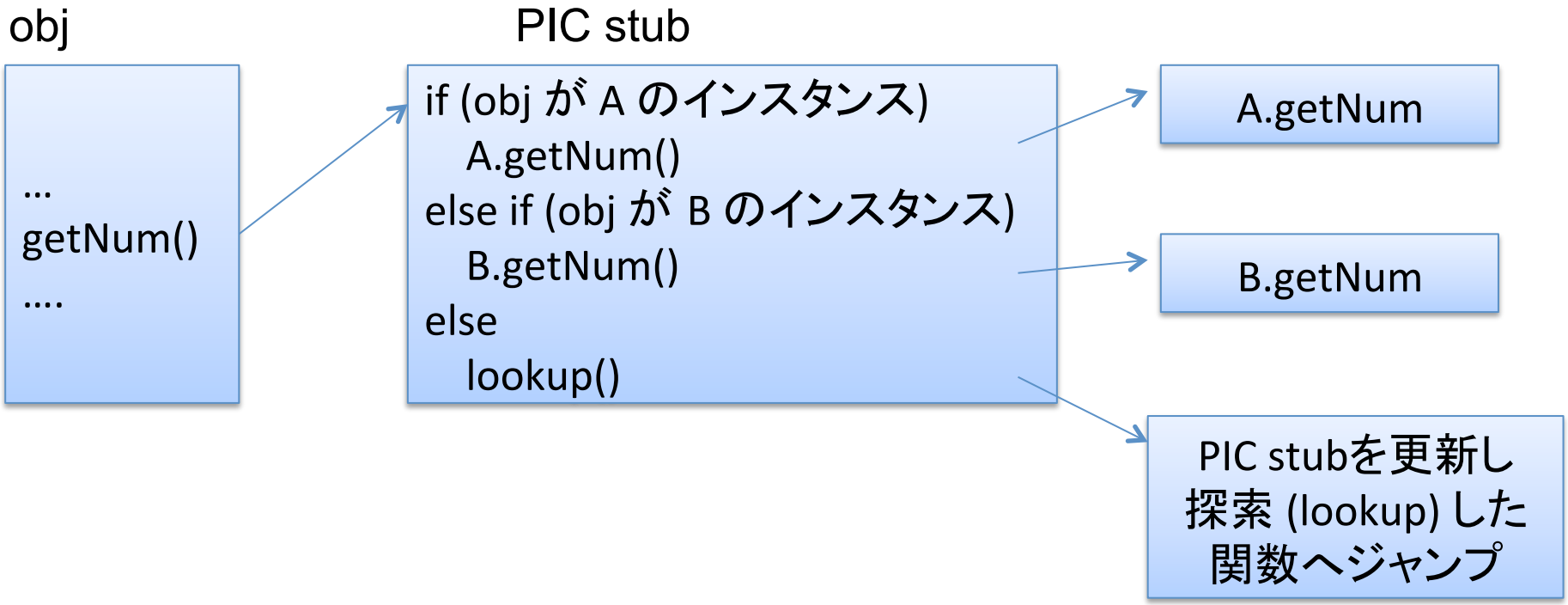
```
if(obj が B のインスタンス){
  r = B.getNum();
} else {
  r = obj.getNum();
}
```

インラインキャッシングの問題点

- 一番最近呼出した関数をキャッシュする場合:
  - obj が交互に B, C, B, C,... となる場合に
    連続してキャッシュミスが起こる
- 一番良く呼出される関数をキャッシュする場合:
  - 型の偏りがあまりなかったり
    型の種類が多くなると効果が薄れる

解決法: 多相インラインキャッシング

- キャッシュする関数の数を増やす



実装上の注意

コンパイルしたコードを格納する配列の確保

- Linux:
  - mmap(2) で PROT_READ | PROT_EXEC | PROT_WRITE 属性を指定
    - あるいは mprotect(2) で設定
- Windows:
  - VirtualAlloc で PAGE_EXECUTE_READWRITE属性を指定

参考: libjit (http://freecode.com/projects/libjit)の
jit_malloc_exec関数 (jit/jit-alloc.cを参照)

参考文献

- Smith, J.E., Nair, R., Virtual Machines: Versatile Platforms for Systems and Processes, 2005
- 中田育男, コンパイラの構成と最適化 第2版, 2009
- libjit (C言語, 複数プラットフォーム対応)
  http://freecode.com/projects/libjit
- asmjit (C++, x86/x64のみ)
  http://code.google.com/p/asmjit/
- shuJIT
  http://www.shudo.net/jit/index-j.html
- Javaバイトコード処理系の作り方
  http://www.shudo.net/publications/klab-200106/

課題

共通課題

- 2 個の課題のうち**どちらか**を解いてください
- 両方解いてもOK

共通課題 (1)

- JIT コンパイルに関する論文等の文献を一つ選び内容を要約し、考察せよ
  - 例えば以下の論文から選んでも良い
    - Hölzle, U et al., Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches, In Proceedings of the European Conference on Object-Oriented Programming, p. 21-38, July 15-19, 1991
    - Gal, A et al., Trace-based just-in-time type specialization for dynamic languages, In Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation, June 15-21, 2009, Dublin, Ireland
    - Bolz, C et al., Tracing the Meta-Level: PyPy's Tracing JIT Compiler, In Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, pp. 18-25, 2009.
  - もちろん他の文献でもよい

共通課題 (2)

- 何らかのバイトコードインタプリタに
  何らかの JIT コンパイラを実装せよ
  - 対象のバイトコードインタプリタとしては
    CPU 実験で作成したシミュレータを用いてもよい
  - OCaml, Ruby, Python, Perl などの
    バイトコードインタプリタでも良い

コンパイラ係用課題

- 自分たちの実機もしくはシミュレータ上で動く JVM を実装せよ
  - こちらが用意する JAR ファイルを実行できる程度の JVM のサブセットでよい
- さらに上記の JVM に
  何らかの JIT コンパイラを実装せよ

課題の提出先と締め切り

- 提出先: compiler-report-2011@yl.is.s.u-tokyo.ac.jp
- 締め切り: 3 週間後 (1/12) の午後 1 時 (JST)
  - コンパイラ係向け課題締切 : 2012/2/27
- Subject: **`Report␣11␣`**<学籍番号: 5 桁>

  - 例 : **`Report 11 11099`**
- 本文にも氏名と学籍番号を明記のこと

◆ 質問は compiler-query-2011@yl.is.s.u-tokyo.ac.jp まで