Java JITコンパイラの高速化手法

首藤 一幸

2001年 8月 23日

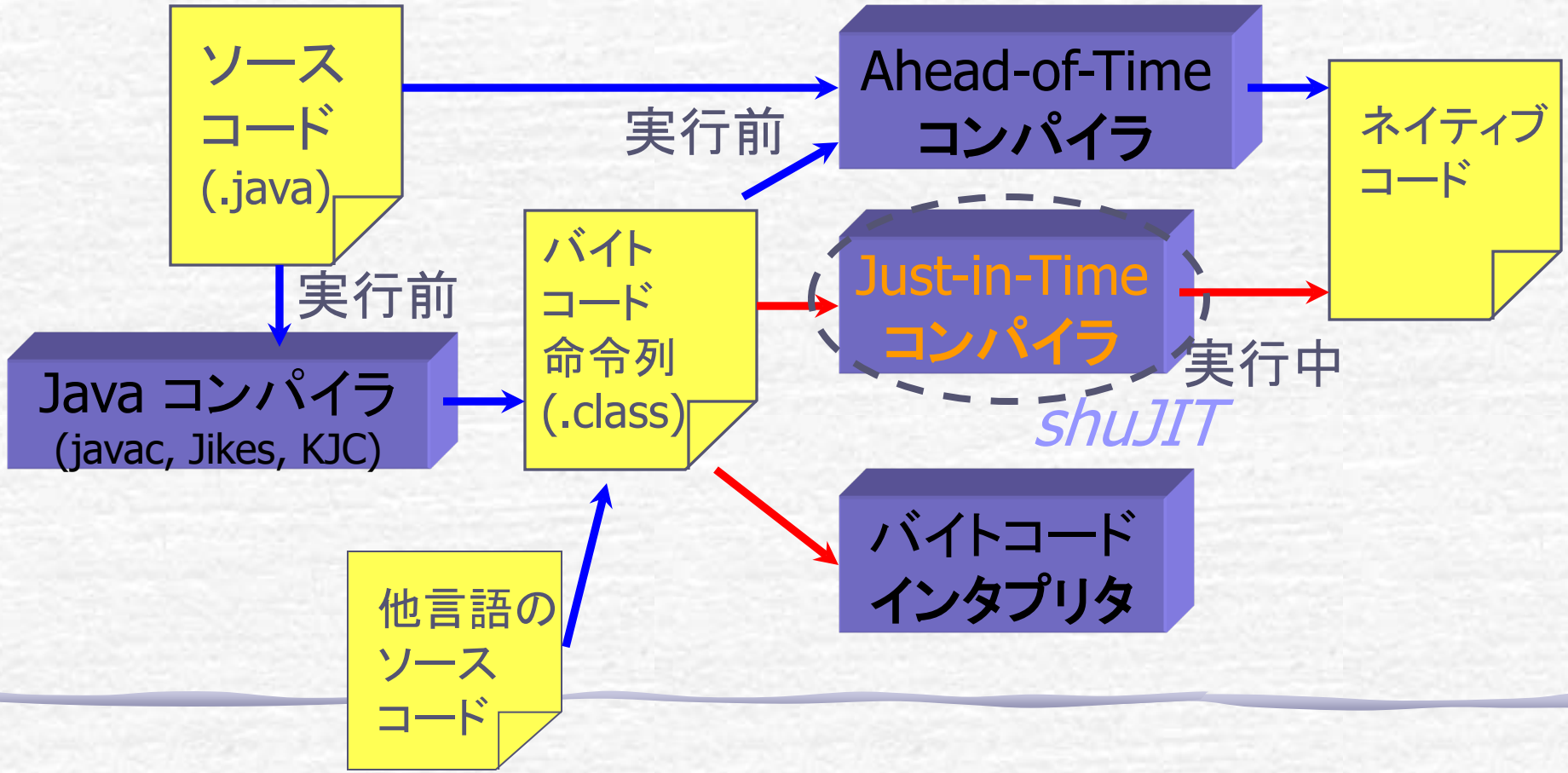
プログラミングシンポジウム

shuJIT

- 首藤作、x86/FreeBSD & Linux用 Javaバイトコード Just-in-Time コンパイラ。
- 短いコンパイル時間と実用的な性能。
  - JITは実行中にコンパイルするため、コンパイル自体の速さも重要。
- 諸研究の基盤として利用されてきた。
  - MetaVM：Java分散仮想マシン (首藤)
  - strictfpの効率的な実装 (首藤)
  - Jaguar (Matt Welsh, UCB)
- 1998年 9月公開。
- ダウンロード数 (2001年 3月9日まで)
  - ソースコード 7558, バイナリ 8476

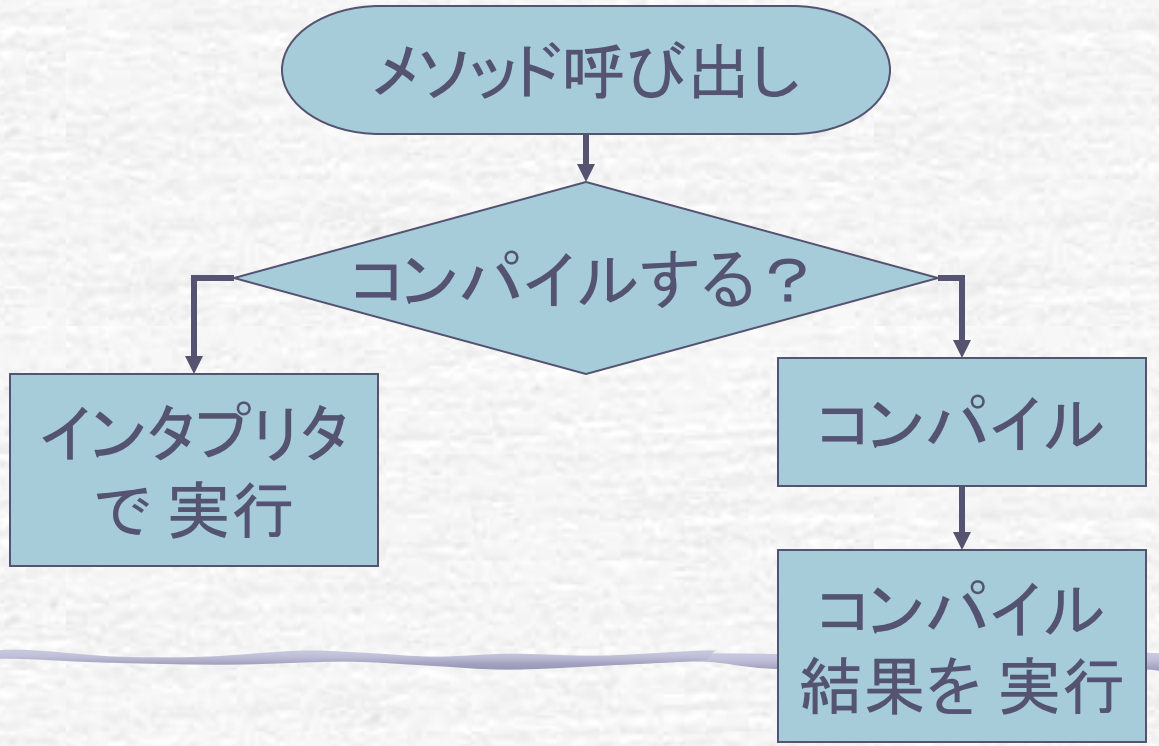
Just-in-Time コンパイル

- 実行時コンパイル



Just-in-Time コンパイル

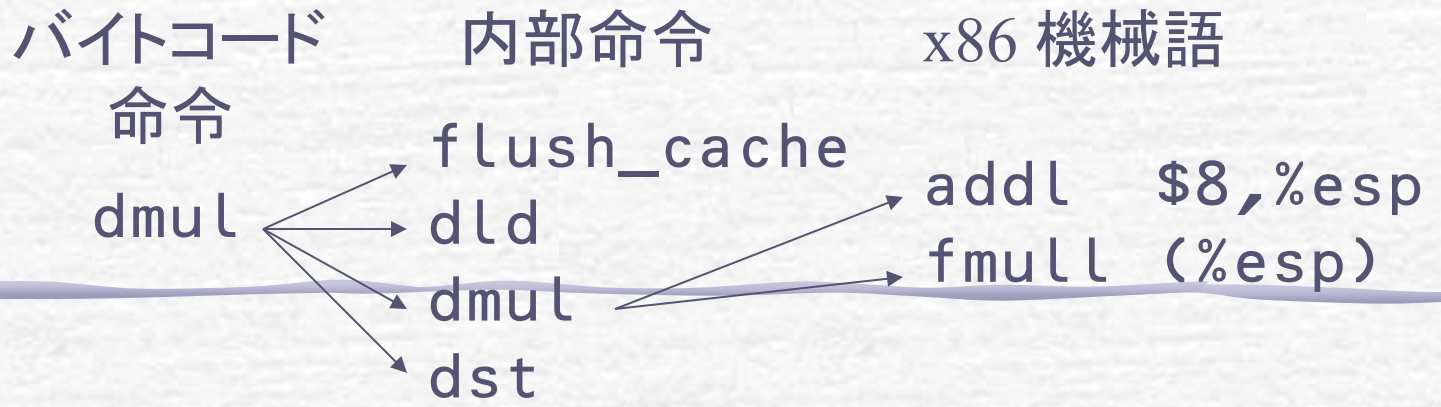
- メソッドを、呼び出された時点でコンパイル。
  - 性能向上 vs. コンパイル時間, メモリ消費 の トレードオフがある。要 コンパイル戦略。



インタプリタではなく、
コンパイル処理の軽い
JITを持つ処理系もある。
→ Level *N* JIT, baseline compiler
例) Intel ORP, IBM Jalapeño

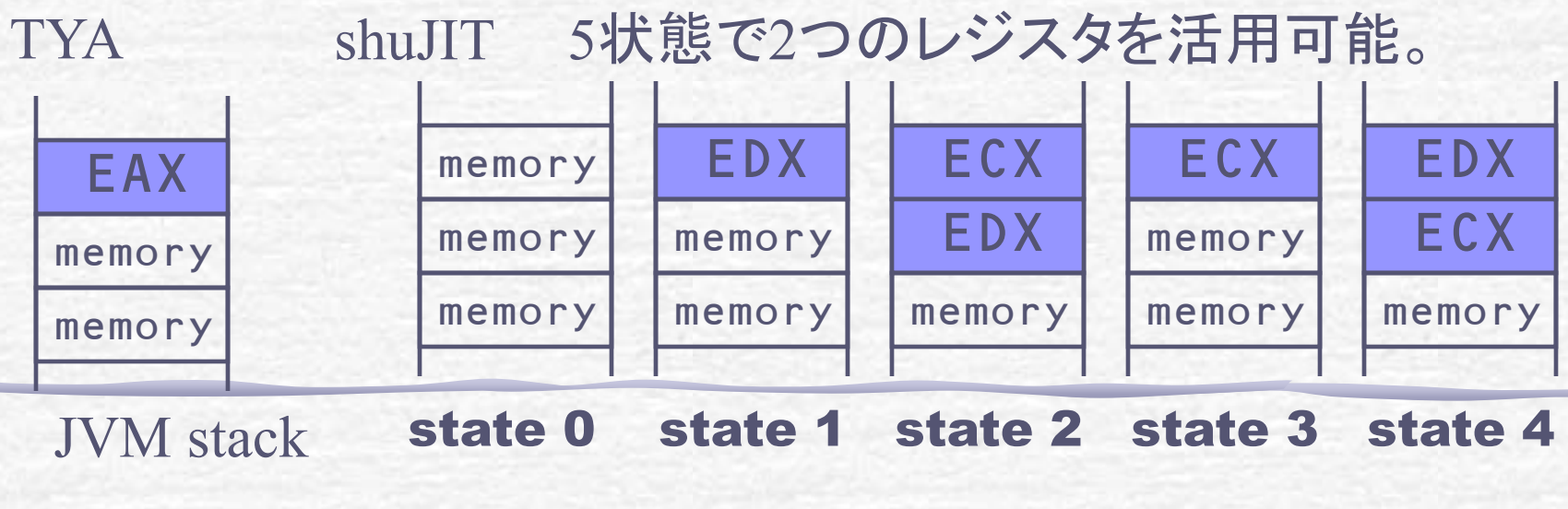
shuJITのコンパイル手法

- Javaバイトコード命令 → shuJIT 内部命令 → compiled code
  - たいていの内部命令はバイトコード命令に対応。
  - コンパイル処理が軽い。
    - コード長 n として O(n) という短いコンパイル時間。
    - 一般的なコンパイラが行うある種の解析はよりオーダが高い。
  - JIT内部に アセンブラ不要。
    - 開発コストが低い。

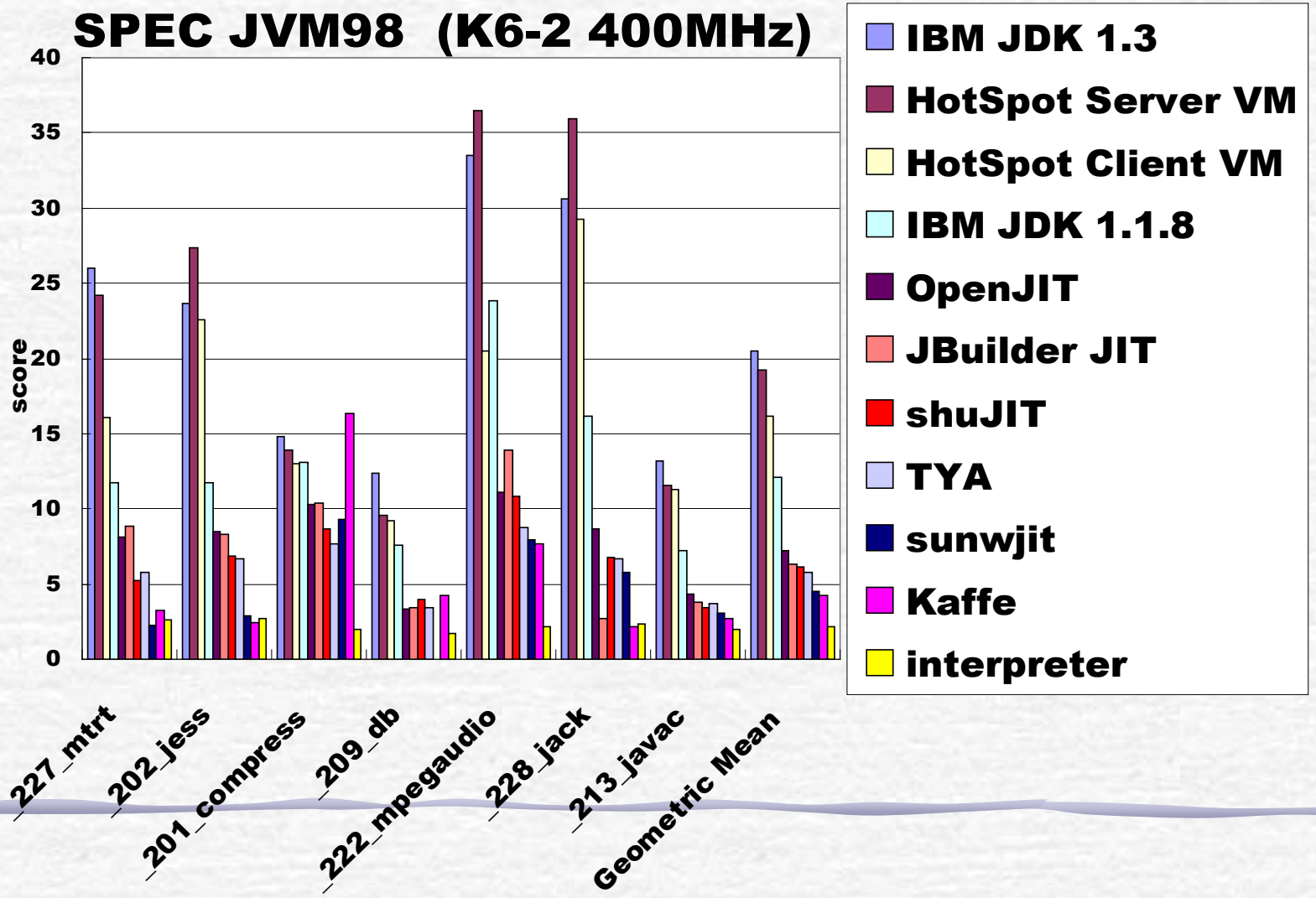


コンパイル手法 (cont'd)

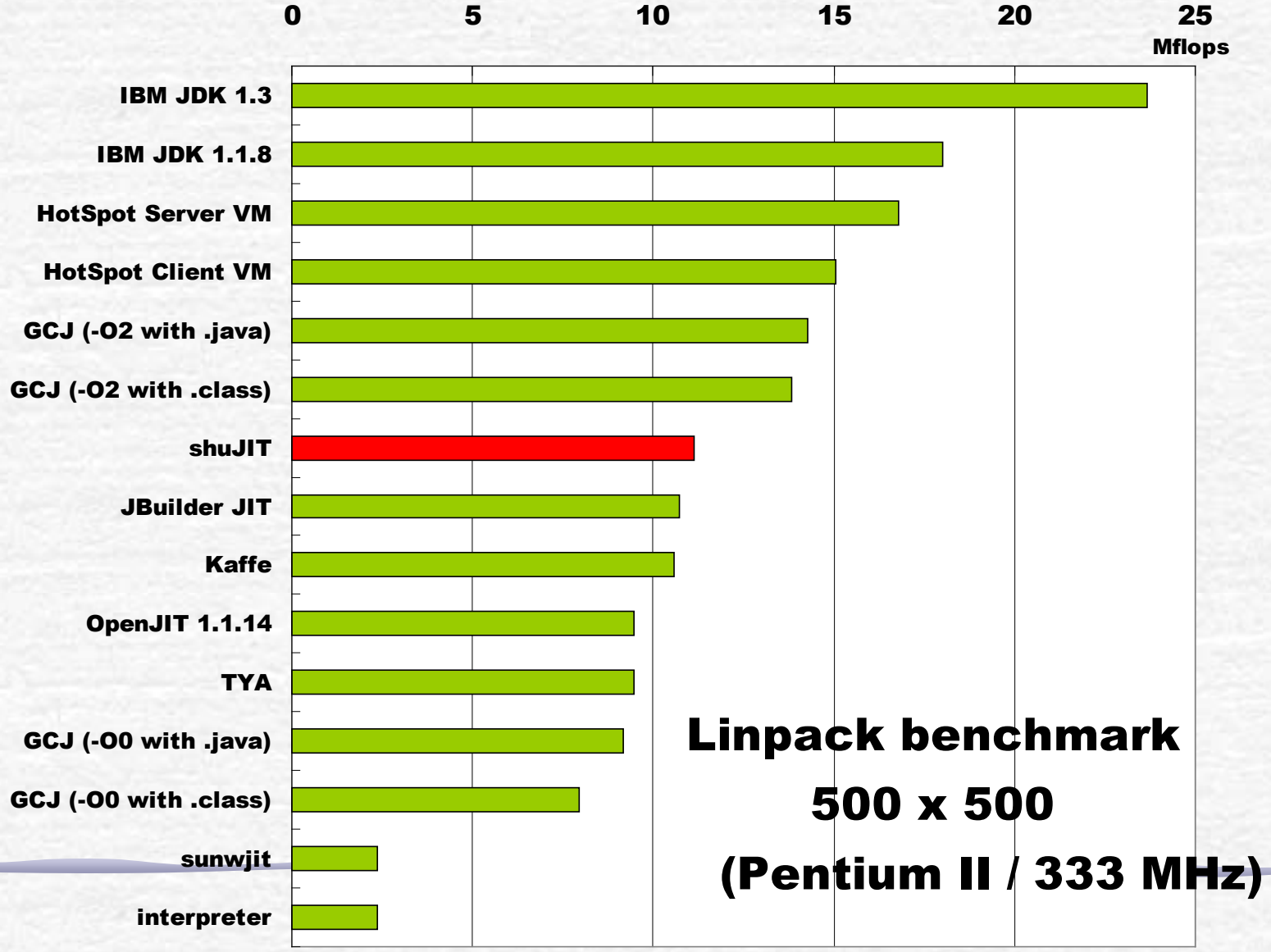
- 問題: いかにして複数のレジスタを活用するか。
  - 生成コードが固定のため、レジスタスケジューリングが難しい。
  - cf. TYA : 同様の手法でEAXレジスタのみ活用。
- 「スタック状態」(stack state) という考えを導入。



性能


SPEC JVM98 (K6-2 400MHz)
score
0
5
10
15
20
25
30
35
40
_227_mtrt
_202_jess
_201_compress
_209_db
_222_mpegaudio
_228_jack
_213_javac
Geometric Mean
IBM JDK 1.3
HotSpot Server VM
HotSpot Client VM
IBM JDK 1.1.8
OpenJIT
JBuilder JIT
shuJIT
TYA
sunwjit
Kaffe
interpreter

性能


0
5
10
15
20
25
Mflops
IBM JDK 1.3
IBM JDK 1.1.8
HotSpot Server VM
HotSpot Client VM
GCJ (-O2 with .java)
GCJ (-O2 with .class)
shuJIT
JBuilder JIT
Kaffe
OpenJIT 1.1.14
TYA
GCJ (-O0 with .java)
GCJ (-O0 with .class)
sunwjit
interpreter
Linpack benchmark
500 x 500
(Pentium II / 333 MHz)

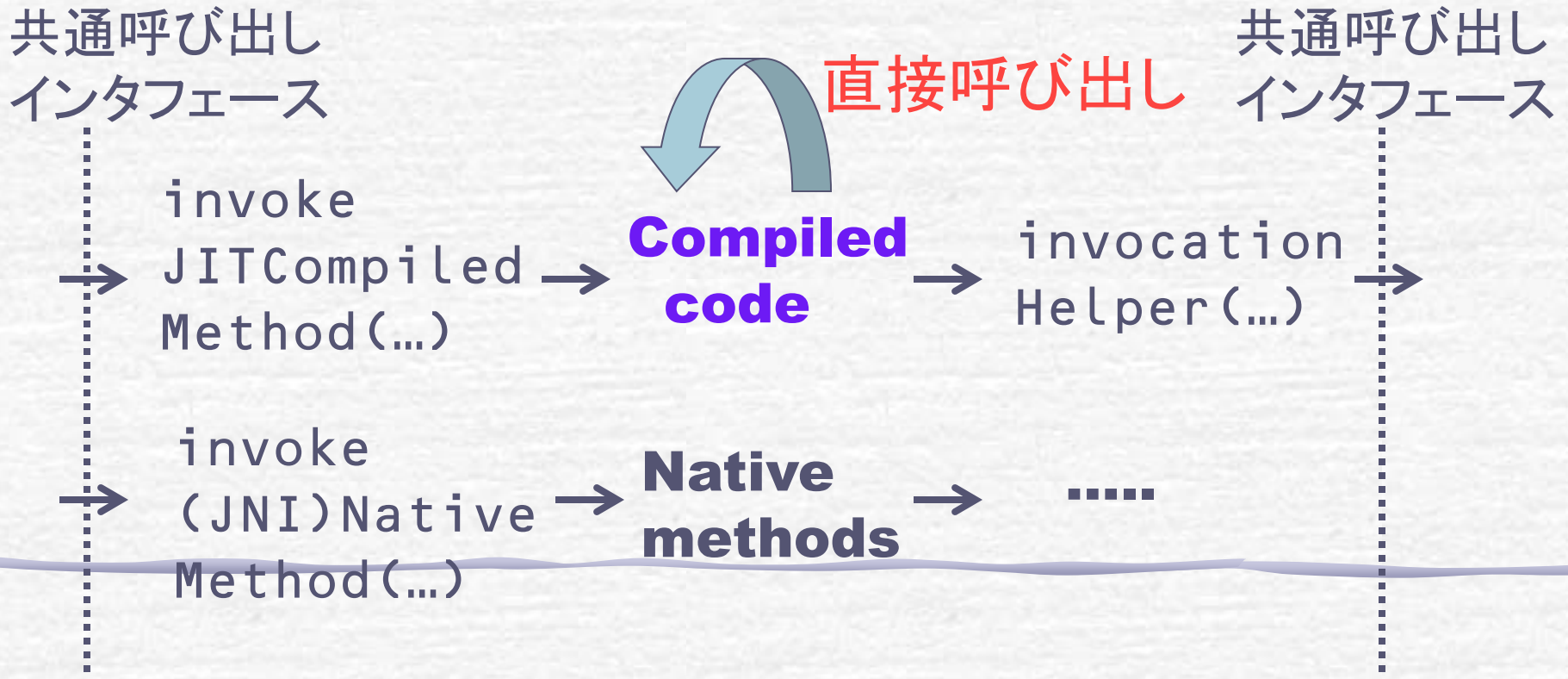
高速化手法

高速化手法 いろいろ

- メソッド呼び出し関係
  - compiled code間の直接呼び出し
  - 末尾再帰の除去
  - インライン展開 (一般, java.lang.Math の特定メソッド)
- ジャンプ関係
  - ループ先頭の alignment
  - 命令長のより短い命令への書き換え
  - 静的分岐予測を意識した条件分岐命令の選択
- その他
  - 各種 peephole 最適化
  - 1度きりの処理を自己書き換えで 2度目以降スキップ
  - 生成したコードの、次回実行時の再利用

Compiled Code間の直接呼出し

- メソッド呼び出しを高速化。
  - Compiled code間の呼び出しのみ速くなる。
  - 呼出し規約を決め、wrapper関数をそれに合わせた。



Compiled Code間の直接呼出し

- メソッド呼び出し高速化の効果
  - CaffeineMark 3.0のMethod
    - 直接呼び出しを行う 1812
    - 行わない 441
    - スコアが 4.1倍 に
      (Pentium II 333 MHz, JDK 1.1.8)
  - SPEC JVM98はメソッド呼び出し性能に非常に敏感なので、影響大。特にmtrt (レイトレーシング)。

末尾再帰の除去

- 末尾再帰 (tail recursion): 呼び出しの後、その返り値を返すだけの再帰呼び出し。
  - Lispなどの関数型言語で、繰り返しを記述するためによく用いられる。

```
public int aMethod(int a) {
    ……
    return aMethod(b);
}
```

末尾再帰

- 除去: 呼び出しをジャンプに変換する。
  - 容易に繰り返しに変換でき、スタックの消費を避けることができる。
  - Scheme処理系には実装が義務付けられている。
    - 末尾再帰を積極的に制御構造として用いることが推奨される言語であるため。
  - IBMのJITは実装している。
- shuJIT への実装。
  - バイトコード命令 invokespecial, invokestaticを独自の内部命令 invoke_recursive に変換。
    - IBM JITはversioningによってvirtual呼び出しに対しても適用。
  - 活きる局面が少ないため、性能的な効果は高くない。
    - SPEC JVM98 でもたかだか数ヶ所。

末尾再帰

- 失敗談
  - 当初、再帰呼び出しの直後に return している場合を末尾再帰だとみなしていた。
    - 「invokeなんとか, なんとかreturn」←末尾再帰
  - ところが、この単純な判定法には問題があった。
    - 呼び出しによって例外が throwされ、returnの前にcatch節が実行される場合があった。
    - Thanks 前田修吾さん。

実は
末尾再起ではない！

```
try {
    return aMethod(…);
}
catch (Exception e) {
    ……
}
```

シグナルの活用

- OSのシグナルを利用して例外を捕捉。
  - 条件分岐 (if) で判定するコストを省ける。
    - 正常系のコストをゼロにできる。
  - SIGSEGV で NullPointerException を検出する。
    - null はJava仮想マシン内で「0」で表現されている。
  - SIGFPE で ArithmeticException を検出する。
    - ゼロ除算。
  - SIGSEGV で StackOverflowError を検出する。
  - SIGTRAP (x86 INT 3 命令) + code patching で…
    - 後述

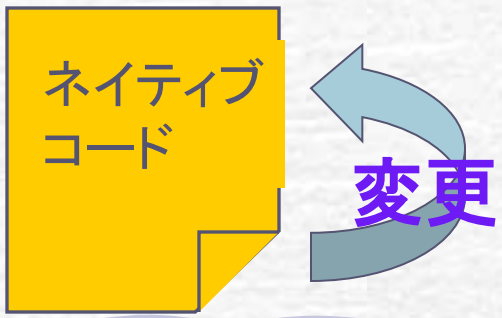
シグナルの活用

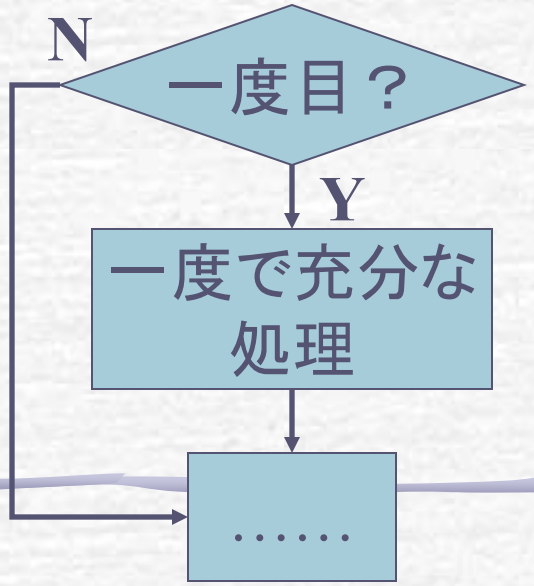
- シグナルハンドラの概要:

```
void signal_handler(int sig, …) {
        struct sigcontext *sc = …
                シグナルコンテクストを取得;

        switch (sig) {
                シグナルの種類に応じた処理
                        (例外のthrowなど)
        }

        シグナルコンテクスト中の
        プログラムカウンタを書き換える;

        return;
}
```

自己書き換え

- Compiled codeが自分自身を書き換える。
- 初めて実行されたときだけ行いたい処理を2度目以降はパスするために使用。
  - 条件分岐はペナルティが非常に大きい。
    - Pentium Proで分岐予測失敗時 9～26 クロック。
    - Pentium 4ではさらにひどいことになる。





自己書き換え

- 初回実行時のみの処理:
  - クラスの初期化
    - static変数アクセス、staticメソッド呼び出し、インスタンス生成
  - インスタンス生成時の諸チェック
    - interface や abstractクラスではないか？
      もしそうなら IllegalAccessError を throw
  - interface呼び出しの呼び出し先解決結果チェック
    - メソッドが見つからなければ、
      IncompatibleClassChangeError を throw

自己書き換え

- 生成される機械語命令列:

```
nop
nop
初回実行時のみの処理
「nop」を「jmp done」に書き換え
done:.....
```

- 一度 実行された後:

不要な処理を
スキップ

```
jmp done
初回実行時のみの処理
「nop」を「jmp done」に書き換え
done:.....
```

Code Patching

- 前述の自己書き換え方法では、
  コードの生成量が多い。
  - 一度しか実行されないコードが多数生成されてしまう！

生成されるコード:

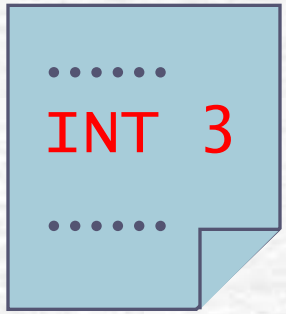
```
        jmp  done
        初回実行時のみの処理
        「nop」を「jmp  done」に書き換え
done:……
```

一度しか
実行されない

Code Patching

- ソフトウェア割り込みを使い、コード生成量を削減。
- x86 の INT 3 命令。
  - SIGTRAP を発生させる 1バイト命令 (0xCC)。
  - INT $0x10 といった 2バイト命令でもよいのだが、1バイト命令なら書き換えの atomicity を気にせずに済む。

コンパイル時:
生成したコードを
「INT 3」で上書き
しておく。



シグナルハンドラ:

```
void signal_handler(…) {
    ※ 一度きりの処理を行う;

    INT 3 を元のコードで
    上書きする;

    シグナルコンテクスト中の
    プログラムカウンタを設定;
}
```

※ 本当は、シグナルハンドラの文脈での
シグナル発生を防ぐため、トランポリンを作成する。

Code Patching

- コード生成量は減ったが、
  メモリ消費量は減らせなかった…
  - 生成コード量:
    - ジャンプ上書き法　1516 KB
    - INT 3 法　1408 KB

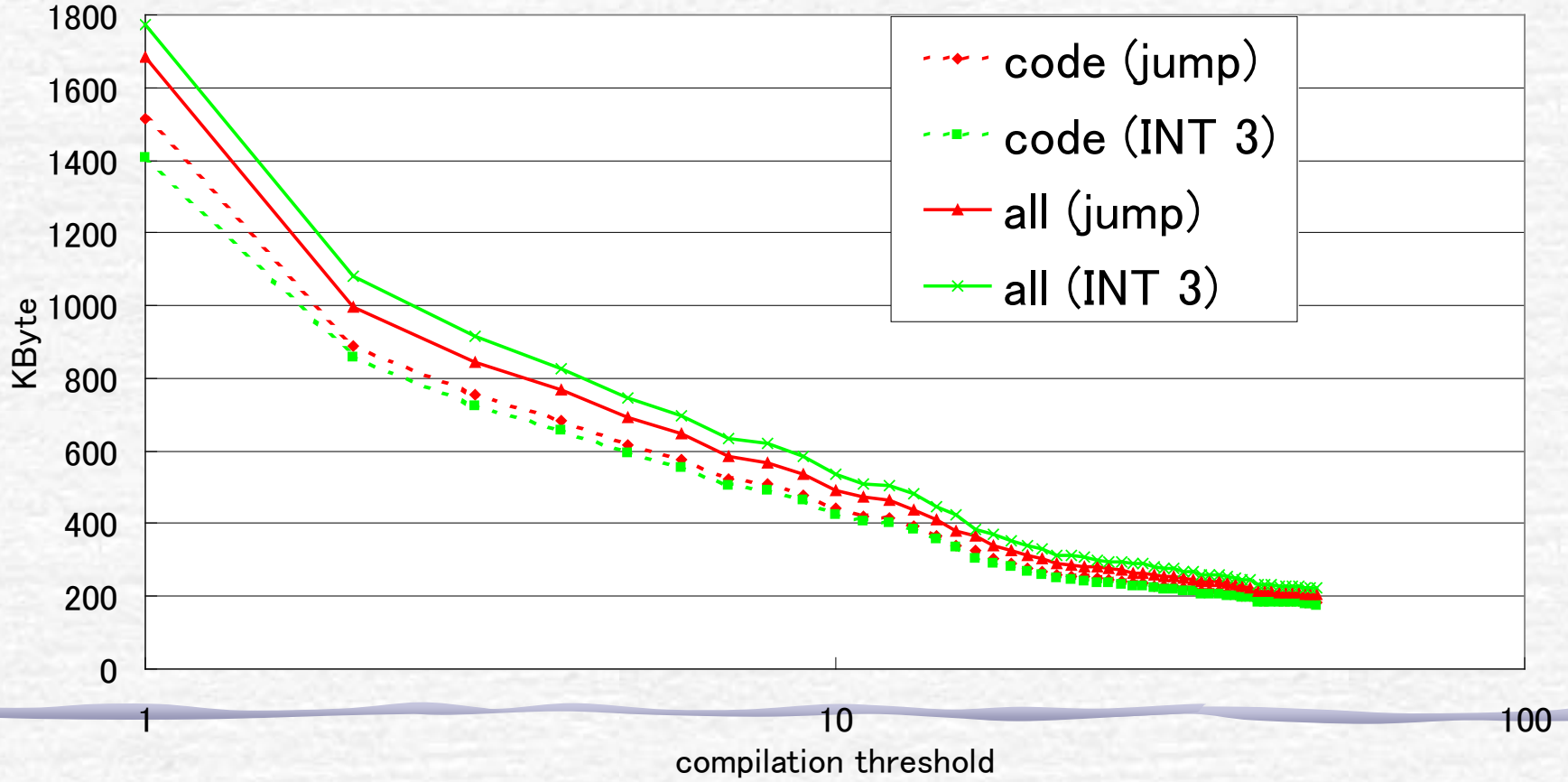
    ∨ 減った (^o^)/
  - 例外表まで含めると…:
    - ジャンプ上書き法　1682 KB
    - INT 3 法　1773 KB

    ∧ 増えた（泣）

例外表: シグナル発生時に、Javaバイトコード上のプログラムカウンタを得るための表。INT 3 法のために、ひとつのエントリが 7バイトから 14バイトになった。

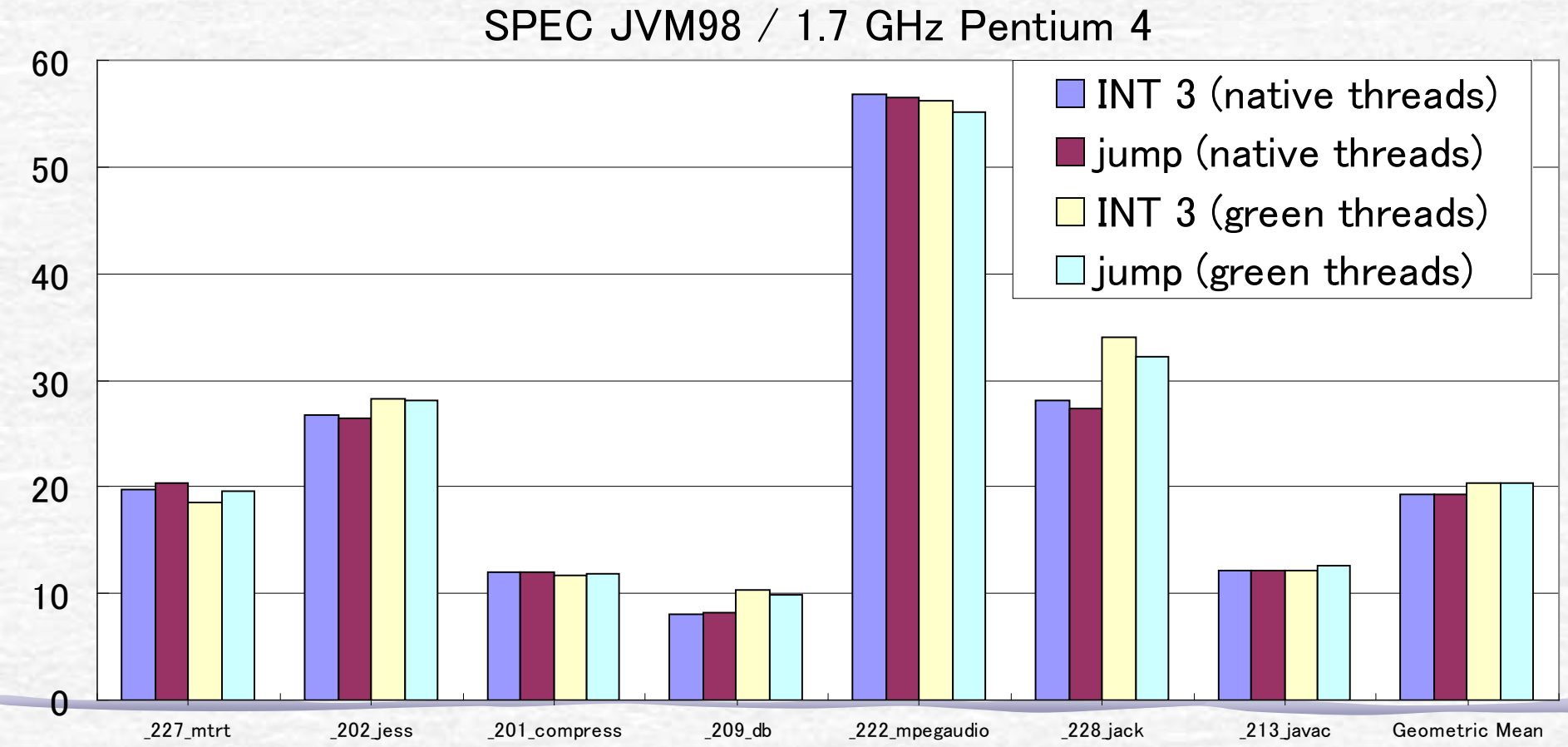
Code Patching

- コード生成量と、表も含めたメモリ消費量
  - プログラム: 何もしないアプレット



Code Patching

- 性能もほとんど変わらず…



インライン展開

- 通常の inlining
  - 条件: 展開対象メソッドが
    - catch節を持たない
    - ジャンプ命令を含まない
    - shuJIT 内部命令列の状態で、長さが **20**以下
  - ネストは **2** 重まで。 ← 変更可能
  - ↑ inliningの効果が高いメソッドは該当する。
    - 例) getter, setter メソッド
- 特定メソッドを特別扱い
  - java.lang.Math クラスのメソッド
    - sqrt,sin,cos,tan,atan2,atan,log,floor,ceil
    - 浮動小数点演算ユニットの命令を直接呼ぶようにコンパイル。
- 中身が空のメソッドの呼び出しを省く。
  - java.lang.Object のコンストラクタ呼び出しを省ける。

Peephole 最適化

- peephole: のぞき穴
  - ごく限られた範囲のコード列を見てのコード書き換えの総称。
- shuJIT での最適化
  - 浮動小数点数のコピー回数削減。
    - 例)「配列 → レジスタorスタック → FPUレジスタ」を直接一度のコピーに変換。
  - スタックとローカル変数間のコピー回数削減。
    - pop, push を、ローカル変数へのストアに変換
- 効果
  - プログラムによっては非常に大きい。
  - Linpack (500x500): 有 11.277, 無 6.351 (Mflops)

コードデータベース

- 生成したコードを保存し、
  次回以降の実行で再利用する仕掛け。
  - 無選択時設定では無効化されている。
- デメリットも大きい。
  - 実行時になってみないと行えない処理が多い。
    - 各種アドレスの解決
      - static変数、メソッド、Cの関数、…
    - 実行時に作られる表への書き込み
      - メソッドの例外表
  - あるクラスが初期化済みであることを仮定できない。
    - 通常は、コンパイル時に初期化済みでさえあれば、static変数アクセスなどの際の初期化コードを省けるのだが。
    - 結果、static変数アクセスやstaticメソッド呼び出しが遅くなり、生成コード量が大きくなる。

ループ先頭のalignment

- ループ先頭を 16バイト境界に整列させる。
  - 命令フェッチで確実に余計なクロックを防げる。
  - P6 (Pentium Pro以降) は命令を 16byteずつ最大3命令/クロック デコードする。
- 必ずしも速くなるとは限らない。
  - 整列によってできた空白には NOP を詰めたり、ジャンプ命令を入れたりするので、そのペナルティがかさむこともある。

ジャンプ命令の縮小

- 5バイト命令を 2バイト命令に変換。
  - 相対アドレスが 1バイトに収まるなら、「0xE9 XX XX XX XX」を「0xEB XX」に変換。
  - 命令フェッチ、デコードに良い効果。
  - CaffeineMark 3.0 の Logic: 4400 から 4900 に上昇。
  - 一般には効果は大きくない。

配列の境界チェックの改良

以前の方法　　条件分岐 2つ

```
if (index < 0)  exception;
if (index >= length)  exception;
```

現在の方法　　条件分岐 1つ

```
If (((unsigned)index - length) < 0)
     exception;
```

- キャリーフラグを見ることで、一度の条件分岐で済む。
- Thanks 石崎さん@ IBM 東京基礎研。

生成コード量の削減

生成コード量

- Compiled codeは大きい。
  - 一般にバイトコードの数倍～十数倍のサイズ。
    - shuJIT では 10倍～
  - Javaバイトコードは非常にコンパクト。
    - ネットワーク経由での転送を考慮している。
    - スタックマシン。(cf. レジスタマシン)
- 一般に、コード量は少ない方がよい。
  - 短いコードは速いことが多い。
  - I キャッシュに載りやすい。
  - 性能とコンパクトさのトレードオフもあり得る。
    - x86の高機能な命令 vs. シンプルな命令の組み合わせ
    - インライン展開。
  - コンパイル自体を行わないという選択もある。
    - コンパイル時間の節約にもなる。

生成コード量の削減手法

- 複数メソッドに共通の処理を、メソッドの外に移す。
  - 例外の扱い。
    - catch節の探索やcatch節へのジャンプ。
    - 例外のインスタンス作成。
- 例外を発生し得ないメソッドについては、例外処理のコードを生成しない。
  - メソッド内の全命令をチェック。

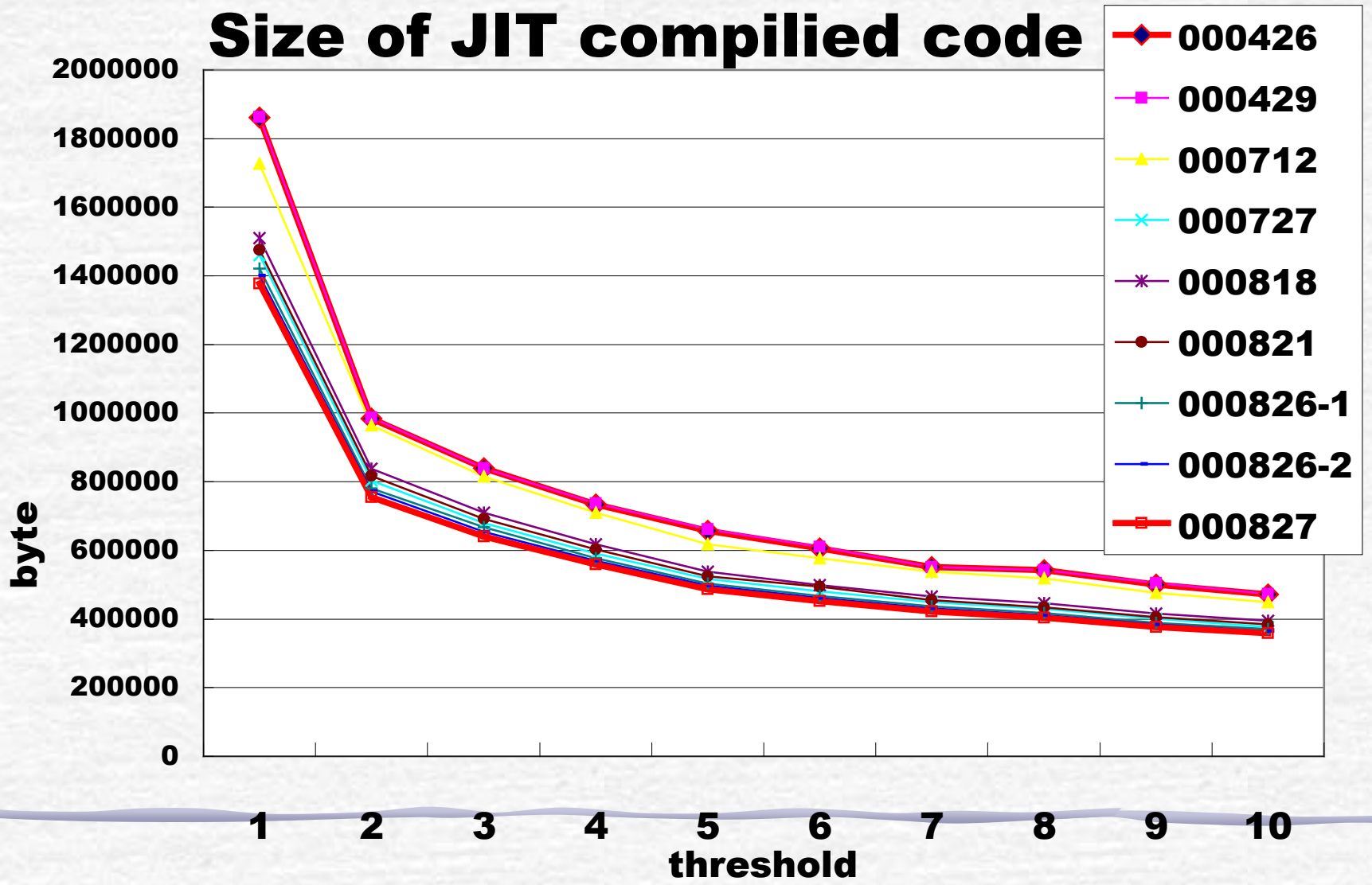
選択的コンパイル

- クラスの初期化メソッドはコンパイルしない。
  - クラスがロードされた時しか実行されないため。
- n回呼び出された時点で初めてコンパイル。
  - Adaptive compilationの一種。
    Lazy compilation
  - あまり呼ばれないメソッドをコンパイルする無駄を防げる。
    - コンパイル時間とメモリ、両方の無駄。
  - shuJITでは n を任意の値に設定可能。

生成コード量の変遷

- 生成コード量 / バイトコードの量
  - n : Lazy compilation の threshold
  - n = 2
    - 2000年 7月 12日　15.28
    - 2000年 8月 28日　11.43
  - n = 50
    - 2000年 7月 12日　12.92
    - 2000年 8月 28日　9.97

生成コード量の変遷


Size of JIT compilied code
2000000
1800000
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
byte
1
2
3
4
5
6
7
8
9
10
threshold
000426
000429
000712
000727
000818
000821
000826-1
000826-2
000827

まとめ (?)

- 報告した内容
  - JIT コンパイルの基礎
  - shuJIT のコンパイル手法, 性能
  - 高速化手法 いろいろ
  - 生成コードの削減法とその成果
- 結論
  - 言語処理系は楽しい。