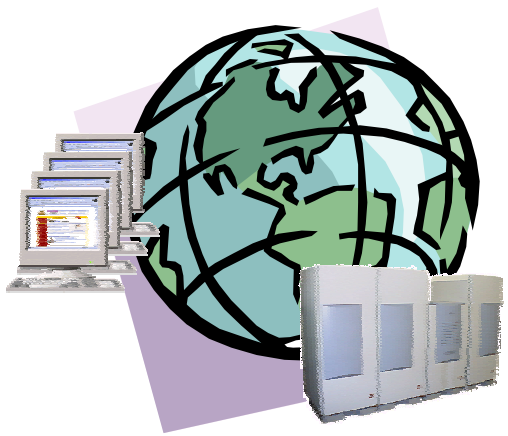
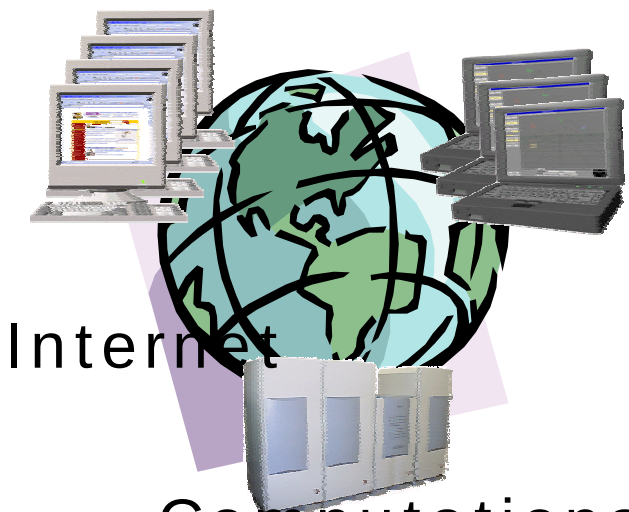
メタコンピュータ構成方式の研究

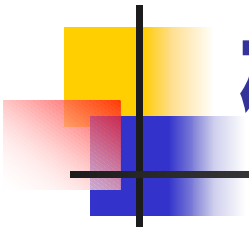


情報科学専攻
村岡洋一 研究室
首藤 一幸

メタコンピュータ

- 概念：
  - 単一の目的に利用できる、ネットワーク上に分散した コンピュータ群。
  - 一台の仮想コンピュータ。
  - グローバル(広域) コンピューティング研究の目的のひとつ。
- 目的
  - 高性能計算。
    - 並列処理, 負荷分散
  - 分散したデバイスの利用。
    - 天体望遠鏡, 大容量ストレージ

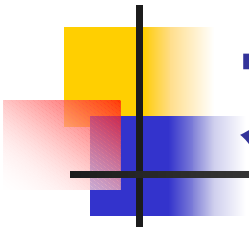




研究の目的

- **プログラマにメタコンピュータを提供する。**
  - **コンピュータ群を 一台であるかのように扱えるように。**
  - **分散処理のための労力を減らす。**
  - **かつ、性能や機能を最大限引き出す。**

- **現在の分散処理**
  - **通信についての労力が非常に大きい。**
    - **ソケット**, MPI, PVM　　- **明示的に記述**
    - RPC, **分散オブジェクト**(ORB)
      - **独特の作法, 配置の手間, できることが限られる**
  - **単一コンピュータと、様々な点で異なる。**

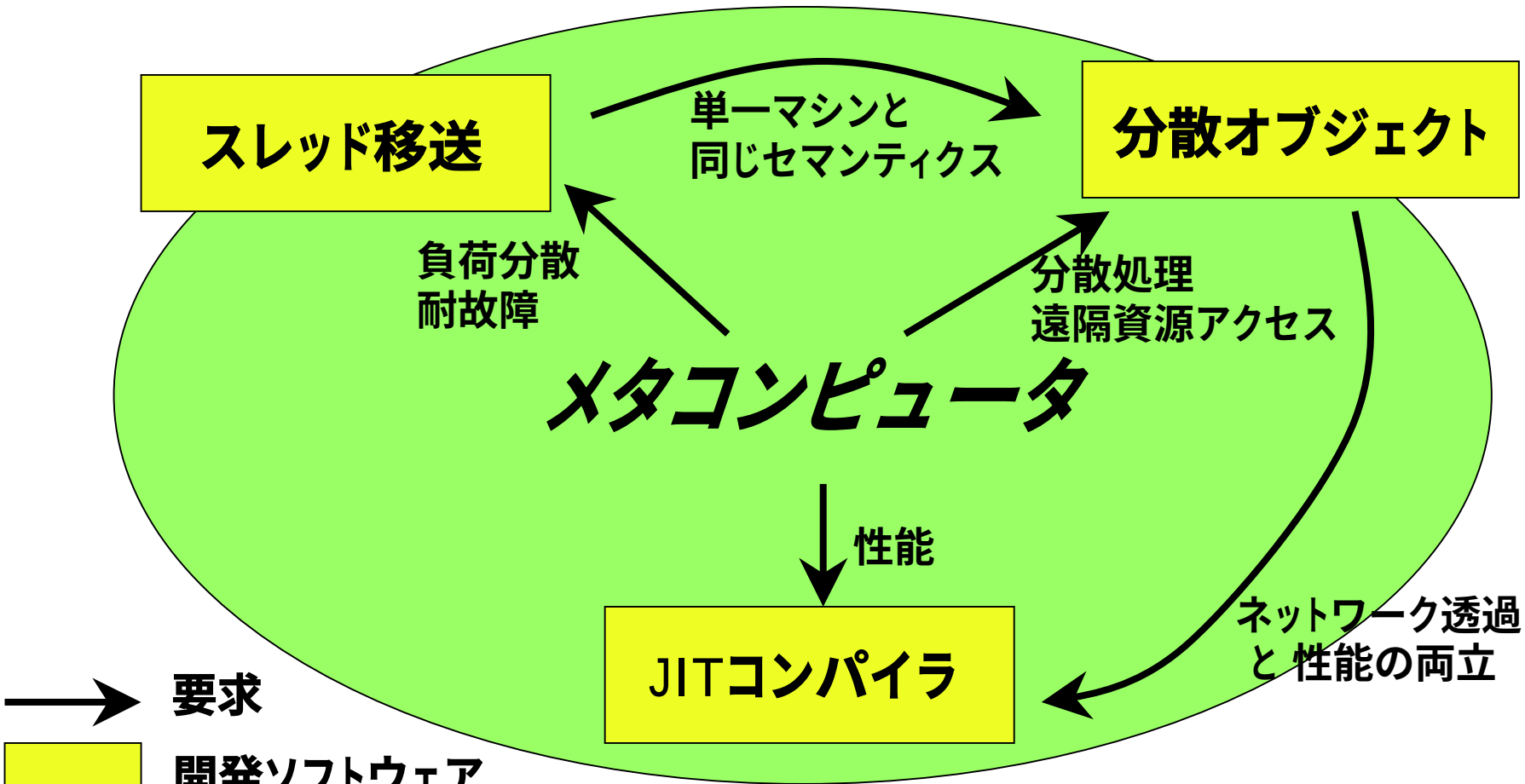


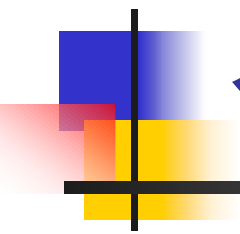
方針

- Java 仮想マシン (JVM) に機能を付加する。
  - なぜJava ?
    機種非依存, 安全性, 高速実行 をはじめて両立。
    - 移動コードを安心して実行できる。
      code verification & sand box
    - 実行時コンパイラによる高速実行。
  - → 分散処理に非常に適している。

- メタコンピュータ、すなわち
  分散仮想マシン の構成に不可欠な機能を付加。
  - スレッド移送
  - オブジェクトの分散

分散 Java仮想マシン

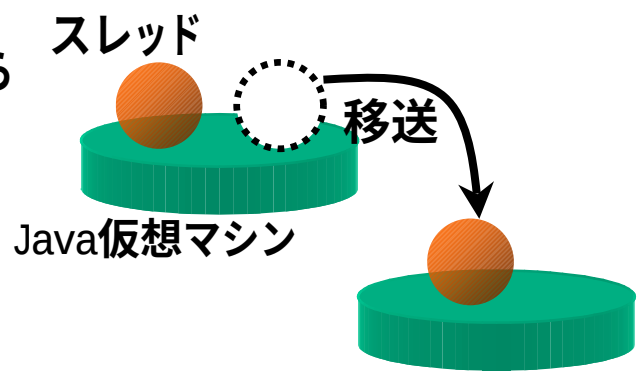




スレッド移送

スレッド移送

- スレッドを Java仮想マシン (JVM) 間で移送。
  - スレッド: プログラム中の、ひとつの実行の流れ。
  - 実行中の状態を保ったまま、別コンピュータに移動、実行を再開。
- 特徴
  - 異機種間移送。
    - Java仮想マシンの実行状態を移送することで達成。
  - 非同期移送。
    - 移送対象の外 (他スレッド, 利用者) から移送を指示できる。
      - cf. プログラム変換による手法。
- 応用
  - 動的 負荷分散
  - 移動エージェントの記述, disconnected operation
  - 耐故障 - 別コンピュータに逃がす
  - …

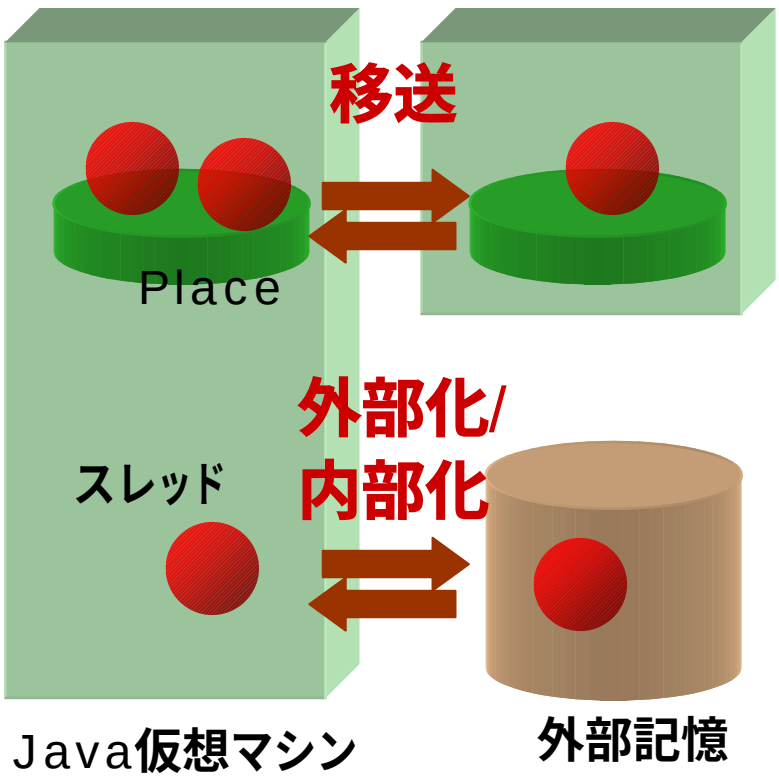


ユーザインタフェース

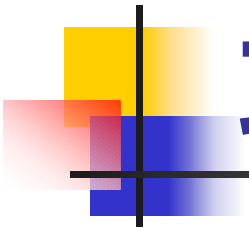
- **プログラミングインタフェース**
  - **通常のプログラムに移送文を挿入できる。**
    …**任意の処理**…
    MobaThread.goTo(**移送先**);
    …**任意の処理**…
  - cf. Aglets (IBM), Voyager (ObjectSpace)
    - **移送を意識して記述せねばならない。コールバック。**
- **グラフィカル** UI
- **キャラクタ** UI

GUI

移送の方法



- 手順
  - 状態をバイト列に変換(外部化)
  - 転送
  - バイト列からスレッドを生成(内部化)
- その他、すべきこと。
  - 排他制御，スレッドの停止
  - 自身の外部化のためのスレッド生成
  - JVMの下 (ライブラリ，OS) のスレッド生成

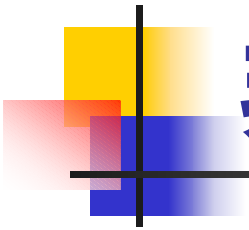


取り組み

- 実装することで…
  - Java仮想マシンで可能であることを示した。
  - 種々の問題を明らかにし、解決法を示した。
    - 非同期移送
    - スタック上の要素の型判定
    - 移送するオブジェクトの選択
    - 位置依存オブジェクトの扱い
    - …

問題の例：非同期移送

- 任意タイミングでの移送
- 移送対象スレッドの外からの指示による移送
  - 利用者や他のスレッドからの指示
  - スレッド自身による移送はプログラム中に記述されるため、タイミングが既知
    - 例）goTo(移送先), migrateTo(移送先)
- 動的負荷分散には必須

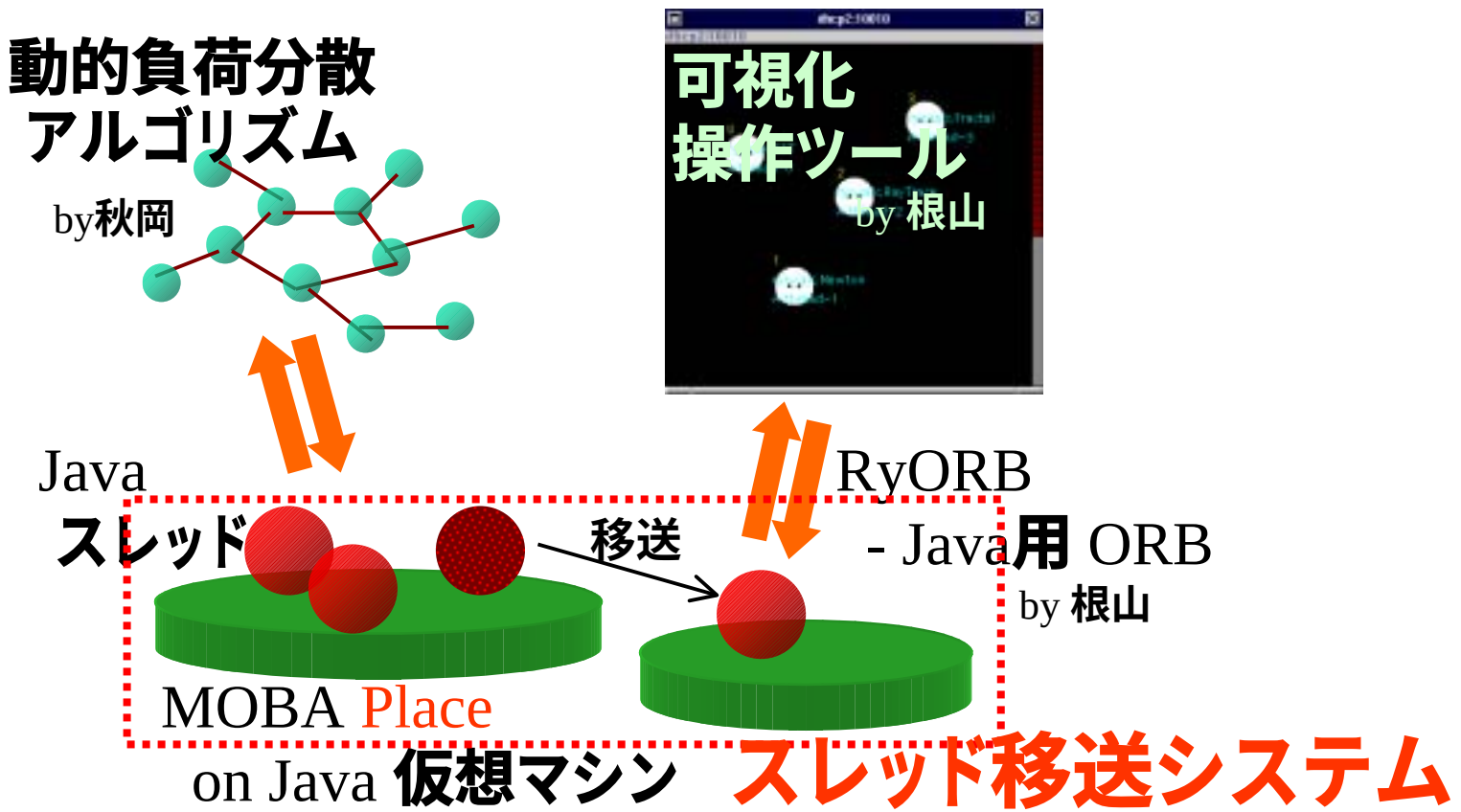


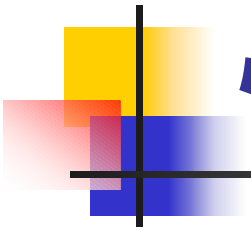
非同期移送 (cont'd)

- 問題
  - Migration safe point でのスレッドの停止
    - JVM の一命令のために、プロセッサは多くの命令を実行する。
    - 停止させたスレッドの状態は JVM 的に一貫していない恐れがある。
      - cf. スタック, プログラムカウンタ, ヒープ, …
- 解決法
  - JVM 下のスレッドライブラリとしてユーザ空間の実装を用いる。
- よりよい解決法
  - Code patching
    - Sun Lab. の JVM で、GC (garbage collection) safe point での停止を保証するために用いられている。

デモシステム

- メタコンピュータのプロトタイプ





性能評価

- 負荷分散のためには、移送にかかる時間は短い方が良い。
  - 移送するかしないかは、負荷分散による利得と移送にかかる時間的損失 で判断するため。
- 評価
  - 移送遅延
  - データ転送のスループット

移送の遅延

- Voyager (**移動エージェント**) と比較。
- 2**台のコンピュータ間を往復させ、片道の時間を算出。**
- **環境**:
  - UltraSPARC-II 167 MHz，UltraSPARC-II 296 MHz
  - SunOS 5
  - 100 Mbps **イーサネット**

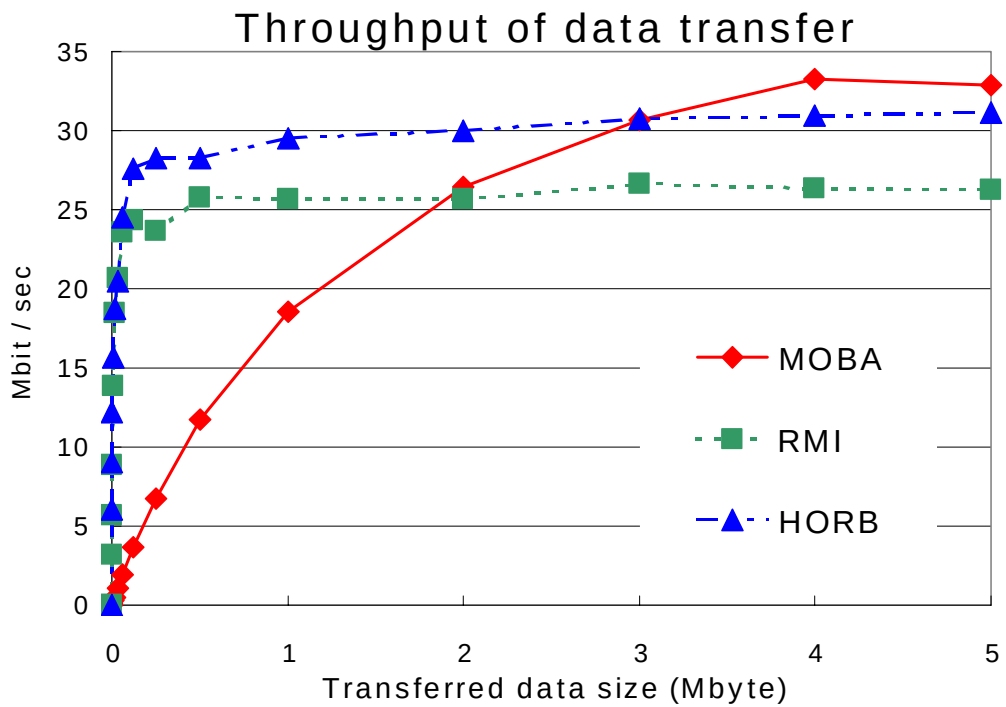
往復回数	1	10	20	50
MOBA	191.0	109.3	105.53	105.32
Voyager	292.5	57.05	44.00	37.08

ミリ秒

- Voyager **は** start-up **に時間がかかる。**
- MOBA **は、実行状態を移送する割には 速い。**

スループット

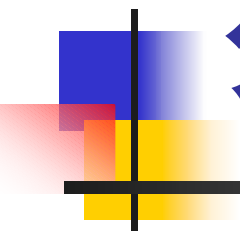
- いくつかの Object Request Broker (ORB) と比較。
  - 遠隔実行に応用した場合を想定。
- 64 bit 浮動小数点数の 配列の転送性能
  - MOBA: 配列を伴って移動。
  - ORBs: 配列を引数に遠隔メソッド呼び出し。
- 実行状態を移送する分、オーバヘッドが大きい。
- 転送量が多いと、オーバヘッドが隠れる。



スレッド移送のまとめ

- Java 仮想マシン上で可能であることを実証。
- 実用的なシステムを提供。
- 異機種間移送 と 非同期移送を両立。
- 問題を明らかにした。
  - 非同期移送, 実行時コンパイラとの共存, スタック上要素の型判定, 移送するオブジェクトの選択, 位置依存オブジェクトの扱い

- 分散オブジェクトの必要性：
  - 分散オブジェクトがないと…
    移送に伴い、オブジェクトがコピーされてしまい、
    一台上での実行と分散実行で、セマンティクスが変わってしまう。



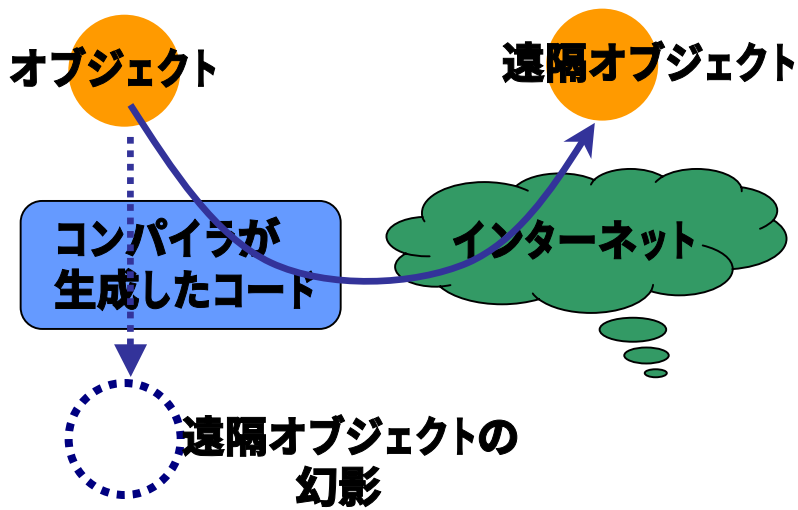
分散オブジェクト

ネットワーク透過な 分散オブジェクト システム

- **遠隔**オブジェクトと **ローカル**オブジェクトを 全く同じに扱える。 区別不要。
  - オブジェクトの位置を意識せずに済む。
  - 分散処理を容易に記述できる。

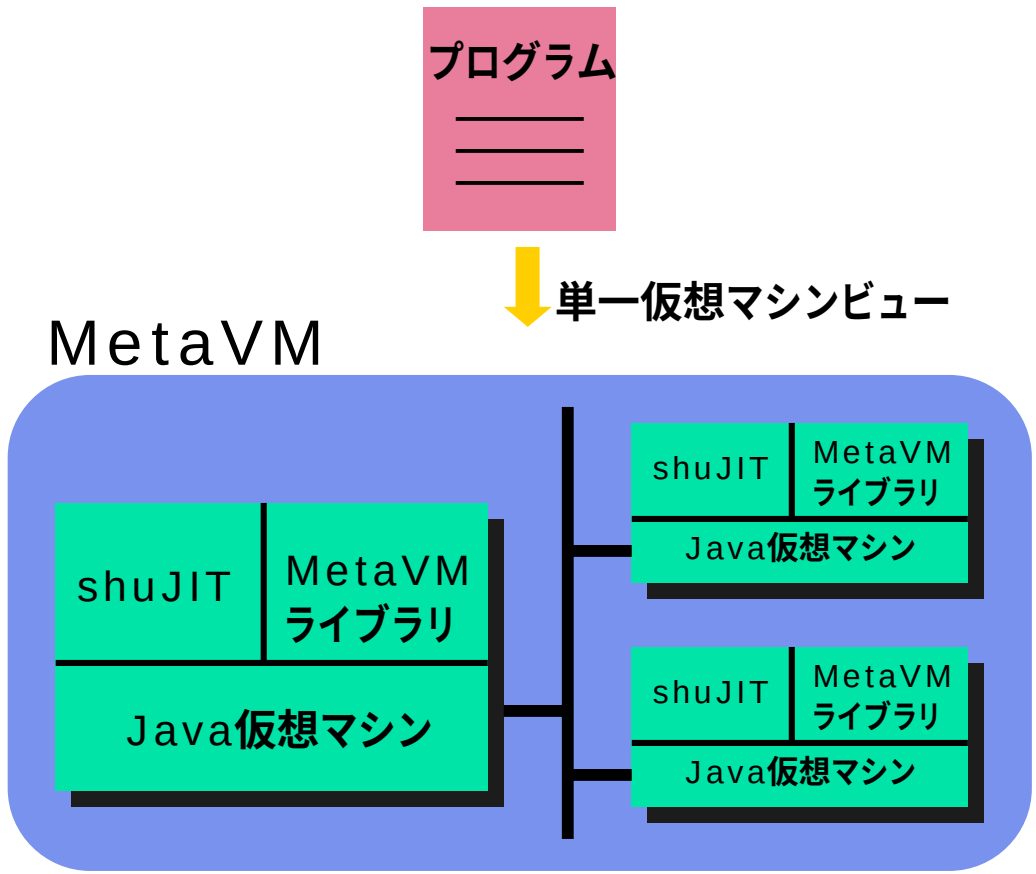
```
...
remoteObj = new Object();
remoteObj . variable = 500;
...
```

- 手法: **実行時コンパイラ(JIT)** による意味拡張
  - 分散と高速実行の両立。
    - cf. cJVM (IBM) ←分散のみ
  - Semantic Expansion (右図)
    - Java仮想マシンの命令の 意味を変更。
  - コンパイラから開発。
    - 研究基盤、かつ、実用的。



システムの構成

- shuJIT
  - Java バイトコードの実行時 (JIT) コンパイラ
  - 遠隔参照を扱えるネイティブコードを生成。
- ライブラリ
  - 分散オブジェクトシステムの機能を提供。



プログラミングインタフェース

- Java 言語 + オブジェクト生成先の指定 のみ

```
VMAddress addr = new VMAddress(“ホスト名”);
MetaVM.instantiationVM(addr);
          // 生成先の指定
Object obj = new  Object();
          // オブジェクトの遠隔生成
MetaVM.instantiationVM();
          // 今後の生成はローカルに
```

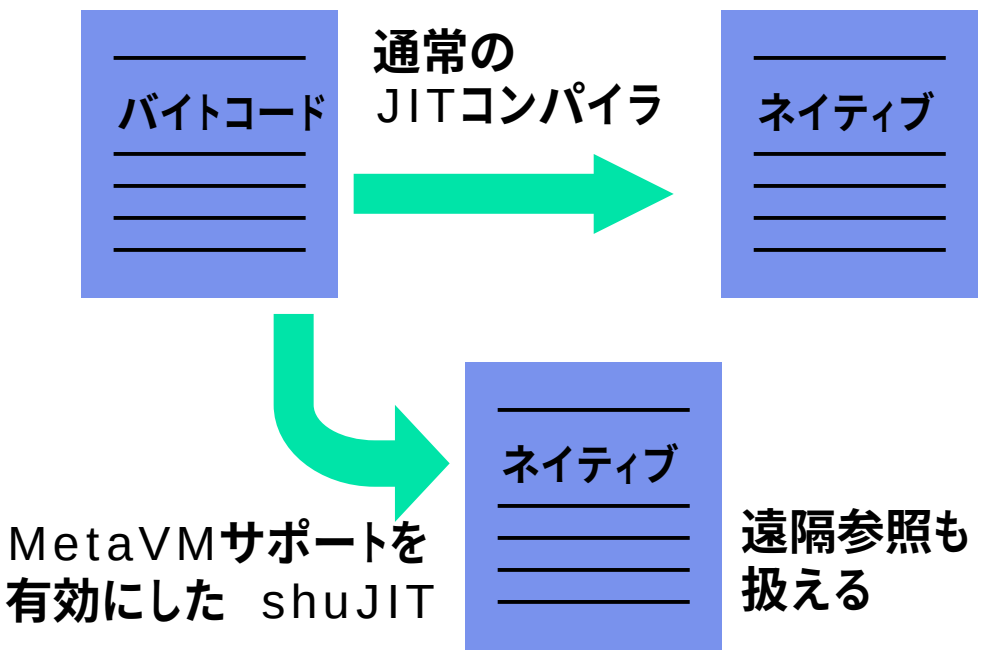
- 発展： 生成先の自動決定

手法：
実行時コンパイラによる 意味拡張

- 通常、JIT コンパイラに求められるのは
  - JVM の仕様通りで
  - 高速な
    ネイティブコードの生成　のみである。
- しかし…
  - プログラムはバイトコードで表現されるものの、
    実際に実行されるのは JIT が生成したコードである。
  - JVM によるバイトコード命令の解釈は JIT が決めている。
  - JIT はバイトコード命令の意味を変えることができる。

意味拡張の
分散オブジェクトへの応用

- JIT による意味拡張を用いた
  ネットワーク透過な 分散オブジェクトシステム

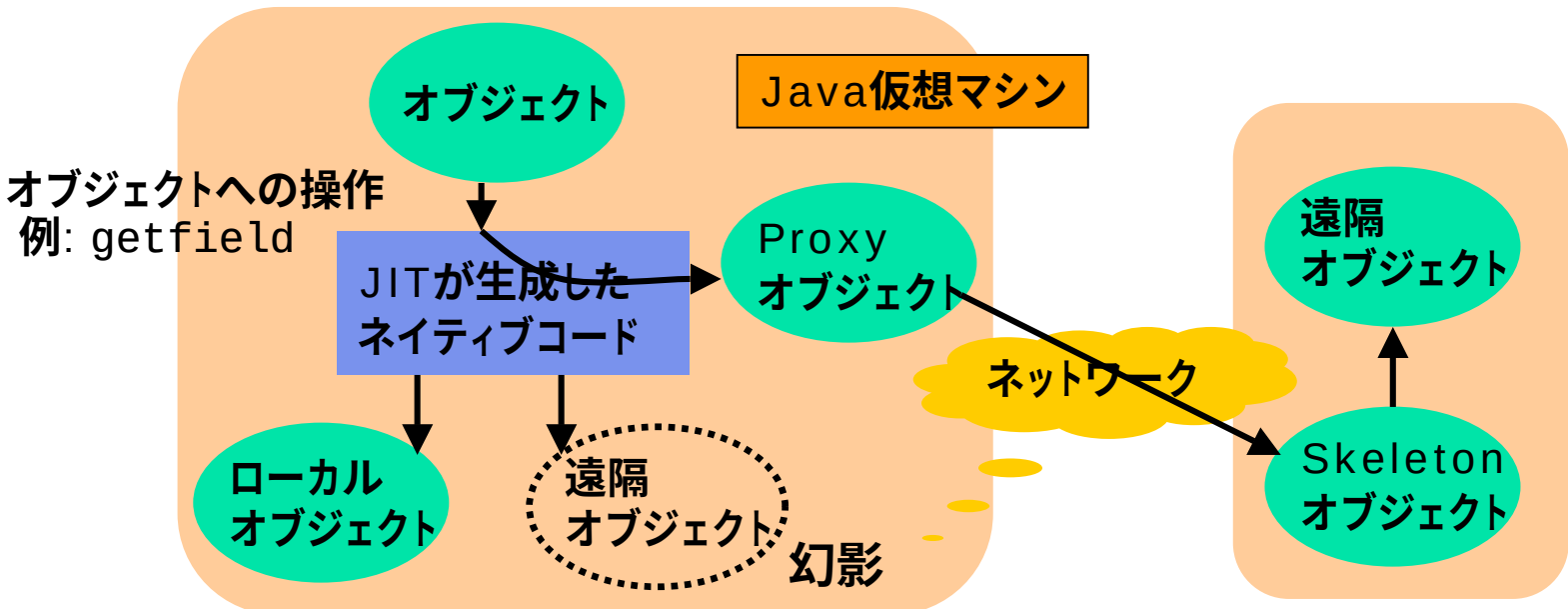


意味拡張の対象となるバイトコード命令

- 200 **強のうちの** 30 **命令。**
  - **オブジェクトの生成**
    - **配列以外** new
    - **配列** newarray, anewarray, multianewarray
  - **アクセス**
    - **フィールド** getfield, putfield
    - **配列の要素** [ailfdbcs]aload, [ailfdbcs]astore
  - **配列長の取得** arraylength
  - **メソッド呼び出し** invoke{virtual, special, interface}
  - **モニタの扱い** monitorenter, monitorexit
  - **型チェック** checkcast, instanceof

オブジェクトへの操作の遠隔への中継

- ネイティブコードが 操作を遠隔に中継する。
  - MetaVM ライブラリを呼び出す。
- ユーザプログラム，バイトコードからは、遠隔オブジェクトがローカル…と同じに見える。



遠隔参照対応コードの生成

- **オブジェクトへの操作：**
  操作対象が遠隔参照かどうかチェックする。

```
if ((obj instanceof Proxy) && remote_flag)
    obj は遠隔参照。 MetaVM ライブラリに処理させる。
 else
    obj はローカル参照。 通常の処理を行う。
```

- **注釈)**
  - Proxy
    - **遠隔参照を表す** MetaVM **ライブラリ中のクラス**
  - remote_flag
    - **スレッドに属するフラグ**
    - **プログラムに対して、**Proxy **オブジェクトを遠隔オブジェクトとしてみせるか、そのまま見せるか。**
    - MetaVM **ライブラリの実装に必要。**

遠隔参照対応コードの生成 (cont'd)

- **オブジェクトの生成：**
  遠隔に生成すべきかチェックする。

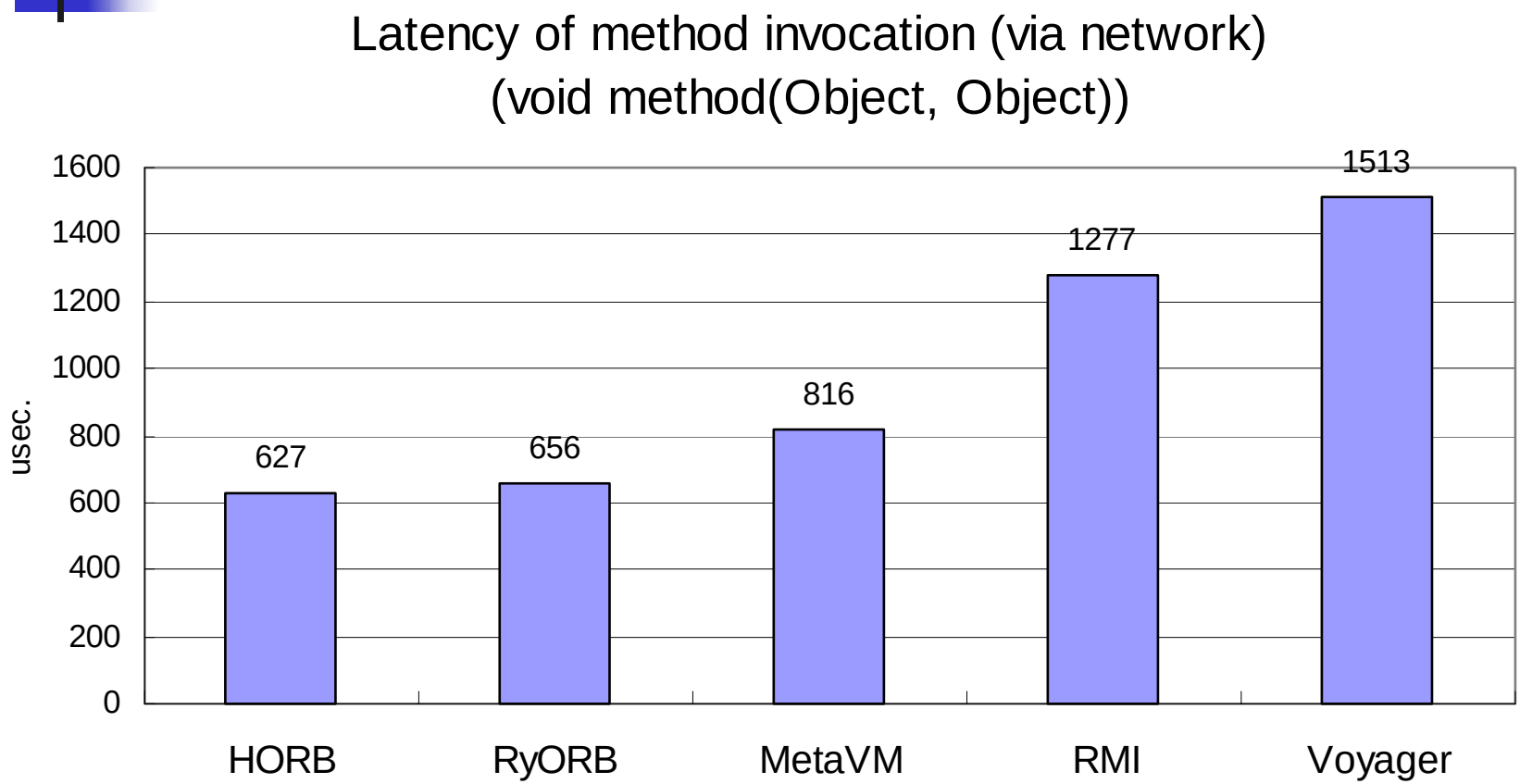
```
if ((生成先が遠隔) && !(コピー渡しのクラス))
    遠隔に生成
else
    ローカルに生成
```

- **注釈）　コピー渡しのクラス**
  - ネットワーク越しに、参照ではなくてコピーが渡るクラス。
  - `Throwable, InetAddress`
    - 実装の都合。
  - `Number, Boolean, Character`
    - Immutable であるため、コピーしても差し支えない。

性能評価

- 評価項目
  - 遠隔メソッド呼び出しと、遠隔アクセスの遅延。
  - 遠隔操作を行わないプログラムへの影響。
    - 操作対象が遠隔にあるかローカルかを判定するオーバヘッドが常にある。
- 比較対象
  - Java RMI, Voyager 2.0.2, HORB 1.3 beta4
- 環境
  - マシン 1: Pentium III 600 MHz, Linux 2.4.0, JDK 1.2.2
  - マシン 2: K6-2 400 MHz, Linux 2.2.16, JDK 1.2.2
  - マシン1から 2に対して遠隔操作。
  - 100 Mbps イーサネット。スイッチを 1台経由。

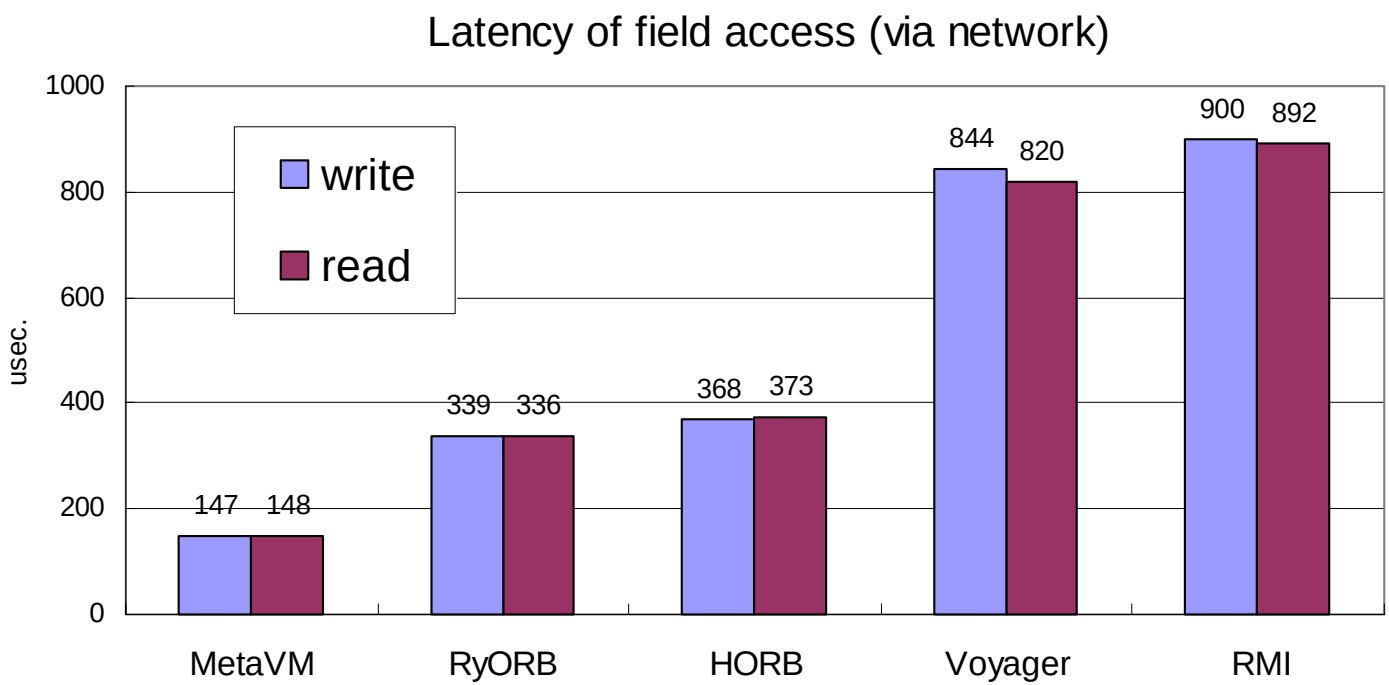
遅延: 遠隔メソッド呼び出し



- RMI, Voyager **よりある程度速く**
RyORB, HORB **より若干遅い。**

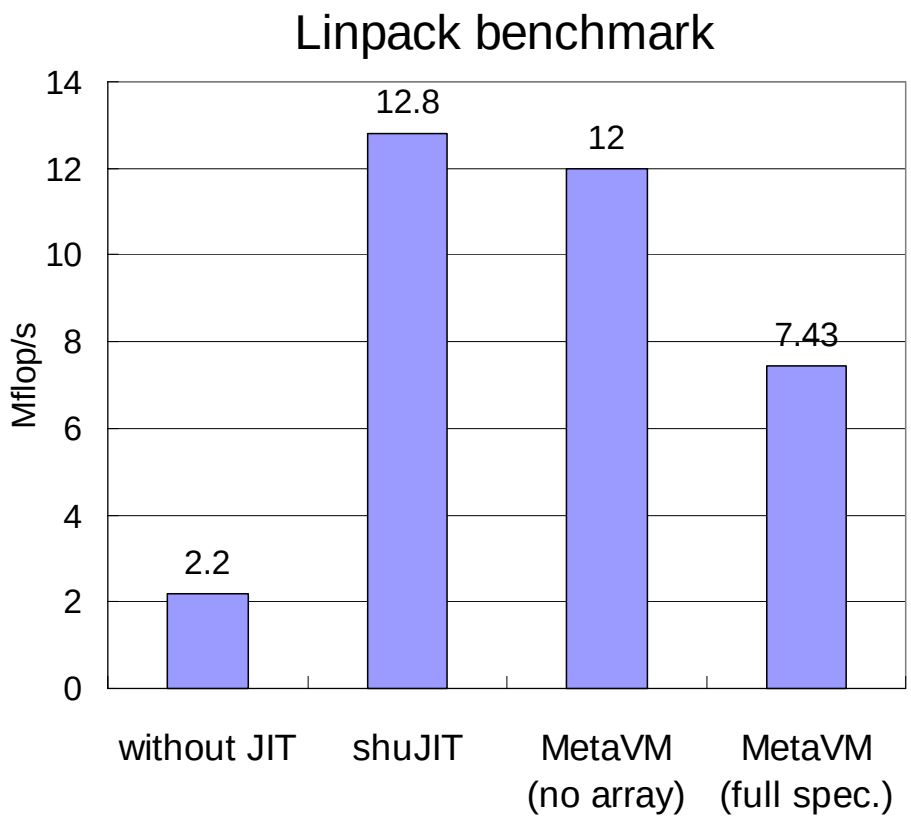
遅延: 遠隔フィールドアクセス

- int型フィールドの読み書き。
- MetaVM 以外はこの機能を持たないため、
  そのためのメソッドを用意した。
  - int get(), void set(int value)

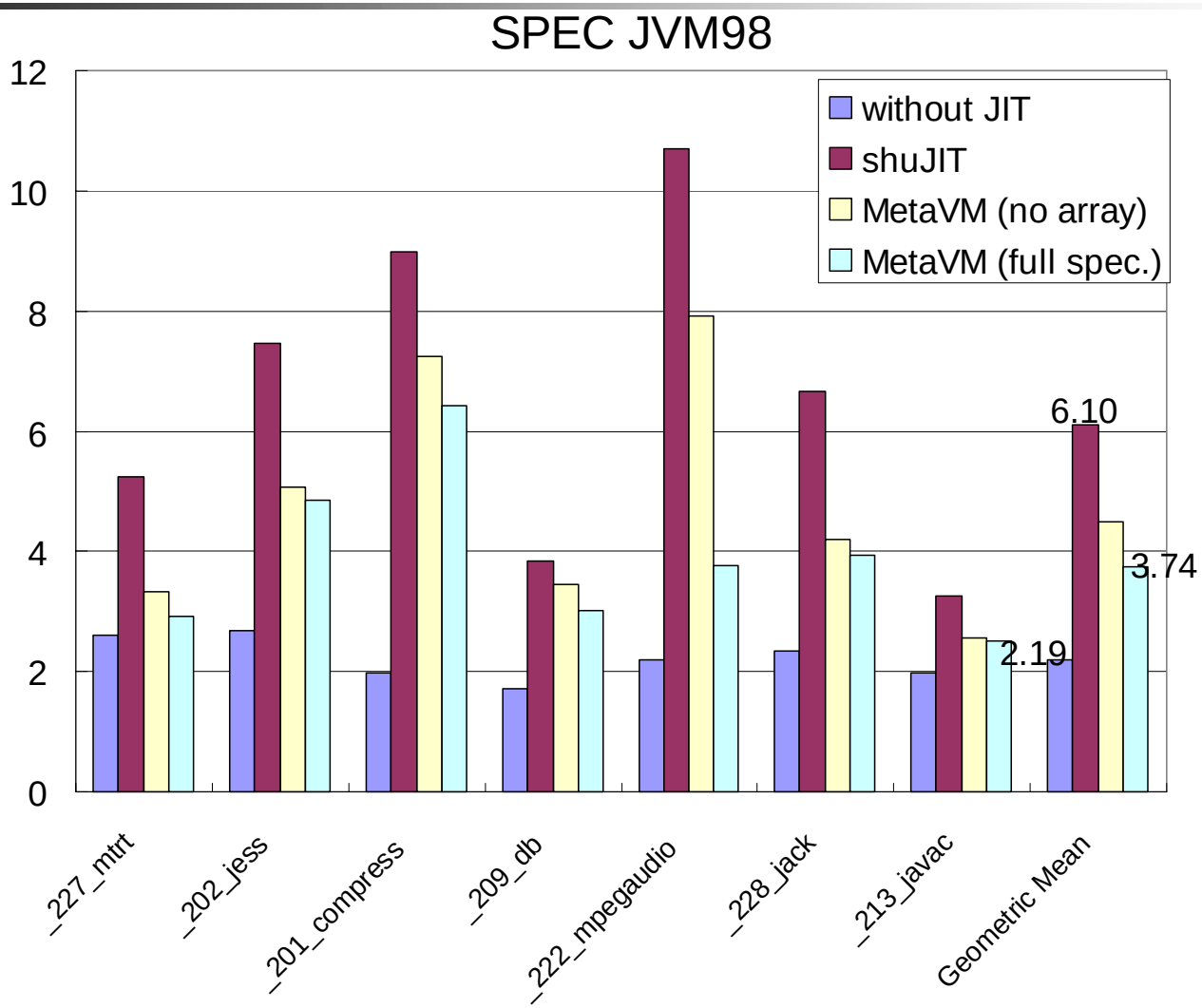


ローカル実行性能への影響

- Linpack benchmark
  - 500 x 500
  - **配列アクセスが多いため、`no array’ ではあまり性能が低下していない。**



ローカル実行性能への影響



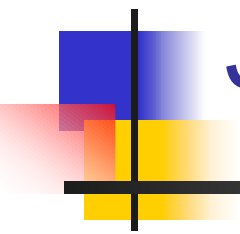
性能評価結果

- **遠隔操作の遅延**
  - メソッド呼び出しの遅延は、既存システム並。
  - 固有の機能であるフィールド、配列アクセスでは 高性能。
- **ローカル実行性能への影響**
  - 常に インタプリタより良い結果。
  - 配列を遠隔参照する機能を無効にすることで、プログラムによっては性能低下を軽減できる。(i.e. グラフ中の `no array')

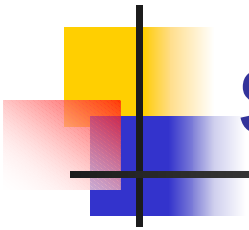
分散オブジェクトのまとめ

- 既存 ORB のような制約のない、ネットワーク透過なシステムを構成できた。
  - cf. RMI, HORB, CORBA, …
- 実行時 (JIT) コンパイラによる意味拡張を提案し、応用して見せた。

本システム MetaVM は shuJIT
(JIT コンパイラ) を元に実装されている…



Java Just-in-Time コンパイラ

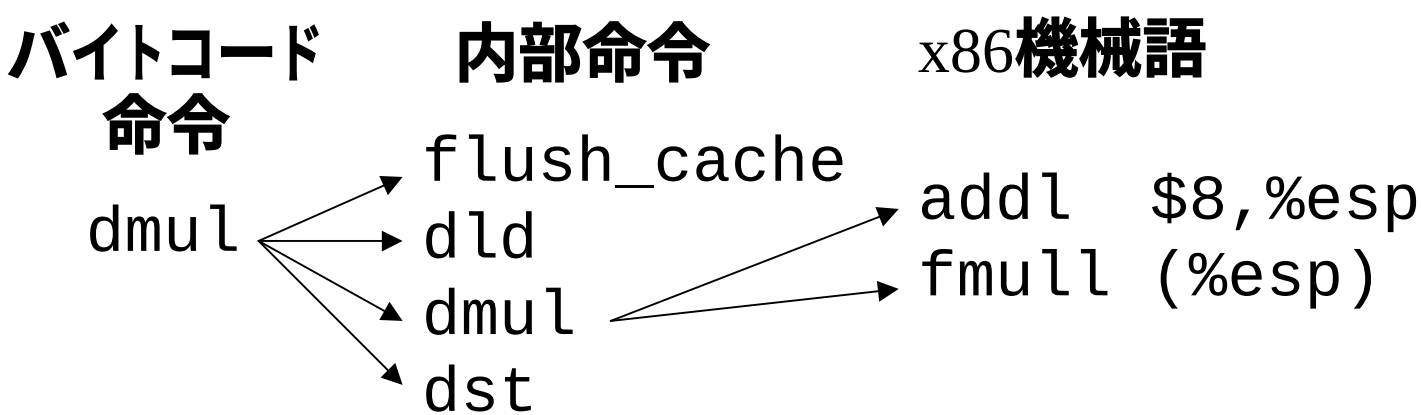


shuJIT

- x86, Sun**社の**JVM, Linux & FreeBSD **用の** JIT **コンパイラ。**
- **短いコンパイル時間と 実用的な性能。**
  - **実行中にコンパイルする**JIT**にとって 速いコンパイルは重要。**
- **諸研究の基盤として利用されている。**
  - MataVM : **分散**Java**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**: **ソース** 7055, **バイナリ** 8143

コンパイル手法

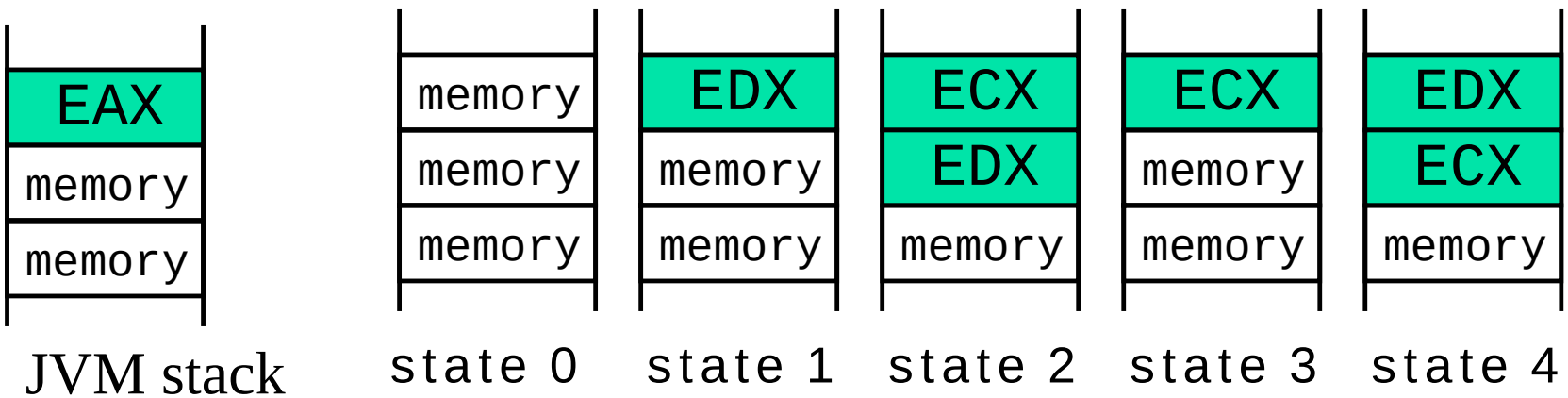
- Java**バイトコード命令** → shuJIT **内部命令** → compiled code
  - **たいていの内部命令はバイトコード命令に対応。**
  - **コード長 n として** O(n) **という短いコンパイル時間。**
    - **一般的なコンパイラが行うある種の解析はよりオーダが高い。**
  - **コンパイル処理が軽い。**

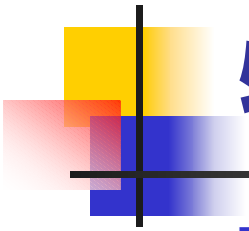


コンパイル手法 (cont'd)

- 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、レジスタ割り付けが難しい。
  - 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。
- → 短いコンパイル時間で実用的な性能を達成。

TYA　　shuJIT　　5状態で2つのレジスタを活用可能。





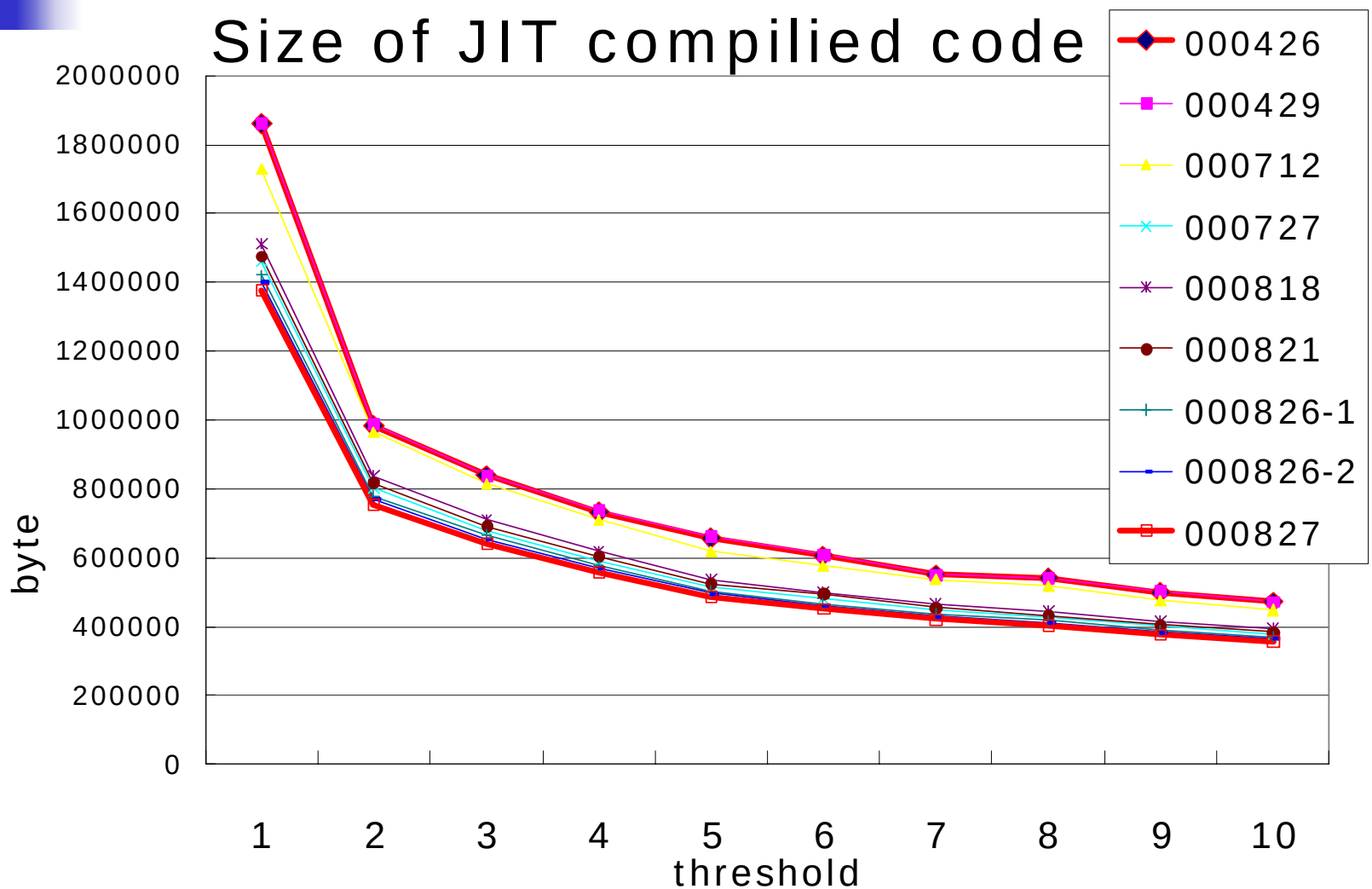
特徴

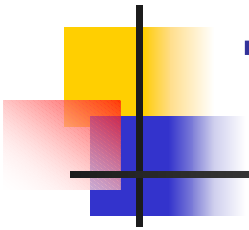
- 短いコンパイル時間で実用的な性能。(前述)
- strictfp に対応。
  - 知る限り、対応している唯一の Java 実行系。
  - 浮動小数点演算のセマンティクスを、仕様通りに忠実に実装。i.e. IEEE 754
- コンパイル結果をファイルに保存、後の実行で再利用可能。
  - 静的に行える処理と、実行時にしか行えない処理をはっきりと区別した。
  - コンパイル結果を、コンパイル時の JVM 状態に依存させない。
    - 例) 再利用する場合、コンパイル時に初期化済みのクラスが、実行時にも 初期化済みだとは仮定できない。

shuJIT を利用した研究例：選択的コンパイル

- 生成されたネイティブコードは メモリを消費する。メモリ消費量を 抑えたい。
- 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。

選択的コンパイル (cont'd)

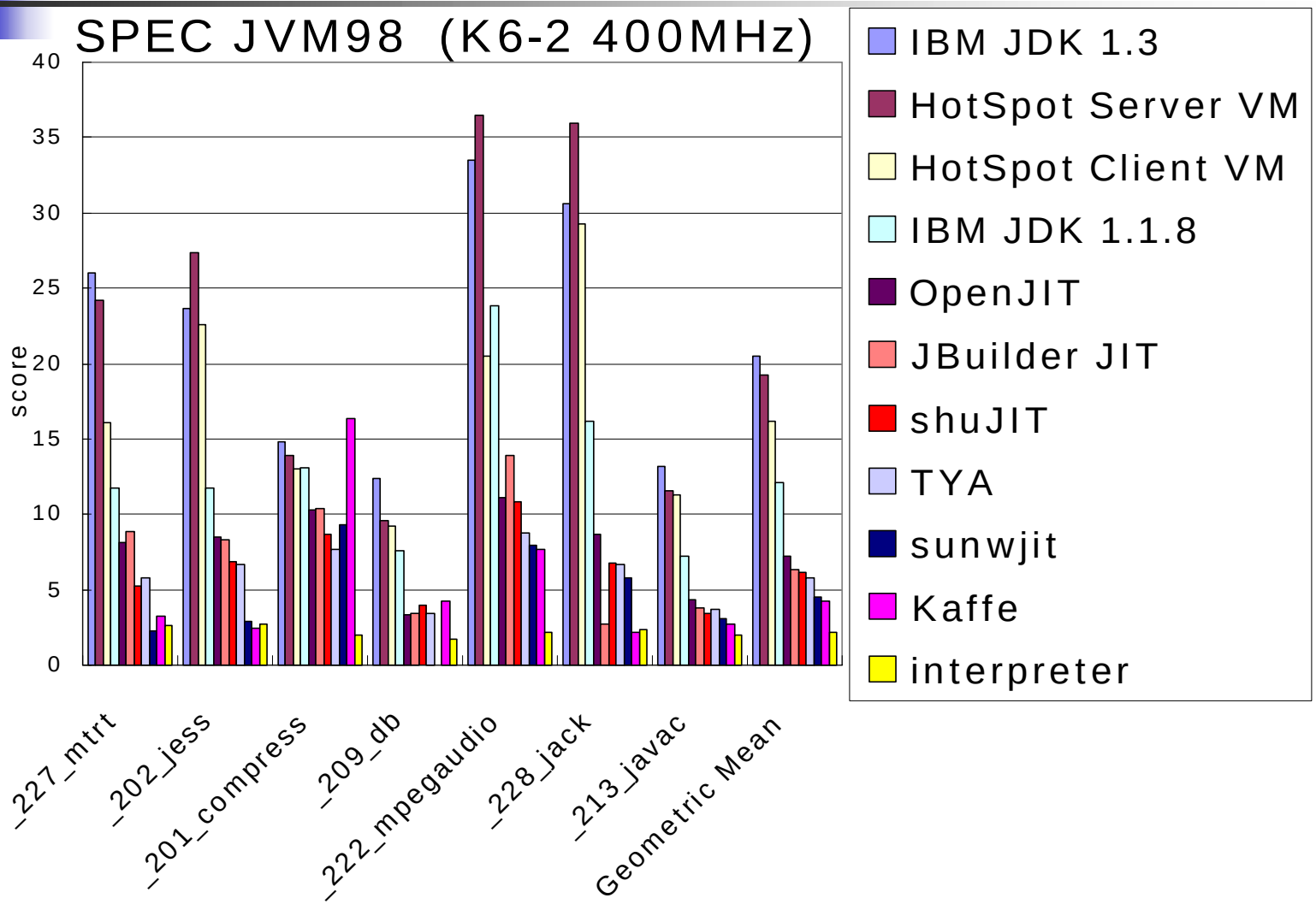




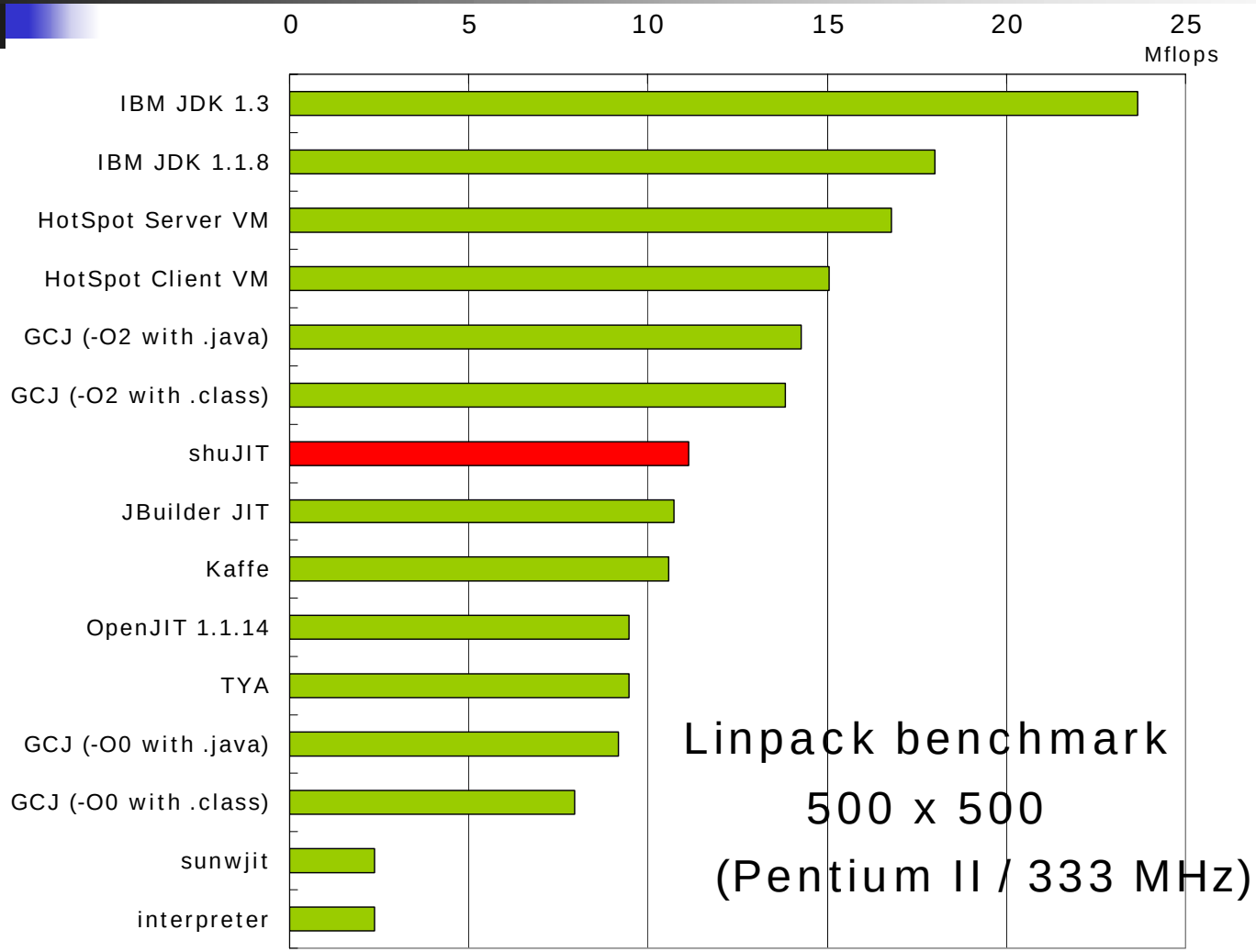
高速化手法 - いろいろ

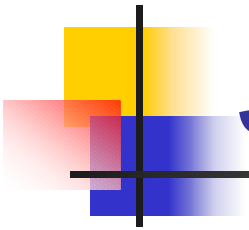
- メソッド呼び出し関係
  - compiled code間の直接呼び出し
  - 末尾再帰の除去
  - Method inlining
- ジャンプ関係
  - ループ先頭の alignment
  - より命令長の短い命令への書き換え
- その他
  - 各種 peephole 最適化
  - 1度きりの処理を自己書き換えで 2度目以降スキップ
  - 生成したコードの、次回実行時の再利用
  - シグナルを利用した例外の捕捉

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

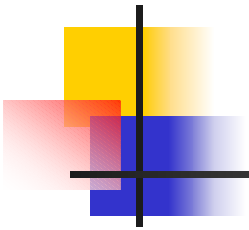
性能 (cont'd)


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)



JITコンパイラのまとめ

- 短いコンパイル時間で実用的な性能を達成。
  - 独特のコンパイル手法。
- 各種研究の基盤として利用されている。
  - MetaVM, strictfp, Jaguar, 選択的コンパイル
- 世界中で広く使われている。
  - 最近は特にFreeBSD用として。
- 各種 高速化技法を実装。



まとめ

- プログラマにメタコンピュータを提供する一手法を提示。
- Java仮想マシンにてスレッド移送が可能であることを示し、問題を明らかにした。
- バイトコード命令の意味拡張を提案し、ネットワーク透過な分散オブジェクトに応用して見せた。
- 低コストで実用的な性能が得られる JIT コンパイル手法を提案し、実用品質のソフトウェアを提供した。