高性能計算基盤

自習 APDX07:高性能JAVA-VM

http://archlab.naist.jp/Lectures/ARCH/x07/apdx07j.pdf

Copyright © 2021 奈良先端大 中島康彦

JAVA仮想マシンとは何か

NAIST

JAVA言語とJAVA仮想マシン

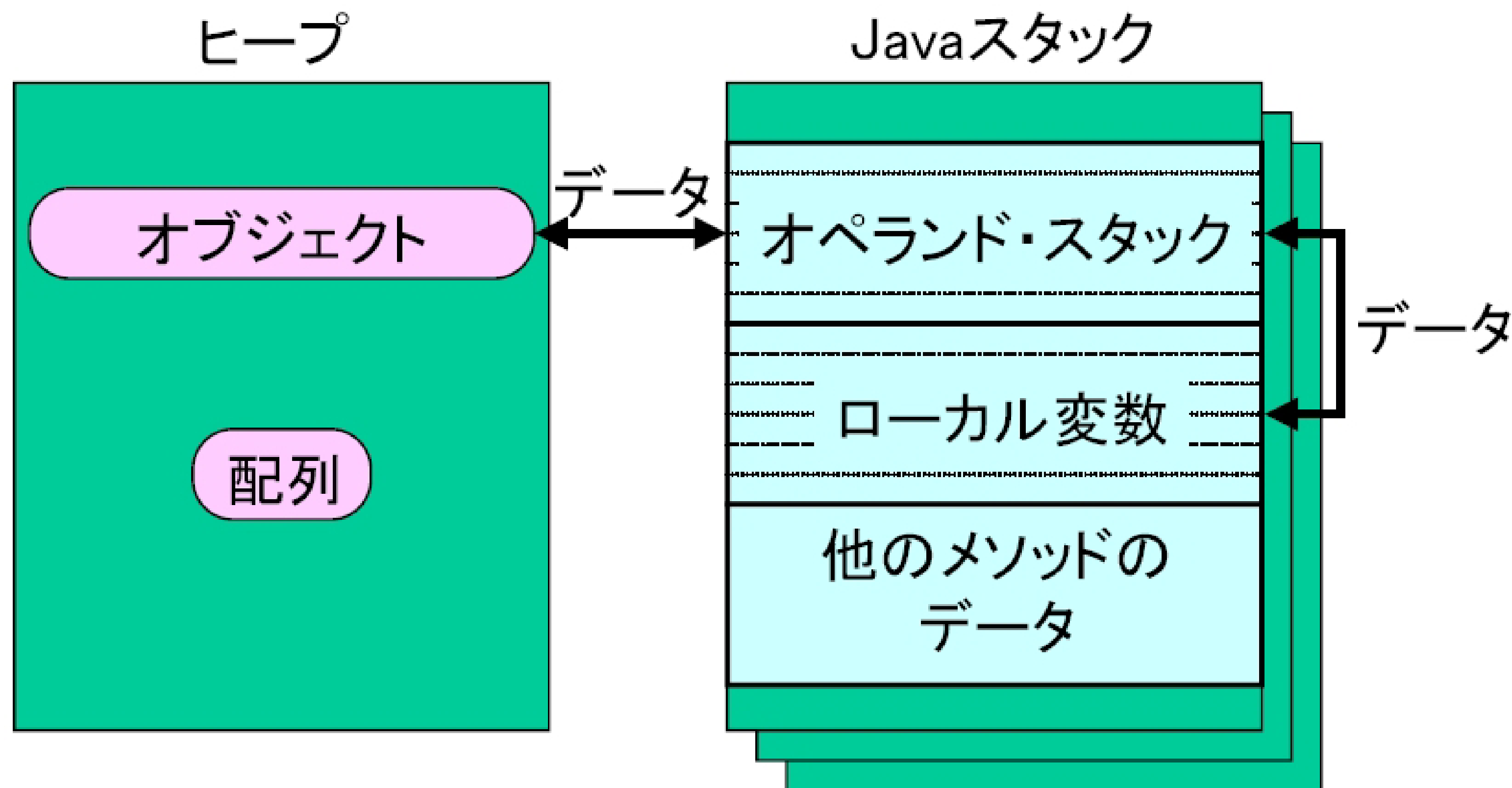
JAVA言語とJAVA仮想マシンは，独立のもの

- JAVA言語は 「オブジェクト指向言語」
  汎用プロセッサ用にコンパイルして実行できる
- JAVA仮想マシンは 「スタックマシン」
  どんな言語でもコンパイラがあれば実行できる

JAVA仮想マシンの命令セットが目指すもの

- ネットワーク利用を考慮した小さなコードサイズ
  可変長バイトコードの採用
- 様々なプラットホーム上でのエミュレーションが容易な実行モデル
  仮想マシンと呼ばれる理由
  スタックマシンの採用
- 安全な実行モデル
  検査可能なバイトコード形式と実行前検査機構

Java仮想マシンの概要



オペランド・スタックへとデータを移動しないと、操作は不可能

NAIST

汎用プロセッサとは異なる実行モデル…何が違うのか

汎用プロセッサでは，アドレス空間が１つしかない

- スタック構造
  局所変数とサブルーチンからの戻りアドレスが入っている
- オブジェクト
  直接入出力の対象となるデータ

以上が混在すると，戻りアドレスが破壊される可能性がある

- スタックオーバフロー攻撃に対する脆弱性

利点は，スタックポインタを戻すだけで局所変数を解放できること

- ただし，局所変数へのポインタが残っているとメモリ破壊の原因

NAIST

汎用プロセッサとは異なる実行モデル…何が違うのか

JAVAプロセッサでは，スタック構造と「オブジェクト」の空間が分離

- 入出力によるスタック構造の直接破壊は困難
- ただし，局所変数に対する入出力ができない

一時的入出力に対しても「オブジェクト」が必要

- スタックポインタを戻すだけでは「オブジェクト」を解放できない
- いずれにせよ残存ポインタの検査をするのであれば，ガベージコレクション方式にならざるを得ない

JAVAプロセッサがガベージコレクション機能を備える必然性

バイトコード

- Stack操作

0x00	nop	: 何もしない
0x57	pop	: Stackの先頭ワードをpop
0x58	pop2	: Stackの先頭2ワードをpop
0x59	dup	: Stackの先頭ワードを複製
0x5a	dup_x1	: Stackの先頭ワードを2つ下に複製
0x5b	dup_x2	: Stackの先頭ワードを3つ下に複製
0x5c	dup2	: Stackの先頭2ワードを複製
0x5d	dup2_x1	: Stackの先頭2ワードを3つ下に複製
0x5e	dup2_x2	: Stackの先頭2ワードを4つ下に複製
0x5f	swap	: Stackの先頭2ワードを入れ換える

- Stackに定数をpush

0x10	bipush [sbyte]	: Byteを符号拡張してpush
0x11	sipush [sbyte1/2]	: Shortを符号拡張してpush
0x02	iconst_m1	: Int定数（-1）をpush
0x03-8	iconst_0-5	: Int定数（0-5）をpush
0x09-a	lconst_0-1	: Long定数（0-1）をpush
0x0b-d	fconst_0-2	: Float定数（0.0-2.0）をpush
0x0e-f	dconst_0-1	: Double定数（0.0-1.0）をpush
0x01	aconst_null	: NULLをpush

NAIST

バイトコード … Stackに/からローカル変数を/にpush/pop

- Push

0x15	iload [index]	:	ローカル変数のIntをpush
0x1a-1d	iload_0-3	:	ローカル変数#0-3のIntをpush
0x16	lload [index]	:	ローカル変数のLongをpush
0x1e-21	lload_0-3	:	ローカル変数#0-3のLongをpush
0x17	fload [index]	:	ローカル変数のFloatをpush
0x22-5	fload_0-3	:	ローカル変数#0-3のFloatをpush
0x18	dload [index]	:	ローカル変数のDoubleをpush
0x26-9	dload_0-3	:	ローカル変数#0-3のDoubleをpush
0x19	aload [index]	:	ローカル変数の参照をpush
0x2a-d	aload_0-3	:	ローカル変数#0-3の参照をpush

- Pop: 参照をストアする場合はガベージコレクション用に登録

0x36	istore [index]	:	ローカル変数にintをストア
0x3b-e	istore_0-3	:	ローカル変数#0-3にintをストア
0x37	lstore [index]	:	ローカル変数にLongをストア
0x3f-42	lstore_0-3	:	ローカル変数#0-3にLongをストア
0x38	fstore [index]	:	ローカル変数にFloatをストア
0x43-6	fstore_0-3	:	ローカル変数#0-3にFloatをストア
0x39	dstore [index]	:	ローカル変数にDoubleをストア
0x47-a	dstore_0-3	:	ローカル変数#0-3にDoubleをストア
0x3a	astore [index]	:	ローカル変数に参照をストア
0x4b-e	astore_0-3	:	ローカル変数#0-3に参照をストア

バイトコード … Stack検査，2オペランド比較と条件分岐

- 3バイト長命令なので，分岐しない場合，PC = PC + 3

```
0x99        ifeq [brc1/2]           : PC += (int ==  0 )?[brc1/2]: 3
0x9a        ifne [brc1/2]           : PC += (int !=  0 )?[brc1/2]: 3
0x9b        iflt [brc1/2]           : PC += (int <   0 )?[brc1/2]: 3
0x9c        ifge [brc1/2]           : PC += (int >=  0 )?[brc1/2]: 3
0x9d        ifgt [brc1/2]           : PC += (int >   0 )?[brc1/2]: 3
0x9e        ifle [brc1/2]           : PC += (int <=  0 )?[brc1/2]: 3
0xc6        ifnull [brc1/2]         : NULL の場合に分岐
0xc7        ifnonnull [brc1/2]      : NON-NULL の場合に分岐
0x9f        if_icmpeq [brc1/2]      : PC += (int1==int2)?[brc1/2]: 3
0xa0        if_icmpne [brc1/2]      : PC += (int1!=int2)?[brc1/2]: 3
0xa1        if_icmplt [brc1/2]      : PC += (int1< int2)?[brc1/2]: 3
0xa2        if_icmpge [brc1/2]      : PC += (int1>=int2)?[brc1/2]: 3
0xa3        if_icmpgt [brc1/2]      : PC += (int1> int2)?[brc1/2]: 3
0xa4        if_icmple [brc1/2]      : PC += (int1<=int2)?[brc1/2]: 3
0xa5        if_acmpeq [brc1/2]      : PC += (val1==val2)?[brc1/2]: 3
0xa6        if_acmpne [brc1/2]      : PC += (val1!=val2)?[brc1/2]: 3
```

バイトコード … 無条件分岐，サブルーチン

0xa7	goto [brce1/2]	: 無条件分岐
0xc8	goto_w [brc1/2/3/4]	: 無条件分岐(ワイド添字)
0xa8	jsr [brc1/2]	: サブルーチンにジャンプ(PC+3をpush)
0xc9	jsr_w [brc1/2/3/4]	: サブルーチンにジャンプ(PC+3をpush)
0xa9	ret [index]	: 復帰(PC = ローカル変数[index])

NAIST

バイトコード … 演算(倍長演算,浮動小数点,型変換は省略)

- Stackから1つpopして結果をpush

0x74	ineg	:	Intの符合反転
0x91	i2b	:	IntをByteに変換 (int[7:0]をIntに符号拡張)
0x92	i2c	:	IntをCharに変換 (int[15:0]をIntに0拡張)
0x93	i2s	:	IntをShortに変換 (int[15:0]をIntに拡張)

- Stackから2つpopして結果をpush

0x60	iadd	:	Intの加算
0x64	isub	:	Intの減算
0x7e	iand	:	Intの論理積
0x80	ior	:	Intの論理和
0x82	ixor	:	Intの排他論理和
0x78	ishl	:	Intの左シフト (val[4:0])
0x7a	ishr	:	Intの算術右シフト (val[4:0])
0x7c	iushr	:	Intの論理右シフト (val[4:0])
0x68	imul	:	Intの乗算
0x6c	idiv	:	Intの除算 (int2が0なら例外発生)
0x70	irem	:	Intの剰余 (int2が0なら例外発生)

バイトコード … 配列

- Stackのarrayrefとindexをpopして要素値をpush

0x33	iaload	： 配列からIntをロード
0x33	baload	： 配列からByte/Booleanをロード(符号拡張)
0x34	caload	： 配列からCharをロード(ゼロ拡張)
0x35	saload	： 配列からShortをロード(符号拡張)
0x2f	laload	： 配列からLongをロード
0x30	faload	： 配列からFloatをロード
0x31	daload	： 配列からDoubleをロード
0x32	aaload	： 配列から参照をロード

- Stackのarrayrefとindexと要素値をpopしてストア

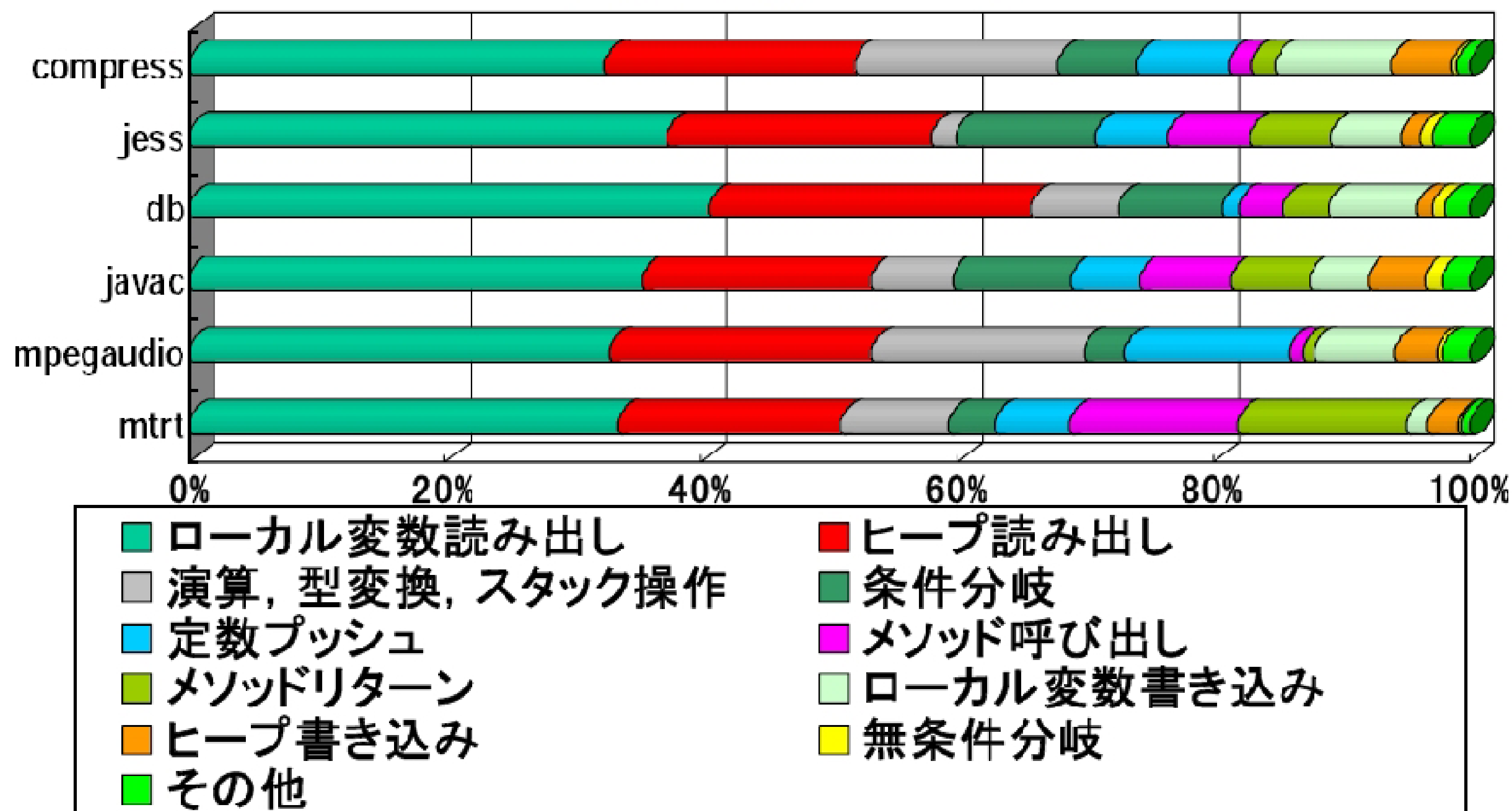
0x4f	iastore	： Intの配列にストア
0x54	bastore	： ByteもしくはBolleanの配列にストア
0x55	castore	： Charの配列にストア
0x56	sastore	： Shortの配列にストア
0x50	lastore	： Longの配列にストア
0x51	fastore	： Floatの配列にストア
0x52	dastore	： Doubleの配列にストア
0x53	aastore	： 参照型の配列にストア

バイトコード… メソッドとオブジェクト

```
0xb6  invokevirtual [CPindex1/2]  :  ディスパッチ,メソッド呼出し

0xac  ireturn                     :  メソッドからIntをリターン
0xad  lreturn                     :  メソッドからLongをリターン
0xae  freturn                     :  メソッドからFloatをリターン
0xaf  dreturn                     :  メソッドからDoubleをリターン
0xb0  areturn                     :  メソッドから参照をリターン
0xb1  return                      :  メソッドからVOIDをリターン

0xbb  new [index1/2]              :  新たなオブジェクトを作成する
0xbc  newarray [atype]            :  新たに配列を作成する
0xb4  getfield [index1/2]         :  オブジェクトからフィールドをフェッチ
0xb3  putfield [index1/2]         :  オブジェクトフィールドを設定する
```



NAIST

JAVAプロセッサを高速化するための工夫

NAIST

高速Javaプロセッサの基本構成

- Javaバイトコードを直接実行
- 内部レジスタの導入による命令畳み込み
  - オペランド・スタックを介さず直接レジスタ上のローカル変数を扱う
- メソッド呼び出し、ヒープ参照時のアドレス計算の高速化
  - オブジェクト変換表、フィールド変換表、メソッド変換表の3つの変換表を使用
- キャッシュによる主記憶参照の高速化

NAIST

メモリ構造をレジスタとキャッシュに写像する

オペランドスタック

- TOP付近しか使わないことを利用．TOP付近の数個をレジスタに写像
- 同様に，TOP付近をキャッシュに写像

ローカル変数

- スタックTOP付近しか使わないことを利用
- 同様に，TOP付近をキャッシュに写像

ヒープ

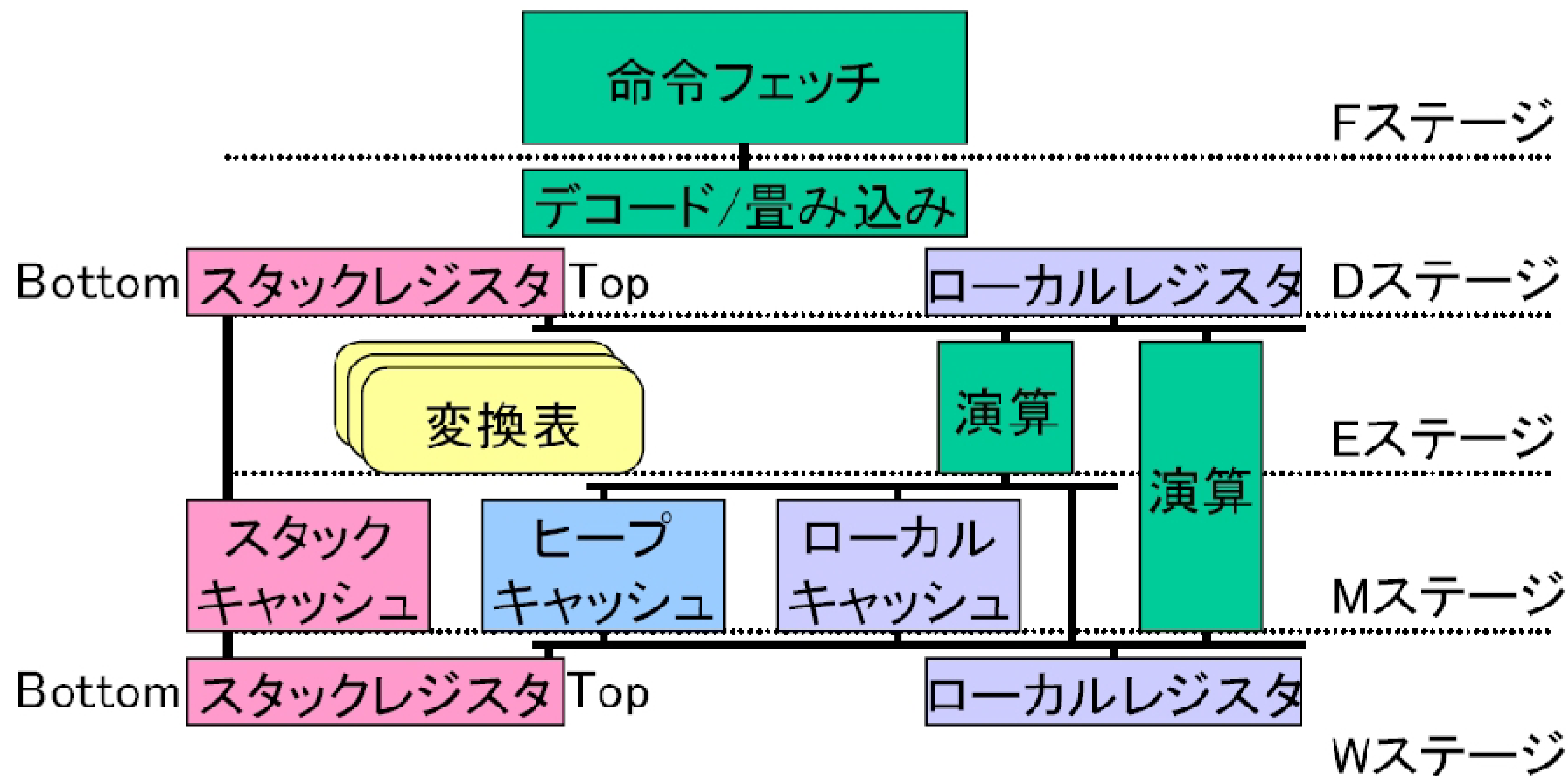
- オブジェクトはアドレスではない
  オブジェクト⇒アドレス変換表（汎用プロセッサのデータTLB相当）
- 同様にフィールド（構造体メンバに相当）についても変換表

メソッド

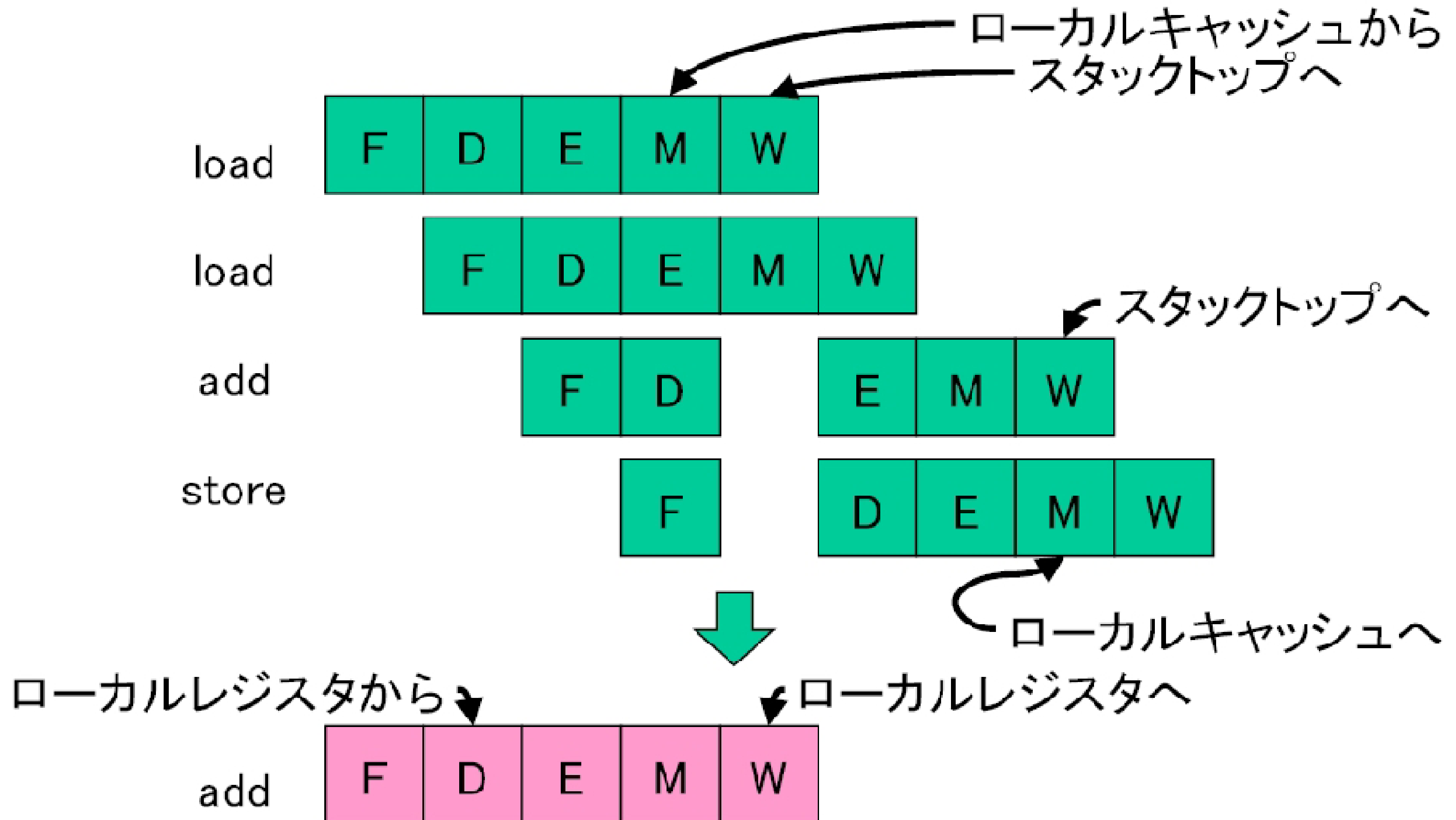
- メソッドはアドレスではない
  メソッドをアドレスに変換する変換表（同じく命令TLB相当）

NAIST

Javaプロセッサの構成



NAIST

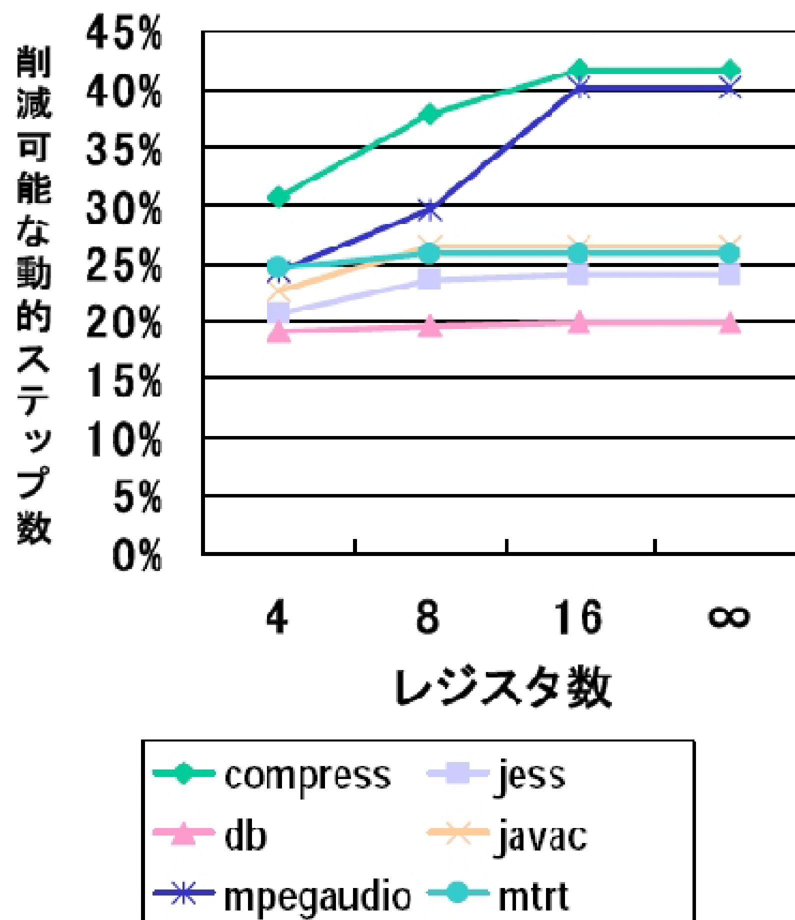


NAIST

最適なローカルレジスタ数

- 本プロセッサ構成においては、ローカルレジスタに存在する値を用いる命令が命令畳み込みの対象

ローカルレジスタ数と削減可能な動的ステップ数の関係を調査し、最適なレジスタ数を8と決定



最適な変換表のサイズ

- 全変換対を登録することが可能である変換表のサイズを3つの変換表それぞれ調査

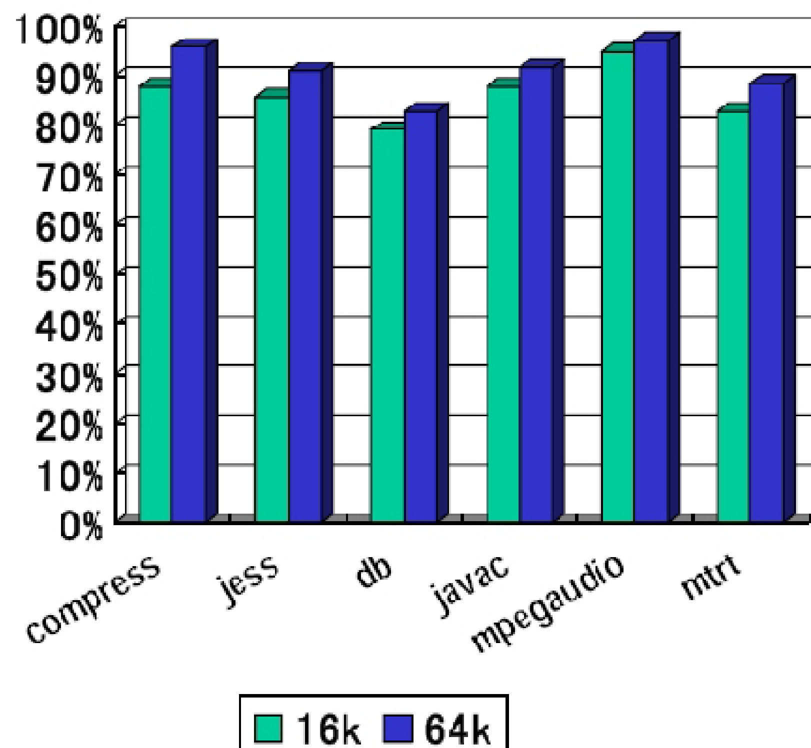
	compress	jess	db	javac	mpegaudio	mtrt
オブジェクト変換表	2229	80163	135977	94920	4477	503672
フィールド変換表	112	148	111	167	223	128
メソッド変換表	199	223	208	267	226	225

- オブジェクト変換表は4kエントリを使用可能とした場合にも約93.7%のヒット率であり、オブジェクト変換表の参照オーバヘッドは無視できない

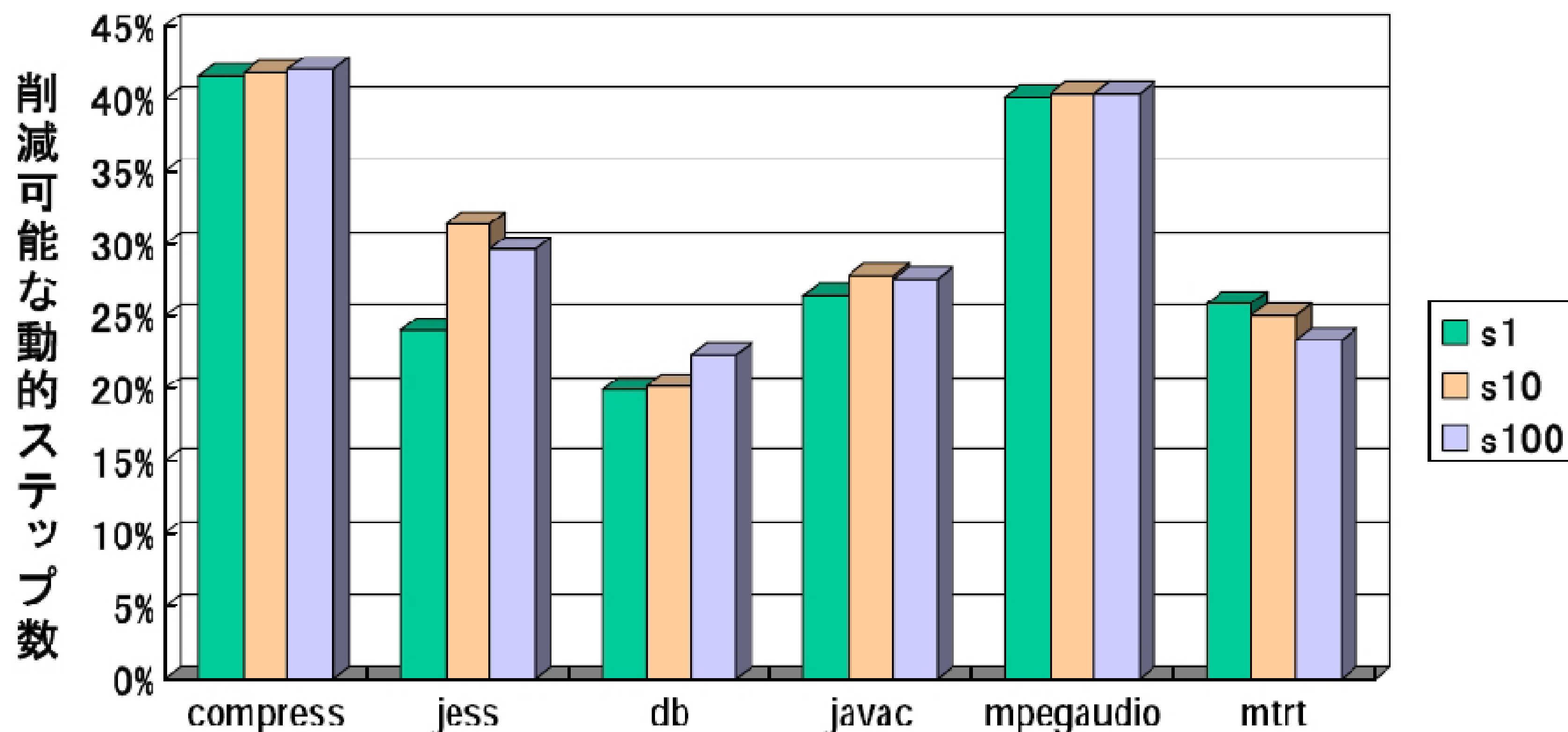
NAIST

キャッシュのヒット率

- ローカルキャッシュおよびスタックキャッシュは100エントリもあれば100%に近いヒット率を得ることが可能
- ヒープキャッシュを64kエントリとしても82.8%～97.0%のヒット率であり、ミスヒット時のオーバヘッドを無視することは不可能



NAIST

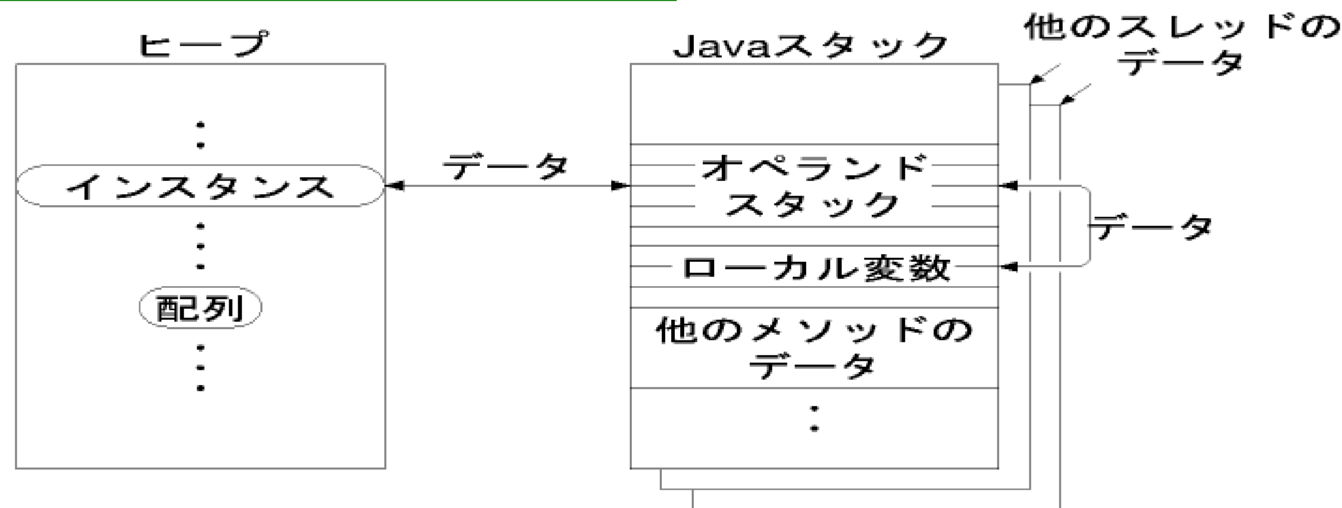


NAIST

アグレッシブな高速化の試み

NAIST

投機と再利用（Java仮想マシンの場合）



演算/ロード結果は必ずスタック・トップへ
引数の数および位置が明確（オペランド・スタック）
参照先が明確（スタック，ローカル変数，ヒープ）

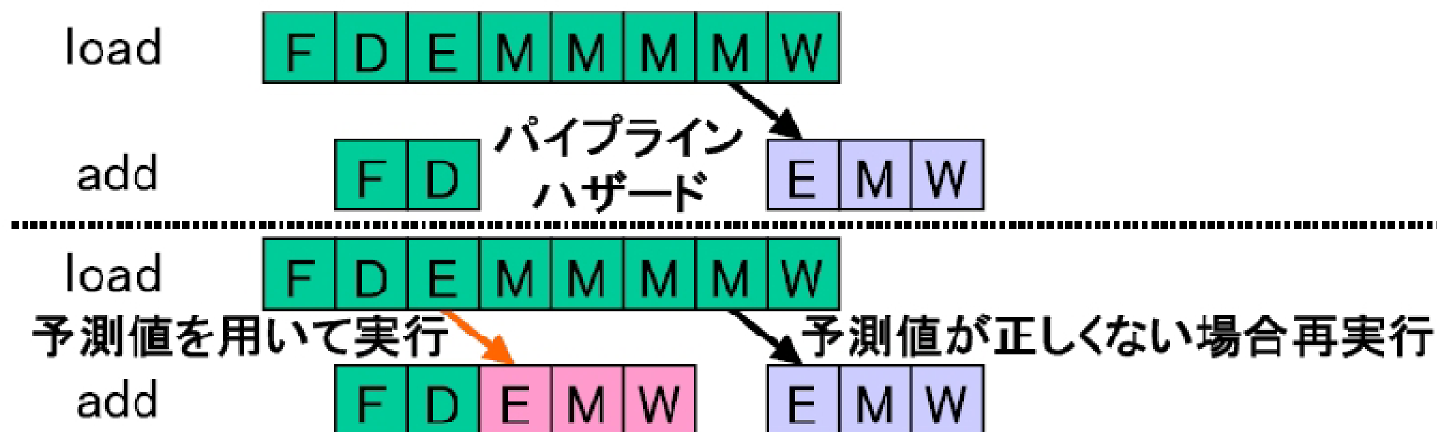
データ投機 … 実行結果を仮定し後続命令を開始
- スタック・トップの予測

データ再利用 … 入出力を記録し同一入力時に省略
- アシスト命令を用いずに再利用単位を特定：メソッド
- メソッド引数，入出力データの特定

NAIST

データ投機

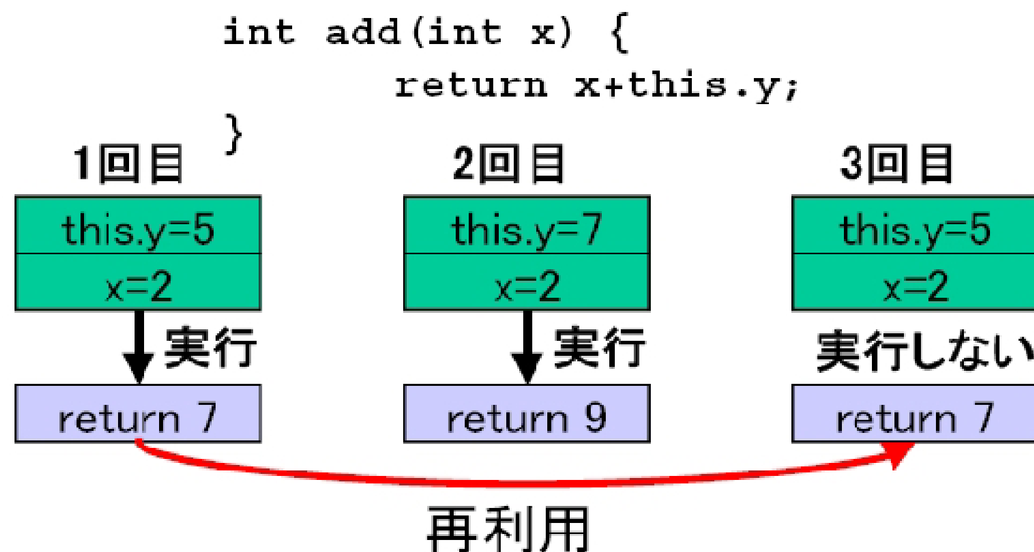
- 予測値を用いて後続命令の実行を開始する高速化手法
- 予測値が正しくない場合、予測した時点まで処理を戻す
- 対象となる命令はパイプラインハザードを生じるもの
  - ヒープ参照、ローカルキャッシュからの読み出し結果を直後に使用する命令、整数乗除算、浮動小数点演算
- 最も簡単なLast Value Predictionを仮定



NAIST

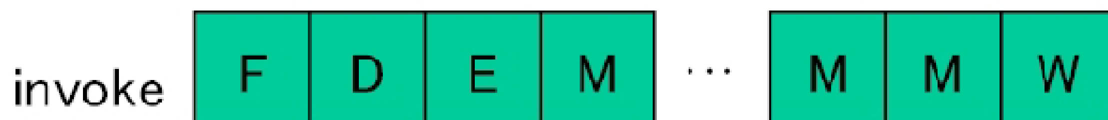
データ再利用(1/2)

- 実行結果を保存しておき再度同じ入力データを用いて命令を実行する場合に実行結果を再利用する高速化手法
- データ再利用はメソッドの実行を単位とすることにより、実行メソッドの切り替え時のパイプラインハザードを解消



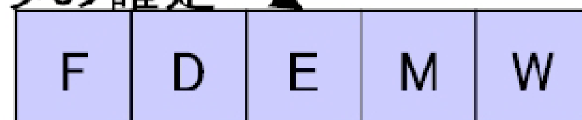
データ再利用(2/2)

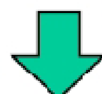
invoke



メソッドが使用するデータの確定

呼び出されたメソッドの命令

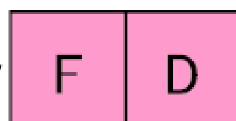


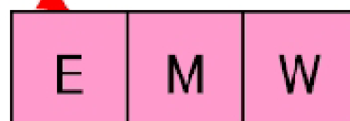


これから実行するメソッドが再利用可能であると判断

invoke

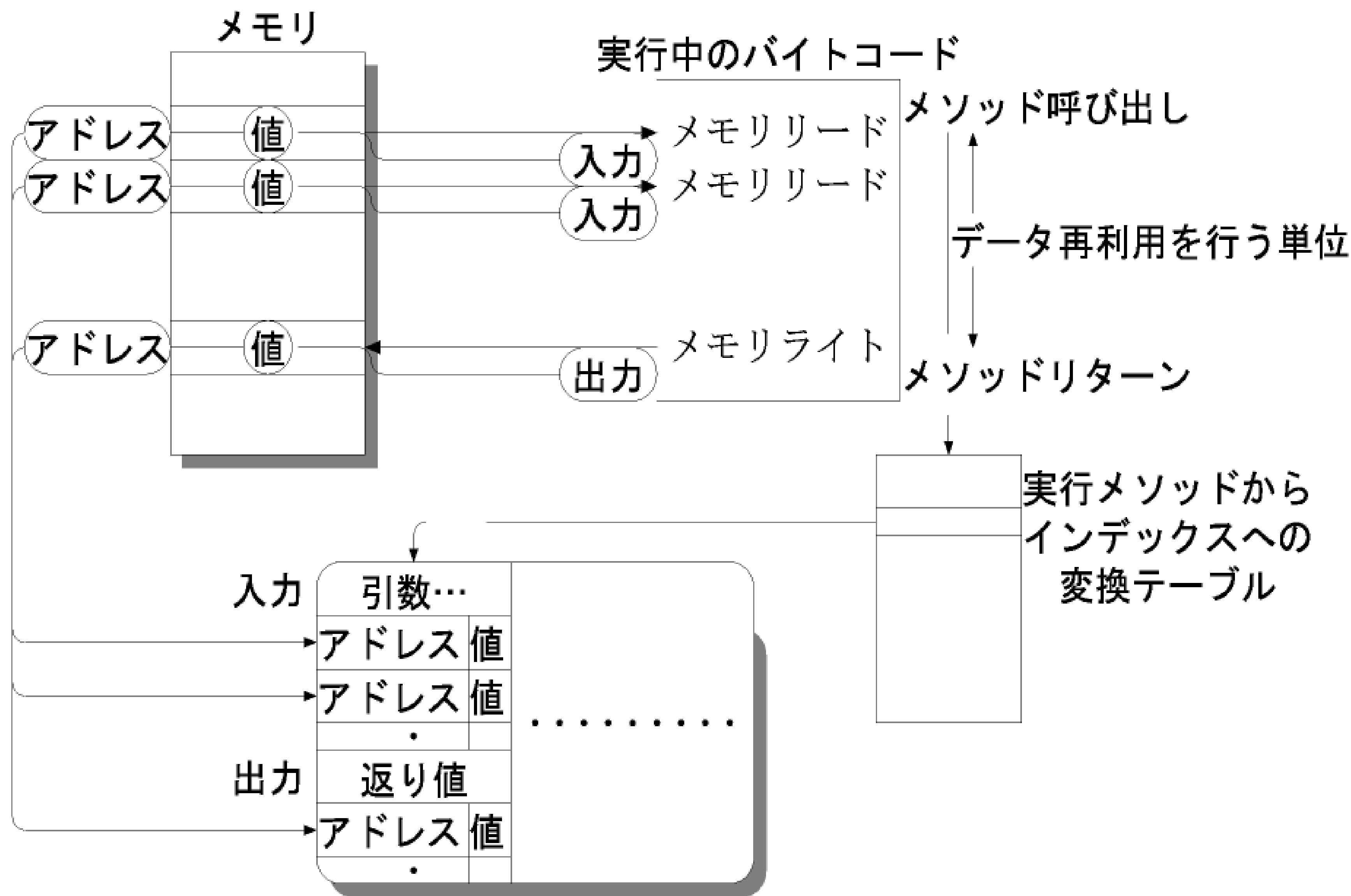
呼び出したメソッドの命令





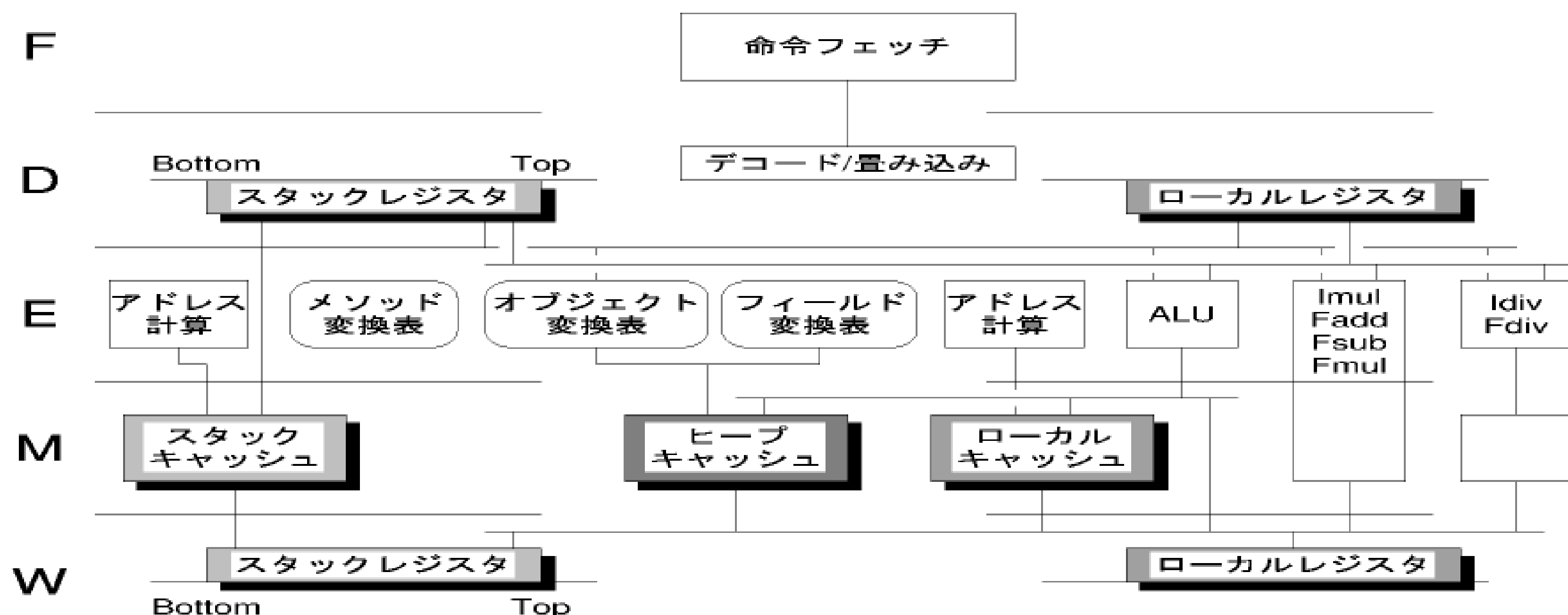
NAIST

メソッド呼び出しと再利用



NAIST

比較のための基本構成



オブジェクト変換表（4Kエントリ：ヒット率93.7%：ミス時20 τ ）

ヒープキャッシュ（64Kバイト：ヒット率91.2%：ミス時20 τ ）

メソッド呼び出し（18 τ ）

ローカルReg. ⇒ ローカルキャッシュ

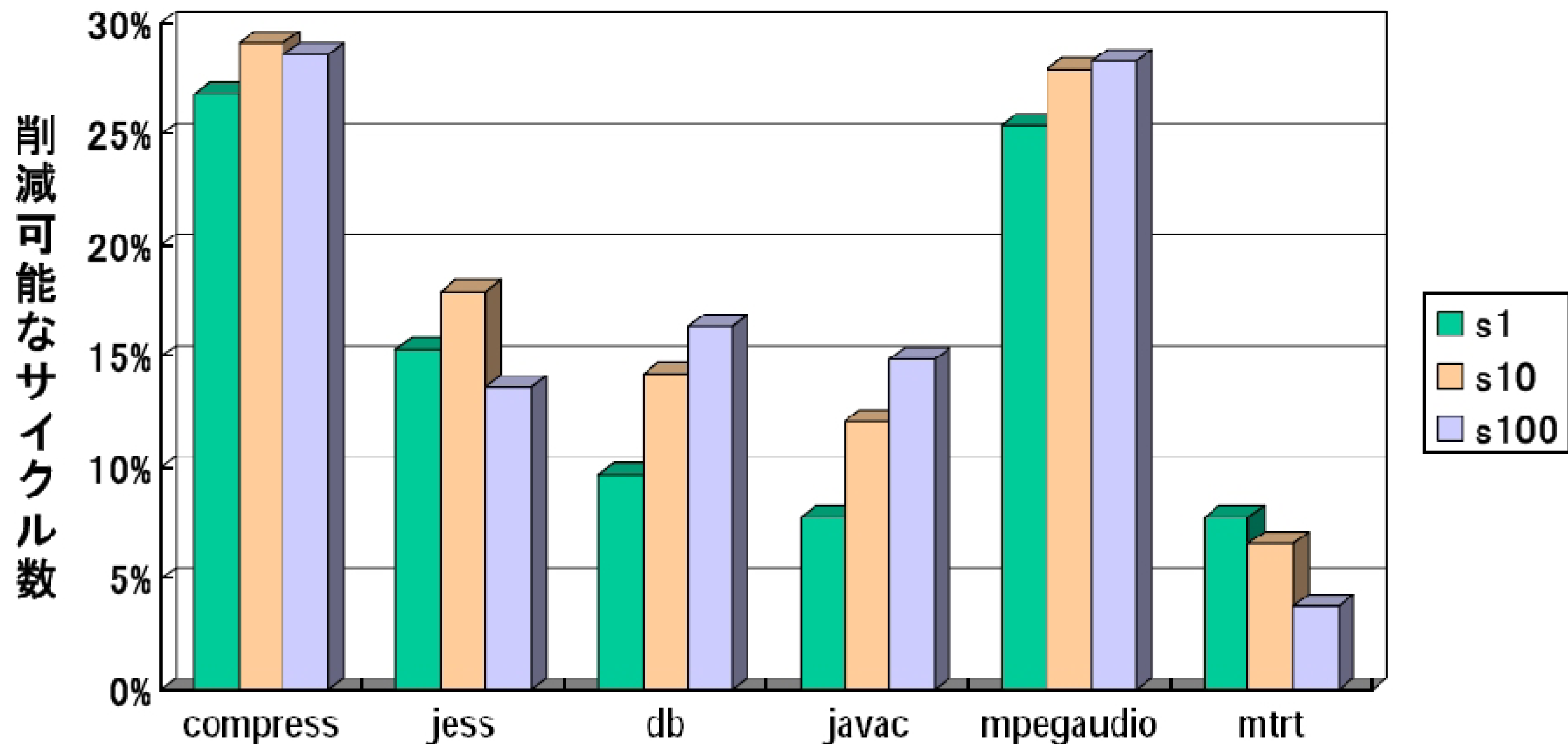
スタックReg. ⇒ スタックキャッシュ/ローカルReg.

復帰（18 τ ）

スタックReg. ← ローカルReg./スタックキャッシュ

ローカルReg. ← ローカルキャッシュ

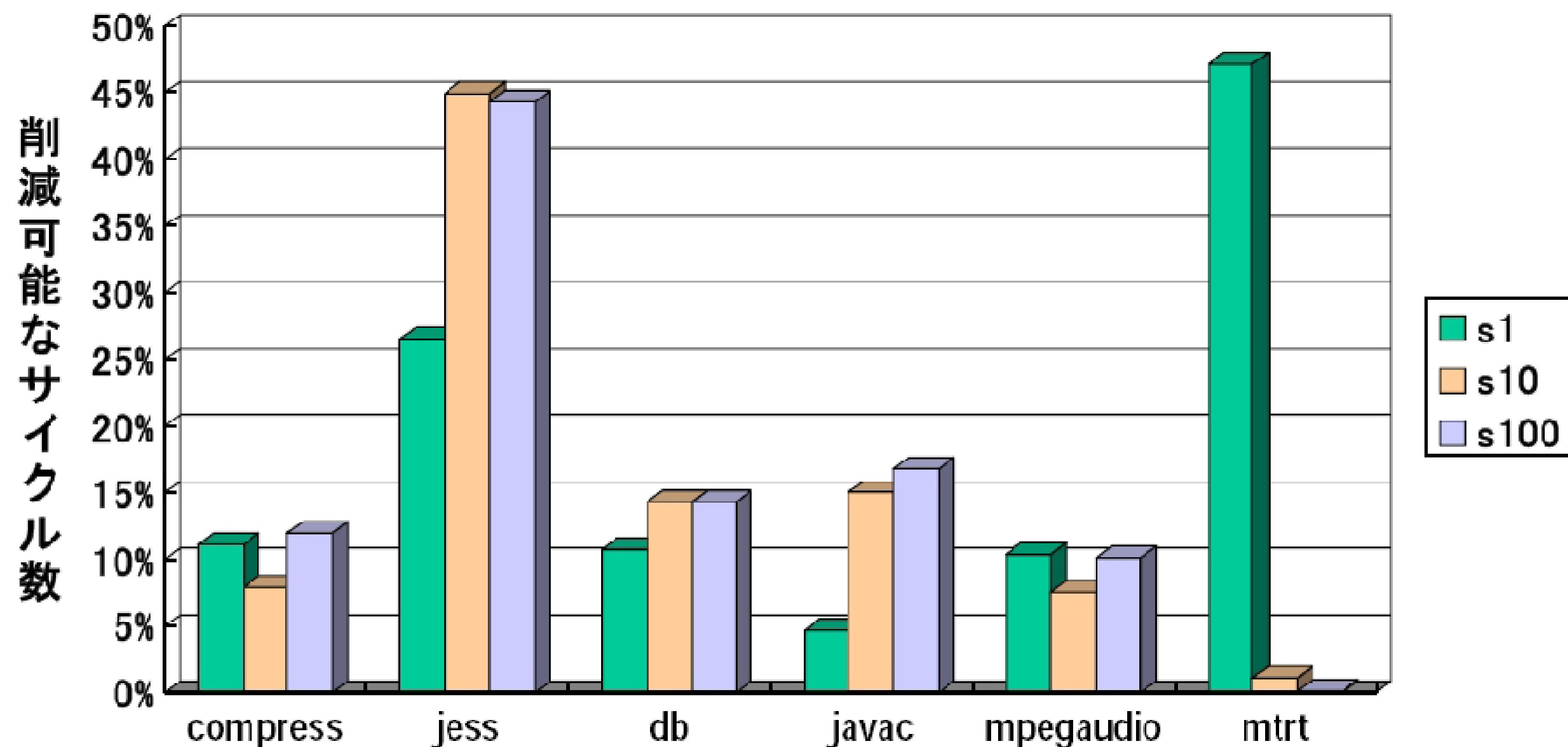
NAIST



NAIST

データ再利用の効果

命令畳み込みは使用していない



NAIST

投機と再利用の比較（SpecJVM98：s1,s10,s100）

投機の対象（ヒープ→スタック等バブルを生じる命令）
- 33.0%~47.2%（平均38.5%）

うち**Last Value Prediction**が正しい命令
- 24.3%~66.9%（平均54.8%）

投機による削減可能サイクル
- 3.8%~29.1%（平均17.0%）

再利用による削減可能サイクル
- 0.10%~47.1%（平均16.6%）CM3.0では94.0%

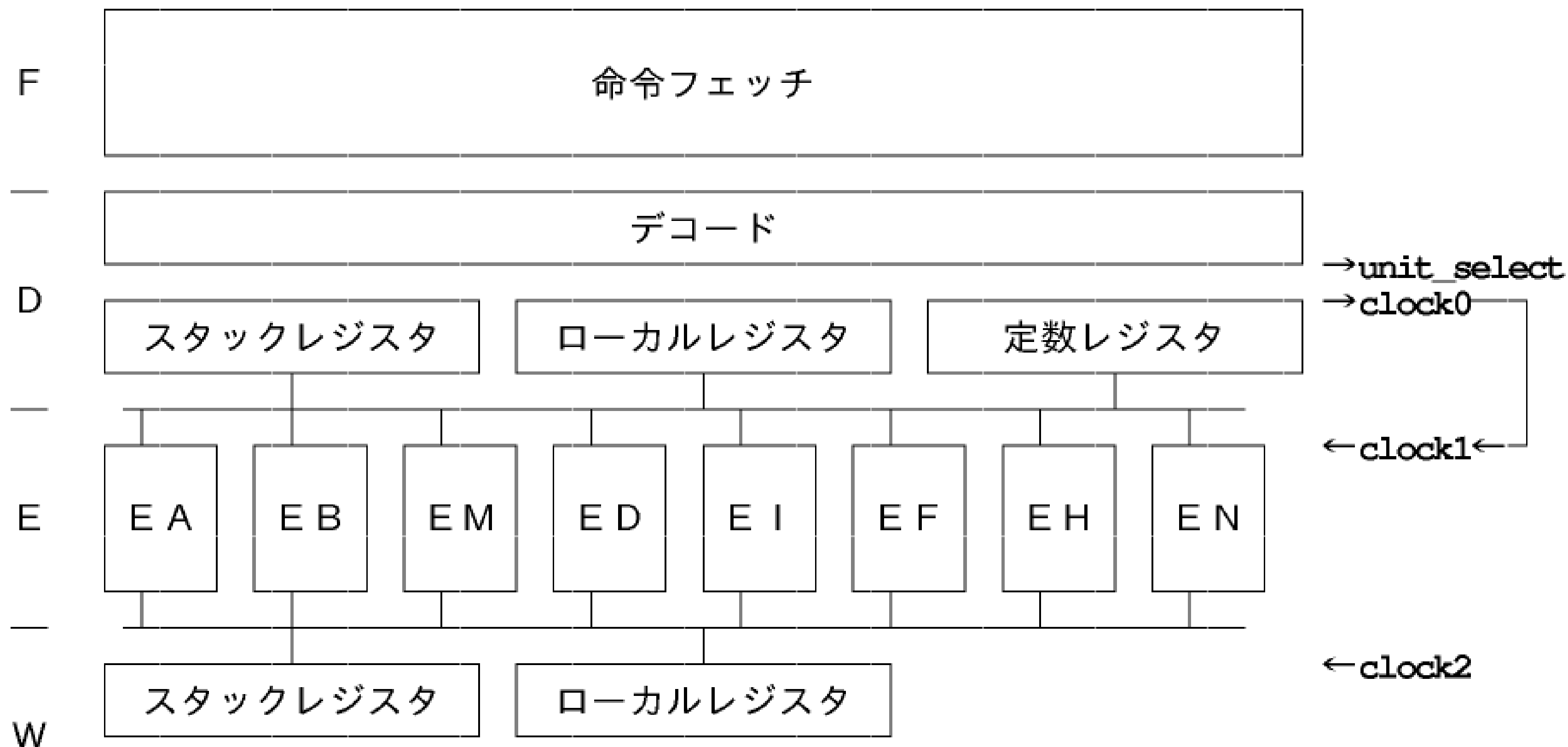
NAIST

アグレッシブな低電力化の試み

プロセッサモデル

使う演算器にだけ電源を入れるという方法

- ただし，電源を入れても，すぐには使えない
- どうやって，「あらかじめ」，電源をON/OFFするかという話



NAIST

モデルのパラメタ

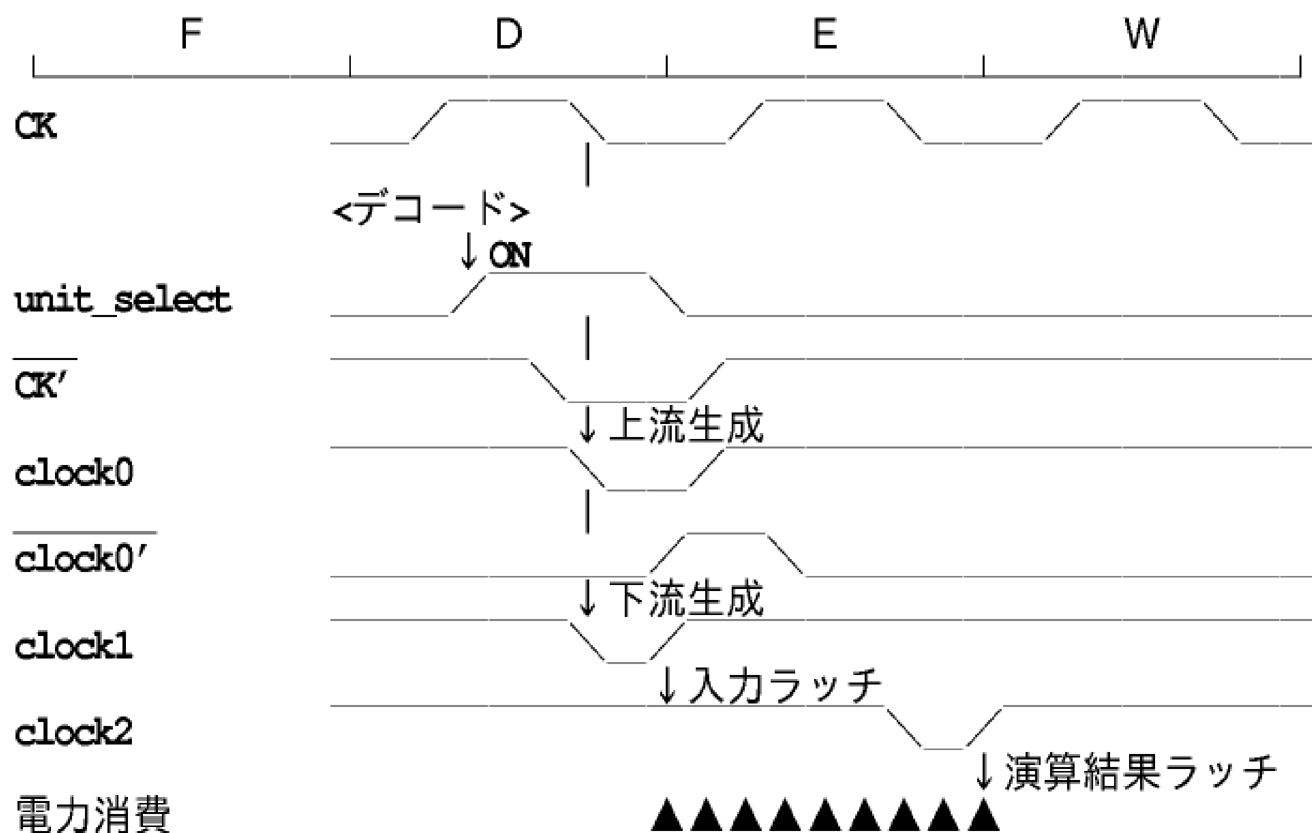
大きな演算器は電力消費も大きいと仮定
複雑な演算には時間がかかると仮定

演算ユニット名	EA	EB	EM	ED	EI	EF	EH	EN
ゲート数	8k	8k	18k	1k	8k	21k	11k	8k
ラッチ数	51	51	196	7	119	333	87	93
消費電力比率 (%)	10	10	22	2	10	24	12	10
nop,popX,dupX	1	0	0	0	0	0	0	0
Xconst,Xipush,Xstore	1	0	0	0	0	0	0	0
IALU,SFT,I2I,Fneg	1	0	0	0	0	0	0	0
Xload	2	0	0	0	0	0	0	0
if,Icmp,goto,jsr,ret	0	2	0	0	0	0	0	0
Xswitch	0	8	0	0	0	0	0	0
Imul	0	0	4	0	0	0	0	0
Idiv,Irem	0	0	16	0	0	0	0	0
swap,dup_X	0	0	0	10	0	0	0	0
dup2_X	0	0	0	20	0	0	0	0
invokeX,Xreturn	0	0	0	0	18	18	0	0
Xaload,Xastore	0	0	0	0	0	0	10	0
ldcX_X,astore	0	0	0	0	0	0	10	0
Xstatic,field	0	0	0	0	0	0	10	0
arraylength	0	0	0	0	0	0	10	0
new,Xnewarray	0	0	0	0	0	0	0	20
Fadd,Fsub,Fmul,Fcmp	0	0	0	0	0	4	0	0
I2F,F2I,F2F	0	0	0	0	0	4	0	0
Fdiv,Frem	0	0	0	0	0	16	0	0
Ddiv,Drem	0	0	0	0	0	32	0	0
invokeinterface	0	0	0	0	0	18	0	0
athrow	0	0	0	0	0	18	0	0
checkcast,instanceof	0	0	0	0	0	10	0	0
multianewarray	0	0	0	0	0	10	0	0
monitorX	0	0	0	0	0	9	0	0

NAIST

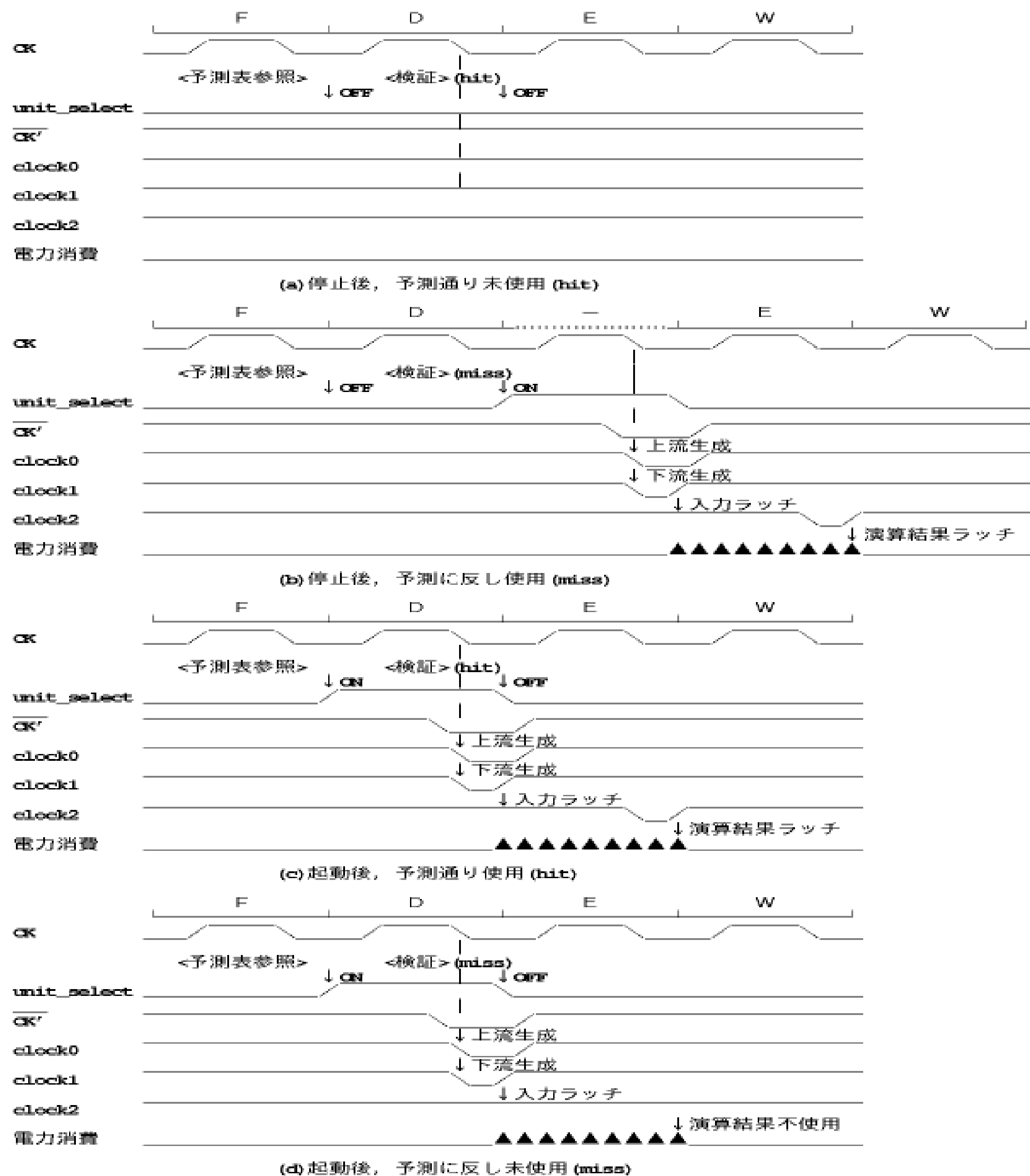
クロックと電力消費モデル

なるべく大元で電源を**ON/OFF**したいが，末端に届くまでに時間がかかると仮定



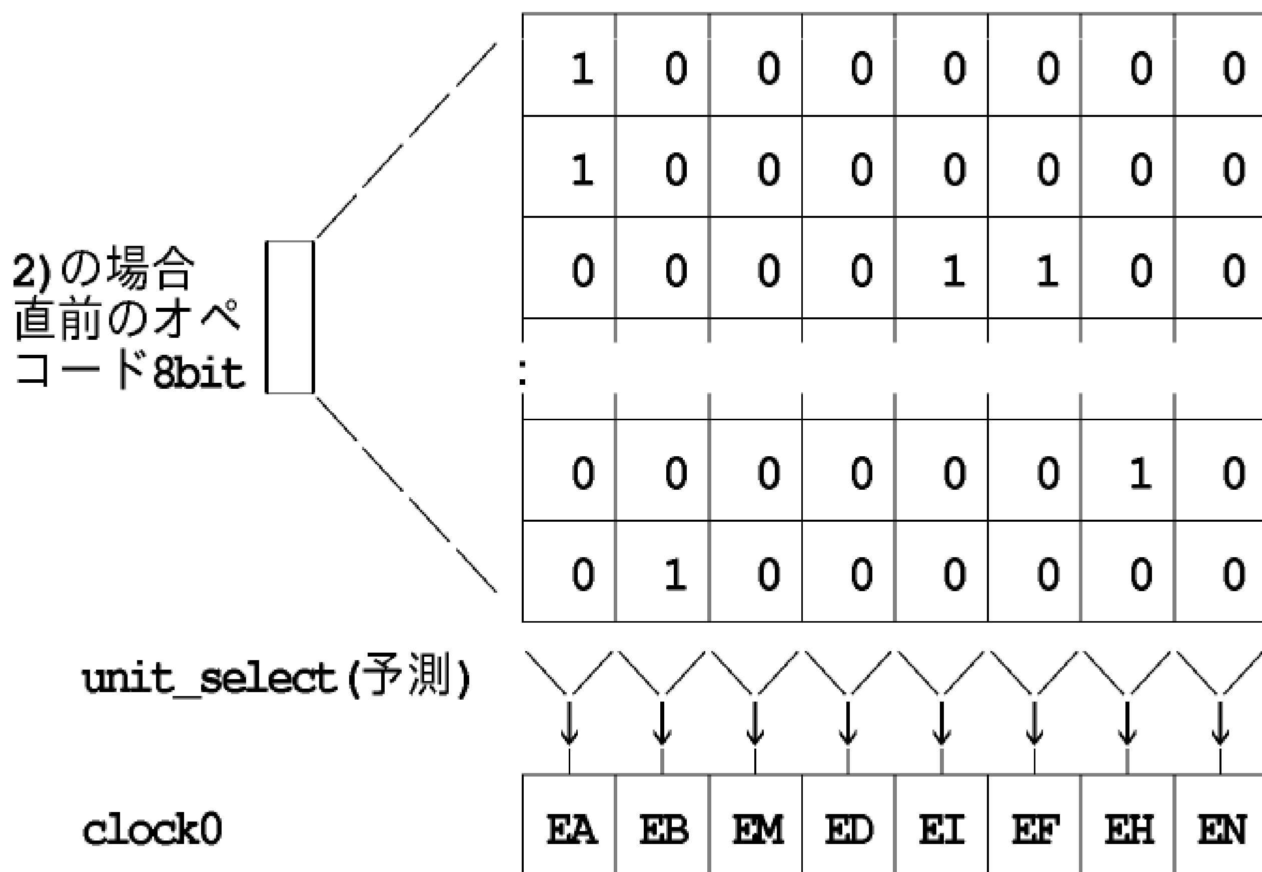
NAIST

投機的クロック制御と電力消費モデル



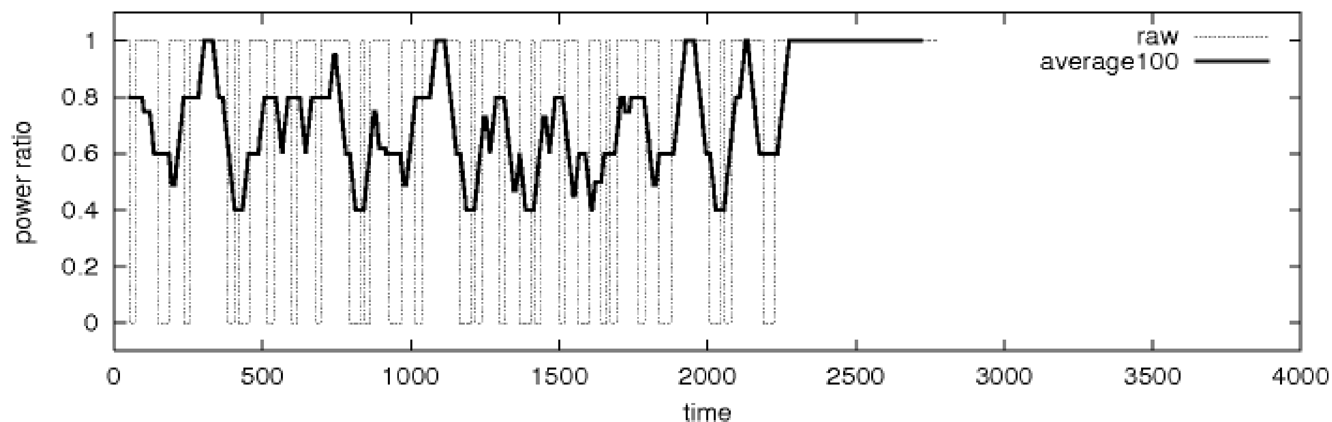
投機的クロックの生成

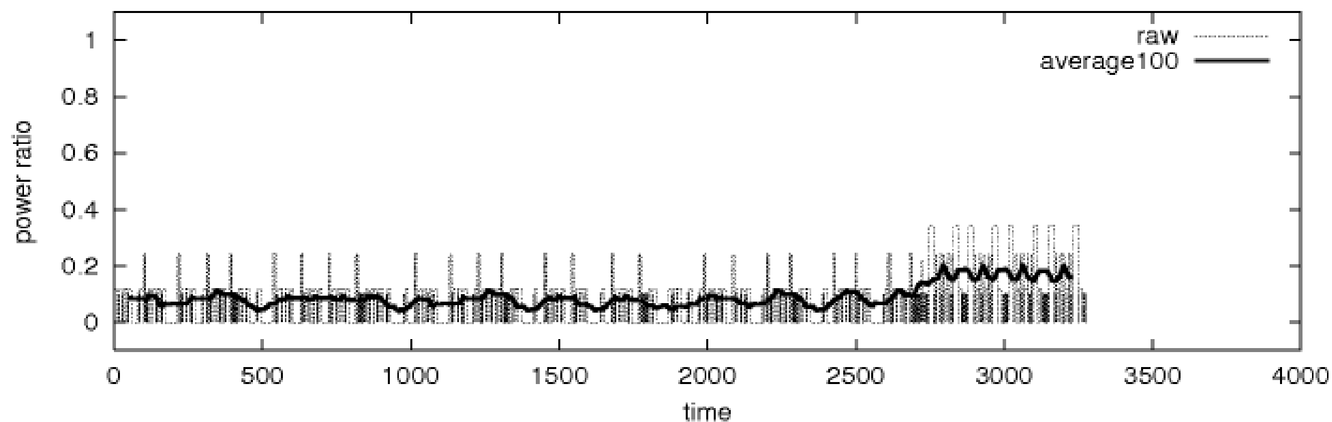
演算器毎に，次の電源**ON/OFF**を予測する
スタックマシンなので，直前のオペコードが参考になると予想

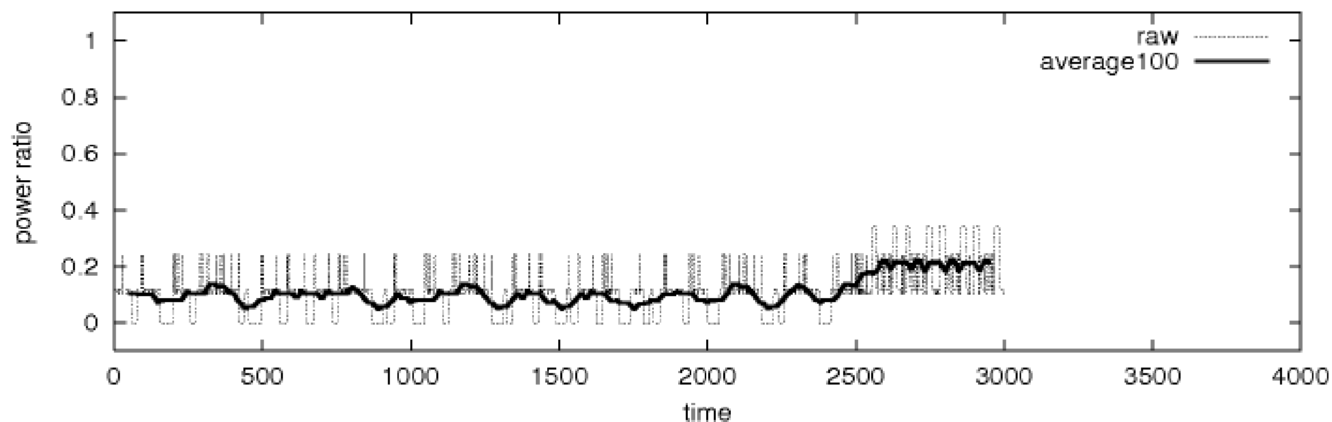


NAIST

電力消費(クロック常時供給,必要時に供給,投機的制御)

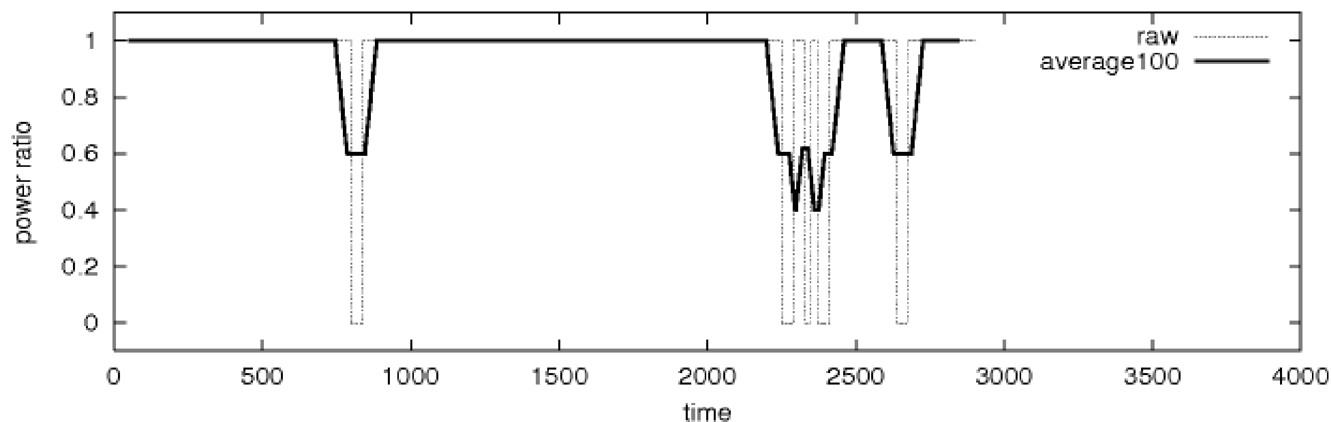


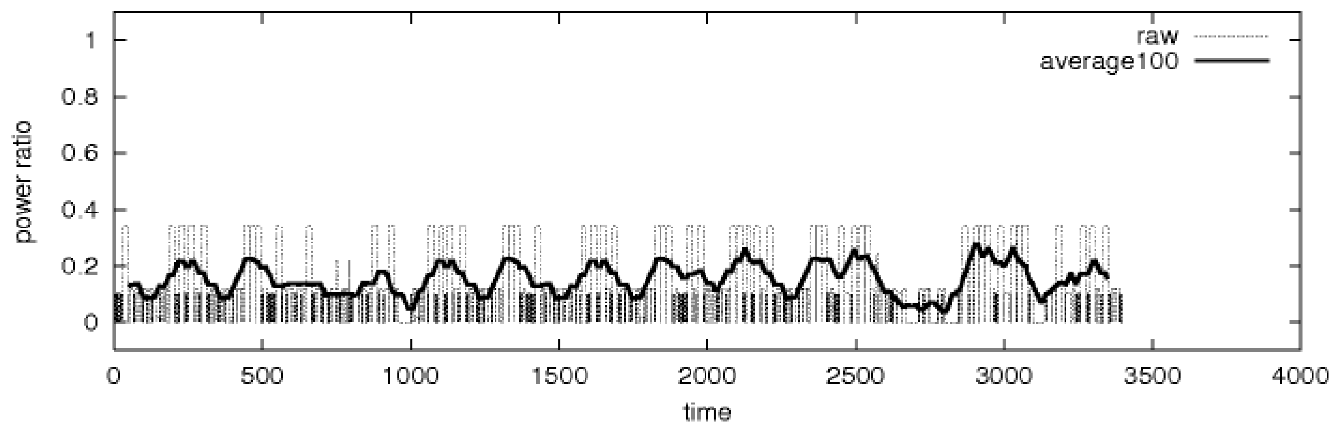


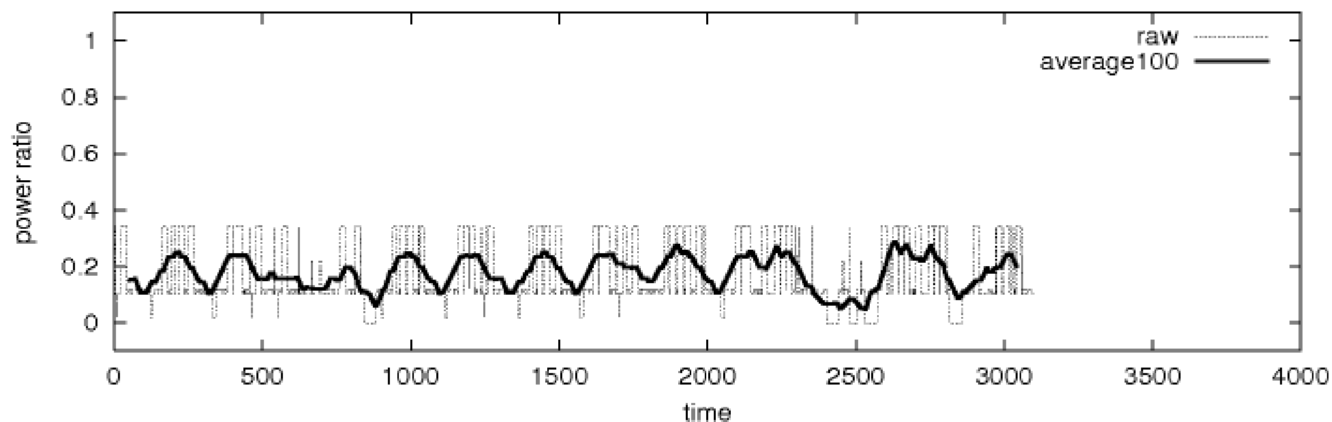


NAIST

電力消費(クロック常時供給,必要時に供給,投機的制御)







NAIST

評価尺度

$$Mips/Watt = \frac{総実行命令数/実行時間}{総電力/実行時間} = \frac{総実行命令数}{総電力}$$

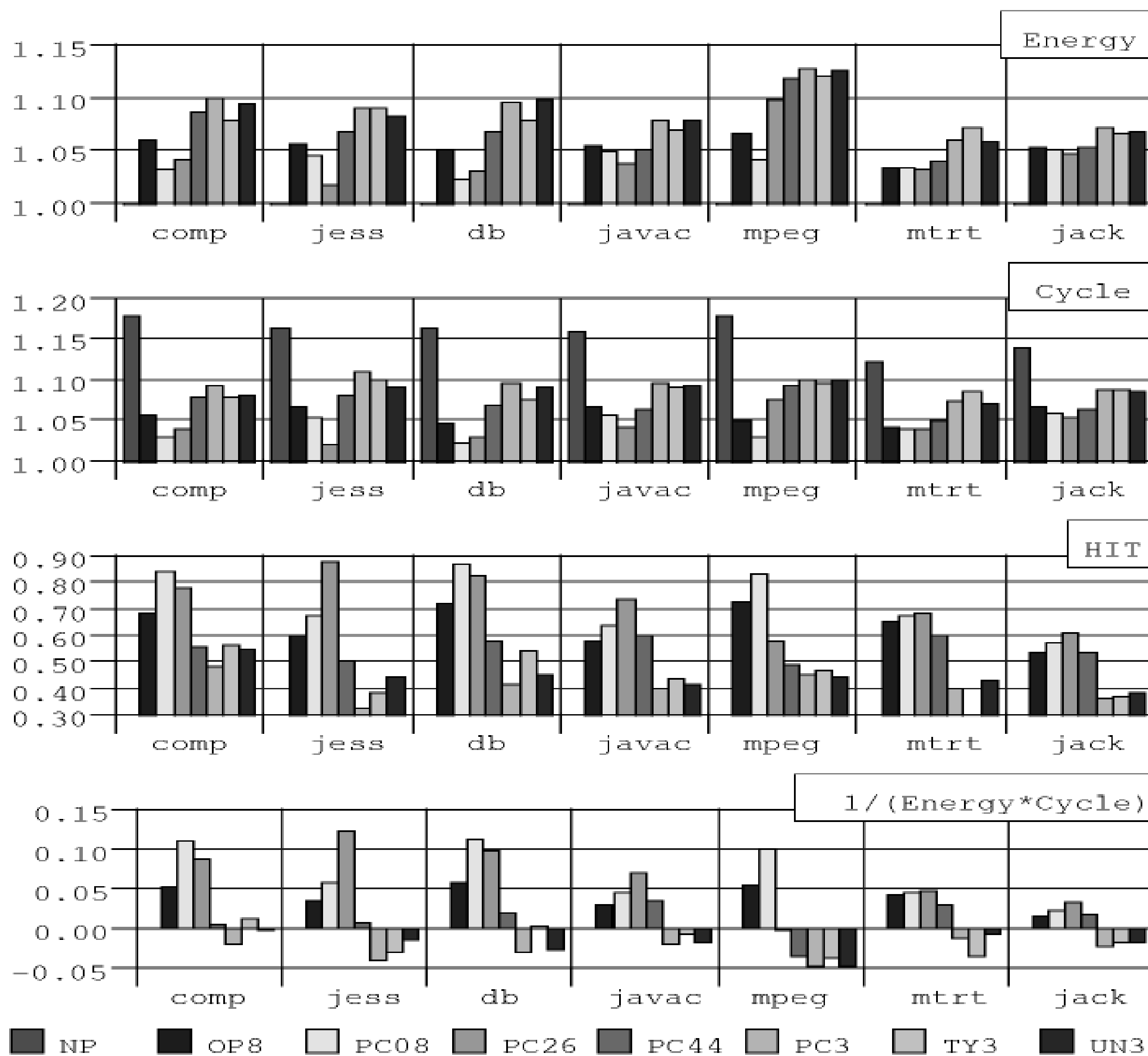
- 命令実行数が同じであれば，総電力のみの比較

$$Mips^2/Watt = \frac{(総実行命令数/実行時間)^2}{総電力/実行時間} = \frac{総実行命令数^2}{総電力 * 実行時間}$$

- 命令実行数が同じであれば，総電力*実行時間の比較

NAIST®

プログラムの消費電力（起動ペナルティ＝１）



NAIST

今日のまとめ

命令セットはプログラムのエンコード手段

- エンコードの考え方を変えると，命令セットや最適化方法も変わる．
- ただし，最適化を進めると，最終的には似たような仕組みに到達することもある．

「投機」は，プロセッサ高速化のキーワード

- 次回は，様々な場所に適用されている「投機」について説明する．

「再利用」は，比較的新しい考え方

- 最後の講義で紹介する．

今日はここまで

NAIST