社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2005－ARC－164（15）
2005／8／4

インタプリタ方式による命令エミュレーション処理性能

近江谷 康人†

あらまし コンピュータ製品開発において，市販の高性能マイクロプロセッサを用いてバイナリ互換を実現するアーキテクチャエミュレーション手法は，開発効率面で有効である．特に動作原理が単純かつホストアーキテクチャ依存度が低い C 言語実装によるインタプリタ方式は，開発費，品質，保守性の点で実際的である．本稿では，将来の製品性能の予測を行なうため，インタプリタの性能（CPI）を分析している．複数種のインタプリタを複数種のホスト上に構築して，命令頻度とともに実行時間を計測した結果，コアループが 44～70%占めていることが判った．また，上位約 20 命令の処理時間と頻度から見積もった CPU 時間は実測値とほぼ同等であり，見積りの正確性を示している．

Performance Analysis for Instruction Set Emulator Based on Interpreter Technique

Yasuhito OHMIYA †a)

Abstract - Architectural emulation technique using high-performance microprocessors is a cost-effective tool for developing a new computer product with keeping the binary compatibility. Especially the interpreter written in C language, based on simple structure and architecture-free implementation, is practical in development-cost, quality and maintainability. Here, CPI(Clock cycles Per Instruction) of interpreters is analyzed in order to forecast the future product performance. Through the analysis by counting frequency of each instruction and measuring emulation time of several interpreters versions on multiple host architectures, it appears that the core-loop wastes 44 to 70 percent of time. This paper also shows that an estimated total CPU time, calculated from top-twenties of each instruction's time and frequency, is well matched to the real emulation time, thus, it is useful to prospect the real performance.

1. まえがき

コンピュータの新機種開発においては，ユーザ資産の継承のため前機種との互換性が必須要件である．一方，半導体技術の進歩に伴う集積度向上により LSI 開発費が高騰し，プロセッサアーキテクチャの淘汰が進み，その結果, 従来と命令セット互換でより高性能の CPU チップの入手や CPU チップ開発が困難となってきている．

プログラムコードの殆どは高水準言語にて記述されており，リコンパイルによるアーキテクチャ変更に対応は可能ではあるが，潜在障害の露見やシステム試験費用確保などの課題がある．市販の高性能マイクロプロセッサを利用して，例外条件も含めて命令仕様をシミュレーションしバイナリ互換を実現するエミュレータ [1][2] は，産業的な価値を持つ．

エミュレータには，プログラム実行前に適用が必要な静的変換方式，プログラム実行時に行なう動的変換方式と古典的なインタプリタ方式がある．静的変換はコードの最適化による高い性能が期待できるが人手介入が必要など運用上の課題がある．動的変換は実際に実行されかつ実行頻度の高い部分のみ変換する方式で，実用的かつ性能に優れている[3]ものの，その最適化処理と例外処理が複雑化し安定的な動作を保証するには膨大な開発費と評価期間が必要となる．

インタプリタ方式では，命令語をメモリより取り出し，命令コードとレジスタ番号などのフィールドのデコード，アドレス計算，演算，結果の格納の一連動作を 1 命令ずつ解釈しながら実行する方式である．その結果，命令による命令書き換えや，商用機では厳密性が要求される例外動作も正確に処理できる．また，命令間に渡る相互作用が限定され，既存の診断プログラムなどによる検証により評価が可能なため，開発費は動的変換方式に比べ 1 桁以上少なくて済むと言われている．更に，必要メモリ量が増えないなど長所が多いが，動的バイナリ変換に比べ数～十倍ほど遅いのが欠点である．

インタプリタ方式のエミュレータでは，高性能を狙うためにはアセンブリ言語で記述する必要があるが，その結果，改良開発および保守を行なう技術者の維持が困難となる，ホストマシンの命令セットアーキテクチャやパイプライン構造への依存度が強く将来のホスト変更が困難になることが懸念される．そのような背景の下，性能を犠牲にしても C 言語記述を用い，開発費が安くホスト依存性が少なく将来ともに使用可能なエミュレータを短期に開発したいという要望が強い．

C 言語実装によるインタプリタ開発を計画する場合にホストの選択肢の拡大に伴い，どのプロセッサを選択し，周波数はどうするか，現行製品のどのレンジまで性能達成が可能か，性能見積りの確度はどうか，製品の全レンジまで適用するにはいつまで待てばよいか，という問いに対する事前検討が必要となるが，客観的な指標による予測が困難であり経験と勘で予測していた．予測困難な理由は，エミュレータの実装には，技巧的なものが多くその構造と性能は公表されにくく，高速化のための工夫がどれだけ効果をもたらしているかという一般的な評価基準が無いためである．

本稿では，C 言語実装のインタプリタ方式のエミュ

†三菱電機株式会社情報技術総合研究所，鎌倉市
Information Technology R&D Center, Mitsubishi Electric Corporation, 5-1-1 Ohfuna, Kamakura-shi, 247-8501 Japan
a) E-mail: ohmiya@isl.melco.co.jp

レータ性能について評価を行ない，ホストアーキテクチャとアプリケーションへの依存性が低いということを示し，その結果，エミュレータ搭載製品の性能予測が容易であることを示す．

本稿の構成を示す．2．では，評価に用いるインタプリタ方式のエミュレータの構造を述べる．3．では評価に用いた模擬対象のアーキテクチャ，エミュレーション実行をするホストアーキテクチャ，模擬実行する評価プログラムについて記述する．4．ではアプリケーションへの依存性について評価結果を示す．5．では 1 命令の処理時間性能について述べる．6．では関連研究に触れ，7．でまとめる．

2. インタプリタ方式エミュレータの構造

本章では，評価対象として作成したエミュレータの構造を解説する．

2.1. エミュレータの構成

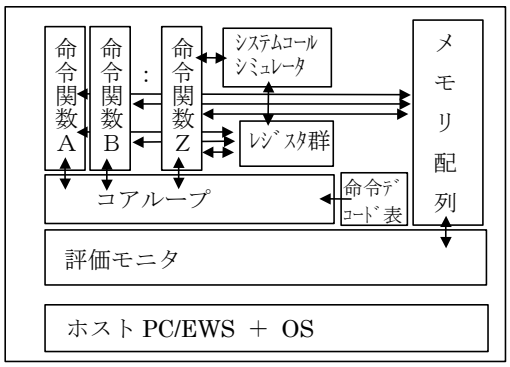

図 1 エミュレータのプログラム階層

エミュレータの構成を**図 1**と下記に示す．

- コアループ：各命令に共通な処理を実行
- 命令関数：コアループより呼び出される個々の命令種類に対応した関数
- システムコールシミュレータ：ファイル入出力，動的メモリ確保(malloc)などの OS 動作を模擬
- レジスタ群：プログラムカウンタ(PC)，汎用レジスタ(GR)，浮動小数レジスタ(FR)，制御レジスタ(CR)，状態語などアーキテクチャ規定のレジスタ変数
- メモリ配列：模擬アーキテクチャのメモリを模擬
- デコード表：命令語より命令関数へのポインタを取得するデコード表，2 フィールドによる命令コードまでデコード可能
- ホスト PC/EWS：OS 制御下で C コンパイルとエミュレータ起動が可能な環境
- 評価モニタ：バイナリローダ，実行時間計測機能など．

実システムとは下記の点が異なる．

- 実システムではスタンドアロンか専用 OS を使用．
- 実システムではメモリのアドレスがそのまま見え，OS と共有される．
- 本評価システムでは評価のため命令数計数を行なう．
- 本評価システムでは，I/O 動作，制御命令，アドレス変換は対象から除外する．

2.2. レジスタと共有変数

模擬対象のアーキテクチャで規定されたレジスタ配列(gr[],fr[],cr[])，プログラムカウンタ(PC)，状態語の他に，下記の変数を持つ．

exp_cond：例外発生とその内容を示す
pc_next：分岐後の PC 値
br_taken：分岐発生を示す
icode：命令語
dec1, dec2 または dec2a：命令デコード表

2.3. コアループの構造

このエミュレータ実装では，開発後の保守の容易化のため，命令に依存しない共通処理を行なう「コアループ」と「命令関数」を別ける．コアループでは，

1) exp_cond をチェックし例外発生がない場合には命令実行を繰り返す
2) プログラムカウンタ(pc)の示すメモリ配列より命令語を読み出し icode へセット
3) 命令語 icode の一部フィールドを切り出しデコードを行い，該当する命令関数への分岐
4) 命令関数内での例外発生事象のチェックのため exp_cond の値を判定
5) br_taken による分岐発生の判断，プログラムカウンタ pc のインクリメントまたは pc_next の値を pc にセット
6) 評価のための命令数の計数

を行なう．例外検出は命令語の読み出し前と命令実行後にそれぞれ行なう．

2.4. 命令デコード表の構造

命令デコード表の構造をコードにて示す．

```
struct dec1_s {
        int sft;            // 命令語の右シフト数
        int msk;            // それに AND するマスク
        int ix;             // 詳細デコードのインデックス
        int dummy           // サイズを 2 冪に調整
} dec1[MAIN_OP_SIZE];
struct dec_all_s {
        void (*func)(void); //命令関数へのポインタ
        // その他の情報
} dec2[MAIN_PLUS_SUB_OP_SIZE];
void (*dec2a[MAIN_OP_SIZE])(void); //最適化用
```

図 2に命令形式の例を示すが，命令コードは単一または複数のフィールドから構成され，命令関数のアドレスをセットされた配列 dec2 または dec2a の該当エントリを連想する．

0-5	6-10	11-15	16-20	21-25	26-30	31
Opcode	Reg#	Reg#	UIMM/SIMM/BD			
Opcode	LI				AA	LK
Opcode	Reg#	Reg#	Reg#	SubOp		Rc
Opcode	Reg#	Reg#	Reg#	Reg#	SubOp	Rc

図 2 PowerPC[4]の命令形式の例

2.5. 命令関数の処理

各命令関数では，下記の処理を行なう．

1) 命令処理に必要な各フィールド（イミーディエト値，レジスタ番号）を命令語 icode よりシフトとマ

スクにより切り出し
2) 例外条件があれば検出
3) 演算数をレジスタ配列 gr[]などからの読み出し
4) 演算の実行またはメモリをアクセス
5) 演算結果をレジスタ配列に格納
6) 分岐命令では分岐判定 br_taken のセットしプログラムカウンタ値 pc_next を計算

2.6. C 言語による実装例

C 言語によるコアループの記述例を示す.

```
// コアループ
void core_loop() {
    int t, x;
    // 宣言,初期化の記述は省略
    while(exp_cond == 0) {
      if(pc >= mem_limit) { // 命令取り出し例外の例
            exp_cond = ADR_OVER;
            goto exception;
      }
      icode = mem[pc>> 2];
      // デコードと命令関数分岐
      if(x=dec1[t =icode>>26 & 63].ix ==0)
            dec2[t].func();     // メインフィールドより決定・分岐
      else                      //  サブフィールドより決定・分岐
            dec2[x+(icode>>dec1[t].sft&dec1[t].msk)].func();
      instr_cnt++;              // 評価用の命令数計数
      pc += 4;
      if(br_taken){             // 命令関数にて pc 値変更有り?
            pc = pc_next; br_taken =0;
      }
    }
exception:// 例外発生あり
}
```

簡単な命令関数の例を示す．また，不正命令検出はコアループで行なうと遅くなるため専用の関数を用意し判定を減らしている.

```
// 命令関数の例
void ins_ori() {     // イミディエト値[16:31]との論理和
        int rs,ra;
        rs = icode>>21& 0x1F;       //レジスタ番号[6:10]の抽出
        ra = icode >>16 & 0x1F;     //レジスタ番号[11:15]の抽出
        gr[ra]= gr[rs] | icode & 0xFFFF;
}
```

3. エミュレータ性能の評価方法

評価にあたり，エミュレーション対象とする命令セットアーキテクチャ(レガシーと省略)，エミュレーション実行をするホストプラットフォーム（ホストと省略)，エミュレーション実行するバイナリモジュール(評価プログラム)について，次のように選定した．評価の指標は，レガシー命令 1 個を実行するのに必要なホストのクロックサイクルを CPI(Clock cycles Per Instruction)と定義する.

3.1. レガシー命令セットアーキテクチャの選定

下記の理由により，レガシーアーキテクチャの評価対象として PowerPC[4]を選択した.

- 実装が容易な RISC アーキテクチャである.
- 命令セットとして一般に知られている.
- エミュレーション動作時間をネイティブ動作時間で割ったスローダウン値の評価にも使用するため，ホストアーキテクチャにも選定可能である.

また，命令セットアーキテクチャへの依存性の評価のため，別のアーキテクチャとして M32R[5]を選択した.

3.2. ホストアーキテクチャの選定

ホストアーキテクチャには，OS 制御下でコンパイルと実行が可能なことに加え，周波数やアーキテクチャのバリエーション，パイプラインアーキテクチャが公開されホスト動作のクロックレベル動作が判ること，big と little の両方のエンディアンの評価を考慮し，PowerPC を基準とし，Pentium III，Xeon と Ultra Sparc も選択した．詳細を表 1に示す.

表 1 ホストマシンの仕様

Architecture	CPU	Cache Memory	Machine	OS, Compiler
x86	Pentium III-M 800MHz	I:16K,D:16K, S:256K	Note PC	Win-2000 Vc++6.0
	Xeon 1.7GHz	I:16K,D:16K, S:256K	WS	Win-2000 VC++6.0
Sparc	Ultra sparc IIIi 1.28GHz	I:32K, D:64K S:1M	SunFire V240	Solaris SUN C5.6
Power PC	MPC750 300MHz	I:32K,D:32K, S:512K	Power Mac G3	OSX(BSD) gcc 3.1

3.3. 評価プログラムの選定

エミュレーション対象の評価プログラムには，CPU 単体性能評価に一般的である SPEC CINT95 ベンチマーク[6]より 099.go，129.compress，130.li を選び，PowerPC は OSX 上の gcc3.1, M32R は Linux 上の gcc2.9 で各々-O オプションによりコンパイルした.

4. 評価結果

4.1. SPEC CINT95 による実行性能の測定

SPEC CINT95 の 099.go(20,9), 129.compress(1400000 e 2231), 130.li を用いてエミュレーション性能を測定した結果を表 2(単位は MIPS, 括弧内は CPI)に示す．先に述べたようにレガシーアーキテクチャは PowerPC と M32R，ホストは 3 種類，周波数は 4 種類による結果である．このように，実行する命令セットや，パイプラインアーキテクチャ，コンパイラなどの違いにもかかわらず，CPI 値が似た値となり**図 3**のグラフのようにコア周波数への依存度が高い.

表 2 エミュレータの実行性能

Legacy	Program	PowerPC 300MHz	UltraSparc 1.28GHz	PIII-M 800MHz	Xeon 1.7GHz
PowerPC	099.go	4.8(62.5)	21.4(59.9)	13.3(60.3)	21.7(78.3)
	129.compress	5.3(57.1)	22.3(57.3)	14.1(56.9)	22.8(74.5)
	130.li	5.2(57.3)	22.7(56.3)	14.0(57.1)	23.5(72.2)
	Average	(59.0)	(57.8)	(58.1)	(75.0)
M32R	099.go	4.4(67.9)	19.4(66.1)	15.51(51.5)	20.8(81.9)
	129.compress	4.3(69.8)	15.1(53.0)	15.1(53.0)	20.9(81.5)
	Average	(68.9)	(66.6)	(52.3)	(81.7)

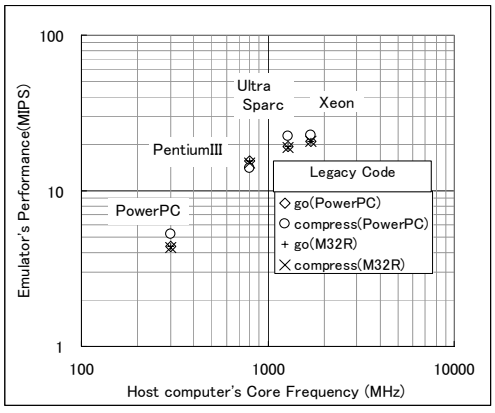

図 3 ホスト CPU コア周波数とエミュレーション性能

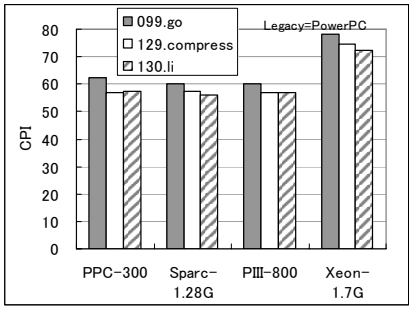

図 4 エミュレーション CPI 値

評価プログラまたはホストの違いによるエミュレーション CPI 値への影響が小さく 50～60 の範囲にあることを図 4にグラフで示す．P4 アーキテクチャの Xeon は，シフトや16ビット超えの加減算が2サイクル掛かるなど 1 クロックあたりの性能が PIII の 60～80%程度といわれておりこの CPI 値は妥当な結果と考える．

4.2. 評価結果の考察

4.2.1. ホスト CPU アーキテクチャへの依存性

CPU アーキテクチャ評価にはベンチマークプログラムが使われ, 一般に下記のような項目が性能を左右する．

- 並列に動作可能な実行ユニットの個数
- スーパスカラーの同時デコード／ディスパッチ数
- メモリ／演算パイプラインの深さ
- 分岐ペナルティとなるパイプライン長
- 命令／オペランドキャッシュ（容量，連想方式，置換方式，階層方式）
- 分岐ターゲットバッファ，分岐履歴バッファ

最近の高性能 RISC と x86 マイクロプロセッサではピーク時の CPI は 1 以下であり, 各種ペナルティの合計も CPI 換算で, 通常は数分の 1, 大きくても 1~2 程度迄である．

インタプリタ方式のエミュレーションにてベンチマークプログラムを実行する時には，ネイティブ動作で発生していた命令フェッチや分岐ペナルティは消滅し，オペランドアクセスのペナルティに代わる．ネイティブ動作のオペランドに関するパイプラインハザードは緩和されるがキャッシュミスペナルティは残る．インタプリタ動作として以下のオーバーヘッドが加算される．

1) インタプリタ動作処理の時間
2) エミュレーション対象命令語に対するメモリオペランドとしてのアクセス
3) レジスタ群の模擬やエミュレータ変数などのメモリオペランドアクセス

この結果，実行 CPU での命令キャッシュメモリは，殆どヒット状態となる．オペランドキャッシュや二次キャッシュメモリについては，ネイティブ動作時よりキャッシュミス回数の増加は免れないが，ミス回数増加量は元の数分の 1 以下と考えられる．

図 5にスローダウン(SD)の測定値を示すが，30～60 である. これらの CPU アーキテクチャ依存のオーバーヘッドの割合は 1／SD に薄まるため，ネイティブ動作で仮に 60%を占めていても 1～2%となるためベンチマークプログラムへの依存性が低くなるといえる．

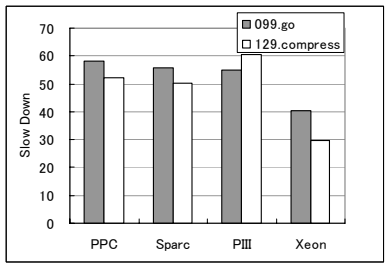

図 5 エミュレーションのスローダウン

4.2.2. 命令の種類とその実行頻度との関係

評価プログラムに依存して，実行されるレガシー命令の種類が異なり，エミュレータの命令関数ごとに実行されるホスト命令数の違いにより実行サイクル数が異なることから，エミュレーション性能は評価プログラムに大きく影響されると思いがちである．以下，それらの影響が小さいことを説明する．

別のシミュレータを用いて計測したベンチマークプログラムの命令実行頻度と累積頻度を図 6に示す．099.go, 129.compress と 130.li の平均頻度を棒グラフに，各ベンチマークの累積を折れ線に示している．このように, いずれも単純な命令である上位 20 位の命令が約 90%の頻度を占めており，命令種類の評価プログラムへの依存性は低い傾向を示すと言える．それは，言語仕様とコンパイラ毎の癖により，評価プログラムによる命令出現頻度の差が小さく，また, RISC では命令間の機能重複が少ないことによる．

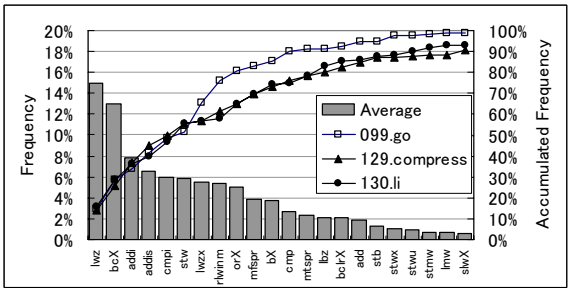

図 6 評価プログラムの命令実行頻度

表 3 処理時間の長い命令の頻度

	099.go	129.compress	130.li
Multiplication	0.190%	0.026%	0.001%
Division	> 0.001%	0%	0.001%
Floating Point	0%	0.367%	0%

次に，処理時間が長い乗除算と浮動小数命令の頻度を**表 3**に，フラグ生成を行なった命令の実行頻度を**表 4**に示す．

表 4フラグ生成を伴う命令の実行頻度

	099.go	129.compress	130.li
Arith-flag gen	0.058%	0.429%	0%
Logical-flag gen	0.013%	0%	1.684%
Shift-flag gen	0%	0%	0%

乗除算や浮動小数演算の正確なエミュレーションにはサイクル数が多く掛かるが，頻度が 2 桁以上小さいためその影響が出ていない．また，キャリーやオーバフ

ロー検出などCプログラムで記述が複雑になるフラグ生成を伴う命令は, RISC では特にコンパイラが多用しないため出現頻度が低くその処理時間の影響度は小さくなる.

以上，インタプリタ方式のエミュレーションでは SD 値が大きいためハードウェアアーキテクチャ依存のオーバーヘッドが薄まりホストアーキテクチャへの依存性が低くなる，また評価プログラムによらず出現する命令頻度の分布が似ており，更に処理時間の長い命令の頻度が極めて低いことにより，評価プログラムへの依存性が低いことを示した. それにより CPI が一定値になる. なお, その CPI 値の内容については 5 で細かく議論をする.

5. 命令あたりの処理時間の分析

本章では，1 命令の実行時間について言及する.

5.1. 命令単体の性能測定

個々の命令の単体性能については，次のような命令列より構成される簡易プログラムにて時間計測を行い評価した．r1 には予めループ回数を初期設定しておく．測定対象命令の処理時間は，N の個数をゼロにしたものを base として用意しそれとの時間差から計算する.

```
// or, or.命令の計測用の PowerPC 用のコード
START:  mtspr   ctr,r1      // ループ回数を設定
                // 計測対象命令を N 個並べる
LOOP:   or      r2,r2,r2    // 1 個目
        or      r3,r3,r3    // 2 個目
                :
        or      r9,r9,r9    // N 個目
        bdnz    LOOP
        .long   0           // 不正命令例外で停止
```

また，コアループ自体の処理時間測定のために仮想的な命令として，命令フィールド 1 個に対応した nop0，2 個に対応した nop1 を追加し, それらについても計測した. なお，これらの nopX 命令関数と各命令関数ではコアループに戻るリターン命令が存在するが，計測結果との対応容易化のためそのサイクルも便宜上コアループとして扱う. また, 本計測にあたり, 選択した x86 マシンでは, コアループから各命令関数へのレジスタ間接またはメモリ間接分岐動作にて，分岐元アドレスを用いていると思われる間接分岐予測が働き計測時間が見かけ上短くなったため, nopX を各 2 種類用意して交互に呼び出すように測定プログラムを変更して計測し適正化している．計測結果例を表 5に示す.

表 5　各命令の処理サイクル数の例

命令	PowerPC	Sparc	PIII	Xeon
nop0	21.7	31.9	24.2	32.3
nop1	38.8	51.4	28.6	45.7
lwz	64.0	54.2	46.3	49.2
bcX (inline)	49.9	48.3	62.9	48.3
bcX (taken)	70.8	59.6	69.1	55.1
addi	40.8	43.6	45.1	46.0
addis	39.4	42.6	42.3	50.6
cmpi (>0)	52.9	55.2	67.2	61.8
stw	45.8	47.2	46.7	55.9

5.2. コアループの処理性能

表 5の nop0/1 が示すように, インタプリタ方式のエミュレータでは，命令デコード処理を含むコアループが処理サイクル数の支配要因となるのでそれについて述べる.

5.2.1. PowerPC750 のコアループ性能

PowerPC 用にコンパイルされた 099.go をホスト PowerPC 750 で実行し，コアループ性能の分析と改善効果を測定した結果を**表 6**に示す．内容の列は，2.6で示したコードに対し, #1～#5 では異なるデコード方式で試行結果を, #6～#8 は#5 をベースに追加した改良を示す．CPI は 099.go の実測時間と命令数より求めた値を示す. 列 core は nop0 と nop1 のそれぞれの平均実測時間より求めたサイクル数を，2 フィールド使用によるデコードの割合(r_{d2})を乗じて加算したものである．この割合 r_{d2} は, 別のシミュレータを使い命令種ごとの実行回数を計数したものであり，099.go では 34.4%である．core%は core の CPI に占める割合である.

表 6　コアループ改善試行項目

#	内容	CPI	core	core%
1	デコードを関数化	68.7	31.2	45.5
2	ﾃﾞｺｰﾄﾞ結果をｷｬｯｼｭ(命令番地を索引, 命令語と命令関数ｱﾄﾞﾚｽを中身とした 16K ｴﾝﾄﾘ)に蓄積	65.7	30.7	46.6
3	#1 をインライン関数化	64.5	28.6	44.3
4	2.6に示したコードの dec2[]参照を関数配列 dec2a[]()に最適化	65.3	29.2	44.7
5	(*dec2a[icode>>26])()と簡略化，更にその分岐先で (*dec2a[icode>> 1 & 0x1FF])()と分岐	62.5	27.5	44.0
6	命令関数に icode を引数として追加	55.1	26.2	47.7
7	命令関数の戻り値に pc_next, br_taken, exp_cond>0 を追加	54.6	26.2	48.0
8	レジスタ群の構造体ﾎﾟｲﾝﾀを命令関数の第２引数に追加 (global 変数ｱｸｾｽｵｰﾊﾞｰﾍｯﾄﾞ削減)	52.8	28.1	53.2

関数戻りを除く間接分岐回数が 1 回の#4 と 2 回の#5 を core サイクル数で比較すると，nop1 の 32.1 が 38.8 と遅くなるが，nop0 が 27.6 から 21.7 に短縮され，r_{d2} の確率 34.4%を考慮すると#5 が有利となる.このようにコアループの処理サイクル数や命令関数の改善が進むに従いコアループの割合が#8 の 53%に達した.

5.2.2. 高周波数のホスト CPU のコアループ性能

高性能のプロセッサでは CPU コア周波数の向上に対応しパイプライン段数が深くなり，分岐命令とりわけ間接分岐はパイプラインペナルティ増加要因となるため，コアループは益々支配項になると考えられる．関数戻りを除く間接分岐回数 1 回の#4 と 2 回の#5 に着目し，各ホストのコアループ時間を計測から求めた．結果は**図 7**に示すように Sparc が 63%，PentiumIII が 65%となった．Sparc では，#5 に更に命令関数に対する符号拡張とインデックス指定値ゼロの計算処理の改良適用(#10)して CPI が 52.4 に減り 70.4%まで上がった. なお，PowerPC ではこの傾向が出にくいのは，パイプラインが浅いことと gcc コンパイラがレジスタを効率よく割り付けられないことによる．x86 はレジスタ本数が少ないためこれ以上の大幅改善は得られていない.

5.3. エミュレーション性能の予測

インタプリタ方式のエミュレーションではコアループが支配的である，上位 20 種の命令の実行頻度が約 90%を占める, SD 値が大きくハードウェア固有のオー

バーヘッドは薄まることから, 上位 20 種程度の命令単体実行時間を計測よりエミュレータ性能予測ができる．

具体的には, 命令頻度が上位 22 位の命令について処理時間を計測し，頻度と掛け合わせて加算した．cmp 命令などは判定結果によりサイクル数が異なるため正／負／零の平均を使用し，条件分岐命令では分岐の成立回数を計測，可変長命令(lmw/stmw)では転送データ語数を計測して補正している．残りの約 10%の頻度を占める命令については上位命令の処理時間の平均を使用した．ホスト 4 種類，評価プログラム 2 種類，デコード方式 2 種類について本方式で計算した CPI 予測を行なった．実測 CPI 値を 100%とした結果を**表 7**に示すが, 3 種のホストではそれだけで 90%以上の精度を得られた．残りの時間にはエミュレーション処理にて発生する分岐ターゲット／履歴バッファミス，投機的命令実行のオーバーヘッド，キャッシュミスなども含まれる．Xeon は 20 段のパイプ段数があり複雑な動作をするため投機実行の影響などが捕捉できていないと予想している．

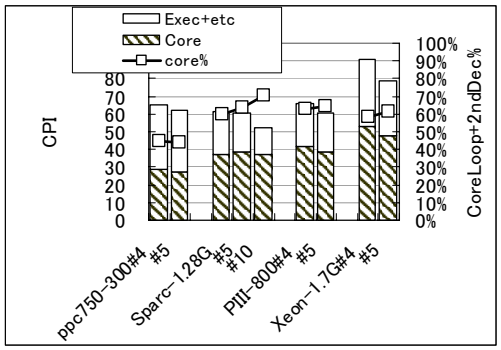

図 7　コアループの割合

表 7 実行頻度の上位 22 位より予測した CPI 値

	099.go#4	099.go#5	129.Compr#5
PPC750-300	95%	97%	94%
Sparc1.28G	95%	99%	98%
PIIIM-800	97%	90%	98%
Xeon-1.7G	76%	78%	78%

6. 関連研究と課題

エミュレーションは，元々シミュレーションの解釈実行の低速性をハードウェアの助けを借りて高速化しようとして考え出され，IBMsystem/360 による 1401 のエミュレーションや汎用エミュレータ MLP-900 などマイクロプログラミングによって発展してきた [1] ．近年では，文献[2]に紹介されているように，静的変換による HP3000(1987 年)， VAX や MIPS から Alpha へ変換する MX/Vest(1993 年)，静的と動的変換を併用して x86 から Alpha に変換する FX!32(1996 年), PA-RISC より IA-64 に動的変換する Aries[7]などのなどが著名である．

SimOS[8]のように教育や研究目的としてコンピュータ全体をシミュレーションしその高速化を研究したものも多いが，例外動作を含む命令仕様の厳密性を必要とする本エミュレータとは探求点が異なる．また，プロセッサアーキテクチャ研究によく使用される SimpleScalar ツールセット[9]をベースに行なわれるプロセッサ動作のシミュレーションでは実行時間が数百～数千倍となり本エミュレータとは高速化の課題や手法が異なる．

従来，インタプリタ方式の高速エミュレータはアセンブリ言語記述の機械語にて実現されてきたが，ホストの RISC 化に伴ったコンパイラの発展により機械語による局所的な最適化効果が薄くなった．機械語による現在での長所は，関数を使わない間接分岐，アドレス制約を課した分岐の最適化，使用頻度を意識したグローバル変数のレジスタ割り付けなどである．更に，パイプラインハザードを減らすため変数をレジスタ割付し命令関数内で次命令の解釈を並行処理する手法[10][11]により C 言語実装の 2.5～5 倍の性能に改善できるが，C 言語記述部との混在が課題として残る．本稿は，ホストのパイプラインアーキテクチャへの精通が不要で，移植性が高く一般性があるインタプリタ実装を対象に絞り，その定量評価を目的としている．

7. むすび

本稿では，SPEC CINT95 のベンチマークを用い，PowerPC, Sparc, x86 上に PowerPC と M32R の命令エミュレータを実装し評価を行なった．その結果より周波数性能への依存性が高く，また CPI 値もレガシーアーキテクチャや評価プログラムに依存しないことを示し，その理由を命令頻度の類似性と複雑な命令処理実行の少なさより裏付けた．インタプリタ方式では命令に依存しないコアループが支配的であり 44~70%の時間を占め，エミュレータを最適化し高性能プロセッサを用いるほどその傾向が高まることを示した．実行頻度の上位約 20 命令について各命令の単体処理時間を計測し頻度を乗じて加算した予測値は実測値と一致するため，エミュレーションにより実現するコンピュータ製品の性能予測に有効であることも確認できた．

文　献

[1] 萩原 宏, マイクロプログラミング， 産業図書，1977.

[2] E.Altman et al., “Welcome to the Opportunities of Binary Translation”, IEEE computer, 33, 40-45(2000)

[3] R. Cmelik, D. Keppel, “Shade: A Fast Instruction Set Simulator for Execution Profiling,” pp.128-137, ACM SIGMETRICS, Nashviville, TN, 1994 (also available from http://portal.acm.org)

[4] “PowerPC™ Microprocessor Family: The Programming Environments for 32-bit Microprocessors”, Motorola, 1997.

[5] http://www.renesas.com/avs/resource/japan/jpn/pdf/mpumcu/j32rsm.pdf

[6] SPEC CINT95, http://www.spec.org/cpu95

[7] C. Zheng, C. Thompson, “PA-RISC to IA-64:Transparent Execution, NoRecompilation” IEEE computer, 33, 47-52(2000)

[8] M. Rosenblum, S. Herrod, E. Witchel, A.Guputa, M. Rosenblum, S. Herrod, E. Witchel, A.Guputa, “Complete Computer Simulation: The SimOS Approach”, pp.34-43, IEEE Parallel and Distributed Technology, 1995.

[9] D. Burger, T.M. Austine, “The SimpleScalar tool set, version 2.0”, Tech. Report 1342, Computer Science Department, University of Wisconsin-Madison, June 1997

[10] 命令エミュレーション方法，近江谷康人,特開 2002-182928

[11] 平岡精一，近江谷康人，西川浩司，山崎弘巳，“ソフトウェア資産活用に有効なバイナリトランスレーション技術” 三菱電機技報, vol.77, No7, pp.59-62, July 2003.