SimCore/Alpha Functional Simulatorの設計と評価

吉瀬謙二[†,††] 片桐孝洋[†,††]
本多弘樹[†] 弓場敏嗣[†]

シンプルで可読性の高い記述，高速な実行という特徴を同時に満たす機能レベルの Alpha プロセッサシミュレータとして SimCore/Alpha Functional Simulator Version 2.0 (**SimCore Version 2.0**) を構築した．SimCore Version 2.0 は，同様の機能を提供する SimpleScalar ツールセットの sim-fast と比較して，19%の高速化を達成する．また，コンパイラと最適化オプションを変更した場合には，最大で 51%の高速化を達成する．本稿では，これらの高速化を実現する手法を中心に，SimCore Version 2.0 の設計と評価結果をまとめる．

An evaluation of a SimCore/Alpha Functional Simulator

Kenji Kise,[†,††] Takahiro Katagiri,[†,††] Hiroki Honda[†]
and Toshitsugu Yuba[†]

We developed a SimCore/Alpha Functional Simulator Version 2.0 (**SimCore Version 2.0**) for research and education activities. Its design policy is to maintain the simplicity and the readability of source code. In addition, a new version was tuned up for operating at high speed. As compared with a sim-fast in SimpleScalar tool set, SimCore Version 2.0 attains 19% of improvement in the simulation speed. Moreover, when a compiler and an optimization option are selected properly, the maximun 51% of improvement is attained.

1. はじめに

プロセッサアーキテクチャ研究のツールとして，あるいはプロセッサ教育のツールとして様々なプロセッサシミュレータ[4] が利用されている．近年の PC の高速化やクラスタの普及により，プロセッサシミュレータを動作させる環境は劇的に向上している．一方で，シミュレータ構築に費す時間は，実装したいアイデアの複雑化に伴い増加する傾向にある．既存のツールをベースとして開発を進めたとしても，シミュレータの構築に数カ月を必要としながら，その評価は数週間で終るようなケースも珍しくない．プロセッサ研究などの目的で広く利用されているプロセッサシミュレータとして SimpleScalar Tool Set[1] (以下 SimpleScalar) が有名だが，高速なシミュレーションを目的の一つとして実装されているために，必ずしも変更が容易なコードにはなっていない．

シンプルで理解しやすいコードであること，コードの変更が容易であることを第一の条件として Alpha プロセッサシミュレータ[2),8)] の構築をおこなってきた．これらのプロセッサシミュレータに加えて，計算機アーキテクチャの研究と教育のための重要なソフトウェアツールを提供するプロジェクトとして**SimCoreプロジェクト**を立ち上げ，その設計と実装を進めている．

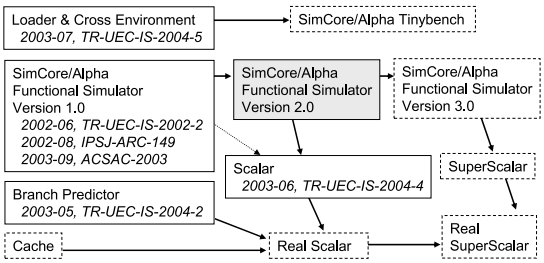

図 1　SimCoreプロジェクトの開発計画
Fig. 1　**The SimCore project** development plan.

図 1 に，SimCore プロジェクトの開発計画[5)~7)] を示す．提供するツール名とソースコードの依存関係を矢印で示した．SimCore プロジェクトの中心となるソフトウェアは，SimCore/Alpha Functional Simulator Version 1.0, 2.0, 3.0 という機能レベルのプロセッサシミュレータと，RealScalar, RealSuperScalar と

† 電気通信大学 大学院情報システム学研究科
Graduate School of Information Systems, The University of Electro-Communications
†† 独立行政法人科学技術振興機構 さきがけ
“Information infrastructure and applications”,
PRESTO, Japan Science and Technology Agency(JST)

呼ばれるクロックレベルのシミュレータである．

SimCore/Alpha Functional Simulator のことを，省略して SimCore と記述することがある．本稿では，図 1 の中央に示した SimCore Version 2.0 の設計と評価結果をまとめる．

サイクルレベルのアーキテクチャ・シミュレーションの高速化を目指す研究[9] は多い．本稿では，アーキテクチャ・シミュレーションではなく，機能レベルのシミュレータの設計と性能を議論する．

2. SimCore/Alpha Functional Simulator Version 2.0 の設計

SimCore Version 2.0 は，Version 1.0[2),8)] が持つ高い可読性を継承しながら，劇的な高速化を達成している．Version 1.0 と同様に，Version 2.0 は C++を用いて記述され，そのコード量はインクルードファイルを含めて **2,800 行以下**と非常に少ない．ソースコードの可読性の向上を優先して設定した SimCore Version 2.0 の設計方針を列挙する．

- グローバル変数を利用しない．
- 条件付きコンパイルを利用しない．
- 定数以外のマクロを利用しない．
- goto 文を利用しない．
- コード中の 1 行の幅を 50 文字以内とする．
- 簡潔な記述を保ちながら高速化を目指す．
- 多くのプラットフォームへの対応を目指す．

動作確認をおこなったプラットフォームを列挙する．これらの環境においては，dhrystone 及び SPEC CINT95 と CINT2000 の 20 本のプログラムの動作が正しいことを確認している．

(1) Pentium 4, RedHat 7.3, gcc version 2.96
(2) Pentium 4, RedHat 7.3, Intel C++ 7.1/8.0
(3) Pentium 4, RedHat 7.3, PGI Compiler 5.1
(4) Pentium 4, Cygwin version 2.340, gcc 3.2
(5) Opteron, Turbo Linux 8, gcc version 3.2.2
(6) Alpha 21264, Tru64 V5.1, gcc version 2.95.2

次に列挙する環境では，dhrystone を用いた動作確認をおこなっている．

(1) UltraSPARCIII, Solaris 8, gcc version 2.95.3
(2) MIPS R14000, IRIX6.5, MIPSpro C++

2.1 命令フロントエンド計算の再利用

本節では，命令パイプラインのフロントエンド部分の計算に関する再利用性を用いた SimCore Version 2.0 の高速化手法に関して述べる．

図 2 に，高速化を施す前のシンプルなメインループを示す．SimCore における 1 命令のシミュレーションは命令パイプラインに対応する 7 個のメソッドを呼び出すことで実現する．

図 2 の Fetch, Slot, Issue という 3 つのメソッドは，命令パイプラインのフロントエンドに対応する，32

```
void simple_chip::loop_slow(){
  while(ev->sys->running){
    p->Fetch(&ev->as->pc); /* pipe stage 0 */
    p->Slot();             /* pipe stage 1 */
    p->Issue();            /* pipe stage 3 */
    p->RegisterRead();     /* pipe stage 4 */
    p->Execute();          /* pipe stage 5 */
    p->Memory();           /* pipe stage 6 */
    p->WriteBack();

    ev->e->retired_inst++;
    house_keeper(p);
  }
}
```

図 2 シンプルな実装の SimCore のメインループ
Fig. 2 A simple implementation of SimCore main loop.

ビットの命令コードのフェッチ，デコード，即値の計算を担当する．これらは，静的に同じ命令の処理であれば，毎回同様の計算を繰り返すという特徴がある．このため，計算した結果を保存しておくことで，同様の処理の繰り返しを省略する．この高速化手法を命令フロントエンド計算の再利用と呼ぶことにする．

```
#define IMSK 0x0ffff  /* mask of inst_buf */
void simple_chip::loop(){
  instruction **ib = new instruction*[IMSK+1];
  for(int i=0; i<IMSK+1; i++){
    ib[i] = new instruction(ev);
  }

  while(ev->sys->running){
    int index = (ev->as->pc>>2) & IMSK;
    instruction *pt = ib[index];

    if(pt->Cpc!=ev->as->pc){
      pt->Fetch(&ev->as->pc);
      pt->Slot();
      pt->Issue();
    }
    pt->RegisterRead();
    pt->Execute();
    pt->Memory();
    pt->WriteBack();

    ev->e->retired_inst++;
    if(ev->sc->slow_mode) house_keeper(pt);
   }
}
```

図 3 命令フロントエンド計算の再利用を施したメインループ
Fig. 3 A main loop with the pipeline frontend reuse.

図 3 に，命令フロントエンド計算の再利用を施したメインループを示す．ダイレクトマップ方式の命令キャッシュと同様に，過去の計算結果を格納する instruction 型の配列を用意する．定数 IMSK で指定した数がキャッシュのエントリ数であり，コードでは，64K のエントリ (16 進数表記で 0xffff) に設定されている．プログラムカウンタから，配列のインデックス

を生成し，そこに格納されている命令と今実行している命令が同じであれば，過去の履歴を利用できる．この時，命令フロントエンド計算を省略する．

64K エントリ，ダイレクトマップ方式の命令キャッシュのヒット率は高い．このため，ほとんどの命令の実行において，Fetch, Slot, Issue という命令パイプラインの前半部分の処理を省略できる☆．

図 2 のコードから僅か 10 行の追加で，本手法を実装できることに注意する必要がある．実装は簡潔だが，次章の評価結果で示すように，2 倍以上の高い速度向上を達成する．

2.2 関数呼び出しオーバヘッドの削減

命令フロントエンド計算の再利用を施すことで，SimCore の実行時間の多くは命令パイプラインのバックエンドで費やされる様になる．この時，図 3 に示した RegisterRead, Execute, Memory, WriteBack というメソッドの呼び出しオーバヘッドが目立つようになる．このオーバヘッドを削減するために，命令パイプラインのバックエンドを構成する 4 つのメソッドの処理を 1 つのメソッド BackEnd として記述する．このチューニングを施した後のメインループを図 4 に示す．

```
while(ev->sys->running){
  int index = (ev->as->pc>>2) & IMSK;
  instruction *pt = ib[index];

  if(pt->Cpc!=ev->as->pc){
    pt->Fetch(&ev->as->pc);
    pt->Slot();
    pt->Issue();
  }
  pt->BackEnd();

  ev->e->retired_inst++;
  if(ev->sc->slow_mode) house_keeper(pt);
 }
```

図 4 命令フロントエンド計算の再利用と関数呼び出しオーバヘッドの削減を施したメインループ
Fig. 4 A main loop with the pipeline frontend reuse and the function call overhead elimination.

SimCore Version 2.0 では，図 4 に示したメインループを利用する．

これまでに述べた高速化手法の他に，8 バイト単位のメモリ参照 (ロード・ストア) を基本とする SimCore の実装を，4 バイト単位のメモリ参照に変更することで数%の高速化を達成できる．しかしながら，この高速化は，ソースコードが複雑になるという理由で採用しなかった．

☆ ダイレクトマップ方式から，セットアソシアティブ方式に変更することで数%の速度向上を期待できる．コードが複雑になることを避けるため，この高速化は断念している．

3. SimCore/Alpha Functional Simulator Version 2.0 の評価

本節では，SimCore Version 2.0 の動作速度を評価し，既存のツールと比較して，高速に動作することを示す．また，シミュレーション時間を見積もる際に重要となる実時間性能比 (slow down) を概算する．

3.1 ベンチマークプログラム

シミュレーションの対象となるアプリケーションとして SPEC CINT95 の 8 本のベンチマーク，あるいは dhrystone を利用する．SPEC CINT95 のバイナリは DEC C コンパイラ，最適化オプション O4 を用いて生成する．各ベンチマークの実行命令数が約 1 億から 2 億命令の範囲となる様に入力のデータセットを調整する．

3.2 SimCore と sim-fast の実行速度の比較

本節では，SimCore の動作速度を測定し，SimpleScalar に含まれる sim-fast の動作速度と比較する．

本節の評価には，現在利用できる計算機の中でコスト性能比が高く，広く普及しているという理由から Pentium4 Xeon を搭載する計算機を利用する．計算機の仕様を以下に列挙する．DELL PowerEdge2650, Pentium4 Xeon (2.8GHz, 512KB L2 cache) を 2 個搭載，メインメモリ 2GB，RedHat 7.3.

評価基準として，1 秒当たりに処理される命令数 (MIPS: Million Instructions Per Second) を採用する．この値が高い方が高速なシミュレータとなる．

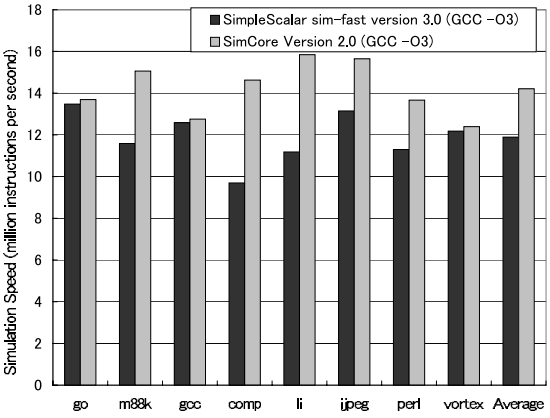

図 5 SimCore と sim-fast の実行速度の比較
Fig. 5 Simulation speed of SimCore and sim-fast.

SimCore と SimpleScalar に含まれる sim-fast の動作速度の結果を図 5 に示す．2 つのシミュレータは，GCC，最適化オプション O3 を用いてコンパイルした．全てのベンチマークプログラムにおいて，SimCore が高い動作速度を示す．compress のシミュレーションにおいて，SimCore は 50%という最も高い速度向上率を

達成する．8 本のベンチマークの平均では，SimCore の動作速度は 14.2MIPS で，sim-fast の 11.9MIPS と比較して，19%の高速化を達成する．

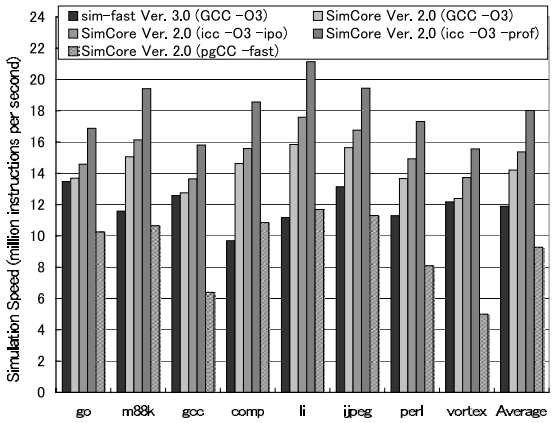

図 6 コンパイラと最適化手法を変化させて測定した実行速度
Fig. 6 Simulation speed measured with various compilers and optimizations.

次に，コンパイラと最適化手法を変化させて測定した動作速度を図 6 にまとめる．図 6 には，ベンチマーク毎に 5 つのデータを表示している．左端の 2 本のデータは図 5 に示した sim-fast と SimCore の動作速度である．

3 番目のデータは，Intel C++ Version 7.1 コンパイラ，適化オプション O3 とファイル間最適化 (-ipo) を施したデータである．この場合の動作速度は平均 15.3MIPS であり，sim-fast と比較して 28%の高速化を達成する．なお，sim-fast は Intel C++ コンパイラを用いてコンパイルすることができなかった．

4 番目のデータは，3 番目の最適化オプションに，プロファイル情報を利用する最適化 (-prof_use) を加えたデータである．プロファイルデータとして，10,000 回ループの dhrystone の実行履歴を用いている．この履歴を得るための実行を含むコンパイル時間は 5 秒以内と非常に短い．Intel C++コンパイラのプロファイル最適化を用いることで動作速度は平均 18.0MIPS に達する．sim-fast と比較して 51%の高速化を達成する．

右端の 5 番目のデータは，商用コンパイラの PGI Compiler 5.1，最適化オプション -fast を用いた結果である．動作速度の平均は 9.3MIPS と GCC を下回る結果となった☆．

これらの結果から，sim-fast の動作速度と比較して，SimCore は GCC を用いた場合に 19%，Intel C++ コンパイラを用いてプロファイル情報を利用する場合に 51%の高速化を達成することを確認した．

3.3 SimCore Version 2.0 に施した高速化手法の影響

本節では，2 章において議論した高速化手法の影響を定量的に評価する．

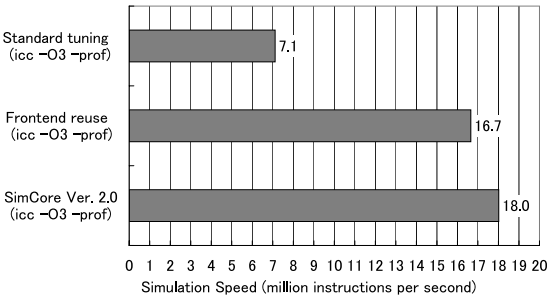

図 7 SimCore に施した高速化手法の影響
Fig. 7 Influence of the tuning methods on a SimCore.

幾つかの版の SimCore のソースコードを Pentium4 Xeon 搭載の計算機で動作させた結果を図 7 にまとめる．ここに示した動作速度 (MIPS) は SPEC CINT95 の 8 本のベンチマークの平均である．

一番上のデータは，図 2 に示したシンプルな実装のメインループを持つ版の SimCore の動作速度である．この版の SimCore には，メモリアクセス量の削減，実行頻度を考慮した switch 文内の case の入れ替え，データ構造の簡略化などの一般的なチューニング手法が施されているが，プロセッサシミュレータに特有の最適化は施されていない．この版の SimCore の動作速度は 7.1MIPS である．

2 本目のデータは，標準的なチューニングに加えて，命令フロントエンド計算の再利用を施したメインループ (図 3) を持つ版の SimCore の動作速度である．これにより，倍以上の高速化を達成し，動作速度は 16.7MIPS となる．このように，命令フロントエンド計算の再利用の効果は大きい．

3 本目のデータは，標準的なチューニング，命令フロントエンド計算の再利用に加えて，関数呼び出しオーバヘッドの削減を施した SimCore Version 2.0 (図 4) の動作速度である．関数呼び出しオーバヘッドの削減が 7%の速度向上をもたらし，この時の動作速度は 18.0MIPS に到達する．

このように，命令フロントエンド計算の再利用と関数呼び出しオーバヘッドの削減は，簡潔な記述で実装可能な手法でありながら高い性能向上を達成する．

3.4 様々な計算機における SimCore Version 2.0 の動作速度の比較

これまでに示したデータは Pentium4 Xeon を用いて測定したものである．本節では，様々な計算機における SimCore Version 2.0 の動作速度を比較する．利

☆ -O3 オプションより，-fast 指定の方が速かった．GCC より遅い理由は詳しく調査していないが，最適化オプションの指定が適切ではない等の理由が考えられる．

用する計算機の構成をまとめる．

R14000 SGI Origin 3400, MIPS R14000 500MHz を 32 個搭載，メインメモリ 16GB, IRIX 6.5.

UltraSPARCIII Sun Blade2000, UltraSPARC III Cu 1.05GHz を 1 個搭載，メインメモリ 2GB, SOLARIS 8.

Alpha21264 HP AlphaStation DS10, Alpha21264 (600MHz, 2MB L2 cache) を 1 個搭載，メインメモリ 256MB, Tru64 UNIX 5.1A.

PentiumIII Xeon DELL PowerEdge6400，PentiumIII Xeon (700MHz, 2MB L2 cache) を 4 個搭載，メインメモリ 512MB，RedHat 7.3.

Opteron LIBRAGE TWIN 244，Opteron 244 (1.8GHz, 1MB L2 cache) を 2 個搭載，メインメモリ 2GB, TurboLinux8 for AMD64.

Pentium4 Xeon DELL PowerEdge2650，Pentium4 Xeon (2.8GHz, 512KB L2 cache) を 2 個搭載，メインメモリ 2GB，RedHat 7.3.

アプリケーションとして，100 万回ループする Dhrystone Version 2.1 (Language: C) を利用する．この実行ファイルは gcc version 2.95.2 を用いて作成する．

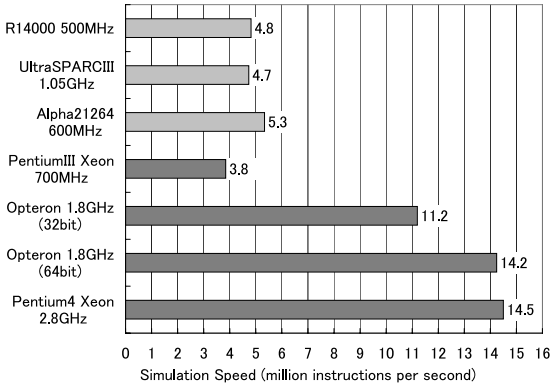

図 8 様々な計算機を用いて測定した SimCore の動作速度
Fig. 8 SimCore simulation speed on different computers.

様々な計算機を用いて測定した SimCore Version 2.0 の動作速度を図 8 にまとめる．SimCore のコンパイルには，下の例外を除いて，GCC，最適化オプション O3 を利用する．R14000(Origin 3400) では，MIPSpro C++, 最適化オプション-Ofast を利用する．Opteron プロセッサの場合には，オプション-m32 を用いて 32 ビットのコードを生成した場合と，64 ビットのコードを生成した場合とに分けて結果を示す．

1 世代前の PentiumIII プロセッサと比較すると，R14000，Alpha21264，UltraSPARC といった RISC 系の CPU を搭載した計算機の方が性能が高いことがわかる．しかし，これらと比較して，最新の Pentium4 や Opteron プロセッサは圧倒的に高い値を示す．

Opteron プロセッサの場合には，64 ビットコードを生成することで，32 ビット版と比較して，26%の高速化を達成できる．しかしながら，64 ビット版のコードを利用したとしても，Pentium4 プロセッサにやや劣る結果となった☆．また，Pentium4 の場合には，Intel C++コンパイラを用いて 18.0MIPS まで速くなることに注意する必要がある．

ここに示した 7 種類の計算機は RISC と CISC，リトル・エンディアンとビッグ・エンディアンといった異なる特徴を持つ計算機である．これらの相違にも関わらず，様々な計算機の上で，条件付きコンパイルを利用しない全く同一のコードが動作するという特徴は SimCore の利用の一つである☆☆．

3.5 アプリケーション 1 命令を処理するための SimCore の命令数と実時間性能比

アプリケーションとしてループ回数を 50 万と 100 万に設定した dhrystone と，SPEC CINT95 に含まれる 2 つのベンチマークを用いて，アプリケーション 1 命令を処理するために必用となる SimCore の命令数を計測した結果を表 1 にまとめる．

表 1 1 命令をシミュレーションするための SimCore の命令数 (2 列目と 3 列目の単位は million instructions)
Table 1 The number of SimCore instructions to simulate one application instruction.

プログラム名	app inst	sim inst	sim/app
dhrystone(500k)	27.3	4,804	175
dhrystone(1000k)	54.6	9,453	173
099.go 9 9	138	24,279	175
129.compress(30k)	142	24,603	173

あるアプリケーションを実行するために必用となる命令数は，SimCore の上で当該アプリケーションを走らせることで正確に知ることができる．このようにして得られた命令数を表 1 の 2 列目に示す．2 列目と 3 列目の単位は million instructions である．

同様にして，Alpha-AXP アーキテクチャの上でコンパイルした SimCore のバイナリを SimCore の上で走らせることで，当該アプリケーションを走らせた SimCore の命令数を正確に知ることができる．このようにして得られた命令数を 3 列目に示す．

これらの比率から，アプリケーションの 1 命令を処理するために，SimCore の約 174 命令を費やしていることがわかる．もし，アプリケーションを実行した場合の IPC(Instructions Per Cycle) と，SimCore を実行した場合の IPC が等しいとすると，SimCore の実時間性能比は 174 となる．

☆ 結果は CPU 以外に，メモリの性能に強い影響を受ける．Pentium4 が Opteron に勝っているということを主張したい訳ではなく，幾つかの利用できる計算機で測定をおこなったところ，たまたま Pentium4 が最速だったということに過ぎない．

☆☆ この特徴から，ベンチマークプログラムとして SimCore を利用すると面白いのではないだろうか．

3.6 実時間性能比 (slow down) の測定

本節では，シミュレーション時間を見積もる際に重要となる実時間性能比 (slow down) を測定する．アプリケーションとして dhrystone と，SPEC CINT95 に含まれる 3 つのベンチマークを利用する．AlphaStation DS10 (Alpha21264 600MHz プロセッサ，256MB メモリ) で動作させた実時間が数十秒となるように，アプリケーションの入力データセットを調整する．

表 2 AlphaStation DS10 で測定した実時間性能比
Table 2 Slow down measured on AlphaStation DS10.

プログラム名	実時間	sim 時間	slow down
dhrystone (100m)	40.7 秒	10,027 秒	246
go 50 21	54.2 秒	7,411 秒	136
compress (4MB)	26.5 秒	3,145 秒	119
perl primes	24.1 秒	3,568 秒	147

測定結果を図 2 に示す．AlphaStation DS10 の上で，time コマンドを利用して測定したアプリケーションの実時間を 2 列目に示す．AlphaStation DS10 上で，SimCore のシミュレーション時間を計測した結果を 3 列目に示す．これらの比率から計算した実時間性能比 (slow down) を 4 列目に示す．

SPEC CINT95 のベンチマークと比較して，dhrystone の実時間性能比は 246 と高い．先の節で求めた，アプリケーションの 1 命令を処理するために SimCore の約 174 命令を費やすというデータを考慮すると，dhrystone ではアプリケーションを実行した場合の IPC が SimCore の IPC より高いことがわかる．

より現実的なアプリケーションに近い SPEC CINT95 の実時間性能比は 120 から 150 の範囲となった．dhrystone とは対象的に，これらの場合には，アプリケーションを実行した場合の IPC が SimCore の IPC より低いという興味深い結果となった☆．

4. おわりに

プロセッサアーキテクチャ研究とプロセッサ教育におけるツール群の提供を目的として SimCore プロジェクトを立ち上げ，その開発をおこなっている．本プロジェクトにおける中心的な役割を果たす機能レベルのプロセッサシミュレータ SimCore Version 2.0 の設計と評価結果をまとめた．

SimCore Version 2.0 は，シンプルで可読性の高い記述と高いポータビリティを保ちながら，高速に動作するという特徴を持つ．

SPEC CINT95 と dhrystone を用いた評価から，同様の機能を提供する SimpleScalar ツールセットの sim-fast と比較して，SimCore Version 2.0 は 19%の高速化を達成する．また，コンパイラと最適化オプションを適切に変更した場合には，最大で 51%の高速化を達成することを明らかにした．

SimCore Version 2.0 では，一般的なチューニングに加えて，プロセッサシミュレータに特化したチューニングを施している．これらチューニング手法の影響を定量的に評価し，簡潔な記述で実装可能な手法でありながら高い性能向上を達成することを確認した．

シミュレーション時間を見積もる際に重要となる実時間性能比 (slow down) を測定した．SPEC CINT95 の幾つかのベンチマークにおいて，実時間性能比が 120 から 150 の範囲となることを示した．

SimCore/Alpha Functional Simulator Version 2.0 のソースコードは次の URL からダウンロードできる．
`http://www.yuba.is.uec.ac.jp/~kis/SimCore/`

☆ Alpha21264 プロセッサで実行した場合には，dhrystone, SimCore, SPEC CINT95 という順番で，多くの命令レベル並列性を抽出していると考えられる．

参 考 文 献

1) Doug Burger and Todd M. Austin: The Simplescalar Tool Set, Version 2.0, Technical Report CS-TR-1997-1342, University of Wisconsin, Madison (1997).
2) Kise, K., Honda, H. and Yuba, T.: SimAlpha Version 1.0: Simple and Readable Alpha Processor Simulator, *Lecture Note in Computer Science (LNCS)*, Vol. 2823, pp. 122–136 (2003).
3) R. E. Kessler: The Alpha 21264 Microprocessor, *IEEE Micro*, Vol. 19, No. 2, pp. 25–36 (1999).
4) Shubhendu S. Mukherjee, Sarita V. Adve, Todd Austin, Joel Emer and Peter S. Magnusson: Performance Simulation Tools, *IEEE Computer*, Vol. 35, No. 2, pp. 38–39 (2002).
5) 吉瀬謙二，片桐孝洋，本多弘樹，弓場敏嗣: SimAlpha-Loader の実装とクロス開発環境の構築, 技術報告 UEC-IS-2003-5, 電気通信大学 大学院情報システム学研究科 (2003).
6) 吉瀬謙二, 片桐孝洋, 本多弘樹, 弓場敏嗣: スカラプロセッサシミュレータの実装と動作検証, 技術報告 UEC-IS-2003-4, 電気通信大学 大学院情報システム学研究科 (2003).
7) 吉瀬謙二, 片桐孝洋, 本多弘樹, 弓場敏嗣: 高性能プロセッサのための代表的な分岐予測器の実装と評価, 技術報告 UEC-IS-2003-2, 電気通信大学 大学院情報システム学研究科 (2003).
8) 吉瀬謙二, 本多弘樹, 弓場敏嗣: SimAlpha: C++ で記述したもうひとつの Alpha プロセッサシミュレータ, 情報処理学会研究報告 2002-ARC-149, pp. 163–168 (2002).
9) 中田尚, 大野和彦, 中島浩: 高性能マイクロプロセッサの高速シミュレーション, 先進的計算基盤システムシンポジウム SACSIS2003 論文集, pp. 89–96 (2003).