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

吉瀬謙二[†] 本多弘樹[†] 弓場敏嗣[†]

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利用を目的として、プロセッサシミュレータ SimAlpha Version 1.0 を構築した。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 C++を用いて記述した。インクルードファイルを含むコード量は約 2800 行と少ない。本稿では、実際のコードを参照しながら、SimAlpha のソフトウェアアーキテクチャを明らかにする。また、SimAlpha の利用例として SPEC CINT95 と CINT2000 の理想的な命令レベル並列性を測定した結果を報告する。

SimAlpha: Yet Another Alpha Processor Simulator in C++

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

We developed a processor simulator SimAlpha Version 1.0 for research and education activities. Its design policy is to maintain the simplicity and the readability of source code. SimAlpha is written in C++ and is only 2,800 lines of code. This paper describes the software architecture of SimAlpha referring to its source code. As an actual example of use, we report the oracle IPC of SPEC CINT95 and CINT2000 benchmarks measured with SimAlpha.

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。その一方で、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する一方で、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 SimAlpha Version 1.0 を構築した。SimAlpha は SimpleScalar/Alpha の機能シミュレータと同等の機能をもつ。現在の実装では、命令間でオーバーラップすることなく、1 つの命令毎にプロセッサの動作を模倣する。パイプライン処理やアウトオブオーダ実行を模倣するクロックレベルのシミュレータではないが、それらへの拡張を視野に入れたコード記述となっている。SimAlpha は SimpleScalar とは異なるポリシーで、ゼロから記述したシミュレータである。C++を用いて記述され、インクルードファイルを含むコード量は約 2800 行と少ない。SimAlpha の実装においては可読性を考慮して、グローバル変数、goto 文、条件付きコンパイルを利用していない。SimAlpha は SimpleScalar の置き換えを狙っているものではなく、異なるポリシーを持つもう一つの実装を示すことが構築の狙いである。プロセッサシミュレータをツールとして捉えると、多くの選択肢の中から適切なツールを選択できる利点は大きい。SimAlpha は研究と教育のためのツールとしてのもう一つの選択肢を提供する。

2 章では SimAlpha の動作環境と検証手法を、3 章では実際のコードを参照しながらソフトウェアアーキテクチャを明らかにする。4 章では SimAlpha の利用例として理想的な並列性を測定した結果を示し、5 章で本稿をまとめる。

2. SimAlpha の実行と動作検証

SimAlpha は Intel アーキテクチャ上の Linux 環境で動作する。レジスタの内容とメモリの内容を列挙した SimAlpha 独自の実行イメージを与えることでシ

† 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

ミュレーションを開始する。ELF や COFF といった実行ファイル形式の知識が不要であり、複雑なローダを SimAlpha の実装から切り離すことができるという利点から、図 1 に示す独自形式の実行イメージファイルを採用した。実行イメージファイルは Alpha バイナリから構成できる☆ 。

```
/* SimAlpha 1.0 Image File */
/*** Registers ***/
/@reg 16 0000000000000003
/@reg 17 000000011ff97008
/@reg 29 0000000140023e90
/@reg 30 000000011ff97000
/@pc  32 0000000120007d80
/*** Memory    ***/
@11ff97000 00000003
@11ff97008 1ff97138
@11ff9700c 00000001
```

図 1 SimAlpha の実行イメージファイル

Pentium III 1GHz、メインメモリ 512MB の計算機上で SPEC CINT95 と CINT2000 の 20 本のプログラムを用いて SimAlpha と sim-safe[1] の動作速度を測定したところ、sim-safe のシミュレーション速度 3.1 MIPS に対して、SimAlpha のシミュレーション速度は 1.1 MIPS という結果となった。SimAlpha のコンパイルには egcs-1.1.2 release、最適化オプション O2 を用いた。SimpleScalar と比較して、約 3 倍のシミュレーション時間が必要となる点が SimAlpha の欠点の一つとなっている。ただし、シミュレータ開発の時間が支配的な場合にその開発時間を短縮できるとすれば、シミュレーション速度で 3 倍の速度差は問題にならない範囲といえる。

SimAlpha は、アーキテクチャステートの動作レベルで、SimpleScalar との互換性を確認している。SPEC CINT95 と CINT2000 の 20 個のベンチマークプログラムを利用して、1 つの命令を実行する度に、SimAlpha のアーキテクチャステート (プログラムカウンタ, 32 本の整数レジスタ、32 本の浮動小数点レジスタ) と、SimpleScalar のアーキテクチャステートの全ての値を比較し、一致することを確認している☆☆。従来のシミュレータでは、グローバル変数の存在などにより、一つのプロセス内に 2 つのシミュレーションのイメージを作ることは難しかった。このため、正しい実行結果をトレースとしてファイルに保存し、これと比較することで、もうひとつのシミュレータの検証をおこなっていた。SimAlpha の場合には、高速かつ確実に検証をおこなう手段として、SimpleScalar などのプログラムに SimAlpha のオブジェクトを埋め込む機能を提供する。これを実現するために、C のインターフェースを提供するとともに、ファイル識別子の値を厳密に一致させるための工夫を凝らしている。検証のために必要となる SimpleScalar のコード変更は 1ヶ所で約 40 行と非常に少ない。

SimAlpha の提供する、複数のシミュレーションイメージを作成したり、あるシミュレータの中にオブジェクトを埋め込んで検証する手法は、SimAlpha と SimpleScalar の間の検証に利用されるだけではなく、開発中の幾つかのシミュレータの動作を検証するためにも利用することができる。この機能を利用することで、開発中のシミュレータのバグを早期に発見したり、シミュレータの正当性を確認することが可能になる。

3. ソフトウェアアーキテクチャ

本章では、SimAlpha のコードの高い可読性を示すために、変更を加えていない C++の本物のコード (疑似コードではない) を示しながら、SimAlpha の実装内部の説明を試みる。まず、メイン関数の説明から始め、そこで生成されるオブジェクト chip のコンストラクタが、更に 7 つのオブジェクトを生成することを見る。それらオブジェクト間の関係を見た後に、重要な役割を果たすクラス instruction の定義とコードを説明する。

```
int main(int argc, char **argv){
  if(argc==1) usage();
  char *p    = argv[argc-1]; /* program name */
  char **opt = argv;         /* options      */

  simple_chip *chip =  new simple_chip(p, opt);
  while(chip->step());
  delete chip;

  return 0;
}
```

図 2 SimAlpha のメイン関数

図 2 に示すメイン関数では simple_chip 型のオブジェクト chip を生成する。1 つの命令を実行するメンバ関数 step は、実行する命令がなくなった時点 (アプリケーションのシミュレーションが終了した時点) で値 0 を返す。while ループにより、関数 step が値 0 を返すまで繰り返すことでシミュレーションを進めていく。実行が終わるとオブジェクト chip を解放し、その際に呼ばれるデストラクタがシミュレーション結果を表示する。

クラス simple_chip のコンストラクタを図 3 に示す。

☆ SimAlpha の実行イメージファイルは SimpleScalar のローダを利用して作成できる。また、ハンドアセンブル等により命令列を作成してステップ実行することで、短い命令列の挙動を容易に把握できる。

☆☆ 2 つのシミュレータのアーキテクチャステートの値を一致させるために、SimAlpha の動作を、SimpleScalar の望ましくない挙動と一致させた箇所が 3 つ存在する。また、一部のシステムコールの実装において SimpleScalar のコードを変更した。

```
simple_chip::simple_chip(char *prog, char **opt){
  sc  = new system_config(prog, opt);
  e   = new evaluation_result;
  as  = new architecture_state(sc, e);
  mem = new memory_system(sc, e);
  deb = new debug(as, mem, sc, e);
  sys = new system_manager(as, mem, sc, e);
  p   = new instruction(as, mem, sys, sc, e);
}
```

図 3 simple_chip のコンストラクタ

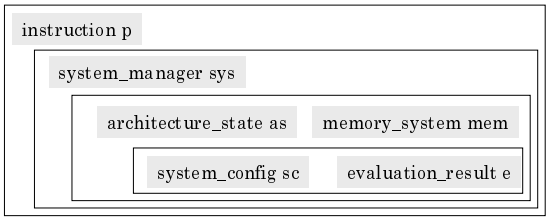


図 4 simple_chip で作られる 6 つのオブジェクトの参照関係

```
class architecture_state{
 public:
  data_t pc;    /* program counter      */
  data_t r[32]; /* general purpose regs */
  data_t f[32]; /* floating point  regs */
  architecture_state(system_config *,
                     evaluation_result *);
};
```

図 5 アーキテクチャステートの定義

simple_chip 型のオブジェクトの生成時に、7 種類のオブジェクトが生成される。デバッグのために利用する deb を省略した 6 つのオブジェクトの参照関係を図 4 に示す。system_config 型のオブジェクト sc には、シミュレーションの対象となるシステム設定の情報などを格納する。これらの値はシミュレーションの開始前に定義され、原則として、シミュレーション中に値が変わることはない。evaluation_result 型のオブジェクト e には評価中のデータを保存する。変数の値はシミュレーションの実行中に更新されるが、これらの値がシミュレーションの実行に影響を与えることはない。オブジェクト sc と e は、その他の全てのオブジェクトから参照される。プログラムカウンタ、整数レジスタ、浮動小数点レジスタから成るクラス architecture_state の定義を図 5 に示す。architecture_state 型のオブジェクト as と、ロードストア命令などにより参照されるメモリシステムを実現する memory_system 型のオブジェクト mem がシミュレーション実行における計算結果などの状態を保持する。これら 2 つのオブジェクトは、instruction 型のオブジェクト p と、システムコールなどを処理する system_manager 型のオブジェクト sys によって更新される。本稿ではオブジェクト sys の説明は省略する。

ステップ実行をおこなうクラス simple_chip のメンバ関数 step のコードを図 6 に示す。instruction 型のオブジェクト p に対して、パイプラインステージに対応する 7 つの関数と WriteBack の 8 つの関数を順番に呼び出すことで 1 つの命令を実行する。SimAlpha Version 1.0 では機能レベルシミュレータの能力しか提供しないが、コードの可読性と拡張性を考慮して、Alpha21264[3] の命令パイプラインを参考にしながら命令の動作を 8 つのステージに分割して記述した。以降、クラス instruction の定義と、8 つのステージに対応する関数のコードを説明する。ただし、関数 Rename では処理をおこなっていないので、この関数の説明は省略する。

```
int simple_chip::step(){
  p->Fetch(&as->pc);    /* pipeline stage 0 */
  p->Slot();            /* pipeline stage 1 */
  p->Rename();          /* pipeline stage 2 */
  p->Issue();           /* pipeline stage 3 */
  p->RegisterRead();    /* pipeline stage 4 */
  p->Execute(&as->pc);  /* pipeline stage 5 */
  p->Memory();          /* pipeline stage 6 */
  p->WriteBack();

  /* split a conditional move,see README.txt */
  execute_cmovb(p, as);

  e->retired_inst++;
  house_keeper(sys, sc, e, deb);

  return sys->running;
}
```

図 6 関数 step のコード

クラス instruction の定義を図 8 に示す。パイプラインステージに対応する関数が呼びだされ、命令の処理が進むに従って、オブジェクト内のプライベート変数の値が計算されていく。

```
int instruction::Fetch(data_t *pc){
  mem->ld_inst(pc, &ir);
  Npc.init(pc->ld() + 4);
  return 0;
}
```

図 7 命令フェッチステージのコード

命令フェッチステージのコードを図 7 に示す。プログラムカウンタ PC の示すアドレスから 4 バイトの命令をロードして、変数 ir に格納する。また、PC に値 4 を加えて得られる次命令のアドレスを Npc に格納する。

スロットステージのコードを図 9 示す。先のステージでフェッチした命令コード ir を用いて、幾つかの変数の値をデコードする。図 9 の様にデコードした値を変数に代入する替わりに、マクロを用いてコードを記述することができる。後者の方法を用いることで、シミュレーション時間の向上を期待できるが、コードの

```
class instruction{
  evaluation_result  *e;
  architecture_state *as;
  system_manager     *sys;
  memory_system      *mem;

  INST_TYPE ir; /* 32bit instruction code  */
  int Op;       /* Opcode field            */
  int RA;       /* Ra field of the inst    */
  int RB;       /* Rb field of the inst    */
  int RC;       /* Rc field of the inst    */
  int ST;       /* store inst ?            */
  int LD;       /* load inst ?             */
  int LA;       /* load address inst ?     */
  int BR;       /* branch inst ?           */
  int Ai;       /* Rav is immediate ?      */
  int Bi;       /* Rbv is immediate ?      */
  int Af;       /* Rav from floating-reg ? */
  int Bf;       /* Rbv from floating-reg ? */
  int WF;       /* Write to the f-reg ?    */
  int WB;       /* Writeback reg index     */
  data_t  Npc;  /* Update PC or PC + 4     */
  data_t  Imm;  /* immediate               */
  data_t  Adr;  /* load & store address    */
  data_t  Rav;  /* Ra                      */
  data_t  Rbv;  /* Rb                      */
  data_t  Rcv;  /* Rc                      */
 public:
  int Fetch(data_t *);
  int Fetch(data_t *, INST_TYPE);
  int Slot();
  int Rename();
  int Issue();
  int RegisterRead();
  int Execute(data_t *);
  int Memory();
  int WriteBack();

  INST_TYPE get_ir();
  int data_ld(data_t *, data_t *);
  int data_st(data_t *, data_t *);
  instruction(architecture_state *,
              memory_system *,
              system_manager *,
              system_config *,
              evaluation_result *);
};
```

図 8 クラス instruction の定義

可読性を考慮して変数に代入する方法を選択した。また、このような記述方法は Verilog-HDL の記述と近く、C++と Verilog-HDL との間の部分的なコードの再利用を可能とする。

発行ステージのコードを図 10 示す。ここでは、命令の種類に応じて必要となるイミディエート Imm のデータを作成する。これらのデータや、アーキテクチャステート、メモリの内容は図 15 に示す data_t 型のオブジェクトとして定義されている。

レジスタリードステージのコードを図 11 を示す。

```
int instruction::Slot(){
  Op  = (ir>>26) &  0x3F;
  RA  = (ir>>21) &  0x1F;
  RB  = (ir>>16) &  0x1F;
  RC  = (ir    ) &  0x1F;
  WF  = ((Op&MSK2)==0x14 || (Op&MSK2)==0x20);
  LA  = (Op==0x08 || Op==0x09);
  LD  = (Op==0x0a || Op==0x0b || Op==0x0c ||
         (Op&MSK2)==0x20 || (Op&MSK2)==0x28);
  ST  = (Op==0x0d || Op==0x0e || Op==0x0f ||
         (Op&MSK2)==0x24 || (Op&MSK2)==0x2c);
  BR  = ((Op&MSK4)==0x30);
  WB  = (LD || (Op&MSK2)==0x08 || Op==0x1a ||
         Op==0x30 || Op==0x34) ? RA :
        ((Op&MSK3)==0x10 || Op==0x1c) ? RC : 31;
  Af  = (Op==0x15 || Op==0x16 || Op==0x17 ||
         Op==0x1c ||
         (Op&MSK2)==0x24 || (Op&MSK3)==0x30);
  Bf  = ((Op&MSK2)==0x14);
  Ai = (Op==0x08 || Op==0x09 || LD);
  Bi = (BR || (Op&MSK2)==0x10 && (ir & BIT12));

  /** For the CMOV Split Code (CMOV1) **/
  if(cmov_ir_create(ir)){
    RB = RC;
    Bi = 0;
  }
  return 0;
}
```

図 9 スロットステージのコード

```
int instruction::Issue(){
  DATA_TYPE Lit, D16, D21, tmp, d21e, d16e;
  d21e = ((ir & MASK21) | EXTND21) << 2;
  d16e  = (ir & MASK16) | EXTND16;

  Lit = (ir>>13) & 0xFF;
  D21 = (ir & BIT20) ? d21e : (ir&MASK21)<<2;
  D16 = (ir & BIT15) ? d16e : (ir&MASK16);
  if(Op==0x09) D16 = (D16 << 16);

  tmp = (LA||LD||ST) ? D16 : (BR) ? D21 : Lit;
  Imm.init(tmp);
  return 0;
}
```

図 10 発行ステージのコード

浮動小数点レジスタあるいは整数レジスタからデータを読みだし、Rav と Rbv に格納する。

実行ステージのコードを図 12 に示す。実行ステージでは、3 つのデータを更新する。算術命令等は Rav と Rbv を入力として Rcv の値を計算する。ロードストア命令はメモリ参照アドレス Adr を計算する。分岐命令は分岐先アドレス Tpc を計算する。

メモリアクセスステージのコードを図 13 に示す。ストア命令の場合には Rav の値をメモリにストアする。ロード命令の場合にはロードした値を Rcv に保存する。

```
int instruction::RegisterRead(){
  Rav = Ai ? Imm : Af ? as->f[RA] : as->r[RA];
  Rbv = Bi ? Imm : Bf ? as->f[RB] : as->r[RB];
  return 0;
}
```

図 11 レジスタリードステージのコード

```
int instruction::Execute(data_t *Tpc){
  /*** Update Rcv ***/
  if(BR || Op==OP_JSR){
    Rcv=Npc;
  }
  else if(!LD){
    ALU(ir, &Rav, &Rbv, &Rcv);
  }

  /*** Update Adr ***/
  Adr.init(0);
  if(LD || ST){
    ALU(ir, &Imm, &Rbv, &Adr);
  }

  /*** Update Tpc ***/
  *Tpc = Npc;
  if(Op==OP_JSR){
    *Tpc = Rbv;
    Tpc->st(Tpc->ld() & ~3ull);
  }
  if(BR){
    BRU(ir, &Rav, &Rbv, &Npc, Tpc);
  }
  return 0;
}
```

図 12 実行ステージのコード

```
int instruction::Memory(){
  if(ST) data_st(&Adr, &Rav);
  if(LD) data_ld(&Adr, &Rcv);
  return 0;
}
```

図 13 メモリアクセスステージのコード

```
int instruction::WriteBack(){
  if(Op==OP_PAL){
    sys->execute_pal(this);
  }

  if(!WF && WB!=31) as->r[WB] = Rcv;
  if( WF && WB!=31) as->f[WB] = Rcv;
  return 0;
}
```

図 14 ライトバックのコード

ライトバックのコードを図 14 示す。実行している命令が PAL コードである場合には、execute_pal 関数を呼びだす。結果を生成する通常の命令の場合は、データ Rcv をレジスタファイルに格納して当該命令の実行を完了する。

4. SimAlpha の利用例

本章では、SimAlpha の拡張性を示す例として、制御依存関係や資源競合を解消し、データ依存関係のみを制約として得られる理想的な命令レベル並列性を測定するように SimAlpha を拡張する。また、これを用いて測定した結果を報告する。測定する値は文献 2) の oracle IPC と同様の意味を持つ。ベンチマークには SPEC CINT95 と CINT2000 を利用した。

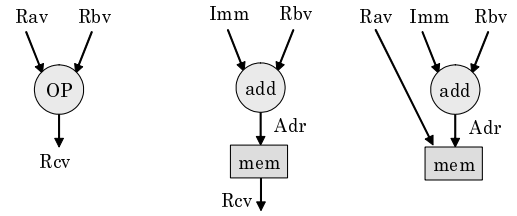
SimAlpha で扱うデータは、単なる unsigned long long 型の値ではなく、data_t 型のオブジェクトとして定義される。理想的な命令レベル並列性を測定するために、計算された全てのデータにデータフローグラフの高さに相当する値 (これをランクと呼ぶことにする) を格納するようにクラス data_t を変更する。物理メモリはクラス data_t のオブジェクトの配列として構築されている。ロードストア命令は 1 バイトから 8 バイトの粒度でメモリを参照するので、メモリ内のデータをオブジェクトとして表現する際の粒度には幾つかの選択肢があるが、8 バイト単位の整列したデータを一つのオブジェクトとした。理想的な命令レベル並列性を測定するために変更されたクラス data_t の定義を図 15 に示す。

```
class data_t{
  uint64_t value;
 public:
  int cmov;
  uint32_t rank;  /* This line is inserted. */
  uint64_t ld();
  int   st(uint64_t);
  int init(uint64_t);
};
```

図 15 データオブジェクト data_t の定義

ランクの計算方法を図 16 に示す。算術論理命令は、ALU によりデータが計算される時に、2 つの入力オペランド Rav と Rbv のランクの最大値に演算レイテンシを加えた値を計算結果 Rcv のランクとして格納する。ロード命令では、Rbv のランクにアドレス計算のためのレイテンシとメモリ参照レイテンシを加えることでランクを計算する。ストア命令は、メモリに書き込むデータ Rav と、アドレス計算により得られるランクとの最大値をデータのランクとしてメモリに格納する。

シミュレーション中には、個々のデータが持つランクとは別に、全データのランクの最大値を更新する。シミュレーションが終了した時点のランクの最大値がデータフローグラフの高さを表すので、この値と実行命令数から理想的な並列性を計算できる。ただし、データはシステムコールを越えてスケジューリングで



(a) rank(Rcv) = max(rank(Rav), rank(Rbv)) + latency(OP)
(b) rank(Rcv) = rank(Rbv) + latency(add) + latency(mem)
(c) rank = max(rank(Rav), rank(Rbv) + latency(add))

図 16 命令タイプ毎のランクの計算方法

きないという制約を加えた。以下の評価において、ランクを計算する際の演算レイテンシとメモリ参照レイテンシを 1 サイクルとした。CINT2000 の 9 つのベンチマークに関しては University of Minnesota が提供している縮小入力セットを利用した。それ以外のベンチマークにおいては、シミュレーションの命令数を抑えるように入力パラメタを調整した。SPEC CINT95 のバイナリは DEC C コンパイラ、最適化オプション O4 を用いて生成した。SPEC CINT2000 のバイナリは、SimpleScalar のサイトからダウンロードしたものを利用した。

理想的な並列性を測定するために SimAlpha に変更を加えた。この変更の多くの部分はデータが生成される際のランクの計算であり、追加した全てのコードはわずか 26 行だった。変更点の詳細は省略するが、変更前後のソースコードは本稿の最後に示す URL からダウンロードできる。

測定結果を表 1 に示す。変更後のクラス data_t の定義に示した様に、ランクを保持するための変数の追加によりシミュレーションのためのメモリ量が増加する。また、ランクを計算するための処理が追加されているにもかかわらず、シミュレーション時間に関して深刻な増加はみられなかった。変更後のシミュレーション速度も、ほぼ 1.1MIPS である。理想的な IPC の測定結果を見ると、124.m88ksim と 253.perlbmk で低い値を示したが、それ以外のベンチマークでは数十というレベルから多いもので 186.crafty の 108 という高い並列性が内在していることを確認できた。

5. 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利用を目的として、プロセッサシミュレータ SimAlpha Version 1.0 を構築した。本稿では、コードの高い可読性を示すために、実際の C++のコードを示しながら SimAlpha のソフトウェアアーキテクチャを明らかにした。また、SimAlpha の利用例として、理想的な命令レベル並列性を測定する方法を示し、

表 1 SimAlpha を用いて測定した実行命令数、シミュレーション速度、理想的な命令レベル並列性
Table 1 Instruction count, million instructions per second, and oracle instruction level parallelism measured with SimAlpha.

Program	instruction count	MIPS	IPC
099.go	138 million	1.12	64.2
124.m88ksim	127 million	1.12	10.5
126.gcc	150 million	1.12	41.8
129.compress	142 million	1.14	56.6
130.li	208 million	1.11	20.0
132.ijpeg	172 million	1.21	107.0
134.perl	153 million	1.10	43.3
147.vortex	184 million	1.07	32.0
164.gzip	596 million	1.19	16.9
175.vpr	17 million	1.00	25.1
176.gcc	551 million	1.10	47.1
181.mcf	188 million	1.12	53.1
186.crafty	4264 million	1.10	108.0
197.parser	611 million	1.10	30.9
252.eon	94 million	0.93	49.7
253.perlbmk	200 million	1.05	8.4
254.gap	1169 million	1.12	32.1
255.vortex	147 million	1.06	29.3
256.bzip2	1819 million	1.12	43.6
300.twolf	91 million	1.00	21.9

これを測定するための機能を数十行という少ない記述で追加できることを述べた。変更した SimAlpha を利用して SPEC CINT95 と CINT2000 の理想的な命令レベル並列性を測定し、その結果を報告した。

SimAlpha Version 1.0 と、理想的な命令レベル並列性を得るために変更を加えた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/SimAlpha/`

謝辞 SimAlpha の初期のバージョンをテストし、貴重なコメントやパッチを頂いた飯塚大介氏に深く感謝致します。

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) Monica S. Lam and Robert P. Wilson: Limits of Control Flow on Parallelism, *19th Annual International Symposium on Computer Architecture*, pp. 46–57 (1992).
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).