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

2005－ARC－164 (17)
2005／8／4

ワークロード最適化によるキャッシュシミュレータの高速化

中田 尚† 津邑公暁† 中島 浩†

高度なマイクロプロセッサの研究・開発や，それを組み込んだ機器のハードウェア・ソフトウェア協調設計においては，その機能・性能を検証するための cycle accurate なシミュレータが不可欠である．しかし，既存のシミュレータは一般に低速であり，開発の効率化の障害となっている．これに対して，スケジューリング計算の高速化や命令エミュレーションの高速化が提案され，効果を上げている．一方，これらの実行時間短縮により，キャッシュシミュレーションの実行時間の割合が相対的に大きくなり，その短縮がシミュレーションのさらなる高速化のための課題となっている．本論文では，個々のキャッシュに対して最適化されたシミュレータを生成することにより，キャッシュシミュレーションの高速化を図る．SPEC CPU95 ベンチマークを用いて評価を行った結果，SimpleScalar の `sim-cache` に対して，最大 14.1 倍，平均 8.3 倍のシミュレーション速度の向上が確認できた．

Fast Cache Simulation Using Workload Optimization

TAKASHI NAKADA,† TOMOAKI TSUMURA† and HIROSHI NAKASHIMA†

Microprocessor simulation is indispensable not only for hardware systems design but also for software development of co-designed embedded systems. In both design fields, cycle accurate (or clock level) simulation is required. However, existing simulators of out-of-order processors run programs thousands of times slower than actual hardware. Thus various techniques for the speedup of instruction scheduling and emulation are proposed and some of them achieve good performance. These improvements, however, reveal that the cache simulation becomes a bottleneck of the further improvement of the simulation in total. This paper proposes a speedup technique which generates an optimized simulator code for each cache. Our evaluation of its implementation shows the simulation speed of SPEC CPU95 benchmarks is improved by up to 14.1-fold and 8.3-fold in average from SimpleScalar's `sim-cache`.

1. はじめに

集積回路技術の進歩にともない，マイクロプロセッサの構造は高度化・複雑化している．このような高度なマイクロプロセッサやそれを組み込んだ機器の研究・開発にはその機能や性能をあらかじめ検証するためのシミュレータが不可欠である．しかし，現状では最も簡単なユニプロセッサのアーキテクチャ・シミュレータでさえ実時間性能比（SD: slowdown）は 1000～10000 と低速であり，研究・開発の効率化の大きな障害になっている．そこで，実行時間の多くを占める命令スケジューリング計算の高速化手法がいくつか提案されており，効果をあげているものも少なくない．

一方これらの高速化の対象であるシミュレータには，命令スケジューリング計算以外の要素である命令エミュレーションやキャッシュシミュレーションも含まれているが，実行時間に占める割合がに小さいためそれらの速度は重要視されていなかった．たとえば，SimpleScalar[1] では命令スケジューラ（`sim-outorder`）の SD が 1000～3000 であるのに対し，命令エミュレータ（`sim-fast`）の SD は 100～300，キャッシュシミュレータ（`sim-cache`）の SD は 300～1000 となっている．ここで，命令スケジューラには命令エミュレーションとキャッシュシミュレーションが含まれ，キャッシュシミュレーションには命令エミュレーションが含まれている．これらのことから，シミュレーション時間全体の約 10% を命令エミュレーション，約 20% をキャッシュシミュレーションが占めていると考えることができる．

以上のことから，命令スケジューリング計算のみを高速化しても，命令エミュレーションやキャッシュシミュレーションが低速である限りシミュレーション全体の高速化には限界があるということがわかる．例えば，命令エミュレータの SD が 100，キャッシュシミュレータの SD が 200，命令スケジューラの SD が 700，つまり全体では SD=1000 のシミュレータがあるとする．このシミュレータの命令スケジューラを 100 倍高速化しても，全体の SD は 307 となるに過ぎず，速度は約 3.25 倍にしかならない．そこでたとえ

† 豊橋技術科学大学
Toyohashi University of Technology

ば，命令エミュレータを 20 倍に，またキャッシュシミュレータを 10 倍にそれぞれ高速化できれば，全体の SD は 32 となりおよそ 30 倍の高速化が達成できることとなる．

そこで本論文では，命令エミュレーションの高速化のために我々が考案したワークロード最適化[2] を利用した，キャッシュシミュレーションの高速化手法を提案する．これはキャッシュ構成をコンパイル時に決定することにより，個々の構成に対して最適化されたキャッシュシミュレータを生成することにより，高速化を達成するものである．

提案手法を sim-cache に適用し，その効果を SPEC CPU95 ベンチマークを用いて測定した．さらに，この手法を高速マイクロプロセッサシミュレータ Burst-Scalar[3] に適用することにより，その効果を測定した．

2. ワークロード最適化

本章ではワークロード最適化[2] の具体的な手法および手順について述べる．

2.1 概　　要

最も単純な命令エミュレータでは様々なワークロードプログラムに対応できるように，1 命令ずつ命令を fetch，decode しながら実行する．この方法は実装が容易であるが，実行速度が遅いという問題がある．

そこで，ワークロードプログラムに最適化したシミュレータのソースコードを自動生成し，それをコンパイルすることにより最適化シミュレータを得る．つまり，機械語レベルの最適化はコンパイラにすべて任せることにする．これにより，理論上オリジナルのシミュレータが対応する全てのホストアーキテクチャで，高速な命令エミュレーションが実現可能である．

本手法の基本的な流れは以下のようになる．

(1) ワークロードを静的に解析して基本ブロックに分割する．
(2) 最適化シミュレータのソースコードを，基本ブロックごとの関数として生成する．
(3) 生成されたソースコードを一般のコンパイラでコンパイルして最適化シミュレータの実行コードを得る．
(4) 生成された最適化シミュレータでワークロードを実行する．

次に，本手法で最も重要な (2) のソースコード生成について詳しく説明する．

2.2 シミュレータコードの生成

最も基本的な命令エミュレータでは以下のように命令を実行する．

(1) 次の PC を決定
(2) 命令を 1 つ fetch
(3) 命令の種類を判断
(4) レジスタ番号や即値などを decode

```
while(1){
  PC = NextPC;                                    ... (1)
  NextPC = PC + 8;
  inst = fetch(PC);                               ... (2)
  switch(inst.op) {                               ... (3)
  case ADD:
    regs[inst.rd] =
      regs[inst.rs] + regs[inst.rt];              ... (4) (5)
    break;
  case ADDI:
    regs[inst.rd] =
      regs[inst.rs] + inst.imm;                   ... (4) (5)
    break;
    :
  }
}
```

図 1　基本的な命令エミュレータ

```
0x00001000:
  add  r5, r3, r2    ;r5 = r3 + r2
  add  r6, r5, r4    ;r6 = r5 + r4
  addi r7, r6, $10   ;r7 = r6 + 10
  j 0x00002000       ;PC = 0x00002000
```

図 2　簡単なソースコード

```
BB_0x00001000(){
  regs[5] = regs[3] + regs[2];
  regs[6] = regs[5] + regs[4];
  regs[7] = regs[6] + 10;
  PC = 0x00002000;
  return;
}
```

図 3　最適化された命令エミュレータ

(5) 命令を実行
(6) (1) に戻る

単純化した例を**図 1** に示す．ここで regs[N] は N 番目のレジスタ，inst.op は命令の種類，inst.rs などはソースレジスタなどの番号，inst.imm は即値を表す．このように，while を 1 周するたびに命令が 1 つずつ解釈，実行される．

次に，基本的な命令エミュレータを利用し，ワークロードに最適化されたシミュレータのソースコードを生成する方法について述べる．

ソースコードは基本ブロックごとに 1 つの関数として生成する．基本ブロック内の命令を 1 命令ずつ fetch，decode し，その命令に対応するソースコードを出力する．このとき，レジスタ番号や即値はできる限り定数に置き換える．

図 2 は 0x00001000 番地から始まる 4 命令で構成された基本ブロックである．この基本ブロックに対応するシミュレータソースコードは，**図 3** のようになる．

この図から，fetch と decode がすでに完了し，実行すべき命令の種類が確定し，レジスタの番号，即値が

```
while(1){
  if(table[PC]!=NULL){
    table[PC]();
  }else{
    inst=fetch(PC);
    :
} }
```

図 4　最適化されたメインループ

定数に置き換わっていることがわかる．

最後に，このソースコードがコンパイルにより最適化され，ワークロード最適化シミュレータが生成される．

また，メインループは図 4 のようになる．ここで，table は PC から最適化されたシミュレータ関数のポインタを取得するための配列である．分岐先アドレスが予測できなかった場合，当該基本ブロックに対応する関数が存在しないため，配列には NULL が格納されている．メインループは NULL を検出した場合，1 命令ずつシミュレーションを行う．

3. キャッシュシミュレーションと最適化

本章では，キャッシュシミュレーションとその最適化の具体的な手法および手順について述べる．なお以下では，キャッシュパラメータをラインサイズ 2^L，セット数 2^S，ウェイ数 W とする．また置換アルゴリズムは LRU を仮定するが，他の方法に対しても原理的に適用可能である．

3.1 キャッシュシミュレーション

基本的なキャッシュシミュレーションの流れを図 5 を用いて説明する．

まず，アクセスアドレスとアドレス幅が決定すると，これらの整合性（alignment）を確認する（b）．つぎに，アクセスアドレスとキャッシュパラメータからセット番号を求める（c）．セット番号は，アクセスアドレスに対して L ビットのシフト演算と，下位 S ビットを抽出するマスク演算によって求められる．

そして，タグを切り出し（d），セットの中に保存されているタグと比較し（e）（f），いずれかのウェイと一致した場合にはヒット（g），すべてのウェイと比較しても一致しなかった場合にはミス（h）となる．

ヒットの場合には LRU 情報を更新する必要がある．ミスの場合にはラインの置換と LRU 情報の更新を行い，さらに下位のキャッシュまたはメモリへアクセスする必要がある．また，必要に応じてヒット／ミス回数のような統計情報の収集も行う．

3.2 パラメータの固定

キャッシュのパラメータを固定することによるキャッシュシミュレーションの高速化について述べる．

図 5 に対して最適化をおこなった図 6 を用いて説明

```
#define set_shift (L)
#define set_mask  ((1<<S)-1)
#define tag_shift (S+L)
cache_access(addr){                          ··· (a)
  check_alignment(addr);                     ··· (b)
  set=(addr >> set_shift) & set_mask;        ··· (c)
  tag= addr >> tag_shift;                    ··· (d)
  for(way=0;way<W;way++){                    ··· (e)
    if(tag==cache[set][way].tag){            ··· (f)
      hit();                                 ··· (g)
      return;
    }
  }
  miss();                                    ··· (h)
  return;
}
```

図 5　基本的なキャッシュシミュレータ

```
cache_access(addr){                          ··· (a)
  check_alignment(addr)                      ··· (b)
  set=(addr >> 5) & 127;                     ··· (g')
  tag= addr >> 12;                           ··· (d')
  if(tag==cache[set].ru_tag){                ··· (f')
    /* hit();  do nothing */                 ··· (g')
    return;
  }
  for(way=0;way<4;way++){                    ··· (e')
    if(tag==cache[set][way].tag){            ··· (f)
      hit();                                 ··· (g)
      return;
    }
  }
  miss();                                    ··· (h)
  return;
}
```

図 6　最適化キャッシュシミュレーション

する．この例では，ラインサイズを $2^5 = 32$ バイト，セット数を $2^7 = 128$，ウェイ数を 4 とした．図中の（x′）は図 5 中の（x）に対して最適化をおこなったことを示す．

実行時にキャッシュパラメータを変更可能にする場合には，シミュレータはこれらのパラメータを変数として扱わなければならない．しかし，これらのキャッシュパラメータをコンパイル時に確定することができれば，キャッシュシミュレーションを簡略化することができ，高速化が可能となる．たとえば，セット番号は定数のシフトと定数のマスクによって求めることができる（c′）（d′）．また，このときウェイ数が少なければループを展開することによる高速化が可能となる．本論文ではループ回数の定数化のみを行い，ループを展開するかどうかはコンパイラの最適化に任せることとした（e′）．

なお対象プロセッサが複数のキャッシュを持つ場合，各々のキャッシュについて固有のシミュレーション関

数を用意する．

3.3 RU 最適化

ウェイが複数存在する場合，ヒットする確率が最も高いウェイは直前にアクセスされたウェイ（以下 RU ウェイ）と考えられる．この特徴を利用してさらなる最適化を行う．

まず，すべてのセットに対して最新のタグを保存しておく．このタグが更新されるのは RU ウェイ以外にヒットした場合やミスが発生した場合である．これらの場合にはラインの入替えや LRU 情報の更新などの処理が必要であり，最新のタグを保存するコストは十分に許容できる．また，このタグは置換アルゴリズムが利用する情報とは独立に管理できるので，置換アルゴリズムの実装には影響を与えない．

このタグを利用することにより，RU ウェイに対するヒット（以下，RU ヒット）を高速に処理することができる（f′）．

キャッシュヒット時に行わなければならない処理には，alignment の確認，ヒット回数のカウント，LRU 情報の更新，Write 時の Dirty bit のセット，レイテンシの計算が考えられる．

ここで SimpleScalar の `sim-cache` のような命令レベルのキャッシュシミュレータに関して，命令 1 次キャッシュと命令 TLB の RU ヒットで必要な処理を考えると，以下のことがわかる．

- alignment 確認
  常に必要．
- ヒット回数
  命令レベルのシミュレーションであるので，実行命令数とキャッシュアクセス数は一致する．そこで，実行命令数とキャッシュミス回数の差がヒット回数となるのでヒット時に回数をカウントする必要はない．
- LRU 情報
  LRU 情報に変更はないので，更新は不要．
- Dirty bit
  フェッチで命令キャッシュに書き込まれることはないので不要．
- レイテンシ
  命令レベルのシミュレーションには不要．

これらのことより，RU ヒット時には alignment のチェックのみが必要であるが，これはすべてのキャッシュアクセスに対して常に必要であるので，RU ヒット時のみに必要な処理はないことがわかる（g′）．

3.4 ワークロード最適化キャッシュシミュレータ

ワークロード最適化とこれまでに述べたキャッシュ最適化を組み合わせることにより，高速なキャッシュシミュレーションを実現する．これらの組み合わせにより，特に命令キャッシュのシミュレーションを高速化することができる．**図 7** の例を用いて説明する．図中の（x*）は図 6 中の（x）に対して最適化をおこなったことを示す．

```
cache_access(0x1000){                                  ... (a*)
  /* check_alignment(0x1000); done */                  ... (b*)
  if(1==cache[0].ru_tag){                              ... (f'*)
    return;
  }
  for(way=0;way<4;way++){                              ... (e')
    if(1==cache[0][way].tag){                          ... (f*)
      hit();                                           ... (g)
      return;
    }
  }
  miss();                                              ... (h)
  return;
}
```

図 7 ワークロード最適化キャッシュシミュレータ

まず，ワークロード最適化を適用することにより，基本ブロック中の命令アドレスを決定することができる（a*）（b*）．これにより，命令キャッシュアクセスに対するセット番号を確定することができる（f′*）（f*）．

また，RU ヒット時には alignment 確認のみを行わなければならなかった．しかし，ワークロード最適化を適用することにより，この確認もコンパイル時に完了することができる（b*）．

さらに，キャッシュ状態を保存する領域を静的に確保しておけば，参照しなければならないキャッシュ状態のアドレスはコンパイル時に決定する．つまり，図 7 中の `cache[0].ru_tag` のアドレスがコンパイル時に決定する．

最後に，この cache_access 関数をインライン展開すると，x86 のホストでは RU ヒットの場合には cmp, jne の 2 命令でキャッシュシミュレーションが完了する．

4. 実　　装

提案手法を SimpleScalar Tool Set version 3.0[1] と BurstScalar[3] に適用した．これによって，広く利用されている SimpleScalar と同じバイナリをシミュレートすることができる．

4.1 SimpleScalar

SimpleScalar については，命令エミュレータにキャッシュシミュレータが加わった `sim-cache` に提案手法を適用した．

`sim-cache` では命令エミュレーションとキャッシュシミュレーションを行う．実行するすべての命令について命令キャッシュシミュレーションを行い，その命令がアクセスするすべてのデータについてデータキャッシュシミュレーションを行う．

命令エミュレータから直接キャッシュシミュレータを呼び出しているので，ワークロード最適化とキャッシュ

最適化の相乗効果が最大限期待できると考えられる.

4.2 BurstScalar

高速マイクロプロセッサシミュレータである BurstScalar への提案手法の適用方法について述べる.

4.2.1 概　　要

BurstScalar は SimpleScalar の命令スケジューラに計算再利用を適用した高速マイクロプロセッサシミュレータである.

BurstScalar は，命令エミュレーションを行うモジュールである予備実行および先行実行と，命令スケジューリング計算を高速に行うモジュールである詳細実行および高速実行から構成されている.

予備実行では論理的な命令エミュレーションを高速に行う．一方，先行実行では分岐予測ミス時のミスパスを含めた命令エミュレーション行う．これらの命令エミュレータは，命令スケジューラに命令トレースとロードストアアドレスを供給する.

キャッシュシミュレータは，命令スケジューラである詳細実行と高速実行の部分から呼び出される.

4.2.2 提案手法の実装

BurstScalar に提案手法を適用することを考える．まず，2 つの命令エミュレータの双方にワークロード最適化を適用し，キャッシュシミュレータにキャッシュ最適化を適用する方法が考えられる.

しかし，キャッシュシミュレータは詳細・高速実行から呼び出されるので，このままでは，ワークロード最適化とキャッシュ最適化の相乗効果を得ることができない.

ここで，予備実行と詳細・高速実行は完全に同じ命令列をシミュレートしていることに注目すると，これまで詳細・高速実行から呼び出されるキャッシュシミュレータで行っていた alignment の確認を，予備実行で行うことが可能である．予備実行にはワークロード最適化が適用されるので，alignment の確認はコンパイル時に完了する.

以上により，キャッシュアクセス時に alignment の確認をする必要が無くなり，高速化の効果が期待できる.

5. 評　　価

5.1 評価条件

最適化シミュレータの評価として，ワークロード最適化の効果を，SPEC CPU95 ベンチマークを用いて測定した．データセットには train を用いた.

評価には Opteron（Dual 2.0GHz, Memory 8GB, Linux 2.4.21）を用いた．ターゲットアーキテクチャは PISA とした．コンパイラは gcc version 3.3.2 を用い，最適化オプションは '-O2' とした.

`sim-cache` のキャッシュの構成を**表 1** に，BurstScalar の評価モデルの構成を**表 2** に示す．キャッシュの構成は `sim-cache` と同じとした.

表 1 キャッシュの構成

キャッシュ	L1 命令	16KB/32B ライン/1-way
	L1 データ	16KB/32B ライン/4-way
	L2 統合	256KB/64B ライン/4-way
TLB	命令	16 エントリ/4-way
	データ	32 エントリ/4-way

表 2 プロセッサの構成

命令発行幅		8
RUU		256
LSQ		128
メモリポート		4
機能ユニット	INT-ALU	8
	INT-MUL/DIV	3
	FP-ALU	8
	FP-MUL/DIV	3
分岐予測	予測方式	2bit カウンタ/2K エントリ
	BTB	512 エントリ/4-way
	RAS	8 エントリ

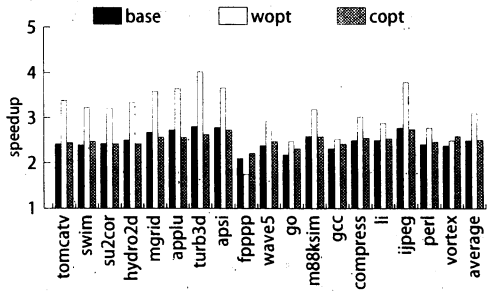

図 8 sim-cache の高速化率（その 1）

5.2 sim-cache

`sim-cache` では以下の高速化を適用した.

- 基本的な最適化のみ（base）
- base とワークロード最適化（wopt）
- base とキャッシュ最適化（copt）

ここで，基本的な最適化（base）は，文献[2] で示したメモリやキャッシュアクセスに関するコーディング改良など，本稿で提案する手法以外の最適化である.

図 8 は，上記の 3 手法の各々について，`sim-cache` に対する高速化率を示したものである．base では最大 2.80 倍（125.turb3d），平均 2.49 倍，wopt では最大 4.01 倍（125.turb3d），平均 3.10 倍の高速化が達成でき，copt では最大 2.74 倍（132.ijpeg），平均 2.51 倍の高速化が達成できた.

base と copt の結果の比較から copt の効果が非常に少ないことがわかる.

また，wopt を適用したときに 145.fpppp の高速化率が減少している．これは，145.fpppp の最内ループ

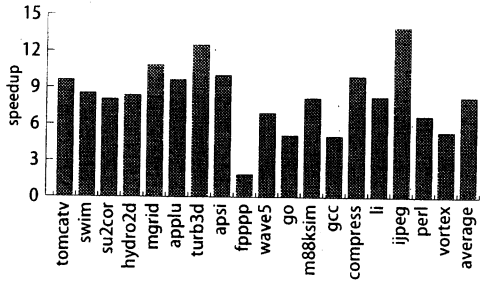

図 9 sim-cache の高速化率（その 2）

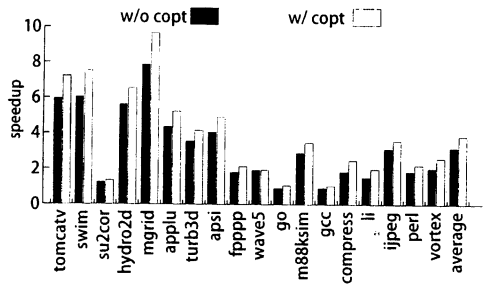

図 10 BurstScalar の高速化率

に対応した最適化シミュレータが大きく，命令キャッシュミスが増大したためと考えられる．実際，wopt では最内ループに対応した部分は 1.05MB であり，L2 キャッシュの容量を超えている．

次に，すべての最適化を適用した場合の高速化率を測定した．測定結果を**図 9** に示す．すべての最適化を適用することにより，最大 14.05 倍（132.ijpeg），平均 8.31 倍の高速化が達成できた．

この結果より，wopt と copt を個別に適用した場合と比較して，両者を同時に適用した場合に非常に大きな効果を得られていることがわかる．つまり，キャッシュ最適化はワークロード最適化と組み合わせることによって，その効果を発揮するといえる．

5.3 BurstScalar

5.3.1 高速化の効果

以下の条件で，SimpleScalar の `sim-outorder` に対する高速化率を測定した．

- キャッシュ最適化なし（w/o copt）
- キャッシュ最適化あり（w/ copt）

測定結果を**図 10** に示す．w/o copt では最大 7.87 倍（107.mgrid），平均 3.20 倍であった高速化率が，w/ copt では最大 9.67 倍（107.mgrid），平均 3.84 倍と，平均で 20%高速化率が向上した．

5.3.2 実行時間の内訳

BurstScalar に対する提案手法の効果を詳しく調べるために，実行時間の内訳を測定した．102.swim の結果を**表 3** に示す．ここで，キャッシュはキャッシュシミュレータの実行時間，その他はキャッシュシミュレータ以外の実行時間を示す．

表 3 BurstScalar の実行時間の内訳（102.swim）

	キャッシュ	その他
w/o copt	43.2	180.9
w/ copt	11.0	166.1

（単位：秒）

この結果より，キャッシュ最適化によりキャッシュシミュレータの実行時間がおよそ 1/4 に削減されており，提案手法による高速化の効果を確認できた．

6. まとめ

本論文では個々のワークロードに対して最適化されたシミュレータを生成するワークロード最適化と，個々のキャッシュ構成に最適化したキャッシュシミュレータを組み合わせることにより，高速なキャッシュシミュレーションを実現する手法を提案した．

提案手法を SimpleScalar のキャッシュシミュレータである `sim-cache` に適用し，その効果を SPEC CPU95 ベンチマークを用いて測定した．その結果，最大 14.1 倍，平均 8.3 倍の高速化を達成した．さらに，提案手法を高速マイクロプロセッサシミュレータ BurstScalar に適用した結果，適用前には最大 7.9 倍，平均 3.2 倍であった高速化率が，最大 9.7 倍，平均 3.8 倍に向上し，平均で 20%の高速化率の向上を達成した．

以上のことから，提案手法はキャッシュシミュレーションの高速化に非常に有効であるといえる．

謝辞 本研究は，(株) 半導体理工学研究センターとの共同研究「SpecC によるソフトウェア記述の性能検証システム」，文部科学省 21 世紀 COE プログラム「インテリジェントヒューマンセンシング」，および文部科学省科学研究費補助金（基盤研究 (B)，研究課題番号 17300015，「高度情報機器開発のための高性能並列シミュレーションシステム」）の支援によって行われた．

参考文献

1) Austin, T., Larson, E. and Ernst, D.: SimpleScalar: An Infrastructure for Computer System Modeling, *Computer*, Vol. 35, No. 2, pp. 59–67 (2002).
2) 中田尚, 津邑公暁, 中島浩: ワークロード最適化シミュレータの設計と実装, 先進的計算基盤システムシンポジウム SACSIS2005, pp. 329–338 (2005).
3) 中田尚, 中島浩: 高速マイクロプロセッサシミュレータ BurstScalar の設計と実装, 情報処理学会論文誌：コンピューティングシステム, Vol. 45, No. SIG 6(ACS6), pp. 54–65 (2004).