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

2007－ARC－172 (32)
2007－HPC－109 (32)
2007／3／2

時間軸分割並列マイクロプロセッサシミュレータの高速化と評価

矢野聖宗† 高崎透†
中田尚† 中島浩☆

集積回路技術の進歩に伴い，マイクロプロセッサの構造は高度化・複雑化している．高度なマイクロプロセッサの性能検証にはクロックレベルでのシミュレーションが不可欠であるが，現存するシミュレータは一般に低速であり，研究開発の大きな障害となっている．そこで我々は，シミュレーション過程を時間軸方向に分割，並列化することによるマイクロプロセッサのクロックレベルシミュレーションの高速化手法を提案している．並列シミュレーションは，分割点でのマシン状態を一致させること，もしくは分割された区間のシミュレーションの正当性をシミュレーション履歴によって検証することにより，精度を落とすことなく高速化を行う．本論文では，マシン状態を近似的にもとめる命令レベルのシミュレーション部分を高速化するために，ワークロード最適化シミュレーション技術を適用することを提案した．SPEC CPU95 を用いて評価した結果，SimpleScalar の sim-outorder に対して，最大 9.41 倍，平均 6.4 倍のシミュレーション速度の向上が確認できた．

A Speed-up Technique for Time-Division Parallel Microprocessor Simulator

Masahiro Yano,† Toru Takasaki,† Takashi Nakada†
and Hiroshi Nakashima☆

Microprocessor simulation is indispensable to design hardware systems. To estimate performance of highly sophisticated microprocessors, cycle accurate (or clock level) simulation is essential. However, existing simulators of out-of-order processors cost thousands times as long execution time as their targeting actual processors. The ultimate goal of our research is to develop a fast and accurate parallel simulator which is capable of microarchitectural modeling and system level simulation. We proposed a time-division parallel simulator in which each time interval is simulated in parallel with an approximated machine state at the beginning of the interval. The contribution of this paper is to speed up instruction set simulator for machine state approximation, by applying our workload specific simulation technique. Evaluation of its implementation shows the simulation speed of SPEC CPU95 benchmarks is improved by up to 9.41-fold and 6.4-fold on average from SimpleScalar's sim-outorder.

1. はじめに

集積回路技術の進歩に伴い，マイクロプロセッサの構造は高度化・複雑化している．近い将来，組み込み機器等にも高度なマイクロプロセッサが用いられるようになると予想される．高度なマイクロプロセッサや，それらを用いた組み込み機器についての研究開発には，その機能や性能を前もって検証するためにシミュレーションが不可欠である．一般に，マイクロプロセッサのシミュレーションでは，命令の論理的な挙動だけをシミュレートする場合にはその実時間性能比（slowdown:**SD**）は 10～100 であるが，命令の実行順序を入れ換えて実行する out-of-order 実行や，複数の命令を同時に実行するスーパースカラ方式等をクロックレベルでシミュレーションする場合には SD は 1000～10000 となる．SD が 10000 の場合，実機では 1 分で終了する動作をシミュレートするために，約 7 日を要することになり，シミュレータの低速さが研究開発の効率化の大きな障害になっている．

さて，マイクロプロセッサの動作をシミュレートしていく過程において，ある時点でのパイプライン，キャッシュ等の状態を考えてみると，それらの状態は，その時点以前のシミュレーション過程すべてに依存しているわけではないことが分かる．たとえば，パイプラインでは分岐予測ミスが発生するたびにその状態は空，または空に近い状態となり過去との依存関係をほぼ失う．またキャッシュについて言えば，特定のセットは

† 豊橋技術科学大学
Toyohashi University of Technology
☆ 現在，京都大学
Presently with Kyoto University

連想度に等しい数の異なるアドレスに対するアクセスがあれば，過去の状態に関わらず一定の状態となる．したがって，ある時点での状態や，その時点から別のある時点までの区間シミュレーションの結果は，要素と場合によっては，始めからシミュレートしなくても求められる．そのような場合には，区間ごとのシミュレーションを別々のノードで並列に行い，その結果を後で統合することでシミュレーション時間を大幅に短縮できる．

区間シミュレーション結果が正当なものであること，すなわち逐次的にシミュレートした場合と同じ結果が得られることを保証する必要ある．このためには，近似的に求めた各区間の初期状態が先行する区間の終了状態と一致するか否かを判定し，不一致であれば区間シミュレーションのやり直しを行わなければならない．

このような考え方に基づき我々は，マイクロプロセッサのシミュレーション過程を複数の区間に分割し，それらを別々のノードで実行することにより精度を全く落とさずに高速化を図る時間軸分割方式の並列シミュレーション方式を提案している[2]．この並列シミュレーションの高速化手法として，ミスパスを部分的に実行することにより，区間シミュレーションのやり直しを低減する方法がこれまでに提案されている．

時分割並列処理を効率的に行うには，論理シミュレーションが詳細シミュレーションに比べて十分に高速でなければならない．そこで，本論文では論理シミュレーションを高速化するために，我々の研究グループで開発したワークロード最適化シミュレーションの技術[3]を適用し，並列シミュレーションを高速化することを目的とする．

以下，2章で時間軸分割方式の並列シミュレーション方式について述べ，3章でワークロード最適化シミュレーションについて説明する．4章で評価を述べ，5章でまとめる．

2. 時分割並列シミュレータ

本章では，時分割並列シミュレータの概要を説明する．

時間軸分割による並列シミュレーションの概念図を**図1**に示す．図1では，マイクロプロセッサのシミュレーション過程を時間軸方向に4分割し，それぞれを別々のノードPC1~4で並列に実行する様子を表している．シミュレーション精度を保証するためには，それぞれの分割区間を正しくシミュレートしなければならないが，分割点でシミュレーション対象プロセッサの状態（**マシン状態**）が一致している場合には正しくシミュレートされる．

また，分割点 d_{12} でマシン状態が多少違っていても，その違いが分割区間 S_2 のシミュレーションに影響を及ぼさない場合もある．このような場合には，分割区間シミュレーションの履歴の一部を保存しておけば，並列実行後に分割点でのマシン状態の違いによる影響を検証できる．検証の結果，分割区間シミュレーションに影響があった場合には，その区間を並列実行後にやり直すことで，全体の正しいシミュレーション結果が求められる．

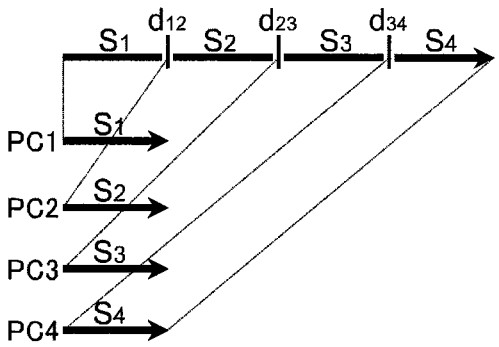

図1 時間軸分割による並列化

なお，以降の説明では，高速化の対象としているクロックレベルの詳細なマイクロプロセッサシミュレーションを**詳細シミュレーション**，命令の論理的な挙動のみ（命令レベル）のシミュレーションを**論理シミュレーション**と呼ぶことにする．

2.1 マシン状態とその一致

本論文では，マシン状態としてメモリ・レジスタ・プログラムカウンタ・パイプライン・キャッシュ・TLB・分岐予測器を想定する．これらの状態が分割点で一致していれば，並列シミュレーションは正しく行われる．

以下，マシン状態のそれぞれの要素について分割点での状態一致を図る手法について述べる．

2.1.1 論理シミュレーション

メモリ・レジスタ・プログラムカウンタについては，並列シミュレーションを開始する前に，論理シミュレーションを行うことにより，分割点での状態を求めればよい．各ノードは論理シミュレーションを分割点まで行い，引続き詳細シミュレーションを行う．

シミュレート対象が単一プロセッサの場合，ロードストアを含むすべての命令はタイミング情報を必要とせず，命令レベルでシミュレートできるので，論理シミュレーションによって分割点での完全な状態を求めることができる．また，論理シミュレーションはSDが小さく短時間で実行できるため，並列実行開始時刻におよぼす影響は少ない．なお論理シミュレーションでは，分割点でのキャッシュ・TLB・分岐予測器の状態を近似的に求めるために，これらの動作の命令レベルでのシミュレーション，すなわちout-of-order実行や投機実行を無視したシミュレーションも同時に行う．

2.1.2 重複実行

パイプラインについては，各ノードが一定区間，詳

細シミュレーションを重複して行うことで状態一致を図る．

パイプラインは，分岐予測ミスが発生するとフラッシュされ，分岐方向および分岐先アドレスを間違えて実行した命令が消去される．パイプラインが完全にフラッシュされる場合であれば，そのたびにパイプラインは空になる．分岐方向や分岐先を間違えて実行した命令のみが消去される場合においても，分岐予測ミスのたびにパイプラインは空に近い状態となるため，予測ミスが繰り返されることによって過去との依存関係の多くを失う．したがって分割点でパイプライン状態が異なっていても，分割点の前後の区間シミュレーションを一定区間，重複して実行し，その区間で分岐予測ミスが何回か発生すれば，パイプライン状態が一致すると予想される．

2.1.3 履歴を用いた正当性検証方法

キャッシュ・TLB・分岐予測器の分割点での状態は，前述のように論理シミュレーションによって近似的に求めるが，投機実行を無視しているためその失敗に起因する状態変化が反映されていない．この近似状態と真の状態の不一致は，L2 キャッシュのように状態数が大きいものでは少なからず存在し，また性能的に許容できる長さの重複実行では完全な一致を期待することはできない．

一方，これらの機構が保持する状態に部分的な差異があっても，その差異が参照されない，あるいは参照されても実行サイクルなどへの影響がないことがしばしばある．そこで，詳細シミュレーションの正当性検証は，キャッシュなどの状態の一致ではなく，詳細シミュレーション中で参照された状態に基づく動作の一致を確認することで行う．具体的には，キャッシュの連想度などにより定まる一定数の参照履歴を詳細シミュレーションの過程で保存し，これを区間終了後に前区間の末尾における正しい状態と照らし合わせ，アクセス遅延などの参照結果が全て等しければ正当であったと判定する．

2.2 並列シミュレーション

我々の研究では，精度を落とさず高速にマイクロプロセッサシミュレーションを行うことを目的としているので，分割区間シミュレーションが正しく行われなかった場合には，その区間のシミュレーションをやり直す必要がある．やり直しのためのシミュレーションは前区間を担当するノードが引続き詳細シミュレーションを行うことで対応する．

なお，分割区間シミュレーションが正しく行われた場合を**分割区間成功**，正しく行われなかった場合を**分割区間失敗**と呼ぶことにする．

2.2.1 並列シミュレーション方法（台数分割）

ここでは，分割数と PC 台数が等しい場合の台数分割シミュレーションについて述べる．

台数分割での並列シミュレーションの様子を**図 2** に

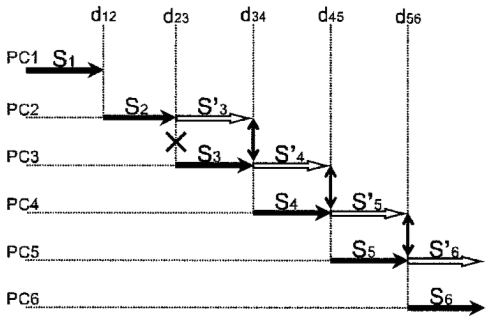

図 2 並列シミュレーション（台数分割）

示す．この方法では S_3 が失敗した場合に，PC2 が再実行を行うのと同時に，それ以降の PC が分割区間失敗のあるなしに関わらず，次の区間をシミュレートする．そして，S'_3:S'_4，S'_4:S'_5，S'_5:S'_6 の順に検証を行う．このような方法を用いると，失敗した分割区間シミュレーションの区間長が重複区間として活用され，再実行による各ノードの分割区間シミュレーションの成功率が飛躍的に増加する．例では S_1 - S_2 - S'_3 - S'_4 - S'_5 - S'_6 の区間を統合して，並列シミュレーションが終了している．

2.2.2 並列シミュレーション方法（多数分割）

ここでは PC 台数よりも多数に分割する多数分割シミュレーションについて述べる．多数分割では，分割区間長が短いため区間失敗時のやり直しのペナルティが小さくてすむ．一方，区間失敗の確率は必ずしも区間数に比例して増加するとは限らない．たとえば，ワークロードの特定の箇所で失敗する可能性が高い場合，区間長が多少変動しても失敗する区間数は同じであることが考えられる．そこでワークロードによっては，ペナルティが小さい多数分割が有利である可能性がある．

多数分割シミュレーションは，基本的に台数分割シミュレーションを繰り返す形で実行する．なお，始めの台数分割シミュレーションから順に第 1，2，3... フェーズと呼ぶことにする．

多数分割での並列シミュレーションの様子を**図 3** に示す．台数分割を単純に繰り返すと，フェーズの最後尾を担当するノードがやり直しに備えて次のフェーズに移行できない問題があるため，図 3 に示すように第 2 フェーズでは，PC6 は S_7 の 詳細シミュレーションを行うことにし，他 PC もそれに応じて詳細シミュレーション区間を後方にシフトさせる．このような方法をとることで，全ての PC が次フェーズのシミュレーションを開始することができる．また，S_7 については S_6 に引き続いて詳細シミュレーションが行われるため，分割区間シミュレーションの検証を行う必要がない．フェーズ数が増えた場合についても，同様の考え方を適用することでフェーズの境を意識することなく並列

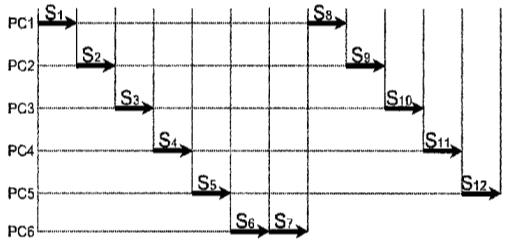

図 3 並列シミュレーション（多数分割）

シミュレーションを実行することができる．

3. 論理シミュレーション高速化技法

n 台の PC を用いた多数分割による時分割シミュレーションで全く失敗が起こらない場合，個々の PC は全体の約 $1/n$ の長さの詳細シミュレーションを行うだけでなく，約 $(n-1)/n$ の長さの論理シミュレーションを行う．したがって，時分割並列処理を効率的に行うには，論理シミュレーションが詳細シミュレーションに比べて十分に高速でなければならない．そこで，論理シミュレーションを高速化するために，我々の研究グループで開発したワークロード最適化シミュレーションの技術[3] を適用する．本章では，このワークロード最適化シミュレータの概要とその適用について説明する．

3.1 概　　要

ワークロード最適化シミュレータは，命令をシミュレートするプログラムコード自体をワークロードの命令ごとに自動生成し，シミュレーションに必要な処理の多くをコンパイル時に一度だけ実施することで，実行時には多くの処理が省略する．また連続して実行される命令間で全く同一の操作が繰り返される場合，コンパイラの最適化処理によって不要な繰り返しを除去できるという効果もある．さらに，ワークロードの命令からシミュレーションコードへの変換は，基本的なシミュレータが持つコードを埋め込む形で実施できるため，シミュレーションの対象マシンやシミュレーションを実行するホストマシンが多様であっても容易に対応することができる．

基本的な流れは以下のようになる．

- 基本ブロックに分割
  ワークロードを静的に解析し，分岐・合流のない連続命令列である基本ブロックに分割する．
- 最適化シミュレータのソースコードを生成
  基本ブロックごとに 1 つのシミュレーション関数を生成する．
- 最適化シミュレータのコンパイル
  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する．
- シミュレーションを実行

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

図 4 メインループ

コンパイルにより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．

3.1.1 基本ブロック分割

基本ブロックとは，連続したアドレスに配置された，分岐・合流のない命令列である．すなわち，プログラム中の分岐元はすべての基本ブロックの終端となり，分岐先はすべての基本ブロックの開始点となる．ここで，分岐元アドレスは分岐命令のアドレスであり明らかである．

3.1.2 シミュレータコードの生成

ソースコードは前節で分割された基本ブロックごとに 1 つの関数として生成される．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．これにより，最適化シミュレータでの実行時には既に decode されているため，高速な実行が可能となる．

すべての命令に対して同様の処理を施すことによって最適化シミュレータのソースコードが生成される．このソースコードでは，fetch と decode が既に完了し，実行すべき命令の種類が確定し，レジスタの番号，即値が定数に置き換わっていることになる．

一方**図 4** に示すメインループでは，関数ポインタを用いて該当する最適化シミュレータの関数を呼び出し，これを順次実行する．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，1 命令ずつ命令シミュレーションを行う処理に切り替える．

3.1.3 コンパイルと実行

生成されたシミュレータコードを一般のコンパイラを用いてコンパイラする．このとき，コンパイラに適切な最適化オプションを指示することにより，シミュレータコードがホストアーキテクチャに対して最適化される．

最適化されたシミュレータはワークロードバイナリを変更しない限り，同一のものを利用できる．つまり，シミュレーションパラメータを実行時引数，入力ファイルを変更しても再コンパイルの必要はない．

3.2 並列シミュレータへの適用

ワークロード最適化は，原理的には任意のレベルのプロセッサシミュレーションに適用できるが，一つの

```
while(1){
 if(table[PC]!=NULL){
  table[PC]();
  if(分岐予測ミス){
   <ミスパスの実行>
  }
 }else{
  inst=fetch(PC);
  ...
  if(分岐予測ミス){
   <ミスパスの実行>
  }
  ...
}}
```

図 5 メインループ

命令に要する処理が大きい場合には基本ブロック関数が複雑かつ大規模になる．一般に複雑な構造をした関数をコンパイラレベルで最適化することは困難であり，また大きな関数は命令キャッシュにも悪影響を及ぼす．したがって時分割シミュレータの全てにワークロード最適化を適用することは得策でなく，具体的には複雑な処理を行う詳細シミュレーションへの適用は困難である．一方論理シミュレーションは比較的処理が単純であり，これまでの研究[3] で SimpleScalar の sim-fast と sim-cache に適用されているなど，この適用事例に類似した性質を持つ．そこで，論理シミュレーションに対してのみ，ワークロード最適化を適用することとした．

なお，キャッシュシミュレーションは，最適化シミュレータから必要に応じてキャッシュシミュレータを呼び出す形式で実装されている．

一方，2.1.1 節で説明したように，論理シミュレーションでは，キャッシュに加え，分岐予測器のシミュレーションも行っている．したがって，分岐予測器のシミュレーションを同時に行うように機能を拡張しなければならない．

3.2.1 分岐予測器シミュレーション

分岐予測器シミュレーションについては，キャッシュシミュレーションと同様の手法を用いることとした．すなわち，最適化シミュレータから必要に応じて分岐予測器シミュレータを呼び出す．分岐予測器シミュレータは分岐命令を実行するタイミングで呼び出すこととした．

また，以前提案したミスパスの実行による高速化手法と併用するためには，最適化シミュレータの実行時にも予測ミスを起こす分岐命令を推定する必要がある．そこで，分岐予測ミス検出機構と同様の機構をシミュレータコード生成時に埋め込むように修正した．

3.2.2 ミスパスの実行との併用

3.1.1 節で述べたように，基本ブロックは分岐命令を境界として分割されている．つまり，すべての分岐命令は必ずシミュレーション関数の最後の命令となる．

表 1 プロセッサの構成

命令発行幅			4	
RUU エントリ数			16	
LSQ エントリ数			8	
メモリポート数			2	
機能ユニット数	INT-ALU		4	
	INT-MUL/DIV		1	
	FP-ALU		4	
	FP-MUL/DIV		1	
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	
メモリ	初期参照レイテンシ		18	
	バースト転送間隔		2	
TLB	命令		16 エントリ/4-way	
	データ		32 エントリ/4-way	
	ミスレイテンシ		30	
キャッシュ	容量	ラインサイズ	way 数	レイテンシ
L1 命令	16KB	32B	1-way	1
L1 データ	16KB	32B	4-way	1
L2 統合	256KB	64B	4-way	6

また前節で述べたように，分岐命令の実行時には予測ミスを起こすか否かが推定される．したがって，メインループにおいて 1 つの基本ブロック関数の実行が終了後に分岐予測ミスを起こすと推定された場合に，ミスパスの実行を行えばよい．メインループは**図 5** のようになる．最適化された基本ブロック関数からミスパスの実行関数を呼び出すのではなく，基本ブロック関数とミスパスの実行関数は分離されているため，関数の複雑化や肥大化を抑止することができる．

4. 評　　価

並列シミュレータは，SimpleScalar Tool Set Version3.0[1] を並列化することで実装されている．

シミュレーション過程の分割には，実行プログラムの命令数を利用した．一つの分割区間の命令数（分割区間長）l を任意で設定するため，全実行命令数を N とすると，分割数 D は $D = \lceil N/l \rceil$ となる．

シミュレーション対象プロセッサモデルは，SimpleScalar のデフォルトモデルとした．主なパラメータを**表 1** に示す．また，評価プログラムには SPEC CPU95 から fp の中で実行時間が短い swim, applu, fpppp を除いたものに加え，int の中で実行時間が長い vortex を用いた．なお，データセットはいずれも train とした．

評価には，OS:Linux 2.4.31，CPU:Intel Xeon /2.8GHz，Mem:1GB，N/W:Gigabit Ethernet のクラスタを用いた．また，並列実行には MPI を用いた．

4.1 高速化の効果

PC 台数を 16，重複区間を 100 万命令としたときのオリジナルの SimpleScalar に対する高速化率を**図 6** と**図 7** に示す．図は，分割区間長がそれぞれ 2000 万

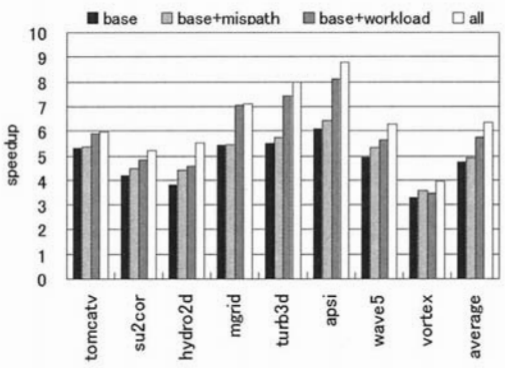

図 6　分割区間長 2000 万命令の高速化率

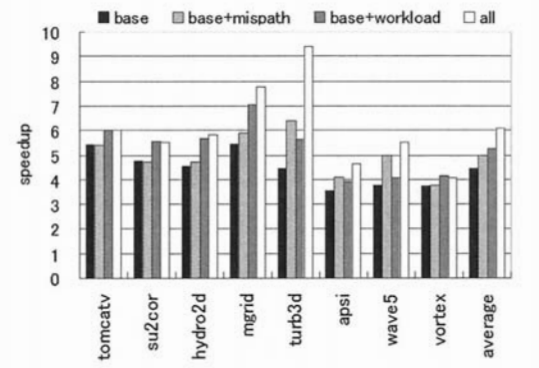

図 7　分割区間長 1 億命令の高速化率

命令・1 億命令のときのものである．また図には，比較対象として以下のシミュレート結果を示した．

- 高速化手法を適用しない (base)
- ミスパス実行による高速化 (base+mispath)
- ワークロード最適化による高速化 (base+workload)
- ミスパス実行とワークロード最適化による高速化 (all)

分割区間長 2000 万命令のとき base+workload では，最大 8.1 倍（apsi），平均 5.7 倍の高速化率を達成し，all では，最大 8.8 倍（apsi），平均 6.4 倍の高速化率を達成した．

また，分割区間長 1 億命令のとき base+workload では，最大 7.0 倍（mgrid），平均 5.3 倍の高速化率を達成し，all では，最大 9.4 倍（turb3d），平均 6.1 倍の高速化率を達成した．

図 6，図 7 の結果より，ワークロード最適化の効果が評価プログラムごとに大きくばらついていることがわかる．たとえば図 6 において，turb3d では 35.7%の性能向上があったのに対し，vortex では 5.2%の性能向上にとどまっている．これは，ワークロード最適化がワークロード自体に存在する命令レベル並列性を引き出す特性を持つことと，並列シミュレーションにおいて分割区間失敗が起こることによる．

また，base+mispath で高速化の効果が無かった評価プログラムにおいても，all では一定の性能向上を達成できていることが分かる．たとえば，分割区間長を 1 億命令のときの su2cor では，ワークロード最適化を適用することにより，約 16%の性能向上となった．

以上のことから，それぞれの高速化手法はともに評価プログラムごとに効果がばらつくが，両手法を組み合わせることによって，すべての評価プログラムで高速化を達成できることがわかる．

5. おわりに

本論文では，時間軸分割並列シミュレータに対する高速化手法として，論理シミュレーションにワークロード最適化シミュレーションの技術を適用する手法を提案した．

提案手法を既存の時間軸分割並列シミュレータに適用し，PC 台数を 16，重複区間を 100 万命令，分割区間長を 2000 万命令・1 億命令として SPEC CPU95 ベンチマークを用いて評価を行った．その結果，分割区間長 2000 万命令で最大 8.8 倍（apsi），平均 6.4 倍の高速化率を，分割区間長 1 億命令で最大 9.4 倍（turb3d），平均 6.1 倍の高速化率を達成した．

以上のことから，ワークロード最適化シミュレーションの技術を適用する手法は時間軸分割並列シミュレーションの高速化に有効であるといえる．

謝辞　本研究の一部は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」）による．

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) 高崎透，中田尚，津邑公暁，中島浩：時間軸分割並列化による高速マイクロプロセッサシミュレーション，情処論 ACS, 46-SIG12 (ACS11), pp. 84-97 (2005).
3) 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，情処論 ACS, 46-SIG12 (ACS11), pp. 98-109 (2005).