Vol. 46 No. SIG 12(ACS 11) 情報処理学会論文誌：コンピューティングシステム Aug. 2005

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

高崎 透[†,☆] 中田 尚[†]
津邑 公暁[†] 中島 浩[†]

マイクロプロセッサは集積回路技術の進歩にともない高度化，複雑化している．高度な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．本論文では，並列化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について述べる．並列化はシミュレーション過程を時間軸方向に分割することにより行い，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った．マシン状態の一致，および分割区間のシミュレーションの正当性を保つためには，並列実行に先行して分割点まで命令レベルシミュレーションを行い，さらに近接するノードでのクロックレベルシミュレーションを一定区間だけ重複実行した．SimpleScalar をベースとする並列シミュレータを実装し，SPEC CPU95 の評価を 8 ノード 8 分割で行ったところ，最大 2.7 倍，平均で 1.8 倍の高速化率が得られた．この値は，命令レベルシミュレーションに低速な sim-cache ベースのものを用いた場合のものであり，これを 5 倍高速なものに置換すれば，最大 5.2 倍，平均で 2.9 倍程度の高速化率が期待できる．

Fast Simulation of High Performance Processor with Time Division Parallelization

Toru Takasaki,[†,☆] Takashi Nakada,[†] Tomoaki Tsumura[†] and Hiroshi Nakashima[†]

Microprocessor simulation is indispensable for designing 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. This paper proposes a time division parallelization for microprocessor simulation. To make the out-of-order simulation for an interval correct, we perform *logical* in-order simulation for the preceding interval in advance. Moreover, two adjacent intervals are overlapped so that the machine state derived from the logical simulation matches that of the real out-of-order simulation. The correctness is verified by comparing pipeline states at the end of the overlapped interval, and by examining access trace of caches, TLBs and the return address stack for branch prediction at the end of the simulation interval. We implemented a SimpleScalar-based parallel simulator on a 8-node PC cluster and evaluated its performance with SPEC CPU95 to find it achieves up to 2.7-fold speedup and 1.8-fold in average. Since these values are obtained by the implementation with a slow sim-cache-based instruction level simulator, it is expected that they will be improved almost twice if we employ a reasonably well-tuned instruction level simulator being five times as fast as 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．近い将来，組み込み機器など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

† 豊橋技術科学大学
Toyohashi University of Technology
☆ 現在，村田機械株式会社
Presently with Murata Machinery, Ltd.

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，現在マイクロプロセッサに実装されている，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スカラ方式など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 1,000〜10,000 となる．SD が 10,000 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，キャッシュなど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いえば，特定のセットは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，過去の状態にかか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．

なお，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がある．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．したがって単純にノード数分の高速化を実現することは容易ではないと予想されるが，予備的に状態の一致確率を評価した結果[15]，たとえば 8 ノードで 4 倍程度の高速化は達成可能であろうという見通しを得た．

このような考え方に基づき本研究では，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まった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法を提案し，実装・評価した．

以下，2 章では関連研究を述べる．3 章で高速化手法を述べ，4 章でその高速化手法を用いた並列シミュレーションの設計について説明する．5 章で実装，6 章で評価を述べ，7 章でまとめる．

2. 関 連 研 究

SimpleScalar[1] や SimOS[12] に代表されるクロックレベルでのマイクロプロセッサシミュレーションは，命令レベルのシミュレーションに比べて，SD が非常に大きくなり研究開発の効率化を低下させる原因になっている．したがって，シミュレーションの高速化手法についてさまざまな研究が行われている．これらの中で代表的な成功例である FastSim[13] や BurstScalar[7] は，SD が大きくなる原因である命令スケジューリングの時間コストを計算再利用技術を用いて削減することで，精度を落とさずに高速化を図っている．

一方本研究と同じように，時間軸方向にシミュレーションを分割して並列化する手法は古くから提案されている[5),9),10]．これらはいずれも，何らかの手法で取得した実行トレースを時間軸方向に分割し，区間ごとの解析を並列に行うことで高速化を図っている．この方法で問題となる複数区間の解析結果の整合性については，解析区間を部分的に重複させることで対処している．この区間重複の考え方は後述するように本研究でも採り入れているが，やはり後述するように完全な整合性を保証するものでない．したがってこの手法は，性能と精度をトレードオフするものであり，精度についてはたとえばトレースのサンプリングよりも高いといった観点で議論されている．

近年では，この手法をクロックレベルシミュレーションに適用した DiST[4] が開発されている．DiST では，シミュレーション中に取得した統計量（IPC など）が，ユーザが設定した誤差の許容量未満になると統計的に判断できるまで，区間の重複度を動的に延長する工夫が行われている．この方法は性能と精度のトレードオフをユーザに委ねる点が特徴であるが，設定した許容誤差を保証するものではなく，また所与の精度に対する性能を事前に見積もるのは困難であろうと思われる．

これらの手法や，DirectRSIM[3] や FastILP[14] などの命令スケジューリングを簡略化して高速化する手法は，いずれも性能と精度をトレードオフするものであり，誤差が数%程度にとどまることで有用性を主張している．しかし得られた誤差はいずれも経験に基づくものであり，所与のアーキテクチャやワークロードに対して論理的に誤差の上限を保証する手段はない．

またクロックレベルシミュレータの代表的用途であるプロセッサアーキテクチャの研究では，プロセッサ性能の 10%内外の差を問題とするものが数多く存在し，それらでは数%のシミュレーション誤差が異なる結論を導く可能性もある[2]．

このほか，マルチプロセッサのアーキテクチャシミュレーションを分散・並列処理によって高速化する試みも提案されており，WWT[11] や Shman[6] では良好な台数効果が報告されている．しかしこれらが対象としている要素プロセッサは，いずれも単純なパイプラインで in-order 実行を行うものであり，out-of-order プロセッサへの適用については報告されていない．

一方本論文で提案するシミュレータは；

- out-of-order ユニプロセッサのクロックレベルシミュレーションを；
- 時間軸方向に分割した並列処理によって高速化し；
- 逐次シミュレーションと同じ結果をまったく精度を落とすことなく得る；

という特質を有しており，上記の従来研究とは本質的に異なるユニークなものとなっている．

3. 高速化手法

本論文で提案する高速化手法の基本的なアイデアは，シミュレーション過程を時間軸方向に分割し，それぞれを並列に実行することである．

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落とさないためには，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．

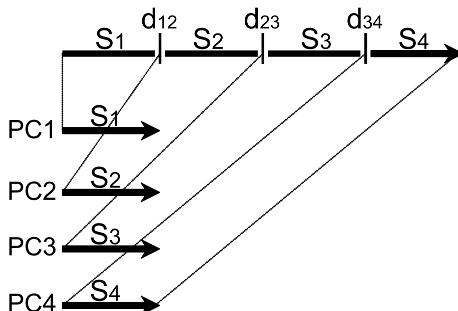


図 1 時間軸分割による並列化
Fig. 1 Time division parallelization.

たとえば，分割点 d_{12} は分割区間 S_1 を担当する PC1 にとっては，シミュレーション終了点であり，分割区間 S_2 を担当する PC2 にとっては，シミュレーション開始点になっている．よって，この点でマシン状態が一致していれば，分割区間 S_2 は正しくシミュレートされ，始めから d_{23} までのシミュレーション結果が求まる．同様にして d_{23} および d_{34} でマシン状態が一致していれば，全体のシミュレーション結果を求めることができる．

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．

以上から，並列シミュレーションでは，分割点において各ノードのマシン状態が一致しているかどうか，もしくは，分割区間シミュレーションが正しく行われるために必要なおおよそのマシン状態が存在しているかどうかがポイントになる．そこでまず，分割点でのマシン状態を一致させる方法を次節より述べていく．

なお，以降の説明では，本研究で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．前章で述べたように，一般に詳細シミュレーションの SD は 1,000〜10,000 であり，論理シミュレーションの SD は 10〜100 である．

3.1 マシン状態とその一致

本論文では，マシン状態として以下の 5 項目を想定する．以下の要素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．

- メモリ，レジスタ，プログラムカウンタ
- パイプライン（out-of-order 実行に必要なすべてのテーブル，キュー，バッファ類）
- キャッシュ
- TLB
- 分岐予測器

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．

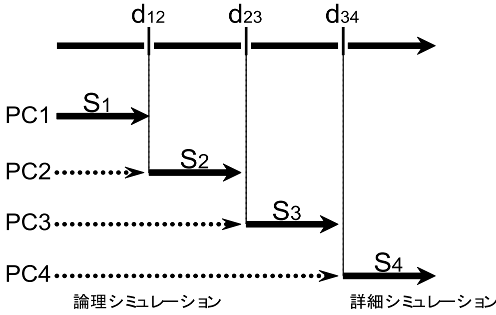


図 2 状態一致のための論理シミュレーション
Fig. 2 Logical simulation for machine state approximation.

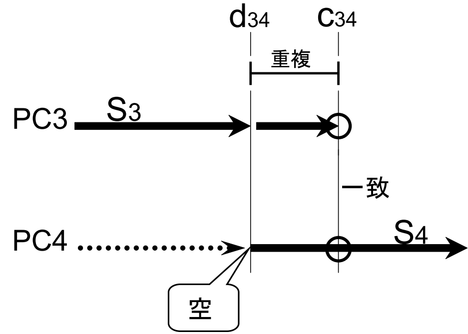


図 3 重複実行
Fig. 3 Overlapped execution.

3.1.1 メモリ・レジスタ・プログラムカウンタの状態一致方法

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，引き続き詳細シミュレーションを行う．**図 2** を例にあげれば，分割点 d_{12} において S_1 終了時と S_2 開始時のメモリ・レジスタ・プログラムカウンタ状態を一致させるためには，PC2 が分割点 d_{12} まで論理シミュレーションを行い，d_{12} の時点のメモリ・レジスタ・プログラムカウンタの状態を求めることで状態一致を図る．PC3，PC4 についても同様である．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．

3.1.2 パイプラインの状態一致方法

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．

図 3 に具体例をあげて説明する．図は分割点 d_{34} において PC3 と PC4 が，パイプラインを一致させるために，詳細シミュレーションを重複して実行する様子を表す．分割点 d_{34} は PC3 にとっては分割区間シミュレーションの終了点，PC4 では開始点となっている．PC4 のパイプラインは空であるから，PC3 と PC4 で状態が異なる．そこで，PC3 は分割点 d_{34} を越えて一定区間詳細シミュレーションを続けて実行し，点 c_{34} で状態を比較する．重複区間 d_{34}-c_{34} で分岐予測ミスが何回か発生すれば，状態は互いに近づいてゆき，点 c_{34} では状態が一致すると考えられる．

3.1.3 キャッシュ・TLB・分岐予測器の状態一致方法

キャッシュ・TLB・分岐予測については，状態一致を図るための基本方針が同じであるので，ここでは，キャッシュを例にあげて説明する．分割区間開始点ではキャッシュについての情報はない．よって，その情報を得るために，メモリ・レジスタ・プログラムカウンタを論理シミュレーションする際にキャッシュも加えて実行する．

論理シミュレーションでは，パイプラインのシミュレーションを行わないため，パイプラインに関係する分岐予測ミス発生についての影響がシミュレーションに反映されない．しかし，キャッシュは分岐予測ミス実行時のアクセスによっても更新される．したがって，キャッシュを正確にシミュレートするには，タイミング情報が必要である．論理シミュレーションによるキャッシュは正確でないため，パイプラインの状態一致方法と同じく，前後のノードで詳細シミュレーションを重複実行することで状態一致を図る．

しかしながら，キャッシュにはパイプラインにおいての分岐予測ミスのような過去と依存関係を大幅に

断ち切るイベントがないので，状態一致のためには不確かなキャッシュブロックが更新されるのを待たなければならない．しかし，実行プログラムによってはあまり参照されないキャッシュブロックも存在し，また，キャッシュは状態数が多いので，すべてのキャッシュブロックを一致させるためには，相当な重複区間を設けなければならず，これは現実的でない．したがって，一定区間を重複実行し，大半のキャッシュブロックを一致させた後，見切りで詳細シミュレーションを開始する．さらに次節で詳しく述べるように，シミュレーション中に発生したキャッシュアクセスとその結果を履歴として保存しておき，区間シミュレーション終了後にこの履歴を用いてその区間のシミュレーションの正当性を検証する．

3.2 履歴を用いたシミュレーションの正当性検証方法

ノード間で分割点におけるマシン状態が異なる場合でも，その違いが分割区間シミュレーションに影響をおよぼさないことが確認できればシミュレーションが正しく行われたことを保証できる．本節では，履歴を用いたシミュレーションの正当性の検証方法を述べる．

3.2.1 キャッシュシミュレーションの正当性検証

ノード間で分割点におけるキャッシュに相違があっても，分割区間シミュレーションが正しく行われる場合は次の 2 通りである．

- 相違のあるブロックは参照されなかった．
- 相違のあるブロックの最初の参照はリプレースであり，ライトバックの有無が一致していた．

キャッシュアクセスとその結果（**キャッシュ参照履歴**）を検証するにあたっては，次の 2 つについて考える必要がある．まず，分割区間シミュレーションにおけるキャッシュ参照履歴は膨大な量であるため，履歴の保存コストを減らす方法を見出す必要がある．2 つ目は，キャッシュ参照履歴の比較方法である．

保存コストについては，キャッシュの容量が有限であることに注目して，記録すべき履歴を定数回で抑える．キャッシュの特定のセットは連想度に等しい数の異なるアドレスに対するアクセスによって，過去の状態にかかわらず一定の状態になる．過去に依存しない一定の状態になるまでのキャッシュ参照履歴が正しければ，それ以降のシミュレーションは正しいので，一定の状態になるために必要なだけの履歴を記録することで保存コストを大幅に減少させることができる．たとえば 128 セット，4 連想のキャッシュの場合，各セットについていえば，異なるアドレスに対する 4 回のアクセスとその際のライトバックのあるなしについて記録すればよいので，最大 128×4 回の保存コストに抑えられる．

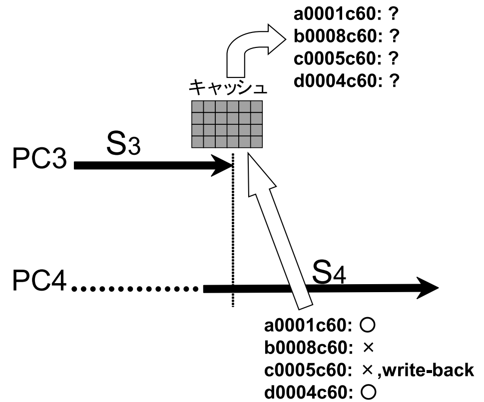


図 4　キャッシュ参照履歴の比較
Fig. 4　Comparison of cache reference history.

次に履歴の比較方法を述べる．並列シミュレーションにおいては，比較にあたってリファレンスとなる分割区間の正しいキャッシュ参照履歴は存在しない．よって，キャッシュ参照履歴が正しいかどうかをチェックするためには，分割点の前区間終了時のキャッシュに後区間のキャッシュ参照履歴を適用し，そのヒット・ミスが同じであるか，またミスの場合はライトバックの有無が一致するかどうかを調べる．

たとえば，**図 4** では分割区間 S_4 のあるセットについて，異なるアドレスに対する 4 回のキャッシュ参照履歴が a0001c60:ヒット，b0006c60:ミス，c0005c60:ミス：write-back，d0004c60:ヒットとなっている．そこでその履歴を分割区間 S_3 終了時のキャッシュに適用し，その結果が同じであるかをチェックする．各セットが同様の結果になっていることが確認された場合は，分割区間 S_4 を PC3 が引き続いてシミュレーションを行っても，同様のシミュレーション結果を示すはずなので，S_4 の結果は PC3 にとって，後続の正しいシミュレーション結果であるといえる．したがって，実行区間が詳細シミュレーションのみで行われる S_1 から，順に後続の分割区間について比較してゆき全体のシミュレーションが正しく行われたかどうかを検証する．

なお通常のアーキテクチャでは，ミスの場合に追い出されるブロックの一致確認をする必要はなく，したがってどのブロックが追い出されるかも記録する必要はない．すなわち，どのブロックが追い出されても当該アクセスに要する遅延などは変わらず☆，また以後

☆ ライトバックされるアドレスによってアクセス時間が異なるようなアーキテクチャであれば話は別であるが，そのような構成は例外的である．

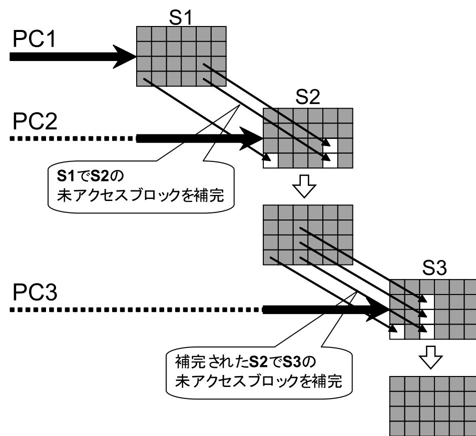


図 5　キャッシュの補完
Fig. 5　Complementing cache line states.

のシミュレーションに影響するのは（追い出されたブロックではなく）残存しているブロックであるからである．また階層型キャッシュの場合，たとえば L1 から L2 へ異なるブロックアドレスがライトバックされると L2 の状態が一致しなくなるが，この相違が後のアクセスで顕在化しなければ問題はなく，また顕在化すれば後に L2 アクセスの不一致として検出される．なお，どのブロックがどのタイミングで追い出されたかを厳密に知りたい場合も，履歴による確認の際に求めることができる．

また，区間終了時のキャッシュ状態については，キャッシュ参照履歴が正しいことが確認されても，分割区間シミュレーション中にアクセスされなかったブロックの内容は論理シミュレーションで得られた区間初期値であるため，区間終了時のキャッシュ状態は正しいかどうかは分からない．そこで，未アクセスのキャッシュブロックについては，前区間終了時のキャッシュを用いて補完する．

S_1 終了時のキャッシュだけは，始めから詳細シミュレーションを行うため正しいキャッシュ状態を保持している．よって**図 5** に示すように，S_1 終了時のキャッシュを用いて S_2 の参照履歴を比較した後，さらにそれを用いて S_2 終了時のキャッシュを補完して正しいキャッシュ状態を得る．こうして得た S_2 終了時の正しいキャッシュは，S_3 の履歴比較と終了時の補完に用い，以後この比較と補完を繰り返して検証を進めてゆく．したがって履歴の比較はつねに，正しいキャッシュをベースとして行われることになる．

また，検証の結果，分割区間シミュレーションが正しく実行されなかったことが判明した場合は，その分割区間シミュレーション結果は使えない．よって，分割区間シミュレーションのやり直しを行うことになるが，その方法については，次章で説明する．また，以降の説明では，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を分割区間失敗と呼ぶことにする．

3.2.2　TLB シミュレーションの正当性検証

TLB についても，キャッシュと同様に一定の状態になるために必要なだけの参照履歴を保存し検証を行う．また参照されることのなかった箇所については，キャッシュ同様，補完を行う．

3.2.3　分岐予測シミュレーションの正当性検証

分岐予測器については，リターンアドレススタックについて履歴を用いて比較する．すなわち，分割区間シミュレーションにおいて分割点での違いが更新される前に分岐先アドレスとして実行されるかどうかをチェックする．分割点でリターンアドレススタックに相違があっても，その違いが分岐先に利用されなければシミュレーションは正しく行われる．履歴の保存コストは，リターンアドレススタックのエントリ数が 8 の場合，最大 8 回である．また分割区間シミュレーション過程において，使われない箇所があれば補完する．なお BTB や BPB も履歴による検証が可能であるが，重複実行によりきわめて高い確率で一致するため省略している．

3.3　ハードウェア機構の変更・追加

本節では，シミュレーション対象マイクロプロセッサのハードウェア機構に対する，ユーザによる変更や追加について考察する．まず，ハードウェア機構の量的な変更，たとえば演算ユニット数，命令ウィンドウサイズ，キャッシュの連想度や容量の変更については，これらの状態や履歴の保存・比較をパラメータ化することにより容易に対応できる．

次に動作メカニズムの変更や，新たなハードウェア機構の追加などの，質的な変更・追加については，(1) 論理シミュレーションによる代替・近似，(2) 状態の保存・比較，(3) 動作履歴の保存・比較，の 3 つのポイントについて考察する．

(1)　論理シミュレーションによる代替・近似

機構の動作が命令や ISA で定められる資源の値によってのみ定まり，投機的実行や動作タイミングには影響されない場合には，その機構をシミュレートする関数などを（詳細シミュレータだけでなく）論理シミュレータからも呼び出すようにするだけでよい．すなわちこのような機構については，内部状態や動作履歴の保存・比較は不要である．一方キャッシュや分岐予測器のように，論理シミュレーションによって近似的な状態を求めることで状態の一致確

率が高くなるものについては，論理シミュレーションから機構の内部状態変更をシミュレートする関数などを呼び出すようにすることが望ましい．なおこの内部状態変更のシミュレーションは，本来必要なシミュレーションの一部であるため，特に新たな設計などを必要とするものではない．

(2) 内部状態の保存・比較
機構の内部状態を表現する配列などのデータ構造を，分割区間の始めに行う詳細シミュレーションの重複実行が完了した時点で保存し，直前の区間を担当する PC の区間終了時の状態と比較すればよい．なお保存または比較の際に，データ構造中に含まれるポインタを位置独立な値（たとえば相対アドレス）に変換する必要があるが，この操作以外は単純なバイト列のコピーや比較で実現することができる．

(3) 動作履歴の保存・比較
キャッシュのように機構の内部状態全体が一致する確率が高くない場合，分割区間失敗の確率を小さくするために動作履歴の保存と比較が必要となる．履歴として保存する情報は機構の仕様に依存するが，一般には機構を有限状態機械と考えてその入力と出力の列を保存すればよい．また履歴を用いた正当性検証は，本来必要なシミュレーションの一部を用いて正しい状態と入力列から同じ出力列が得られるかを判定する操作となり，特に新たな設計などを必要とはしない．また区間終了時に正しい内部状態を得るための補完操作は，テーブルのエントリなど更新単位となる要素ごとに，区間シミュレーション中に更新されたか否かを示すフラグを設け，未更新の部分を前区間終了時の値に置換することで実現できる．なお保存する履歴をできるだけ短くするためには，テーブルのエントリ値などが前区間の値に依存しなくなることを検知して，履歴の保存を中止すればよい．この非依存性の検知は内部状態の更新論理を熟知しているはずのユーザにとっては比較的容易であり，たとえば多くの予測器に用いられる飽和カウンタの場合，増減それぞれの回数の差分がカウンタの最大値に一致した時点で履歴保存を中止すればよい．

なお機構の性質によっては，内部状態の過去への依存性が長期間解消されず，かつ状態参照が頻繁に行われてその結果がシミュレーション対象プロセッサの動作に影響するため，時間軸分割シミュレーションが本質的に困難であることも考えられる．たとえば，タイマ割込みやメモリのリフレッシュカウンタなど，一定のサイクル間隔で作動する機構はこのような性質を持つが，これらをシミュレートすること自体が例外的であると考えられる☆．

4. 並列シミュレーションの設計

並列シミュレーションは，分割区間の並列実行と分割区間シミュレーション結果の検証･統合によって行われる．マシン状態の一致を図る方法とシミュレーション履歴の検証方法については，前章で述べた．ここでは，その高速化手法を用いた並列シミュレーション方法について述べる．

4.1 分割区間失敗時の対応

分割区間のシミュレーションが正しく行われるためには，パイプラインなどについては重複区間終了後の状態一致判定が，またキャッシュなどについては区間シミュレーション終了後の履歴検証が，それぞれ成功しなければならない．本研究では，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，これらのいずれかが失敗して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き続き詳細シミュレーションを行うことで対応する．すなわち，メモリ（主記憶）データの保存が必要となる巻き戻しではなく，再実行開始点でのマシン状態を保持している前区間担当ノードが継続実行することで，再実行コストをできるだけ小さくしている．

4.2 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．説明のために，まず，単純な状態一致のみで分割区間シミュレーションの成功，失敗を判断する（開始状態が不一致の分割区間は失敗となる）場合についての並列シミュレーション方法を述べてから，比較にシミュレーション履歴を取り入れた場合のシミュレーション方法を述べる．

全体の並列シミュレーションの具体例を**図 6** に示す．図では各ノードの分割区間シミュレーション終了後，それぞれの状態を比較した結果，d_{23} と，d_{45} でマシン状態が不一致であった場合を表している．この際，やり直しのための再実行 S'_3 と S'_5 は，同時に行うことができる．再実行の結果，次の分割点でマシン状態が一致すれば，S_1 - S_2 - S'_3 - S_4 - S'_5 - S_6 というシミュレーション区間を統合して，並列シミュレーションが終了する．

しかし履歴による比較を導入すると，図 6 の S'_3 と

☆ たとえば SimpleScalar にはこれらは実装されていない．

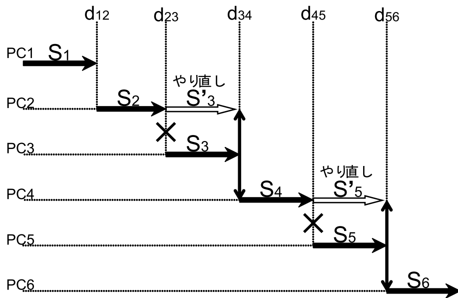


図 6 状態比較による並列シミュレーション
Fig. 6 Parallel simulation with state comparison.

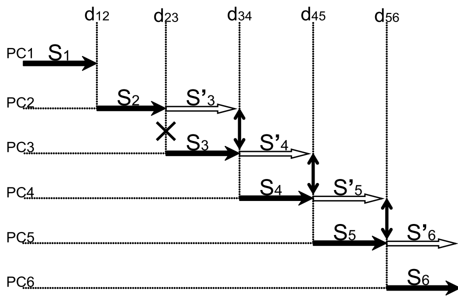


図 7 履歴比較を用いた並列シミュレーション
Fig. 7 Parallel simulation with history comparison.

S'_5 の再実行を同時に行うことができなくなる．すなわち，S'_5 の再実行を確実に行うには，その開始点である S_4 終了時の正しい状態が必要となるが，それを求めるには S'_3 の終了時のキャッシュ状態を用いて S_4 のキャッシュを補完しなければならない．したがって S'_3 と S'_5 の実行は逐次化されてしまうことになる．

この逐次化を避ける方法としては，S_4 終了時の暫定的なキャッシュ状態，すなわち S'_3 終了時の状態で補完する前の状態を用いて，S'_5 を S'_3 と同時に実行することが考えられる．しかしこの場合には S'_5 の実行は失敗する可能性があり，失敗したときにはもう一度再実行を行う必要がある．この再々実行を担当可能なノード，すなわち d_{45}，あるいはそれ以前で d_{45} に最も近接した状態を保持しているノードは PC2 あるいは PC3 である．ここで PC2 が担当するためには S'_3 の後に S_4 を実行する必要があり，また PC3 が担当するためには失敗してしまった S_3 を継続して S_4 に相当する区間を実行する必要がある．また成功したかに見える S_4 や S_6 も，履歴検証のベースとなる正しいキャッシュ状態が得られていないため実は失敗していた可能性もあり，その場合の再実行はさらに複雑なものとなる．

そこで，履歴を導入するにあたっては**図 7** のような方法を用いた．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かかわらず，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．

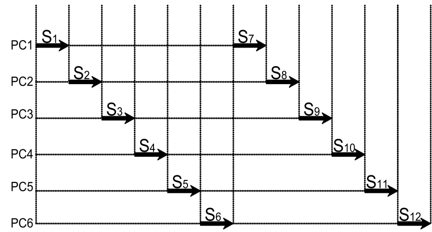


図 8 多数分割方法（1）
Fig. 8 Method to divide simulation by a multiple of number of nodes (1).

4.3 並列シミュレーション方法（多数分割）

前節では，分割数と PC 台数が等しい場合の並列シミュレーションについて述べたが，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**図 8** のように台数分割シミュレーションを繰り返す形で実行する．図 8 は PC6 台で 12 分割をする様子である．以下，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．図 8 は 2 フェーズで構成されている．各 PC は，第 1 フェーズ終了後，第 2 フェーズでの分割区間 S_7〜S_{12} の開始点まで論理シミュレーションを行い，引き続き詳細シミュレーションを行う．台数分割シミュレーションを単純に繰り返すと，第 1 フェーズと第 2 フェーズの間で，PC6 の S_6 と PC1 の S_7 が統合可能であるか検証を行う必要がある．分割区間失敗の場合には，前区間担当のノードが次区間をやり直すため，S_7 が失敗した場合には，PC6 が S_7 の再実行を担当する必要がある．よって，第 2 フェーズでは，PC1〜PC5 は担当分割区間のシミュレーショ

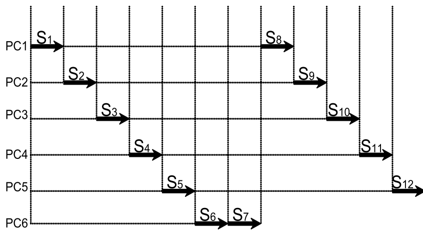


図 9 多数分割方法（2）
Fig. 9 Method to divide simulation by a multiple of number of nodes (2).

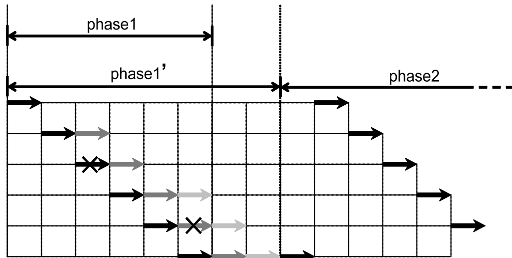


図 10 フェーズ区間長
Fig. 10 Length of a phase.

ンを行うために実行を先に進められるが，PC6 だけは S_7 のやり直しに備え，次の S_{12} のシミュレーションに進むことができない．

そこで，**図 9** 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．この結果，第 2 フェーズの最初の区間である S_7 の実行は，その正しい開始状態を保持している PC6 によって行われるため，第 1 フェーズの PC1 による S_1 の実行と同じように確実に成功する．すなわち第 2 フェーズでは，PC6 が第 1 フェーズの PC1 の役割を果たし，PC1〜5 はそれぞれ第 1 フェーズの PC2〜6 の役割を果たすことになり，第 1 フェーズと同じ考え方で並列シミュレーションを実行することができる．

また，あるフェーズの最終区間担当ノードが次のフェーズの先頭区間を確実に正しくシミュレートするには，フェーズの終了時にそのノードが正しいマシン状態を保持している必要がある．そこでフェーズ内で分割区間失敗が生じた場合には，**図 10** に示すように最終区間担当ノードが正しいマシン状態を得るまでフェーズを延長する．すなわち，1 フェーズあたりの分割区間数は，分割区間失敗のたびに増加する．図では，分割区間失敗が 2 回発生しているので，フェーズ区間長は，失敗がない場合の $phase1$ から，$phase1'$ に 2 区間だけ延長される．

5. 実 装

並列シミュレータは，SimpleScalar Tool Set Version 3.0 [1] を並列化することで実装した．論理シミュレーションは，命令レベルシミュレータモジュールである sim-fast に，キャッシュおよび分岐予測シミュレーションのモジュールである sim-cache と sim-bpred を組み合わせることにより実現した．詳細シミュレーションは，命令スケジューリングなどを行う sim-outorder であり，このモジュールの高速化が並列シミュレーションの目的である．並列実行には MPI を用いた．

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．すなわち，全命令実行数を N，使用するノード数を n，多数分割のフェーズ数を P とすると（$P = 1$ ならば台数分割），1 つの分割区間の命令数（分割区間長）l は $l = N/nP$ となる☆．したがって各々のノードは，各フェーズにおいて，l 命令分の詳細シミュレーションと，その前後 $l(n-1)$ 命令分の論理シミュレーションを実施するのが基本となる．

しかし実際には，詳細シミュレーションを隣接したノード間で重複実行するため，詳細シミュレーションを行う区間は l よりも長くなる．今回の実装では l に一定の値 f を乗じた命令数を重複実行区間長としたので，詳細シミュレーションを行う区間長は $l(1+f)$ となる☆☆．評価では，f の値を 0.1（すなわち区間長の 10%）とした．

なおフェーズの最終区間を担当するノードだけは，担当区間長が l となる．また分割区間失敗により次の区間も担当する場合の詳細シミュレーション区間長は，失敗による延長回数を k としたとき $l(1+k+f)$ となる．すなわち失敗時のペナルティは（$l(1+f)$ ではなく）l 命令分の詳細シミュレーション時間となる．

なお上記の議論は，全命令数 N が並列シミュレーション前に既知であるという前提を用いているが，未知である場合にはあらかじめ l を適切な値に定め，フェーズ数が $lnP \geq N$ となるような最小の P に達するまで，多数分割方式でのシミュレーションを行えばよい．

分割区間シミュレーションの検証は，つねに PC1 が行うことにした．各分割区間の並列実行終了後，PC1 は検証を行い，分割区間シミュレーションのやり直しを行うか，次フェーズに進むかを決定する．各 PC は分割区間シミュレーションの開始時と終了時にマシン

☆ 一般に l は整数とならないため，分割区間長は $\lfloor N/nP \rfloor$ または $\lceil N/nP \rceil$ となるが，ここでは議論を簡単にするために N が nP の倍数であるとする．

☆☆ 説明を簡単にするために，$l(1+f)$ も整数であるとする．

表 1 プロセッサの構成
Table 1 Processor configuration.

<table>
<tr><td colspan="2">命令発効幅</td><td colspan="3">4</td></tr>
<tr><td colspan="2">RUU エントリ数</td><td colspan="3">16</td></tr>
<tr><td colspan="2">LSQ エントリ数</td><td colspan="3">8</td></tr>
<tr><td colspan="2">メモリポート数</td><td colspan="3">2</td></tr>
<tr><td rowspan="4">機能
ユニット数</td><td>INT-ALU</td><td colspan="3">4</td></tr>
<tr><td>INT-MUL/DIV</td><td colspan="3">1</td></tr>
<tr><td>FP-ALU</td><td colspan="3">4</td></tr>
<tr><td>FP-MUL/DIV</td><td colspan="3">1</td></tr>
<tr><td rowspan="3">分岐予測</td><td>予測方式</td><td colspan="3">2 bit カウンタ/2 K エントリ</td></tr>
<tr><td>BTB</td><td colspan="3">512 エントリ/4-way</td></tr>
<tr><td>RAS</td><td colspan="3">8 エントリ</td></tr>
<tr><td rowspan="2">メモリ</td><td>初期参照レイテンシ</td><td colspan="3">18</td></tr>
<tr><td>バースト転送間隔</td><td colspan="3">2</td></tr>
<tr><td rowspan="3">TLB</td><td>命令</td><td colspan="3">16 エントリ/4-way</td></tr>
<tr><td>データ</td><td colspan="3">32 エントリ/4-way</td></tr>
<tr><td>ミスレイテンシ</td><td colspan="3">30</td></tr>
<tr><td>キャッシュ</td><td>容量</td><td>ラインサイズ</td><td>way 数</td><td>レイテンシ</td></tr>
<tr><td>L1 命令</td><td>16 KB</td><td>32 B</td><td>1-way</td><td>1</td></tr>
<tr><td>L1 データ</td><td>16 KB</td><td>32 B</td><td>4-way</td><td>1</td></tr>
<tr><td>L2 統合</td><td>256 KB</td><td>64 B</td><td>4-way</td><td>6</td></tr>
</table>

状態を，シミュレーション中にキャッシュ，TLB，リターンアドレススタックのシミュレーション履歴を保存しておき検証に用いる．

6. 評　　価

シミュレーション対象プロセッサモデルは，**表 1** に示す SimpleScalar のデフォルトモデルとした．また評価プログラムには SPEC CPU95 を，データセットには ‘train’ をそれぞれ用いた．

6.1 予 備 評 価

重複実行後のパイプラインとキャッシュの状態一致率，およびキャッシュ参照履歴の一致率について調査を行った．

6.1.1 パイプライン状態の一致率

通常のマイクロプロセッサシミュレーション実行中にパイプラインを空にすることで，区間シミュレーション開始時点でのパイプライン状態を再現し，それ以後，どの程度シミュレーションが進めば，パイプラインを空にしなかった場合と状態が一致するかを測定した．全命令区間にわたり均等に約 1 万カ所について調査したところシミュレーションが 1,000 命令分実行された後には，99.9%状態が一致することが分かった．よって，パイプラインについては，他の要素が一致していれば，ごくわずかな重複で簡単に状態が一致するため，並列シミュレーションにおいては，他要素について，その状態および履歴を一致させることが重要である．

6.1.2 キャッシュ状態の一致率

キャッシュについて，重複実行区間を 1,000 命令としたときの状態一致率を測定したところ，L1 命令キャッシュ（il1）が平均 23%，L1 データキャッシュ（dl1）が 62%，また L2 統合キャッシュ（ul2）が 34%となり，キャッシュはわずかな重複では一致しないことが分かった．そこで重複区間を増加させて，その一致率を測定した．具体的には，分割数 16，重複実行を分割区間の 10%（= 全命令の約 0.6%）で並列シミュレーションを行うものと想定し，1/16 の分割区間の開始点を全命令区間にわたって，均等に約 90 カ所とり測定した．その結果，10%重複後の状態一致率は，平均 il1: 53%，dl1: 98%，ul2: 62%であった．この結果から重複区間を十分に大きくすれば顕著に一致率が増加することが明らかになったが，3 つのキャッシュのすべてが一致する確率は $0.53 \times 0.98 \times 0.62 = 32\%$ であり，3 区間中の 2 区間で失敗が生じることになってしまう．

そこで同じ条件を用いて，キャッシュ参照履歴の一致率を測定した結果，3 つのキャッシュ全体での一致率が 75%であることが明らかになった．したがって仮に 8 分割でも同程度の一致率であるとすれば，まったく失敗が生じない確率が 13%程度☆はあるといえる．また 1 回だけ失敗する確率は 34%となるが，4.2 節で述べた方法により 1 回失敗した後には重複区間が著しく増加するため，2 回以上失敗する確率は十分に小さくできるのではないかと判断した．そこで結果から，8 ノードでの詳細シミュレーション時間を逐次実行の 1/4 程度とできる可能性が高く，十分な有効性を並列シミュレータが実現できるという結論を得た．

6.2 並列シミュレーション評価

実装した並列シミュレータを用いて，高速化率を測定した．評価には，OS: Redhat Linux 7.2，CPU: Intel PentiumIII/866 MHz，Mem: 512 MB，N/W: Gigabit Ethernet のクラスタを用いた．

PC 台数を 8 とし，分割数を 8 および 16 としたときのオリジナルの SimpleScalar に対する高速化率と，分割区間失敗回数を，それぞれ**図 11**，**図 12** に示す．図に示すように，分割数 8 での最大高速化率は su2cor と ijpeg での 2.66 倍であり，また平均は 1.83 倍である．一方分割数 16 の場合は，mgrid の 2.46 倍が最高値であり，平均は 1.71 倍である．またすべての失敗の原因は，3.1.3 項で述べたように，分岐予測ミス時のキャッシュおよび TLB のアクセスが論理シミュレーションと詳細シミュレーションで異なることであり，その大半は L1 命令キャッシュと L2 統合キャッシュの不一致であった．

図 11 に示したように 16 分割は最大値も平均値も

☆ 最初の区間は失敗しないので 0.75 の 7 乗と計算できる．

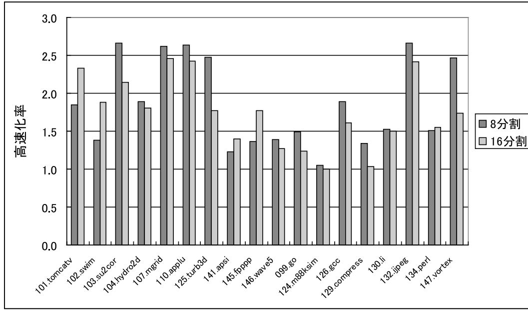


図 11　高速化率
Fig. 11　Speedup.

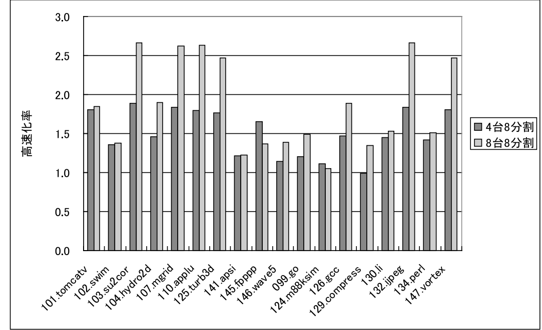


図 13　4 ノードと 8 ノードの性能比較
Fig. 13　Performance comparison of four and eight-node simulation.

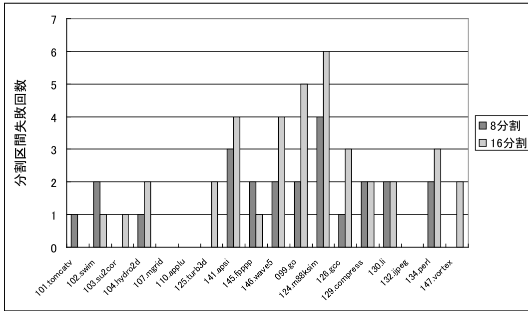


図 12　分割区間失敗回数
Fig. 12　Number of failed intervals.

8 分割より若干劣っているが，tomcatv，swim，および fpppp では，この傾向が逆転している．この理由は図 12 の分割区間の失敗数が，これらのプログラムについては 8 分割よりも 16 分割が少ないことから明らかである．すなわち並列シミュレーションの性能を定める最大の要因は分割区間の失敗数であり，これを最小にするような分割数が最も高い性能を与える．また 16 分割の区間長は 8 分割のほぼ 1/2 であるため 1 回の失敗によるペナルティもほぼ 1/2 となる．したがって，16 分割での失敗数が 8 分割の失敗数の 2 倍未満のプログラムである apsi と perl においても，若干ではあるが 16 分割が有利になっている．ただし m88ksim，compress および，li は失敗数が 2 倍未満であるにもかかわらず，16 分割の性能が劣っており，特に compress では顕著である．この理由はこの 3 つのプログラム実行時間が短く☆，失敗回数に比例する検証オーバヘッドやフェーズ切替えのオーバヘッドが顕在化しているものと考えられる．

また失敗回数が性能に与える影響は，分割数を 8 に固定してノード数を 8 から 4 に減らした場合の性能からも見てとることができる．すなわち**図 13** に示すように，すべての区間が成功する 6 つのプログラムでは，8 ノードの性能が 4 ノードに比べて顕著に高いが，それ以外では差が小さく，失敗数が多い m88kisim では性能が逆転している．

これらの結果から，ワークロードによって失敗の回数やペナルティを最小化する分割数が異なることは明らかである．しかし最適な分割数を事前に求めることは容易ではなく，たとえば実行命令数のように比較的容易に事前評価できる指標との関係は，これまでの解析では明らかにできていない．また前述のように失敗の原因は分岐予測ミスに密接に関係しているが，予測ミス頻度などを分割数の最適化に利用できるかどうかも，今後の課題である．

6.3　結果の解析と考察

前節の高速化結果を解析するために，並列シミュレータの性能モデルを考える．図 7 に示したように，シミュレーション時間は最終区間担当ノードの実行時間によって定まり，逐次詳細シミュレーションの実行時間を T_S，論理シミュレーションと詳細シミュレーションの実行時間比を $R_{D/L}$，1 区間の結果検証のための状態・履歴の保存・比較に要する時間を T_V とすると，まったく失敗がないときの分割数 n の並列実行時間 T_P は下式により近似できる．

$$T_P = T_S \left(\frac{1}{n} + \frac{(n-1)}{nR_{D/L}} \right) + T_V(n-1)$$

上式の第 1 項の括弧内の $1/n$ は詳細シミュレーション区間が全体の $1/n$ であることに対応し☆☆，$(n-1)/(nR_{D/L})$ は論理シミュレーション区間が全体の $(n-1)/n$ であって，それを詳細シミュレーショ

☆ すべての中で最短が compress であり，m88ksim と li がこの順でそれに次ぐ．また次に短い fpppp と比べても li は約 1/2，m88ksim は 1/3，compress は 1/10 以下である．

☆☆ 最終区間は次の区間がないため重複実行分の延長はない．

ンの $R_{D/L}$ 倍の速度で実行することを意味している．また第 2 項は，第 1 区間以外の区間について結果検証が必要であることを示しており，T_V は対象アーキテクチャによって定まりワークロードには依存しない定数と考えることができる．また k 回失敗し，その結果 m 回の区間検証を余分に必要とした場合の並列実行時間は

$$T_P = T_S\left(\frac{1+k}{n} + \frac{(n-1)}{nR_{D/L}}\right) + T_V(n+m-1)$$

と見積もることができる．

ここで今回の実装に使用した論理シミュレータのベースである SimpleScalar の sim-cache は，一般的な命令レベルのキャッシュシミュレータに比べてかなり低速であり，詳細シミュレータである sim-outorder の約 5 倍程度の性能にすぎない．また論理シミュレータには sim-bpred の機能も組み込まれているので，実際の $R_{D/L}$ はさらに小さくなり全ワークロードの平均値は 4.16 である☆．また上式に $n = 8$ と各ワークロードの T_S，k，m およびワークロード独立な実測に基づく T_V の見積り値 2.1 秒を代入してモデル値を求めると，±25% 以内の誤差で実測値を近似できる．また $T_V = 0$ として失敗なしのワークロードの限界性能を求めると，vortex の 3.16 倍が最大値となる．したがって今回の実装の性能を律しているのが，論理シミュレータの性能であることが分かる．

上述のように SimpleScalar の sim-cache は，他の命令レベルのキャッシュシミュレータに比べて非常に性能が低い．実際我々は，ワークロードごとに命令エミュレーションのコードを最適化する技術などによって，SimpleScalar の命令エミュレータである sim-fast の性能を，SPEC95 について最大 34 倍，平均 19 倍に高速化できることを明らかにしている[8)]．またこの研究では，sim-fast の高速化をメモリアクセス時のアライメント確認法の改良などを組み合わせることで，sim-cache の性能を最大 4 倍，平均でも 3.1 倍にできることも明らかにしている．なお，sim-cache には非常に非効率なコード，たとえばセット連想キャッシュのタグの線形リストによる管理などがあるが，上記の改良にはこれらキャッシュ全体のシミュレーションに関するものは含まれていない．したがってこれらを改良すればさらに論理シミュレーションを高速化でき，たとえば 5 倍程度の高速化は十分期待できると考えられる．

そこで，論理シミュレーションの高速化により並列シミュレーションをどの程度高速化できるかを見積もるために，並列シミュレーションに占める論理シミュレーションの実行時間を測定した．**表 2** はその結果を示すものであり，$n = 8$ と $n = 16$ の場合の並列シミュレーションと論理シミュレーションの実行時間，および逐次実行である SimpleScalar の実行時間を示している．この表から，分割区間失敗数の回数が少ないものほど論理シミュレーションが占める割合が高く，全体の高速化率を制限していることが分かる．また前述の技術によって論理シミュレーションの実行速度を 3 倍にすることは比較的容易であると考えられるが，その場合には $n = 8$ の高速化率は最大 2.66 倍から 4.47 倍に（mgrid），平均では 1.86 倍から 2.63 倍に，それぞれ向上するものと予想される．また $n = 16$ の場合は，同様に最大 4.20 倍，平均 2.51 倍になるものと算出される．さらに論理シミュレーションの実行速度を 5 倍にできたとすると，$n = 8$ では最大 5.20 倍，平均 2.90 倍に，また $n = 16$ では最大 4.89 倍，平均 2.77 倍に，それぞれ向上することが期待できる．

また，高速化率を向上させる別のアプローチとして，失敗数の削減があげられる．失敗の主要因であるキャッシュの不一致は，3.1.3 項で述べたように分岐予測ミス時のアクセスが論理・詳細シミュレーションで異なることによる．したがって，論理シミュレーションの際にも分岐予測ミス発生時にミスパスで行われるキャッシュアクセスを部分的にシミュレートすることにより，不一致の可能性を大きく削減できるものと考えられる．

表 2 並列シミュレーション実行時間（秒）
Table 2 Breakdown of the execution time of parallel simulation.

SPEC95	逐次	8 分割		16 分割	
		論理	全体	論理	全体
101.tomcatv	39,634	7,726	21,441	9,665	17,006
102.swim	2,114	311	1,534	463	1,124
103.su2cor	51,069	11,225	19,170	12,014	23,822
104.hydro2d	18,804	3,869	9,930	4,733	10,403
107.mgrid	34,950	8,263	13,335	8,853	14,230
110.applu	1,267	284	481	303	523
125.turb3d	37,795	9,306	15,281	9,970	21,385
141.apsi	6,042	804	4,927	1,492	4,317
145.fpppp	1,036	179	757	265	586
146.wave5	8,006	1,341	5,759	2,071	6,305
099.go	1,594	217	1,071	323	1,284
124.m88ksim	296	28	281	71	295
126.gcc	3,829	665	2,026	824	2,377
129.compress	94	14	70	20	91
130.li	539	76	353	114	359
132.ijpeg	3,094	696	1,162	754	1,280
134.perl	6,798	1,207	4,506	1,547	4,382
147.vortex	7,037	1,591	2,855	1,696	4,054

☆ 3.69〜4.81 の範囲に分布している．

7. ま と め

本論文では，マイクロプロセッサシミュレーション過程を時間軸分割し並列にシミュレーションを行うことで高速化を図った．以下に手法のポイントをまとめる．

- 論理シミュレーションを分割点まで実行し，一定区間詳細シミュレーションを重複実行することで，マシンの各要素について状態を一致または，それに近い状態にする．
- 状態が一致しないままスタートした分割区間シミュレーションについては，履歴を用いて正しく実行されたか検証する．
- 分割区間失敗の場合には，前区間担当のノードが後区間のシミュレーションを引き続き行うことで対応する．また，失敗区間のやり直しを行うノード以降のノードについても，同時に次区間をシミュレートすることで，失敗区間長を重複実行区間にあてる．

8 ノードの PC クラスタに並列シミュレータを実装し，分割数 8 と 16 の性能を SPEC CPU95 を用いて評価した結果，8 分割では最大 2.7 倍，平均 1.8 倍の高速化が，また 16 分割では最大 2.5 倍，平均 1.7 倍の高速化が，それぞれ達成された．また評価結果を近似する性能モデルを用いて，SimpleScalar の sim-cache をベースとする論理シミュレータの性能が高速化率を律していることを明らかにし，我々の研究グループで提案している論理シミュレーションの高速化技法などを用いれば，最大 4〜5 倍，平均 2.5〜3 倍の高速化が期待できることを示した．

今後の課題として，上記の論理シミュレータの高速化が緊急的なものとしてあげられる．また本論文の冒頭で述べた 8 ノードで 4 倍程度の高速化を少なくとも平均的に達成するには，分割区間の失敗率をより小さいものにする必要がある．このための有力な方法として，分岐予測ミス時のキャッシュアクセスを論理シミュレーションでも部分的に実施することがあげられる．またワークロードの性質に応じた分割数や重複実行の長さの調整についても，今後検討する予定である．

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) Desikan, R., Burger, D. and Keckler, S.W.: Measuring Experimental Error in Microprocessor Simulation, *Proc. 28th Intl. Symp. Computer Architecture*, pp.266–277 (2001).
3) Dubhakula, M., Pai, V. and Adve, S.: Improving the Accuracy vs. Speed Tradeoff for Simulating Shared Memory Multiprocessors with ILP Processors, *Proc. 5th Intl. Conf. High-Performance Computer Architecture*, pp.23–32 (1999).
4) Girbal, S., Mouchard, G., Cohen, A. and Temam, O.: DiST: A Simple, Reliable and Scalable Method to Significantly Reduce Processor Architecture Simulation Time, *Proc. Intl. Conf. Measurement and Modeling of Computer Systems*, pp.1–12 (2003).
5) Heidelberger, P. and Stone, H.S.: Parallel Trace-Driven Cache Simulation by Time Partitioning, *Proc. 1990 Winter Simulation Conf.*, pp.734–737 (1990).
6) Matsuo, H., Imafuku, S., Ohno, K. and Nakasima, H.: Shaman: A Distributed Simulatior for Shared Memory Multiprocessors, *12th IEEE/ACM Intl. Symp. Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp.347–355 (2002).
7) 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.SIG6(ACS6), pp.54–65 (2004).
8) 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，先端的計算基盤システムシンポジウム SACSIS 2005, pp.329–338 (2005).
9) Nguyen, A-T., Bose, P. and Wellman, J-D.: PARSIM: A Parallel Trace-driven Simulation Facility for Fast and Accurate Performance Analysis Studies, *Proc. IEEE Intl. Performance, Computing and Communication Conf.*, pp.291–297 (1997).
10) Nguyen, A-T., Bose, P., Ekanadham, K., Nanda, A. and Michael, M.: Accuracy and Speed-up of Parallel Trace-driven Architectural Simulation, *Proc. 11th Intl. Parallel Processing Symp.*, pp.39–44 (1997).
11) Reinhardt, S.K., Hill, M.D., Laurs, J.R., Lebeck, A.R., Lewis, J.C. and Wood, D.A.: The Wisconsin Wind Tunnel: Virtual Proto-

typing of Parallel Computers, *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp.48–60 (1993).

12) Rosenblum, M., Herrod, S., Witchel, E. and Gupta, A.: Complete Computer System Simulation: The SimOS Approach, *IEEE Parallel and Distributed Technology*, Vol.3, No.4, pp.34–43 (1995).

13) Schnarr, E. and Larus, J.: Fast Out-Of-Order Processor Simulation Using Memoization, *Proc. 8th Intl. Conf. Architectural Support for Programming Languages and Operating Systems*, pp.283–294 (1998).

14) Sorin, D., Pai. V, Adve, S., Vernon, M. and Wood, D.: Analytic Evaluation of Sharedmemory Systems with ILP Processors, *Proc. 25th Intl. Symp. Computer Architecture*, pp.380–391 (1998).

15) 高崎 透, 中田 尚, 中島 浩：高性能マイクロプロセッサシミュレータの並列化による高速化, 情報処理学会研究報告, 2004-ARC-159, pp.91–96 (2004).

(平成 17 年 1 月 24 日受付)
(平成 17 年 5 月 18 日採録)

高崎 透

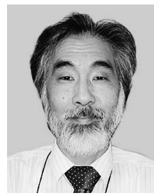
2004 年豊橋技術科学大学大学院工学研究科情報工学専攻修士課程修了．同年村田機械（株）入社．在学中はマイクロプロセッサシミュレータに関する研究に従事．

中田 尚（学生会員）

2004 年豊橋技術科学大学大学院工学研究科情報工学専攻修士課程修了．同年同大学院工学研究科電子･情報工学専攻博士後期課程入学．計算機アーキテクチャとシミュレーションに関する研究に従事．

津邑 公暁（正会員）

1998 年京都大学大学院工学研究科情報工学専攻修士課程修了．2001 年同大学院情報学研究科博士後期課程学修認定退学．同年同大学院経済学研究科助手．博士（情報学）．2004 年豊橋技術科学大学工学部助手．計算機アーキテクチャ，並列処理応用，脳型情報処理等に関する研究に従事．電子情報通信学会，人工知能学会，日本神経回路学会各会員．



中島 浩（正会員）

1981 年京都大学大学院工学研究科情報工学専攻修士課程修了．同年三菱電機（株）入社．推論マシンの研究開発に従事．1992 年京都大学工学部助教授．1997 年豊橋技術科学大学教授．並列計算機のアーキテクチャ等並列処理に関する研究に従事．工学博士．1988 年元岡賞，1993 年坂井記念特別賞受賞．情報処理学会計算機アーキテクチャ研究会主査，同論文誌：コンピューティングシステム編集委員長等を歴任．IEEE-CS，ACM，ALP，TUG 各会員．