2005－ARC－161　(2)
2005／1／21

高性能マイクロプロセッサシミュレータの時分割並列処理による高速化

高崎　透[†]　中田　尚[†]　中島　浩[†]

マイクロプロセッサは集積回路技術の進歩に伴い高度化・複雑化している．高度なプロセッサの性能検証には cycle accurate なシミュレータが不可欠であるが，現状のシミュレータは一般に低速であり，研究・開発の効率化にあたって障害となる．

本論文では，マイクロプロセッサシミュレーションを並列に行うことで高速化を図る．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，シミュレートされたプロセッサのマシン状態を分割点で一致させること，もしくは，分割区間のシミュレーション履歴を一致させることにより，精度を落とすことなく高速化を行った．状態および履歴を一致させるためには，パイプラインが分岐予測ミス時に空，あるいはそれに近い状態になることと，キャッシュの各セットは一定数のアクセスにより一定の状態となることを利用した．

並列シミュレータを実装し 8 並列で評価を行ったところ，SPEC CPU95 で最大 2.7 倍（su2cor），平均 1.9 倍の高速化を達成した．

High Speed Simulation of High Performance Processor by Parallel Processing

Toru Takasaki,[†] Takashi Nakada[†] and Hiroshi Nakashima[†]

Microprocessor simulation is indispensable for hardware systems design. In systems design field, cycle accurate (or clock level) simulation of highly sophisticated microprocessor is required. However, existing simulatiors of out-of-order processors run programs thousands times as slow as actual hardware. 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.

Our parallel simulation is performed by decomposing the problem along time axis so that each simulator node works on a subdivided interval. 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 cache access trace at the end of the simulation interval.

We implemented this parallel simulator on a 8-node PC cluster and evaluated its perfomance with SPEC CPU95 to find it achieves up to 2.7-fold speedup with su2cor and 1.9-fold in average.

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のシミュレータが不可欠である．しかし，現状のシミュレータは一般に低速であり，シミュレーションの実時間性能比（SD）は 1000~10000 となっている．SD がこのような大きな値になる要因は，命令実行順序を動的に定める命令スケジューラのシミュレーションに要する時間コストが大きいことである．命令の論理的な挙動のみをシミュレーションする場合では，SD は 10~100 のオーダとなっている．

マイクロプロセッサの動作を順々にシミュレーション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，ある時点以前のシミュレーション結果すべてに依存しているわけではないことに気づく．たとえば，パイプラインでは分岐予測ミスが発生する度にその状態は空，または空に近い状態となり，過去との依存関係が減少する．またキャッシュについて言えば，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，ある時点からある時点までの部分的な結果を知りたい場合には，それらの要素を始めから詳細にシミュレーション

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

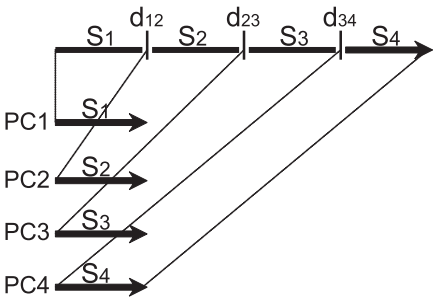

図 1　時間軸分割による並列化

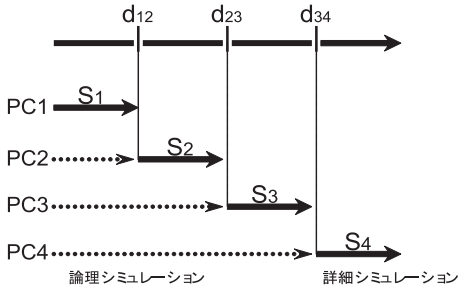

図 2　論理シミュレーション

しなくてもよい場合がある．このことを応用すると，マイクロプロセッサのシミュレーション過程を時間軸で複数の区間に分割し，それらを別々のノードで並列に実行することで高速にシミュレーションを行うことができる．また，時間軸上の途中から実行し始める分割区間シミュレーションが正しく行われたかどうかをチェックし，正しくない場合には再度シミュレーションを行うことで，精度を落とすことなく高速化する．

本研究では，高性能マイクロプロセッサシミュレータの高速化を目的とし，その方法として時間軸分割による並列シミュレーションを提案する．

以降，2 章で高速化手法を説明し，3 章で並列シミュレーション方法，4 章で実装について述べる．5 章で評価を行い，最後に 6 章でまとめる．

2. 高速化手法

シミュレータの高速化は，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により実現する（**図 1**）．以降，説明するにあたって図 1 に示すように分割点を d_{ij}，分割区間を S_i とする．最初の分割点，分割区間はそれぞれ，d_{12}，S_1 である．正確に全体のシミュレーション結果を求めるには，各分割区間が正しくシミュレートされなければならない．分割点 d_{ij} において，S_i のシミュレーション終了時のマシン状態と，S_j の開始時のマシン状態が一致していれば，S_j は正しくシミュレートされる．しかし，マシン状態がやや違っていても，その違いが S_j のシミュレーションに影響を及ぼさなければ，正しくシミュレートされる．よって，並列実行中にシミュレーションの履歴を保存しておくことで，正しくシミュレートされたかを検証する．

次節より，マシン状態の一致を導く手法，履歴を用いた検証方法の順に述べてゆく．説明するにあたり，本研究で高速化の対象としているパイプラインを含む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**，パイプラインを含まない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 状 態 一 致

2.1.1 メモリ・レジスタ・プログラムカウンタ

メモリ・レジスタ・プログラムカウンタの状態を一致させるためには，並列実行をスタートする前に，分割区間の開始点まで論理シミュレーションを行う．各ノードは論理シミュレーションを分割区間開始点まで行い，引続き詳細シミュレーションを行う（**図 2**）．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しないので，論理シミュレーションによって分割点での状態を正しくシミュレートできる．また，前章で述べた通り，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及ぼす影響が少ない．

2.1.2 パイプライン

パイプラインについては，分岐予測ミスを利用することと，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．分岐予測ミスが発生すると，パイプラインはフラッシュされ，分岐方向及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．したがって，分岐予測ミス点を分割点とすることで状態を一致させることができる．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，分岐予測ミスの度にパイプラインは空に近い状態となるため，分岐予測ミスが繰り返されることにより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若干異なっていても，分割点の前後の区間シミュレーションを重複して実行し，分岐予測ミスを何回か繰り返すことでパイプライン状態の一致を導くことができる．**図 3** に具体例を上げて説明する．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，区間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 では区間シミュレーションの終了

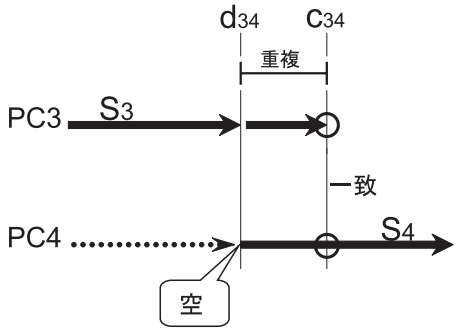

図 3 重複実行

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 で状態が異る．そこで，PC3 は分割点 d_{34} を越えて一定区間実行を続け，点 c_{34} で状態を比較する．重複区間 d_{34}-c_{34} では分岐予測ミスが発生していると予想されるので，状態は互いに近づいてゆき c_{34} では状態が一致していると考えられる．

2.1.3 キャッシュ

分割区間開始点には，キャッシュについての情報がない．よって，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し，その情報を得る．キャッシュを正確にシミュレーションするには，タイミング情報が必要であるので，この論理シミュレーションによるキャッシュ情報は完全でない．よって，パイプライン同様，前後のノードで重複実行し状態一致を目指す．しかし，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に過去との依存関係を大幅に断ち切るイベントがないうえ，その状態数も多いのでキャッシュブロック全て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，シミュレーション中にキャッシュアクセスとその結果を保存しておき，シミュレーション終了後に，それらを用いて正しく実行されたかを検証する．分割区間シミュレーションの検証の項目で詳しく述べる．

2.1.4 分岐予測器

分岐予測器についても，論理シミュレーションを分割点まで行い重複実行することで状態一致を図る．見切りで開始した場合には，そのシミュレーション結果を検証する．

2.2 分割区間シミュレーションの検証

分割点においてマシン状態が異っていても，その違いが分割区間シミュレーションに影響を及ぼさないことが確認できれば正しく実行されたことを保証できる．よって，シミュレーションの履歴を実行中に保存しておくことで，その影響を検証する．

2.2.1 キャッシュ

分割点において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 3 通りである．

- 相違のある箇所を参照しなかった．
- 相違のある箇所は，参照するまでの間にリプレイスされていた．
- 相違のある箇所を参照したが，ミスヒットであり，その際，ライトバックのあるなしが一致していた．

キャッシュアクセスとその結果（以下，キャッシュ参照履歴）を検証するにあたっては，履歴の保存コストを減らす必要がある．さらに，分割区間の履歴には，リファレンスとなる PC1 台でシミュレーションを行った場合の正しい履歴が存在しないので，履歴の比較方法についても考えなければならない．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，過去の状態に関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，それ以降は正しくシミュレーションされるので，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．例えば 128 セット，4 連想のキャッシュの場合，各セットについて言えば，異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．

次に履歴のチェック方法を述べる．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，直接リファレンスと比較して確かめることができない．よって，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・ミスが同じになるか，また，ミスの場合，追い出されるブロックにより，ライトバックが行われるかについてチェックする．例えば，**図 4** では分割区間 S_4 のあるセットついて，異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，b0008c60:ミス，c0005c60:ミス:write-back，d0004c60:ヒットとなっているので，それを S_3 終了時のキャッシュに適用し，その結果が同じになるかどうかをチェックする． 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 が引続いてシミュレーションを行っても，同様

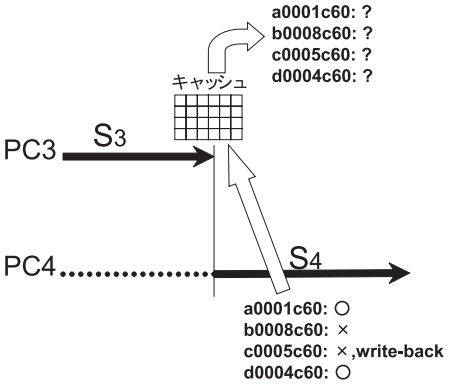

図 4　キャッシュ参照履歴の比較

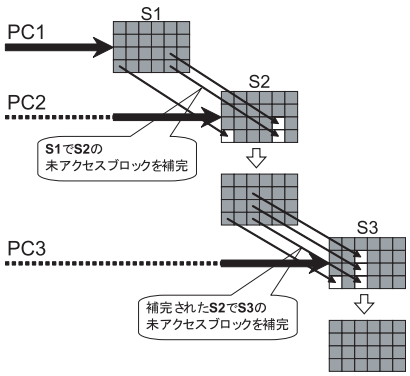

図 5　キャッシュの補完

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言える．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．また，分割区間シミュレーション中にアクセスされることのなかったキャッシュブロックについては，開始時の不確かなキャッシュを区間シミュレーション終了時まで保持するため，区間終了時に正しいキャッシュがあるか分からない．正しいキャッシュが分からないとこの検証方法は使えないが，未アクセスのキャッシュブロックについて，例えば，S_2 終了時のキャッシュであれば，正しい S_1 終了時のキャッシュを用いて，S_2 終了時のキャッシュを補完することができる．S_2 終了時のキャッシュが補完されれば，S_3 終了時のキャッシュも補完できるので，以降の分割区間においても，履歴のチェックと補完を行いながら検証を進めて行く．(**図 5**)．

2.2.2　分岐予測器

分岐予測器については、論理シミュレーション及び重複実行によって状態が一致する可能性が高いが，リターンアドレスについては分割点でやや異る場合がある．よってそ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．またキャッシュ同様，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．

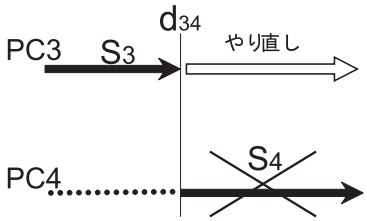

図 6　区間シミュレーションのやり直し

3.　並列シミュレーション方法

並列シミュレーションは，分割区間の並列実行フェーズとそれら分割シミュレーションの結果を比較・統合する検証フェーズによって構成される．

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．

状態一致のみで並列シミュレーションを行う場合について説明する．並列シミュレーションにおいては，間違った区間は，やり直しのため再度シミュレーションする必要がある．**図 6** を具体例として説明すると，分割点 d_{34} において，状態が不一致であったので，分割区間 S_4 は使えない．よって再度 S_4 のシミュレーションを行う必要がある．PC4 が再びやり直すには，その状態を巻戻す必要がある．そのため，S_4 の再実行は PC3 がシミュレーションを引続き行うことで対処する．

全体の並列シミュレーションの具体例を**図 7** に示す．図では各ノードの区間シミュレーション終了後，それぞれの状態の比較した結果，d_{23} と d_{45} で不一致であった場合を表している．この際，S_3 と S_5 のやり直しのための再実行は，同時に行うことができる．再実行の結果，次の区間でマシン状態が一致すれば，S_1- S_2 - S_3' - S_4 - S_5' - S_6 というパスを通って，並列シミュレーションが終了する．

図 7 について履歴による比較を導入すると，S_5 の再実行は，S_3 の再実行が終了するまで，S_4 終了時の正しいキャッシュが分からず開始できなくなる．なぜなら，履歴を用いて検証を行う場合は，比較・補完を

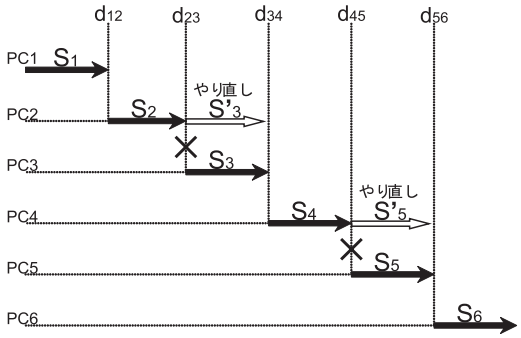

図 7 状態比較による並列シミュレーション

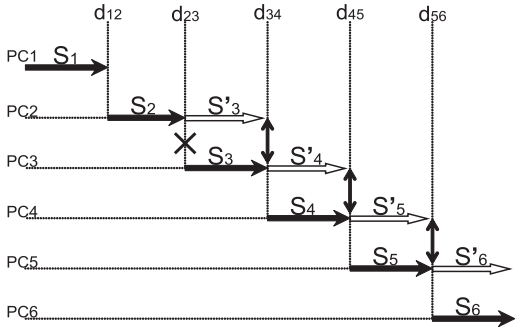

図 8 履歴比較を用いた並列シミュレーション

S_1 から順に行うからである．

S_4 終了時のキャッシュが不確かなまま S'_5 を S'_3 と同時に行い，再実行終了時に S'_3 より補完された S_4 のキャッシュを用いて S'_5 の検証を行ってもよいが，S'_5 が失敗したときに PC4 は d_{56} まで進んでいるため，PC2 が d_{56} までやり直しを行わなければならなくなり，大きなペナルティを背負うことになる．

そこで，履歴を導入するにあたっては**図 8** の様な方法を用いる．この方法では，S_3 のシミュレーションが失敗した場合において，PC2 がやり直しを行うことは，状態比較についての並列シミュレーション法と同じであるが，再実行を PC2 が行う際に，それ以降の他 PC が，次の区間を実行する．そして，比較については分割区間 S'_3:S'_4，S'_4:S'_5 の順に行ってゆく．こうすることで，検証をシリアルに行うことについては変わらないが，分割区間シミュレーションが失敗した区間長が，各ノードについての重複区間長に追加されるため，再実行後の各シミュレーションについての成功率が飛躍的に増加する（補足であるが，図には最初に行う一定区間の重複を表示していない）．例では S_1 - S_2 - S'_3 - S'_4 - S'_5 - S'_6 のパスを通って，並列シミュレーションが終了している．すなわち，分割区間が失敗する度に比較点が右にシフトし，各ノードの重複にあてる区間長が増加する．

4. 実　　装

並列シミュレータは，SimpleScalar Tool Set Version 3.0 を並列化することで実装した．論理シミュレーションには，sim-cache, sim-bpred を合わせて用いた．詳細シミュレーションは，高速化対象の sim-outorder となる．並列実行には MPICH 1.2.6 を用いた．

5. 評　　価

5.1 予 備 評 価

まず，予備評価として 一定区間の重複実行を行った場合のパイプラインとキャッシュの一致率，及びキャッシュ参照履歴の一致率について調査を行った．デフォルト設定の Simple Scalar Tool Set Version 3.0 を評価モデルとした．評価プログラムには SPEC CPU95 を用いた．

5.1.1 パイプライン

任意の時点でパイプラインを空にし，その影響が消えるまでの命令数を調査した．約 10000 箇所について実験したところ 1000 命令で 99.9%影響が消えていることが分かった．よって，パイプラインについては，キャッシュ等，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，その状態及び履歴を一致させることが重要になる．

5.1.2 キャッシュ

キャッシュについて 1000 命令後の一致率を調査した結果，その一致率は，平均 il1:23%，dl1:62%，ul2:34%であり，キャッシュはわずかな重複では一致しないと言える．よって，並列シミュレーションでは，重複実行により状態を近付けた上で，見切りでスタートすることになる．よって，履歴の一致率について調査を行った．調査にあたっては，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と想定し，1/16 の分割区間の開始点を全命令区間にわたり均等に約 90 箇所とり，開始点でのキャッシュと，分割区間におけるキャッシュ参照履歴の一致率を調査した．

その結果，重複後のキャッシュ一致率は，平均 il1:53%，dl1:98%，ul2:62%まで上昇し，キャッシュ参照履歴一致率が平均 75.5%であることが分かった．よって，履歴を用いることで並列シミュレーションを行うことができる．

5.2 並列シミュレーション結果

実装した並列シミュレータを用いて，分割数を 8，重複実行を分割区間の 10%とし，並列シミュレーション

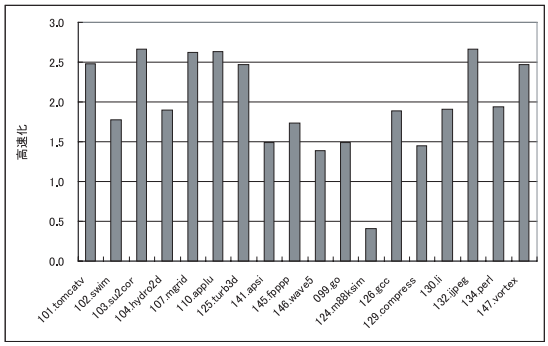

図 9 高速化率

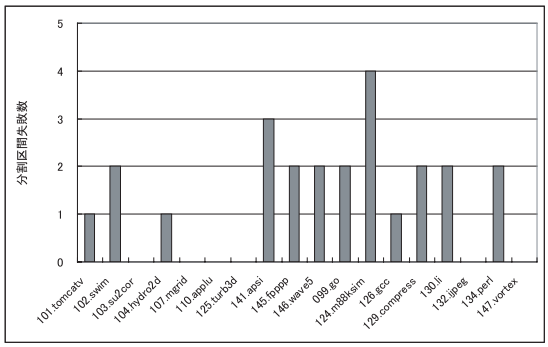

図 10 分割区間失敗回数

を行った．評価環境は，OS:Redhat Linux7.2, CPU:Intel PentiumIII/866MHz, Mem:512MB, N/W:Gigabit Ethernet のクラスタで 8 ノードを用いた．高速化結果と分割区間の失敗回数を，それぞれ，**図 9**，**図 10** に示す．今回，実装のベースに用いた Simple Scalar Tool Set においては，論理シミュレーションと詳細シミュレーションの速度比が約 1∶5 となっているため，失敗なしで実行できた場合の最終区間担当ノードの実行時間は，全体のシミュレーションの 3/10（= (1/8)+(7/8)×(1/5)）となり，高速化の上限は約 3.3 倍となる．

並列シミュレーションを行った結果，su2cor で最大 2.7 倍，平均して 1.9 倍の高速化を達成した．m88ksim では，本来の実行時間が短いことと，分割区間失敗数が 4 と多いため高速化はできなかった．しかし，並列シミュレーションでは，プログラムの命令数（実行時間）が多いほど，利用価値が高く，分割区間の成功率も増加すると考えられるため，tomcatv など他の SPEC において高速化を行えたことでその有用性を実証することができた．

6. 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．
- 分割シミュレーションが正しくない場合には，前区間担当のノードが，後区間のやり直しを行う．その際，他ノードも後区間をシミュレーションし，統合点をシフトすることでやり直し区間を重複実行区間にあてる．

8 分割で並列シミュレーションを行った結果，SPEC95 において最大 2.7 倍，平均 1.9 倍の高速化を達成した．なお高速化率が十分ではない最大の要因は，Simple Scalar の論理シミュレーションが低速であることである．我々の別の研究では，sim-fast を 10 倍以上高速化できることが明らかになっている．これを適用して論理シミュレーションを 10 倍高速化できれば，高速化率の上限が 3.3 倍から 7 倍に向上し，それに応じて実際の高速化率も向上することが期待できる．

現在の PC8 台で 8 分割を行う方法では，分割シミュレーションが失敗した場合にやり直しを行う区間が長くペナルティが大きい．また，一度にシミュレーションする区間が長いと，ベンチマークの特性に影響を受けやすいので，PC 台数よりも多数に分割する細分割シミュレータを実装し評価を行う予定である．

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) 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.SIG06(ACS6), pp.54-65, (2004)
3) 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)