高速マイクロプロセッサシミュレータ BurstScalar の設計と実装

中田 尚[†] 中島 浩[†]

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，現状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．そこで，我々はスケジューリング計算に計算再利用技術を適用した高速シミュレータ BurstScalar を提案する．高性能マイクロプロセッサのシミュレーションでは，スケジューリング計算が最も時間コストが大きい．一方，スケジューリング計算には局所性がある．たとえば最内ループでは少数のスケジューリングパターンが繰り返されることが予想できる．そこで，多数回繰り返されるスケジューリングパターンを検出した場合に，スケジューリング計算の結果を保存することにより，それ以降の同じ結果になる計算を省略し高速化できる．SPEC CPU95 ベンチマークを用いて評価を行った結果，シミュレーション速度が SPECfp で最大 3.7 倍，平均 3.0 倍，SPECint で最大 2.5 倍，平均 1.7 倍の向上が確認できた．

Design and Implementation of a High Speed Microprocessor Simulator BurstScalar

Takashi Nakada[†] 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 of highly sophisticated microprocessor is required. However, existing simulators of out-of-order processors run programs thousands of times slower than actual hardware. In this paper, we propose a high speed simulation method for high performance out-of-order microprocessors. Our primary contribution is the computation reuse to the expensive process of simulating an out-of-order microarchitecture. We record the instruction sequence of a loop together with its behavior and the microarchitecture states resulted from the sequence. When we find a recorded state in the out-of-order microarchitecture simulation, we skip the simulation reusing the state until we encounter a sequence unseen previously. We evaluated the implementation using the SPEC CPU95 benchmarks to find our technique achieves up to 3.7 and 2.5-fold speedups for SPECfp and SPECint respectively, and 3.0 and 1.7 in average.

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．また，マイクロプロセッサ組込機器などにも高度なマイクロプロセッサを搭載するようになると予想される．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，現状のシミュレータは一般に低速であり，最も簡単なユニプロセッサのアーキテクチャ・シミュレーションでさえ実時間性能比（SD: slowdown）は 1000〜10000 となっている．また，高度なワークロードやマルチプロセッサを含む複雑なシステムのシミュレーションではこの数倍〜数十倍の時間を要するため，研究・開発の効率化の大きな障害になっている．

高性能マイクロプロセッサのためのアーキテクチャ・シミュレーションの SD が 1000〜10000 という大きい値になる要因は，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にある．実際，命令の論理的な挙動のみのシミュレーションであれば SD は 10〜100 のオーダであり，命令スケジューラの計算量が実行時間の多くを占めている．一方，ワークロード・プログラムの実行過程には局所性があり，スケジューリング計算にも局

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

所性がある．たとえば，最内ループの実行時にはごく少数のパターンのスケジューリングが繰り返し行われると予想することができる．

そこで，我々が提案する高速シミュレータ BurstScalar では，スケジューリング計算に計算再利用技術を適用し，プロセッサの内部状態と入力（命令流）の繰返しを検出して，同一結果をもたらすスケジューリング計算を削除することで高速化を行う．また，高速化の効果を SPEC CPU95 ベンチマークを用いて測定した．

以下，2 章では関連研究について述べる．3 章では提案する高速化手法の概要を説明し，4 章で設計，5 章で実装について説明する．6 章で提案手法の評価結果を述べ，最後に 7 章でまとめを行う．

2. 関連研究

シミュレータは計算機工学にとって欠かせないツールであり，これまでにさまざまな研究が行われている．

たとえば，精度を犠牲にして超高速のシミュレーションを実現した技術の一例が，ダイナミック・バイナリ変換である．この高速 ISA エミュレーション手法は，ターゲット ISA の命令シーケンスを，それと等価なホスト ISA の命令シーケンスに変換するものである．変換済みコードをキャッシュして再利用することによって，エミュレータは，従来の命令インタプリタのフェッチおよびデコードによるオーバヘッドを大幅に削減できる．この手法の初期の実装には，Shade [1] がある．

Embra [2] では，特権命令，仮想・物理アドレス変換，例外処理，および割込み処理をエミュレートすることにより，OS の実行を完全にシミュレートできるまでに，ダイナミック・バイナリ変換が拡張された．

しかし，これらの高速 ISA エミュレータでは，プロセッサのパフォーマンスをクロック・サイクルのレベルまで細かく予測することはできず，我々の目標とする正確な性能評価には利用できない．また，バイナリ変換では命令シーケンスを変換するための変換テーブルを作成しなければならない．これはそれぞれの ISA が 1 対 1 に対応するとは限らず，作成が困難な場合もある．しかも，ターゲット，ホストのどちらかを変更するたびに命令シーケンスを変換するための変換テーブルを作成し直さなければならず，可搬性に欠ける．

シミュレーション精度や可搬性が優先される場合は，SimpleScalar [3] や RSIM [4] などのシミュレータを用いる．これらはバイナリ変換を使わないので，ホストを変更する場合もシミュレータを再コンパイルするだけでよい．しかし，この精度と可搬性のために，シミュレーション速度が犠牲になっている．たとえば，SimpleScalar で詳細なシミュレーションを行うと SD は 1,000 以上になる．

これらの数字から分かるとおり，シミュレーション速度，精度，可搬性の要素のうちどれかについて妥協しなければならず，SimpleScalar や RSIM は Shade や Embra とお互いに対極に位置している．速度，精度，可搬性のどこに妥協点を見出すかには，いろいろな選択肢がある．

シミュレーションの精度を一定の範囲で犠牲にしたうえで，処理を簡略化することによって高速化する手法がある．これらはいずれも精度と性能がトレードオフの関係にあり，たとえば DirectRSIM [5] は誤差が 1〜2%と小さいが SD は 1000 以上と大きく，FastILP [6] は SD が 100 程度と高速であるが誤差が 10%と大きい．

SimOS [7] は複数のシミュレーションモデルを持ち，必要に応じて切り替えてシミュレーションを行うことができる．たとえば，「詳細なシミュレーションを行いたい部分は精度重視，そうでない部分は速度重視」とすることができる．

また，FastSim [8] は，バイナリ変換と memoization と呼ばれるシミュレーション結果のキャッシュを使うことにより精度を落とさずに高速化を実現している．これにより，SD が 190〜360 で out-of-order プロセッサのマイクロアーキテクチャをシミュレーションできる．

次に SimpleScalar と FastSim について詳しく説明する．

SimpleScalar ツールセットを利用すると，複雑なマイクロアーキテクチャを迅速に記述して，詳細なシミュレーションを行うことができる．SimpleScalar は，シミュレーションのための高い柔軟性を備えており，さまざまなパラメータを簡単に変更することができる．そのため，多くのマイクロアーキテクチャの研究グループで幅広く使用されている．重大な問題点の 1 つは速度が遅いことである．

FastSim は高速な out-of-order シミュレータであり，SimpleScalar と比較して 5〜8 倍高速である．

これはバイナリ変換と memoization の 2 つの技術を使うことにより実現している．しかし，バイナリ変換を用いるので可搬性が犠牲になっている．

memoization ではプロセッサの状態とパイプラインシミュレーションの結果を記録する．そして，以前に記録された状態を再度シミュレーションする場合に以前の計算の結果を利用し，計算を省略する．この省略による誤差は発生しない．

FastSim では，シミュレーションの高速化のために

分岐またはロードストアのたびにプロセッサ状態を保存している．しかし，この方法では保存しなければならない状態の数が膨大になるという問題がある．

3. 設計方針

3.1 概　　要

本研究の高速シミュレータ BurstScalar では，最悪性能解析などへの応用にも利用できるように精度を最優先とし，精度を落としての高速化は考えない．つまり，高速化を行う前とまったく同じシミュレーション結果が得られることとする．また，可搬性を持たせるためにバイナリ変換は使わない．

そこで，スケジューリング計算を省略することにより高速化を目指す．マイクロプロセッサのシミュレーションではパイプラインシミュレーションが大部分の時間を占めており，最内ループのスケジューリングではごく少数のパターンの繰返しが起こることが予想できる．そこで，同一のシミュレーションパターンの繰返しを検出し，以降の計算を省略することにより高速化ができる．

しかし，FastSim のようにあらゆる分岐またはロードストア命令でプロセッサ状態を保存するのは効率的とはいえない．効率的に再利用を行うためには，事前にどの部分が多数回繰り返されるのかを知る必要がある．このためには，パイプラインシミュレーションを行う前に命令エミュレーションのみを行うことが有効である．これによりプログラム全体から再利用による高速化が効果的な部分を検出することができ，プロセッサ状態の保存回数を効果的に削減することが可能となる．

また，ループでは同一のシミュレーションパターンであっても，レジスタの内容やメモリアクセスのアドレスのようにループごとに変化する値もある．これらと，法則性のあるスケジューリングパターンを分離する必要がある．つまり，命令エミュレーションとパイプラインシミュレーションを分離し，パイプラインシミュレーションのみを省略することにより高速化する．

3.2 シミュレーションの分割

まず，シミュレーション全体を以下の 5 つの部分に分割して考える．

- 命令エミュレーション
- 分岐予測シミュレーション
- キャッシュシミュレーション
- アドレスコンフリクトの検出
- パイプラインシミュレーション

すると，本質的に out-of-order 実行が必要なものと in-order で実行可能なもの，命令列のみが必要なものと命令列以外にレジスタやメモリの内容が必要なものがあることが分かる．これらを**表 1** に示す．

表 1　out-of-order 実行とレジスタとメモリの内容の必要性
Table 1　Necessity of out-of-order execution and memory data.

	out-of-order	レジスタやメモリの内容
命令	×	○
分岐予測	○	○
キャッシュ	○	△
コンフリクト	○	△
パイプライン	○	×

○：必要，△：一部必要，×：不要

3.2.1 命令エミュレーション

命令エミュレーションは out-of-order の必要はなく in-order で実行することができる．本シミュレータではユニプロセッサをシミュレートするので，ロードストアを含むすべての命令はタイミングに依存せず実行できる．

また，命令エミュレーションを行うためにはレジスタやメモリの内容はすべて必要である．

3.2.2 分岐予測シミュレーション

分岐予測は過去のすべての実行に依存している．そして，out-of-order 実行では分岐予測器への lookup と update が別々のタイミングで行われ，先行する分岐命令に対する update が行われる前に，後続の分岐命令に対する lookup が発生する可能性もある．これらのタイミングはパイプラインのシミュレーションを行わなければ分からない．さらに，投機的な update を行う場合にはタイミングだけではなく update が行われるかどうかもパイプラインシミュレーションを行わなければ分からない．

つまり，分岐予測シミュレーションには out-of-order 実行の情報が必要である．

3.2.3 キャッシュシミュレーション

キャッシュシミュレーションでは，キャッシュアクセスのタイミングとアドレスからレイテンシを求める．

マイクロプロセッサ全体の正確なシミュレーションのためには，キャッシュの hit/miss だけでなくレイテンシが必要であり，ロードストア命令の実行されるタイミング情報が不可欠である．また，キャッシュへのアクセス順序自体も out-of-order 実行により入れ替わる可能性がある．したがって，キャッシュシミュレーションには out-of-order 実行が必要である．また，メモリアクセスのためには，アドレス計算のソースとなるレジスタやメモリの内容，あるいはアドレス計算結果が必要である．

3.2.4 アドレスコンフリクトの検出

out-of-order 実行では依存関係のない命令は順序を入れ替えて実行することができるが，ロード命令がストア命令を追い越すときは，先行するストア命令が書き込むアドレスとロード命令の読み込むアドレスが一致（コンフリクト）していない場合に限り，順序を入れ替えて実行することができる．このときに両者のアドレスが一致していると，順序の入替えができない．そのため，このコンフリクトの発生を検出する必要がある．アドレスコンフリクトが発生する条件は以下のいずれかの場合である．

- 先行するストア命令の書き込むアドレスが未確定．
- 先行する未完了のストア命令とロード命令のアドレスが一致．

また，アドレスコンフリクトの発生はロードストアアドレスに依存しているので，この検出を省略し高速化することは困難である．

以上により，アドレスコンフリクトの検出には out-of-order 実行が必要であり，キャッシュシミュレーションと同様にアドレス計算のソースあるいはその結果が必要であることが分かる．

3.2.5 パイプラインシミュレーション

パイプラインシミュレーションでは命令の詳細なスケジューリング計算を行う．したがって，パイプラインシミュレーションは当然 out-of-order 実行しなければならない．

また，パイプラインシミュレーションでは命令の種類と命令間の依存関係が重要である．そして，レジスタやメモリの内容によって変化する可能性のある依存関係はアドレスコンフリクトの検出によって解決される．つまり，パイプラインシミュレーションではレジスタやメモリの値は不要である．

3.3 実行の流れ

高速シミュレータは予備実行，先行実行，詳細実行，高速実行の 4 つのモジュールで構成する．構成を**図 1** に，分割方法を**表 2** に示す．

まず，予備実行では in-order でターゲット ISA に基づいて命令を実行する．その結果生成される命令流の中から多数回繰り返されるパターン（ループ）を検出・保存する．詳細実行と高速実行ではこの検出結果を利用することで高速化を行えるかどうかを判断する．これによって，繰返し回数が少なく高速化が期待できない部分でのオーバヘッドを削減することができる．

次に，詳細実行でメモリ系とパイプラインのシミュレーションを行う．ここでロードストアアドレスや分岐予測の結果が必要となるが，これらの値は先行実行を呼び出すことによって取得する．先行実行の詳細は 3.3.1 項で述べる．

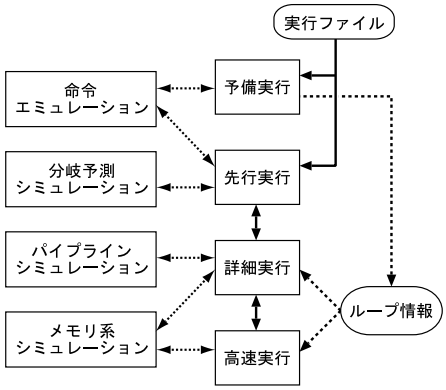

図 1 シミュレータの構成
Fig. 1 Structure of simulator.

表 2 シミュレーションの分割
Table 2 Division of simulations.

	予備実行	先行実行	詳細実行	高速実行
命令	○	○	×	×
分岐予測	×	○	×	×
キャッシュ	×	×	○	○
コンフリクト	×	×	○	○
パイプライン	×	×	○	×

○：シミュレーションする，×：シミュレーションしない

詳細実行では out-of-order 実行でシミュレーションを行う．ループのシミュレーションを行う場合，各ループの開始時点で内部状態を保存する．ループを 1 周するたびに以前に保存した状態と比較を行い，内部状態の繰返しが検出されると，詳細なパイプラインシミュレーションを省略する高速実行に移行する．

高速実行ではメモリ系のシミュレーションのみを行い，その結果が詳細実行での結果と一致している限りパイプラインシミュレーションをスキップする．一方メモリ系シミュレーションの結果が一致しなければ高速実行は中断し，詳細実行に戻ってパイプラインやメモリ系の未知の挙動をシミュレートする．

3.3.1 分岐予測とロードストアアドレス

詳細・高速実行でメモリ系のシミュレーションを行うためにはロードストアアドレスが必要である．また，分岐予測ミスパス上にロードストア命令が存在した場合には，それらの正確なシミュレーションのためにもロードストアアドレスが必要である．しかし，ロードストアアドレスをメモリやレジスタの内容を保持しない詳細・高速実行で計算することは不可能である．

そこで，ロードストアアドレスを取得するためには先行実行で命令エミュレーションを行う必要がある．

このとき，ミスパス上のロードストアアドレスを計算するためにはミスパスの命令エミュレーションも行わなければならない．

また，正確な分岐予測シミュレーションのためにはパイプラインシミュレーションが必要である．たとえば，分岐予測の投機的な update ではパイプラインシミュレーションを行うまで，update が行われるかどうかも分からない．

以上より，先行実行では詳細・高速実行から分岐予測のタイミングを取得しながら，分岐予測を含めた命令エミュレーションを行うこととする．これによって正確な分岐予測シミュレーションとロードストアアドレスの取得を確実かつ効率的に行うことができる．

4. 設　　計

4.1 予備実行

予備実行では命令エミュレーションを行い，ループの検出と記録を行う．

4.1.1 命令エミュレーション

本シミュレータではユニプロセッサをシミュレートするので，ロードストアを含むすべての命令はタイミングに依存せず実行できる．また，in-order での命令エミュレーションであるのでパイプラインのシミュレーションに比べて十分高速に実行することができる．

4.1.2 ループの検出と記録

高速化が可能となるのは多数回繰り返される命令列（ループ）であるので，これを検出する必要がある．

まず，命令エミュレーションで得られた命令トレースを，成立する後方分岐命令を境界として分割する．この分割された部分命令列を iteration と呼ぶことにする．それぞれの iteration に固有の ID を割り振り，その出現回数（ループ情報）を記録する．また，この ID のならびによって命令トレース全体を表現することができる．

例として，**図 2** の左のようなソースコードについて考える．まず，このコードをアセンブラにすると図 2 の右のようになる．この中で分岐命令は 6 行目と 8 行目に存在する．そこでこれらの分岐命令で命令列を区切ると，**図 3** の (A)，(B)，(C) の 3 種類の命令列から構成されており，(A) が 98 回連続し，その間に (B)，(C) が 1 回ずつ挟まれていることが分かる．つまり，このループ全体は (A) × 98,(B),(C),(A) × 98, . . . と表現できる．

4.2 先行実行

先行実行では命令エミュレーションと分岐予測を行う．

```
1:  /* foo.c */          1: ; foo.s
2: #define N 100         2:     li i, 0
3: for(i<N){             3: l1: li j, 0
4:   for(j<N){           4: l2: process
5:     process;          5:     inc j
6:   }                   6:     blt j, 100, l2
7: }                     7:     inc i
                         8:     blt i, 100, l1
```

図 2 多重ループの例
Fig. 2 Example of nested loop.

```
(A)
4: l2:process
5:    inc j
6:    blt j, 100, l2

(B)
4: l2:process
5:    inc j
6:    blt j, 100, l2
7:    inc i
8:    blt i, 100, l1

(C)
3: l1:li j, 0
4: l2:process
5:    inc j
6:    blt j, 100, l2
```

図 3 多重ループの分割
Fig. 3 Division of nested loop.

分岐予測を正確に行うためには lookup と update のタイミングが重要である．これらのタイミングは詳細・高速実行から供給される．

そして，先行実行は分岐予測結果とロードストアアドレスを詳細・高速実行に供給する．

4.3 詳細実行と高速実行

詳細実行と高速実行では，予備実行で保存したループ情報と先行実行での分岐予測結果，ロードストアアドレスを利用し，out-of-order シミュレーションを行う．

詳細実行では，パイプラインシミュレーションとメモリ系シミュレーションを行い，同時に高速実行可能なループの検出を行う．高速実行可能ループが検出されると高速実行に遷移し，パイプライシミュレーションを省略してメモリ系シミュレーションのみを行う．

4.3.1 高速実行可能なループの検出

高速実行を行うためには多数回繰り返される命令列（ループ）であることが必要である．したがって，繰返し回数が少ない部分はではそのまま out-of-order シミュレーションを行い，省略による高速化は行わない．

ループの部分は予備実行により事前に検出されている．しかし，すべてのループが高速実行可能なわけで

はない．高速実行が可能になる必要条件としては以下の 2 つがある．

- ループの開始時のプロセッサの内部状態が一致．
- iteration が一致．

まず，内部状態の一致を検出するために，iteration の先頭ではプロセッサの状態を保存する．保存する内容やデータ量はプロセッサの設計に依存するが，パイプライン中に含まれる命令や演算ユニットの残りレイテンシなどを保存する必要がある．

ループを 1 周するごとに，現在の状態を過去の状態と比較する．過去の状態と等しい場合は高速実行に移る．異なっていた場合は新たな状態として保存し詳細実行を続ける．

4.3.2 高速化の原理

4.3.1 項の条件に加え以下の条件が満たされる場合には，パイプラインシミュレーションを省略して高速化することができる．

- ループ中のメモリ系シミュレーションの結果が過去の結果と一致．
- ループ中の分岐予測の結果が過去の結果と一致．

4.3.3 項で述べるように，メモリ系シミュレーションや分岐予測のタイミングと結果は，過去の詳細実行によって保存されている．したがって高速実行時に行うこれらのシミュレーション結果が過去の結果と完全に一致すれば，パイプラインの振舞いも過去の結果と一致するのでパイプラインシミュレーションを省略した実行を継続する．逆に不一致が検出されると高速実行を継続することができないので，ループの先頭の状態まで戻って詳細実行に移行する．

この様子は状態遷移として表現することができる．「状態」はループ開始時のプロセッサの内部状態，「入力」は iteration の種類，ループ中のメモリ系シミュレーションの結果，分岐予測の結果，「出力」は所要クロック数などの統計情報となる．状態を 1 つ遷移することはループを 1 周することに対応する．

4.3.3 詳細実行結果の保存

ループ中の詳細実行では高速実行に必要なデータを保存する．保存しなければならないデータは以下のとおりである．

- メモリ系シミュレーションの結果
  - キャッシュアクセスのタイミングと結果
  - アドレスコンフリクトの検出のタイミングと結果
- 分岐予測のタイミングと結果
- ループ終了時の状態
- 統計情報

表 3 保存されるデータ例
Table 3 Example of saved data.

開始時の状態 /iteration ID	キャッシュアクセス（T/L） 1 回目	2 回目	3 回目	終了時の状態
$S_1/1$	1/1	2/1	4/1	S_2
$S_2/1$	1/1	2/1	4/1	S_3
$S_3/1$	1/1	3/1	5/1	S_1

T：アクセスタイミング，L：レイテンシ

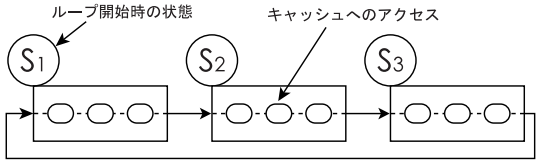

図 4 シミュレーション例
Fig. 4 Example of simulation.

キャッシュアクセスについては，ループの開始から順にすべてのキャッシュアクセスを記録する．アクセスタイミングはループ開始時を 0 とした相対サイクル数で記録する．アドレスコンフリクトの検出についても同様である．

また，ループ終了時には終了時の状態（つまり，次のループの開始時の状態）を保存する．同時に，ループ 1 周にかかったサイクル数や，ループ中のキャッシュ hit/miss 回数などの統計情報の保存も行う．

例を**表 3** に示す．この例はループ 3 周のシミュレーションを行い，それぞれのループの開始時の状態がすべて異なった（S_1，S_2，S_3）場合である．表の 1 行がループ 1 周に対応している．この例ではループ 1 周あたりのキャッシュアクセスが 3 回だが，これはループによって増減する．簡単のため iteration はすべて同一であるとした．また，ここでは省略しているが，分岐予測結果も同様に保存する．この例では，ループ 3 周で状態の繰り返し（$S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_1 \rightarrow \ldots$）が発生している．図で表すと**図 4** のようになる．

このデータ構造を生成するためには詳細実行を 3 周分行う必要があるが，その後はこのパターンに一致し続ける限り，パイプラインシミュレーションを省略した高速実行で行うことができる．

4.3.4 高速実行の成功と失敗

ループ開始時点のプロセッサの状態が一致していても，メモリ系シミュレーションや分岐予測の結果が一致しないと高速実行は失敗する．一致しなかった場合は新しいパターンとして保存する．

高速実行に成功する場合と失敗する場合の両方について，実行の流れを**表 4** と**図 5** の例を用いて説明する．簡単のため iteration はすべて同一であるとする．

表 4 保存データの分岐例
Table 4 Example of branch of saved data.

開始時の状態 /iteration ID /パターン番号	キャッシュアクセス（T/L） 1 回目	2 回目	3 回目	終了時の状態
$S_1/1/1$	1/1	2/1	4/1	S_2
$(S_1/1)/2$	(1/1)	2/10	?/?	$S_?$

T：アクセスタイミング，L：レイテンシ

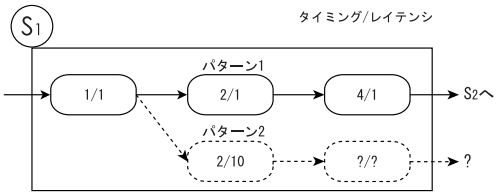

図 5 シミュレーションの分岐例
Fig. 5 Example of branched simulation.

ここでループ開始時の状態が S_1 に一致しているとする．リンクをたどると 1 番目のキャッシュアクセスのタイミング（この例では 1）でキャッシュシミュレーションを行う．この結果が保存されたもの（この例では 1）と同じ場合は次のリンクをたどる．同様に 2 番目，3 番目のキャッシュシミュレーションを行い，すべての結果が一致すれば高速実行は成功する．成功した場合は統計情報に差分を足し，状態を S_2 に更新する（図 5：パターン 1）．したがって前述の表 3 のように詳細実行結果が保存されていれば，$S_2 \rightarrow S_3 \rightarrow S_1 \rightarrow \ldots$ のように高速実行が継続する．

高速実行に失敗すると，パイプラインの状態が保存されているループの先頭に戻って詳細実行を行い，その結果を順に保存する．たとえば図 5 の例では，3 回目のキャッシュアクセス結果を含むパターン 2 のリンクと，遷移先であるループ終了時の状態が完全に生成される．

この終了時の状態が保存されているもの（たとえば S_1）と一致すると再び高速実行に遷移する．ここで，再び S_1 からの高速実行を行う際には，2 回目のキャッシュアクセスのレイテンシが 1 であっても 10 であっても保存された結果と一致するため，高速実行を継続することができる．このように高速実行が失敗するたびに状態遷移のパターンの数が増えていくので，失敗の割合はループを繰り返すに従って急速に減少する．

4.3.5 プロセッサ状態の参照

シミュレーション中にはプロセッサ状態を参照する必要性が生ずることがある．しかし，高速実行ではループ開始時のプロセッサ状態しか保持していないため，ループ途中の任意の状態を参照することは不可能である．そこで，参照の必要が発生した場合には高速実行を中断し，詳細実行に戻らなければならない．

ここで，高速実行で省略されるシミュレーションはこれまでに実行されたシミュレーションであることに注目すると，高速実行が成功した場合にどの程度のシミュレーション区間が省略されるかは事前に知ることができる．この省略される可能性のある区間に観察したい地点が含まれていることを検出したときに高速実行を停止することにより，任意の地点のプロセッサ状態を参照することが可能である．

4.4 メモリ管理

BurstScalar では予備実行で得られたループ情報を利用して効率的な状態保存を行っている．これにより，メモリ使用量を削減している．しかし，それでもメモリ不足が発生することがある．

そこで，メモリ不足が発生した場合にも予備実行で得られたループ情報を利用することにより，再利用の可能性が少ない部分からメモリの解放を行う．再利用の可能性が少ない部分としては以下のものがある．

- シミュレーションが完了したループ
- キャッシュ/分岐予測ミスが発生した状態遷移
- 繰返し回数が少ないループ

解放の対象となるのは状態遷移のみとし，プロセッサ状態は解放しないこととした．

メモリの解放を行っても，繰り返しメモリ不足が発生する場合は「繰返し回数が多い」と判断する閾値を上げていく．閾値を上げることにより解放対象が増加すると同時に保存対象も減少する．これにより，メモリを有効利用することが可能である．

5. 実　　装

高速シミュレータ BurstScalar の高速化手法は一般的なシミュレータに適用することができる．

本論文では，SimpleScalar に本手法を適用した．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．

5.1 予 備 実 行

予備実行は SimpleScalar の命令エミュレータである sim-fast にループの検出・保存機能を追加することで実装した．

5.2 先 行 実 行

先行実行は SimpleScalar の命令エミュレータである sim-fast に分岐予測とロードストアアドレスの保存機能を追加することで実装した．

分岐予測を正確に行うためには lookup/update の順序を保つことが重要である．そこで，先行実行中に

分岐命令を検出すると，lookup を行わずいったん詳細・高速実行へ制御を移す．詳細・高速実行が進むと分岐予測のタイミングが確定するので，再び先行実行を再開することができる．

5.3 詳細・高速実行

詳細・高速実行は SimpleScalar の out-of-order シミュレータである sim-outorder を拡張することで実装した．ただし，命令エミュレーションは行わず，命令トレースやロードストアアドレスは先行実行より取得する．

プロセッサ状態の保存は SimpleScalar 中のパイプラインに関するすべての変数をダンプすることによって行った．この方法ではデータ量が大きくなり，状態の比較にも多くのコストがかかるという問題がある．そこで，プロセッサ状態からハッシュ値を生成し同時に保存する．状態の比較時にはまずハッシュ値を比較することにより，状態の不一致を迅速に検出することができる．

6. 評　　価

6.1 評 価 条 件

高速シミュレータ BurstScalar の評価として，パイプラインシミュレーションの省略による高速化の効果を，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 Xeon（Dual 2.8 GHz，Memory 3 GB，Linux 2.4.20），SimpleScalar Tool Set Version 3.0c，gcc Version 2.95.3 を用いた．最適化オプションは SimpleScalar，BurstScalar ともに-O2 とした．評価モデルの構成を**表 5** に示す．model 2 は model 1 の機能ユニットなどを 2 倍，RUU（Register Update Unit）と LSQ（Load/Store Queue）を 8 倍にした構成である．それ以外のキャッシュや分岐予測器の構成は同一とした．命令セットアーキテクチャは PISA とした．

6.2 予　　測

6.2.1 ループの検出

高速化の効果を予測するために，SPEC CPU95 ベンチマークの各プログラムに含まれるループの出現回数を測定した．具体的には，プログラム全体で同一の iteration が何回現れるかを測定し，その回数を iteration の長さで重み付けをして出現回数別の割合を求めた．結果を**表 6** に示す．

6.2.2 高速化率の予測

高速化の効果はループの出現回数が多いほど効果が高いと予想することができる．そのほかには，メモリ系シミュレーションの一致確率も影響する．たとえば iteration 中に独立したロードストア命令が複数存在すると，すべてのキャッシュシミュレーション結果の一致する確率は相対的に低くなり，結果として高速化の効果が現れにくくなる．ここではループの出現回数についてのみ考えることにする．

SPECfp95 の結果（表 6 上半分）から多くのベンチマークではループの出現回数が多く，高速化の効果が期待できることが分かる．ただし，fpppp はループ出現回数が少なく高速化の効果があまり期待できない．

SPECint95 の結果（表 6 下半分）から jpeg や perl ではある程度の高速化の効果が期待できることが分か

表 5 プロセッサの構成
Table 5 Simulation model.

		model 1	model 2
命令発効幅		4	8
RUU		16	128
LSQ		8	64
メモリポート		2	4
機能ユニット	INT-ALU	4	8
	INT-MUL/DIV	1	2
	FP-ALU	4	8
	FP-MUL/DIV	1	2
分岐予測	予測方式	2 bit カウンタ/2 K エントリ	
	BTB	512 エントリ/4-way	
	RAS	8 エントリ	
キャッシュ	L1 命令	16 KB/32 B ライン/1-way	
	L1 データ	16 KB/32 B ライン/4-way	
	L2 統合	256 KB/64 B 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	
	ミスレイテンシ	30	

表 6 ループ出現回数の割合
Table 6 Frequency of loop.

出現回数	10^0–10^2	10^2–10^4	10^4–10^6	10^6–
101.tomcatv	0.0	0.3	16.0	83.7
102.swim	0.0	0.2	56.1	43.7
103.su2cor	0.0	0.1	8.9	91.0
104.hydro2d	0.0	0.5	8.4	91.1
107.mgrid	0.0	2.1	6.6	91.3
110.applu	0.2	2.6	97.2	0.0
125.turb3d	0.0	0.1	9.0	90.9
141.apsi	0.1	2.8	59.5	37.3
145.fpppp	3.0	58.7	38.3	0.0
146.wave5	0.0	0.2	28.2	71.6
099.go	3.1	45.4	51.5	0.0
124.m88ksim	0.5	1.7	97.8	0.0
126.gcc	1.4	40.7	55.4	2.6
129.compress	0.2	4.7	95.1	0.0
130.li	0.1	4.0	95.9	0.0
132.ijpeg	0.1	4.0	32.3	63.6
134.perl	0.0	1.9	30.5	67.6
147.vortex	0.1	5.0	52.8	42.1

（単位：%）

表 7 高速化率
Table 7 Speedup ratio.

	model 1			model 2		
	高速化あり	高速化なし	高速化率	高速化あり	高速化なし	高速化率
101.tomcatv	5082	19022	**3.74**	8767	22794	2.60
102.swim	307	1025	3.34	513	1429	2.79
103.su2cor	7539	24653	3.27	-	33106	-
104.hydro2d	2473	9127	3.69	-	10940	-
107.mgrid	4917	17074	3.47	9034	23001	2.55
110.applu	222	611	2.75	408	761	1.87
125.turb3d	6350	17791	2.80	10678	19129	1.79
141.apsi	1021	2824	2.77	1863	3774	2.03
145.fpppp	462	481	1.04	638	522	0.82
146.wave5	1290	3867	3.00	-	4637	-
平均			2.99			(2.06)
099.go	904	769	0.85	-	963	-
124.m88ksim	25.5	53.6	2.10	28.1	64.5	2.30
126.gcc	2204	1878	0.85	-	2160	-
129.compress	22.7	46.1	2.03	37.8	59.7	1.58
130.li	134	265	1.98	225	329	1.46
132.ijpeg	582	1460	**2.51**	1299	1719	1.32
134.perl	1827	3420	1.87	3238	3672	1.13
147.vortex	1719	3449	2.01	-	3551	-
平均			1.73			(1.45)

（単位：秒）

る．しかし，go や gcc では高速化の効果があまり期待できない．

6.3 結　　果

6.3.1 高速化の効果

BurstScalar のシミュレーション速度を SimpleScalar の sim-outorder と比較することにより高速化の効果を測定した．結果を**表 7** に示す．

model 1 の結果から，SPECfp95 では最大 3.7 倍（tomcatv）の高速化が達成できていることが分かる．また，その他のベンチマークでも 3 倍前後の高速化が達成できている．しかし，fpppp では予測どおり高速化の効果はわずかであった．SPECint95 では最大 2.5 倍（ijpeg）の高速化が達成できていることが分かる．だたし，予測どおり全体的に SPECfp95 と比べて高速化の効果は少ない．また，go，gcc では高速化ができず，逆に遅くなっている．

また，model 2 ではメモリ不足のため一部のベンチマークで高速実行が完了できなかった．これはプロセッサ構成の複雑化により，状態あたりの保存量が増えたことに加え，状態の一致率が低下により状態保存数が増えたため，メモリが不足したと考えられる．

そこで，高速化の効果を詳しく調べるために，高速実行の失敗率と状態保存の回数を測定した．結果を**表 8** に示す．ここで，失敗率は失敗回数/（成功回数＋失敗回数）である．

この結果から，高速化の効果があるベンチマークでは高速実行の失敗率が 1%以下であり，多数回の高速実行が行われていることが分かる．

逆に，高速化の効果がないベンチマークでは，失敗率が 6〜12%と高くなっている．失敗率が高くなったことにより，保存のオーバヘッドが高速化の効果を打ち消していると考えられる．

次に，状態の保存回数に注目すると，model 2 ではプロセッサ構成の複雑化により，model 1 と比較して状態の保存数が 2〜6 倍に増加していることが分かる．

6.3.2 メモリ使用量

プロセッサ状態の保存に必要なメモリ量は，構成の複雑さやパイプライン中に含まれる実行中の命令数によって変化する．1 状態の保存に必要なメモリ量は，model 1 ではおよそ 6〜8 KB，model 2 では 20〜30 KB であった．

状態遷移に必要なメモリ量はワークロードの複雑さによって変化する．

状態と状態遷移が使用するメモリの割合は，ワークロードによって変化するが，搭載メモリの 2/3 にあたる 2 GB を状態の保存に割り当てられたとすると，model 1 で 250,000 状態，model 2 で 67,000 状態の保存が可能である．表 8 の状態保存数はこの範囲に収まっている．また，実行が完了できなかったベンチマークではこの範囲を超えたためと考えられる．

現在の実装ではアドレスコンフリクトの検出のために LSQ を（未使用の部分を含めて）すべて保存して

表 8 高速実行と状態の保存
Table 8 Reuse and saved state.

	model 1				model 2			
	高速実行				高速実行			
	成功回数	失敗回数	失敗率	保存状態数	成功回数	失敗回数	失敗率	保存状態数
101.tomcatv	183821881	11979	0.0065%	1886	183250286	22326	0.0122%	5051
102.swim	18843751	3589	0.0190%	1545	19010483	50415	0.2645%	36025
103.su2cor	447275665	95660	0.0214%	3928	-	-	-	-
104.hydro2d	284284828	80928	0.0285%	6747	-	-	-	-
107.mgrid	181756201	17083	0.0094%	2743	181648293	22124	0.0122%	9573
110.applu	11985528	2545	0.0212%	1342	12226393	4489	0.0367%	9044
125.turb3d	559531136	6714	0.0012%	1890	568255275	28959	0.0051%	14774
141.apsi	70005595	22942	0.0328%	7337	69956795	34866	0.0498%	26297
145.fpppp	449714	42099	8.5600%	1617	279720	13850	4.7178%	2866
146.wave5	127787919	51265	0.0401%	3673	-	-	-	-
099.go	10994207	1449638	11.6494%	89517	-	-	-	-
124.m88ksim	3579704	2246	0.0627%	1510	3577890	2676	0.0747%	3045
126.gcc	35368658	2270859	6.0332%	243660	-	-	-	-
129.compress	2089461	7144	0.3407%	521	2087551	14811	0.7045%	2263
130.li	9677882	18184	0.1875%	3960	9632632	68348	0.7045%	11817
132.ijpeg	45814245	79343	0.1729%	5775	35513877	131198	0.3681%	35252
134.perl	128514941	490624	0.3803%	8855	115763424	174825	0.1508%	35825
147.vortex	64833735	433332	0.6639%	52006	-	-	-	-

表 9 実行時間の内訳（swim）
Table 9 Breakdown of execution time (swim).

	予備実行	先行実行	詳細実行	高速実行	メモリ	合計
高速化なし	-	-	923	-	102	1025
高速化あり	47	97	10	95	58	307

メモリ：メモリ系シミュレーション （単位：秒）

表 10 実行時間の内訳（go）
Table 10 Breakdown of execution time (go).

	予備実行	先行実行	詳細実行	高速実行	メモリ	合計
高速化なし	-	-	681	-	88	769
高速化あり	23	99	481	202	99	904

メモリ：メモリ系シミュレーション （単位：秒）

いる．しかし，実際のワークロードではつねに LSQ が埋まっているとは限らない．そこで，この実装を改善することによりメモリ消費量の削減ができる．これにより，今回実行が完了できなかったベンチマークも実行可能になることが期待できる．

6.3.3 実行時間の内訳

高速化ができている例として swim，できていない例として go に注目し，それぞれの実行時間の内訳を測定した．結果を**表 9**，**表 10** に示す．

この結果より，swim では詳細実行が 10 秒とほとんど行われておらず，省略による高速化ができていることが分かる．逆に go では詳細実行の時間があまり減っておらず，高速化ができていないことが分かる．これは，状態遷移が大きくなりすぎてメモリ不足が発生し，いったん保存したシミュレーション結果を部分的に破棄していることが原因と考えられる．

また，先行実行に注目すると予備実行と比較して実行時間が約 2〜4 倍かかっている．これは先行実行に必要以上のオーバヘッドが存在すると考えられ，調査・改善が必要である．

6.4 予備実行の効果

本シミュレータでは予備実行と先行実行で命令エミュレーションを 2 回行っている．そこで，予備実行がある場合とない場合を比較することにより予備実行の効果を測定した．プロセッサの構成には model 1 を用いた．

6.4.1 比 較 方 法

実際には予備実行を行わないプログラムは作成せず，予備実行を行うプログラムにいくつかの変更を加えることにより予備実行がない場合の結果を予測した．

予備実行の効果を測定するために以下の変更を加えた．

- iteration の出現頻度にかかわらず，高速実行を行う．
- 命令列の分割はすべての分岐命令とする．
  高速化を行うためには命令列の開始時に iteration ID を知る必要があり，予備実行なしで確実に実行されることが分かるのは次の分岐命令までである．
- 予備実行の実行時間分として，sim-fast での実行時間を引く．

表 11 状態の比較と保存
Table 11 Status comparison and save.

	予備実行あり		予備実行なし		
	比較回数	保存状態数	比較回数	保存状態数	状態削減率
101.tomcatv	24323	1886	32790	16096	88.3%
102.swim	6380	1545	13280	8706	82.3%
103.su2cor	138593	3928	136608	19343	79.7%
104.hydro2d	117211	6747	142541	57393	88.2%
107.mgrid	47071	2743	84781	17013	83.9%
110.applu	9568	1342	16677	4588	70.7%
125.turb3d	28684	1890	66582	11625	83.7%
141.apsi	63998	7337	124540	54182	86.5%
145.fpppp	221963	1617	30288	15116	89.3%
146.wave5	58111	3673	56713	19092	80.8%
099.go	5253852	89517	-	-	-
124.m88ksim	7203	1510	20033	8548	82.3%
126.gcc	14663789	243660	-	-	-
129.compress	10281	521	9726	3375	84.6%
130.li	31768	3960	59813	26492	85.1%
132.ijpeg	146752	5775	152050	32926	82.5%
134.perl	1148425	8855	197357	78670	88.7%
147.vortex	2101013	52006	-	-	-

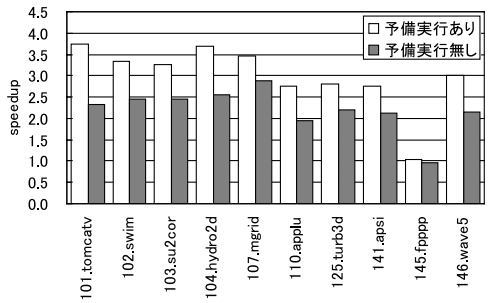

図 6 予備実行の効果（SPECfp95）
Fig. 6 Effect of preliminary execution (SPECfp95).

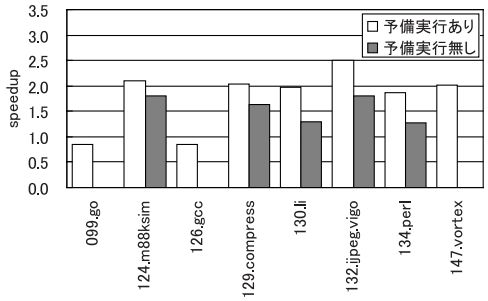

図 7 予備実行の効果（SPECint95）
Fig. 7 Effect of preliminary execution (SPECint95).

6.4.2 比較結果

SPECfp95 の結果を**図 6**，SPECint95 の結果を**図 7** に示す．この結果より，すべてのベンチマークにおいて予備実行の効果があることが分かる．

また，予備実行を行わないと一部のベンチマークでメモリ不足によりシミュレーションが完了できなかった．これは，iteration の出現頻度にかかわらず iteration 開始時にプロセッサ状態の保存を行ったこと，さらに，iteration が細かく分割されたことにより，状態の保存数が急激に増えたことが原因である．

そこで，予備実行を行った場合と行わなかった場合について，状態の比較・保存回数を測定した．結果を**表 11** に示す．この結果より，予備実行を行うことにより状態保存の回数を 1/3 以下に削減できていることが分かる．

また，状態の比較にかかる時間がシミュレーション全体に占める割合は，予備実行ありの go の場合でおよそ 3%（30 秒）である．

7. ま と め

本論文では，スケジューリング計算に計算再利用技術を適用した高速シミュレータ BurstScalar について述べた．

シミュレーション全体を命令エミュレーションのような省略不可能な部分とスケジューリング計算のような省略可能な部分に分割し，命令列とスケジューリング計算の繰返しを検出することで，同一の結果をもたらすシミュレーションを省略し高速化を行った．

SPEC CPU95 ベンチマークを用いた評価を行った結果，最大 3.7 倍の速度向上ができた．このことより，現実的なアプリケーションのシミュレーションを高速化可能であるといえる．

今後の課題としては，メモリ不足時にプロセッサ状態の保存数を効果的に削減することにより，さらなる高速化を目指すことがあげられる．

謝辞 本研究の一部は（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」および文部科学省 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」による．

参 考 文 献

1) Cmelik, B. and Keppel, D.: Shade: A Fast Instruction-Set Simulator for Execution Profiling, *ACM SIGMETRICS Performance Evaluation Review*, Vol.22, No.1, pp.128–137 (1994).
2) Witchel, E. and Rosenblum, M.: Embra: Fast and Flexible Machine Simulation, *Measurement and Modeling of Computer Systems*, pp.68–79 (1996).
3) Austin, T., Larson, E. and Ernst, D.: SimpleScalar: An Infrastructure for Computer System Modeling, *Computer*, Vol.35, No.2, pp.59–67 (2002).
4) Pai, V.S., Ranganathan, P. and Adve, S.V.: RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors, *3rd Workshop on Computer Architecture Education* (1997).
5) Durbhakula, M., Pai, V. and Adve, S.: Improving the Accuracy vs. Speed Tradeoff for Simulating Shared-Memory Multiprocessors with ILP Processors, *International Symposium on High Performance Computer Architecture* (*HPCA*), Orlando, FL, pp.23–32 (1999).
6) Sorin, D.J., Pai, V.S., Adve, S.V., Vernon, M.K. and Wood, D.A.: Analytic Evaluation of Shared-memory Systems with ILP Processors, *ISCA*, pp.380–391 (1998).
7) Rosenblum, M., Herrod, S.A., 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).
8) Schnarr, E. and Larus, J.: Fast Out-Of-Order Processor Simulation Using Memoization, *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.283–294 (1998).

(平成 15 年 10 月 6 日受付)
(平成 16 年 2 月 7 日採録)

中田 尚
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．

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．IEEE-CS，ACM，ALP，TUG 各会員．