高性能マイクロプロセッサの高速シミュレータの設計と実装

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．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る．

そこで，われわれはスケジューリング計算に計算再利用技術を適用したシミュレーション高速化を提案する．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もっと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は明らかである．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の最大 6.2 倍の向上が確認できた．

Design and Implementation of High Speed Simulator for High Performance Processor

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.

Here, we propose high speed simulation of high performance processor. 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 6.2-fold speed up.

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．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 となり，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．

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

そこで本研究では，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）の繰り返しを検出して，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．

これまでの研究により，単純なループであればパイプラインシミュレーションを 8 倍以上高速化できることがわかっているが，単純でないループに対応できないことが問題だった[1]．そこで，本稿では単純でないループも繰り返しと認識できるように，ループの検出方法の改良を行った．これにより一般的なプログラムのシミュレーションを高速化することができるようになった．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．最後に 6 章で提案手法の評価結果を述べる．

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，これまでに様々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．この手法の実装には，Shade シミュレータ[2] や Embra シミュレータ[3] がある．

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，バイナリ変換ではターゲット，ホストのどちらかを変更する度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[4] などのシミュレータを用いる．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．

また，FastSim[5] 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．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度にパイプライン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いけない状態の数が膨大になるという問題がある．

3. 高速シミュレーション

本研究のシミュレーションでは精度を最優先とし，精度を落としての高速化は考えない．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り返しが起こることは明らかである．そこで，同一のシミュレーションパターンの繰り返しを検出し，以降の計算を省略することにより高速化ができる．

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毎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．

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わかる．これらを**表 1** に示す．

4. 設計

4.1 実行の流れ

高速シミュレータは先行実行，詳細実行，高速実行の 3 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，先行実行では in-order でターゲット ISA に

表 1 out-of-order 実行とレジスタとメモリの内容の必要性

	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

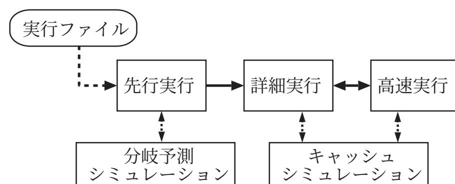

図 1　シミュレータの構成

表 2　シミュレーションの分割

	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

基づいて命令を実行する．その結果生成される命令流を保存し，保存と同時に命令流の中から多数回繰り返されるパターン（ループ）を検出する．またキャッシュシミュレーションとアドレスコンフリクト検出（以下，メモリ系シミュレーションと呼ぶ）に必要な，ロード・ストアのアドレスも命令流に付加して保存する．これらの結果は詳細実行と高速実行に供給される．

次に，この結果を用いて詳細実行でメモリ系とパイプラインシミュレーションを行う．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 周する度に以前に保存した状態と比較を行い，内部状態の繰り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．

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シミュレーションを行う．同時に，命令トレースの記録，ロードストアアドレスの記録，ループの検出も行う．

4.2.1　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．

```
1:  /* foo.c */
2: #define N 100
3: for(i<N){
4:   for(j<N){
5:     process;
6:   }
7: }
```

```
1: ; foo.s
2:     li i, 0
3: l1: li j, 0
4: l2: process
5:     inc j
6:     blt j, 100, l2
7:     inc i
8:     blt i, 100, l1
```

図 2　多重ループの例

(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
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

4.2.2　ループの検出と記録

これまでの実装では完全に同一の命令列が繰り返されているもののみをループとしていため，最内ループ中に分岐命令が含まれる場合や多重ループに対応できないという問題があった．そこで，今回はループの定義を同一の後方分岐で繰り返す命令列とする．

まず，ループの必要条件である同一の後方分岐の連続成立を検出する．同一の後方分岐が連続して成立していることを検出すると，その後方分岐命令を境界として命令列を分割する．この分割された部分命令列を iteraton と呼ぶことにする．それぞれの iteration に含まれる命令列はすべて同一の命令列の場合もあるが，一般には複数のパターンの繰り返しになる．そこで，それぞれのパターンに固有の ID を割り振り，この ID の並びによってループ全体を表現する．

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．この中で連続成立するのは 6 行目の分岐命令である．そこでこの分岐命令で命令列を区切ると，**図 3** の (A) と (B) の 2 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) が 1 回挟まれていることがわかる．つまり，このループ全体は (A)×98,(B),(A)×98,... と表現される．

また，ループ中の分岐予測が外れた場合は高速実行が不可能なループとして記録する．分岐予測が外れた場合の詳細は 4.4 節で述べる．

4.3　詳細実行と高速実行

詳細実行と高速実行では，先行実行で保存したトレー

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ループであることが必要である．したがって，ループで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．

ループの部分は先行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，ループの開始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．

ループを 1 周する毎に，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 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

4.3.3 で述べるように，メモリ系シミュレーションの結果は，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，パイプラインの振る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 の種類とループ中のメモリ系シミュレーションの結果,「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
- ループ終了時の状態
- 統計情報

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．

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**図 4** 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．

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 4:パターン 1）．

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

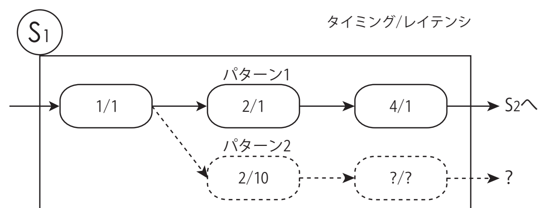

図 4　シミュレーションの分岐例

の結果を順に保存する．たとえば図 4 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．

このように高速実行が失敗する度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したがって急速に減少する．

4.4 分岐予測ミスの影響

分岐予測で予測ミスが発生すると，誤予測パス（実行すべきではない命令列）が実行される．誤予測パスの実行は予測ミスであることを検出したときに巻き戻され，プログラムの機能は正しく実行される．out-of-order 実行においては，この検出までの間に後続の命令が実行される可能性があり，どの命令が実行されるかは，正確なパイプラインシミュレーションを行わなければわからない．

誤予測パス中のメモリアクセスはキャッシュに影響を及ぼすので，正確にシミュレーションする必要がある．しかし，このメモリアクセスのアドレスは先行実行でのみ計算できる．したがって，予測ミスが検出された場合は先行実行と詳細実行は相互にデータを交換しながら動作する必要がある．

5. 実 装

実装は SimpleScalar を拡張することにより行った．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ションすることができる．

5.1 先 行 実 行

先行実行は SimpleScalar の命令エミュレータである sim-fast を拡張することで実装した．現在の実装では分岐予測ミスに対する処理が未実装であるため，分岐予測は必ず成功するものとした．

5.1.1 ロードストアアドレスの保存

ロードストアアドレスはプログラム中で実行されるロードストア命令すべてについて保存する必要がある．つまり，必要なメモリは実行時間に比例する．これは，SD=100 で 1GHz，アドレス幅 32bit，CPI=1，ロードストア命令が 4 命令に 1 回の割合であると仮定すると 10MB/s となる．

これでは，大規模なシミュレーションを行おうとすると全くメモリが足りない．そこで，ロードストアアドレスの記録用には固定的にメモリを割り当てる．割り当てられたメモリを使い切った場合には先行実行を中断する．

詳細･高速実行でロードストアアドレスが不足した場合には，詳細・高速実行を中断し先行実行を再開する．

今回は，ロードストアアドレスの記録用に約 100MB を割り当てた．

5.2 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．

5.2.1 プロセッサ状態の保存

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

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 2.8GHz, 2GB), SimpleScalar Tool Set Version 3.0c を用いた．評価モデルの演算ユニット数は INT-ALU が 4，INT-MUL/DIV が 1，FP-ALU が 4，FP-MUL/DIV が 1 とし，フェッチ幅と発行幅は 4 とした．

6.1 予 測

6.1.1 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．SPECint95 の結果を**表 3** に，SPECfp95 の結果を**表 4** に示す．

6.1.2 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出来る．その他には，メモリ系シミュレーションの一致確率も影響する．例えば 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．ただし，ここではループの出現回数についてのみ考える．

表 3 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わかる．ただし，fpppp はループ出現回数が少なく高速化の効果

表 3 SPECfp95 のループ出現回数の割合

出現回数	10^0–10^2	10^2–10^4	10^4–10^6	10^6–
101.tomcatv	0.1	0.6	2.9	96.4
102.swim	0.1	1.8	34.5	63.6
103.su2cor	0.2	0.2	11.5	88.1
104.hydro2d	0.1	0.6	5.4	93.9
107.mgrid	0.2	1.1	5.0	93.7
110.applu	0.6	22.4	77.0	0.0
125.turb3d	0.0	0.5	44.0	55.5
141.apsi	0.7	10.0	77.7	11.6
145.fpppp	35.6	64.1	0.3	0.0
146.wave5	0.0	1.5	36.2	62.3

（単位：%）

表 4 SPECint95 のループ出現回数の割合

出現回数	10^0–10^2	10^2–10^4	10^4–10^6	10^6–
099.go	93.5	4.0	2.5	0.0
124.m88ksim	1.5	3.2	95.3	0.0
126.gcc	74.3	16.3	7.2	2.2
129.compress	1.9	30.9	67.2	0.0
130.li	10.7	51.9	37.4	0.0
132.ijpeg	13.6	23.2	16.3	46.9
134.perl	58.3	13.1	19.4	9.2
147.vortex	4.9	28.7	64.9	1.5

（単位：%）

表 5 SPECfp95 の高速化率

	高速化あり	高速化なし	高速化率
101.tomcatv	3397	19022	5.60
102.swim	225	1025	4.56
103.su2cor	5556	24653	4.44
104.hydro2d	1463	9127	**6.24**
107.mgrid	3847	17074	4.44
110.applu	175	611	3.49
125.turb3d	4425	17791	4.02
141.apsi	931	2824	3.03
145.fpppp	443	481	1.09
146.wave5	912	3867	4.24

（単位：秒）

表 6 SPECint95 の高速化率

	高速化あり	高速化なし	高速化率
099.go	1236	769	0.62
124.m88ksim	9.7	53.6	**5.51**
126.gcc	1594	1878	1.18
129.compress	13.9	46.1	3.32
130.li	111	265	2.39
132.ijpeg	596	1460	2.45
134.perl	2695	3420	1.27
147.vortex	2924	3449	1.18

（単位：秒）

があまり期待できない．

表 4 から jpeg や m88ksim ではある程度の高速化の効果が期待できることがわかる．しかし，その他のベンチマークでは高速化の効果があまり期待できない．

6.2 結　　果

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．SPECint95 の結果を**表 5**，SPECfp95 の結果を**表 6** に示す．

この結果から，SPECfp95 では最大 6.2 倍 (hydro2d) の高速化が達成できていることがわかる．fpppp では予想通り高速化の効果が出ていない．

SPECint95 では m88ksim で 5.5 倍の高速化が達成できていることがわかる．しかし，その他のベンチマークでは予想通り高速化の効果が少なかった．

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シミュレーション高速化について述べた．

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り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 5.5 倍の速度向上ができた．

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) 中田尚, 大野和彦, 中島浩: 高性能マイクロプロセッサの高速シミュレーション, 先進的計算基盤システムシンポジウム SACSIS2003 論文集, pp. 89–96 (2003).
2) 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).
3) Witchel, E. and Rosenblum, M.: Embra: Fast and Flexible Machine Simulation, *Measurement and Modeling of Computer Systems*, pp. 68–79 (1996).
4) Burger, D. and Austin, T. M.: The SimpleScalar Tool Set, Version 2.0 (1997).
5) Schnarr, E. and Larus, J.: Fast Out-Of-Order Processor Simulation Using Memoization, *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 283–294 (1998).