計算機アーキテクチャ 149－27
（２００２．８．２３）

高性能マイクロプロセッサの高速シミュレーションの構想

中田　尚[†]　大野和彦[†]　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る．

そこで，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の構想を述べる．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，高速化する．

Toward High Speed Simulation of High Performance Processor

Takashi Nakada ,[†] Kazuhiko Ohno [†] and Hiroshi Nakashima [†]

Microarchitectural simulation is an essential tool in the research and design of processors, compilers, and other system software. However, existing simulators of out-of-order processors run programs thousands of times slower than actual hardware.

Here, we toward 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.

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 1000〜5000 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ることは明らかである．

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 1000〜5000 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，パイプラインスケジューラの計算量が実行時間のほとんどを占めている．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，パイプラインのスケジューリングパターンの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ことは明らかである．

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，スケジューラの内部状態と入力（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により高速化を図る．

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

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．これらの研究から，シミュレーション速度，精度，柔軟性の目標は，相容れないものであることがわかる．

たとえば，精度を犠牲にして超高速のシミュレーション速度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ーヘッドを大幅に削減できる．この手法の初期の実装については，Cmelik と Keppel が，Shade シミュレータ[1] に関する論文の中で説明している．

Embra シミュレータ[2] では，特権命令，仮想・物理アドレス変換，例外処理，および割り込み処理をエミュレートすることにより，完全な OS の実行を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．

これらの技術により，Shade や Embra などの高速 ISA エミュレータでは，シミュレートするワークロードの複雑さにもよるが，SD が 10〜40 でシミュレートが可能である．

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，バイナリ変換を行うためには命令シーケンスを変換するために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度に変換テーブルを作成し直さなければならず，柔軟性に欠ける．

シミュレーション精度や柔軟性が優先される場合は，通常 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，この精度と柔軟性のために，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には，実際のハードウェアの 1,000 倍以上の時間がかかる．

これらの数字からわかる通り，シミュレータは，シミュレーション速度，精度，柔軟性の要素のうちどれかを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，柔軟性のどこに妥協点を見い出すかには，いろいろな選択肢がある．

たとえば，SimOS[5] 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．

また，FastSim[6] は，memoization（結果をキャッシュすること）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 の時間でアウトオブオーダー・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．これは，SimpleScalar によるシミュレーションに比べて，8〜5 倍の高速化である．

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

- SimpleScalar
  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．これは，さまざまな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ことからも明らかである．
  唯一の問題点は速度が遅いことである．
- FastSim
  FastSim は高速な out-of-order シミュレータであり，SimpleScalar と比較して 8〜5 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高速化を実現している．
  まず，バイナリ変換を用いるので Shade 等と同様に柔軟性が犠牲になっている．
  次に，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，分岐またはロードストアの度にパ

表 1 各種シミュレータの比較

	Shade, Embra	SimpleScalar	FastSim
精度	×	○	○
速度	○	×	△
柔軟性	×	○	×

イプライン状態を保存している．しかし，この方法では分岐またはロードストアの度に現在の状態が過去の状態と一致するかどうかを調べなければならない．

また，マイクロプロセッサの高性能化により，単位時間あたりのロードストア回数も増え続けている．したがって，FastSim の方法では単位時間あたりの状態の保存の回数も増え，これが性能の障害となることが予想される．

これらのシミュレータの比較を**表 1** に示す．

3. 高速 out-of-order シミュレーション

本構想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，柔軟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．そこで，同一のシミュレーションパターンを検出しこの計算を省略することにより高速化が可能である．

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，カウンタ値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，命令のエミュレーションとパイプラインのシミュレーションを分離し，これらを必要に応じて使い分けることにより高速化する．

3.1 シミュレーションの分割

まず，シミュレーション全体を命令エミュレーション，キャッシュシミュレーション，分岐予測シミュレーション，パイプラインシミュレーションに分割して考える．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 2** に示す．

3.1.1 命令エミュレーション

命令エミュレーションは out-of-order ではなく in-order で実行することが可能である．

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，レジスタやメモリの内容はすべて必要である．

表 2 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	○	○
パイプライン	○	×

○：必要，△：一部必要，×：不要

3.1.2 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．しかし，分岐予測のシミュレーションにはタイミング情報は基本的に不要である．したがって，分岐予測シミュレーションは基本的に in-order で実行できる．

しかし，分岐予測ミスが発生すると本来実行されないはずの命令列が実行されることになる．この命令列に分岐命令があれば，それ以降の分岐予測の挙動が変化する．同様のことがキャッシュシミュレーションにも当てはまる．

また，誤った投機的な実行がどこまで実行されるかは，パイプラインのシミュレーションを行わなければわからない．

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．

3.1.3 キャッシュシミュレーション

最内ループではパイプラインスケジューリングと同様にキャッシュもごく少数のパターンの繰り返しが発生している．そこで，このパターンを検出することによりキャッシュシミュレーションを省略し高速化することが考えられる．

しかし，キャッシュシミュレーションは分岐予測シミュレーションと同様に過去すべての実行に依存しており，キャッシュのシミュレーションを省略することはできない．つまり，仮にループ中のキャッシュシミュレーションを省略することができたとしても，ループを抜けた後のキャッシュシミュレーションを正確に行うことができなくなる．

また，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．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のアドレスの計算にはレジスタやメモリの内容が必要である．

3.1.4 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は当然 out-of-order 実行しなければならない．

しかし，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り，レジスタやメ

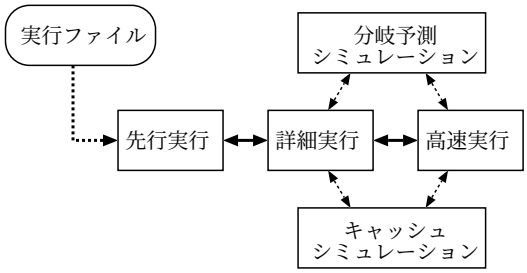

図 1 シミュレータの構成

表 3 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	×	○	○
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

モリの内容は不要である．なぜなら，値によってパイプラインは変化しないからである．

3.2 構　　成

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 3** に示す．

先行実行は in-order 実行で対象プロセッサの ISA に基づいて命令を実行する．そして，その結果生成される命令流（ロードストアアドレスなどのハードウェア・イベントを含む）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を検出する．これらの結果は詳細実行と高速実行に供給される．

詳細実行は out-of-order 実行でシミュレーションを行う．シミュレーション中で多数回繰り返されるパターンのスケジューリング計算を行う場合，各繰り返しの開始時点で内部状態を保存する．この保存された内部状態列の繰り返しが検出されると，高速実行に移行し命令流の繰り返し終了までスケジューリング計算をスキップし，繰り返しあたりのクロック数と繰り返し数から総クロック数を得ることが可能である．

3.3 実行の流れ

マイクロプロセッサシミュレーションは，まず先行実行を行う．

次に先行実行で取得したデータを元に詳細実行を行う．詳細実行ではループの部分をシミュレーションする時に，ループ開始時点のマイクロプロセッサの状態を保存する．ループを 1 周する度に以前に保存した状態と比較を行う．一致した場合には詳細なパイプラインシミュレータを行わない高速実行に移行する．この高速実行では詳細実行と全く同じ結果を得ることができる．

高速実行は，異なるキャッシュの振る舞い，あるいは異なるコントロール・フローによって終了する．これらの変化は常にチェックされ，振る舞いの変化が検出されると早送りは終了する．その結果，詳細実行が新たなシミュレーションを開始する．

4. 詳　　細

4.1 先 行 実 行

先行実行では命令エミュレーション，命令トレースの記録，ロードストアアドレスの記録，分岐の記録，ループの検出を行う．

先行実行には SimpleScalar の命令エミュレータである sim-safe をベースに用いる．

4.1.1 命令エミュレーション

前述の通り，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．

4.1.2 命令トレースの記録

すべての実行された命令のアドレスを記録する．

命令トレースのデータ量は，SD=100 で 1GHz，アドレス幅が 32bit，CPI=1 のマイクロプロセッサをシミュレーションすると仮定すると 40MB/s となる．このデータ量は決して小さくないので，データを圧縮することを考える．

命令トレースは差分のランレングス圧縮が有効である．なぜならば，分岐しないときはアドレスの増分は一定である．また，分岐するときも近いアドレスに飛ぶことが多い．これにより，ほとんどの差分はアドレス幅よりも少ないビット数で表すことが可能になる．

4.1.3 ロードストアアドレスの記録

キャッシュのシミュレーションのためにメモリアクセスのアドレスを記録する必要がある．これにはすべてのロードストア命令にってアクセスされたアドレスが含まれる．命令のフェッチによってアクセスされたメモリのアドレスは命令トレースと一致するので不要である．

ロードストアアドレスはアドレスの値を予測することで圧縮が可能である．これは，配列を順にアクセスするプログラムに対して非常に有効である．

4.1.4 分岐の記録

分岐予測のシミュレーションのために分岐の履歴とターゲットアドレスを記録する必要がある．ターゲットアドレスは分岐予測が外れた場合に必要である．

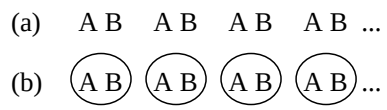

図 2 ループの例

4.1.5 ループの検出

ループとは完全に同じ命令列の実行が繰り返されている間とする．

まず，必要条件である後方分岐の連続成立を検出する．後方分岐が 2 回連続成立した場合，ループの可能性があると判断し，前回のループの命令列と比較を行いながら命令を実行する．そして，すべての命令が完全に一致した場合をループであるとする．また，一致した回数をループ回数とする．

ただし，これだけでは

```
for(i=0; i<N; i++){
  if((i%2) == 0)  /* processA */ ;
  else            /* processB */ ;
}
```

のような場合には processA と processB の命令列がループの度に交互にあらわれ，ループとして認識できないという問題がある（**図 2**(a)）．

しかし，このような場合に対しては命令トレースの比較範囲を拡大することにより，if 文が成立する processA と成立しない processB を 2 つで 1 つの大きなループと考えることができる．これによりこのプログラムを（N/2 回の）ループと認識することができる（図 2(b)）．

同様に多重ループにおける最内以外のループに拡張することも可能である．

4.2 詳 細 実 行

詳細実行では，パイプラインシミュレーション，キャッシュシミュレーション，分岐予測シミュレーション，高速実行可能なループの検出を行う．

ベースには SimpleScalar の out-of-order シミュレータである sim-outorder を用いる．これにより，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可能であるなどの利点がある．

4.2.1 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には SimpleScalar の out-of-order シミュレータである sim-outorder を用いる．

先行実行で保存された命令トレースを元にして命令をフェッチし，パイプラインシミュレーションを行う．

4.2.2 キャッシュシミュレーション

ロードストア命令ではキャッシュシミュレーションを行う．

ターゲットアドレスは先行実行で保存されている．このアドレスと実行のタイミングをキャッシュシミュレータに渡す．すると，キャッシュシミュレータはこれまでのロードストアと渡されたロードストアの情報から，そのロードストアの完了に必要なサイクル数を返す．そして，この結果は高速実行時のために保存される．

4.2.3 分岐予測シミュレーション

分岐命令では分岐予測シミュレーションを行う．

分岐予測の結果もキャッシュシミュレーションの結果と同様に記録される．分岐予測が成功すればターゲットアドレスは記録する必要はないが，外れた場合には分岐予測ミスパスのシミュレーションを行うためにターゲットアドレスも記録する．

分岐予測が外れた場合の詳細は後述する．

4.2.4 高速実行可能なループの検出

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，ループの開始時点でのプロセッサの内部状態が一致している必要がある．

そこで，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．同時にパイプラインに入っている命令が多いほど保存しなければいけないデータ量は多くなる．

ループの実行中は，キャッシュと分岐予測のシミュレーション結果も保存する．

ループを 1 周すると，開始時に保存していた状態と比較する．異なっていた場合は新たな状態として保存する．これを繰り返し，過去の状態と等しくなった場合高速実行に移る．

4.3 高 速 実 行

高速実行ではキャッシュシミュレーション，分岐予測シミュレーション，シミュレーションの早送りを行う．

4.3.1 キャッシュ，分岐予測シミュレーション

高速実行では，キャッシュと分岐予測のシミュレーションのみ行う．詳細実行と同様にキャッシュシミュレーションに必要なロードストアアドレスは先行実行で保存されている．

4.3.2 シミュレーションの早送り

ループ開始時点のプロセッサの内部状態とループ中のキャッシュ・分岐予測シミュレーションの結果が過去の状態・結果と一致した場合にシミュレーションは早送りされる．一致しなかった場合は，詳細実行に戻り以前の保存された状態から再開する．

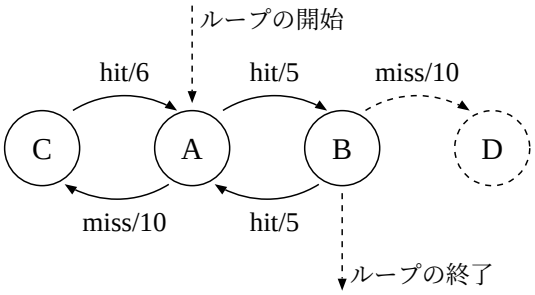

図 3 シミュレーションの状態遷移図の例

この様子は状態遷移図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」はループ中のキャッシュ・分岐予測シミュレーションの結果,「出力」は所要クロック数となる．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．

例を**図 3** に示す．簡単にするために，ループ中にメモリアクセス命令が 1 つだけ存在する場合を考える．入力は，そのメモリアクセスに対するキャッシュの hit/miss（正確にはレイテンシ）となる．この例ではパイプラインが安定（キャッシュに hit）している間は 2 つの状態（A, B）を交互に遷移することを示す．また，状態 C は状態 A からキャッシュミスが発生した場合に遷移する状態を示す．

ここで状態 B からのキャッシュミスが発生したとすると，新たに状態 D が作成される．

4.4 誤予測パスの影響

分岐予測で誤予測が発生すると，(1) 誤予測パス（実行すべきではない命令列）(2) 巻き戻し (3) 正しいパスの順に実行される．巻き戻しにより，誤予測パスで実行した命令は取り消され，プログラムの機能は正しく実行される．

誤予測パスの実行は誤予測であることを検出したときに巻き戻される．out-of-order 実行においては，この検出まで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．分岐予測も同様に誤予測パスの分岐命令により影響を受ける．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，誤予測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. まとめと今後の課題

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の構想を述べた．

現在わかっている問題点としては，SimpleScalar のキャッシュシミュレーションが遅いことがあげられる．sim-outorder では実行時間の約 20%がキャッシュシミュレーションである．これまでは，パイプラインシミュレーションが多くを占めているのであまり大きな問題にはならなかった．

しかし，パイプラインシミュレーションを高速化するとキャッシュシミュレーションの時間が問題になってくる．キャッシュシミュレーションに必要な割合を 20%とするとパイプラインシミュレーションをいくら高速化しても，5 倍以上速くなることはあり得ない．

この原因は，様々な構成のキャッシュに対応しているためである．一般的に多く使われる構成のキャッシュに限定すれば高速化は可能であるので，速度優先と柔軟性優先のキャッシュシミュレータを必要に応じて使い分けることにより解決できる．

謝辞 本研究の一部は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」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) Burger, D. and Austin, T. M.: The SimpleScalar Tool Set, Version 2.0 (1997).
4) Pai, V. S., Ranganathan, P. and Adve, S. V.: RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors, *Proceedings of the Third Workshop on Computer Architecture Education* (February 1997). Also appears in IEEE TCCA Newsletter, October 1997.
5) Mendel Rosenblum, Stephen A. Herrod, E. W. and Gupta, A.: Complete Computer System Simulation: The SimOS Approach, *IEEE Parallel and Distributed Technology*, Vol. 3, No. 4, pp. 34–43 (1995).
6) 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*, San Jose, California, pp. 283–294 (1998).