ISIS-SimpleScalar の実装

薬袋俊也† 塙敏博†† 田辺靖貴† 天野英晴†

近年、コンピューティングシステムの構成が複雑化しているため、開発過程におけるシミュレーションでの事前検証が重要になってきている。本研究室で開発された並列計算機シミュレータ構築支援ライブラリ ISIS は、近年主流となっているような高性能なプロセッサを用いたシミュレーションを行うことができない。また、高性能なプロセッサシミュレータを提供している SimpleScalar は、マルチプロセッサシステムのシミュレーションに対応していない。そこで、SimpleScalar が提供する高性能なプロセッサシミュレータをマルチプロセッサシステムに対応させ、ISIS によって記述されたシステムと接続できるようにした。評価の結果、高性能なプロセッサモデルを使用したマルチプロセッサシステムの詳細なシミュレーションによる評価が可能になったことを示した。

Implementation of ISIS-SimpleScalar

Toshiya Minai,† Toshihiro Hanawa,†† Yasuki Tanabe† and Hideharu Amano†

For developing recent complex computing systems, it is essential to examine many facets of the system using simulator in the early step of design. Although components of parallel systems including cache and networks can be modeled with a parallel simulation library ISIS, recent high-end processors are not supported. In contrast, SimpleScalar tool set provides high-performance processor simulators, but it cannot treat parallel systems with a lot of processors, shared memory and networks. ISIS-SimpleScalar acheives both benefits by incorporating SimpleScalar into ISIS library, that is, parallel systems with high-end processors can be simulated.

1. はじめに

近年、コンピューティングシステムの構成が複雑化しているため、開発過程においてシミュレーションによる事前検証が重要になってきている。特にマルチプロセッサシステムの挙動は予測が困難であるため、正確な性能評価のためには実機動作を考慮した詳細なシミュレーションモデルが必要不可欠である。

SimpleScalar[1)2)] は、プロセッサの詳細なモデルを提供しており、シングルプロセッサシステムやプロセッサアーキテクチャの検証に広く利用されているが、マルチプロセッサシステムのシミュレーションには対応していない。

SimOS[3)] や RSIM[4)] は、近年のプロセッサやキャッシュ等のモデルを提供し、マルチプロセッサシステムのシミュレーションに対応しているが、ネットワーク部の柔軟性や正確性に欠けるため、複雑なネットワークを実装する必要がある場合やネットワーク構成がシステム性能に与える影響の検証には適さない。

ISIS[5)] は、細部まで考慮した記述を行うため動作が正確であり、汎用的な部品をライブラリとして提供しているためシミュレータの実装が比較的容易である。しかし、近年主流となっている out-of-order 実行、分岐予測などをサポートした高性能なプロセッサを提供していない。

このように、近年の高性能なプロセッサを詳細にモデルし、かつマルチプロセッサシステムにとって重要な要素であるネットワーク部についての正確な挙動の再現が必要な場合、利用可能なシミュレータが存在しないという問題点があった。

そこで本研究では，SimpleScalar をマルチプロセッサシステムへ対応させ、ISIS によって記述されたシステムと接続できるようにした。

本稿では、まず、2 章で ISIS、3 章で SimpleScalar について紹介する。4 章において、今回実装した ISIS-SimpleScalar について取り上げ、5 章で評価を行う。最後に 6 章で結論および今後の課題を述べる。

2. ISIS

ISIS は主として並列計算機をターゲットとした、計算機シミュレータのための C++言語によるクラスライブラリである。C++を採用したことによりシステム全体をクラス単位で効率良く管理でき、しかも C 言語と同様に高速に動くプログラムが容易に作成可能である。

2.1 ユニットとユニット間の通信方法

ISIS では計算機内部のプロセッサやメモリなどの機能ブロックがユニットと呼ばれるクラスライブラリとして実装されている。

† 慶應義塾大学 理工学研究科
Department of Science and Technology, Keio University
†† 東京工科大学 コンピュータサイエンス学部
Department of Computer Science, Tokyo University of Technology

図 1 に示したように、各ユニットは、他ユニットと結合するためのポートを持ち、それぞれクロック入力を受けて自律動作し、ポートを介しパケットを送受信する。パケットにはユニット間でやり取りされるすべての情報を定義する。

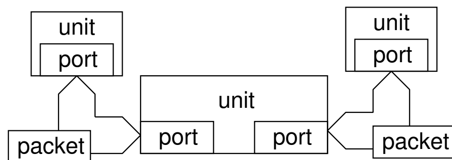


図 1 結合と通信のモデル

2.2 シミュレータの構築手順

シミュレーション対象のアーキテクチャを構築する場合には、ユーザはまず、評価対象となる並列計算機を機能ブロックに分解し、それぞれの機能ブロックに対応するユニットをライブラリ内から選択する。対応するユニットが存在しない場合には、所望の機能を持つユニットを新たに実装する。そして、互いにデータをやり取りするユニットのポート同士を結合していくことにより、全体を構成する。

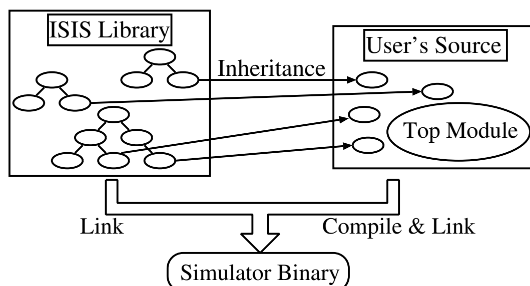


図 2 シミュレータの構築

2.3 ISIS の欠点

近年マルチプロセッサシステムにおいて主流となっているプロセッサは、out-of-order 実行、複数命令同時発行、分岐予測などをサポートした高性能なプロセッサであるが、ISIS では、プロセッサとしては、IDT 社の R3081 をモデルにした単数命令発行、in-order 実行を行う比較的簡素なプロセッサのみを機能ユニットとして提供している。そのため、ISIS において高性能なプロセッサを用いたシステムを想定してシミュレータを構成するには、新たにプロセッサシミュレータを実装する必要がある。

3. SimpleScalar

SimpleScalar は、1992 年に Wisconsin 大学の Multiscalar プロジェクトの一環として開発が開始された。SimpleScalar ツールセットは、コンパイラ、アセンブラ、リンカ、ライブラリ及びシミュレータからなる。特にシミュレータには、非常に高速な機能レベルシミュレータから、ノンブロッキングキャッシュ、投機実行、分岐予測などをサポートした高性能なシミュレータまであり、性能、柔軟性、精度の異なる 6 つのシミュレータを提供している。

SimpleScalar は、MIPS、Alpha、ARM などのアーキテクチャをモデルにしたプロセッサシミュレータを提供している。また、MIPS 互換の PISA、Alpha、Power PC、x86、ARM などの命令セットに対応したインタプリタを用意することによって、高い移植性を提供している。

本研究では、R3081 と同じ MIPS 系であるという理由から、MIPS アーキテクチャをモデルにした PISA 依存のプロセッサシミュレータをマルチプロセッサシステムに対応させた。以後の説明は、MIPS アーキテクチャをモデルにした PISA 依存のプロセッサシミュレータについて行う。

3.1 命 令 定 義

命令定義フォーマットは図 3 のような形式になっており、すべての命令が 1 つのファイル (DEF ファイル) 中で、定義されている。1 番目の要素は、シミュレータプログラム内で使用する識別子である。5 番目の要素は実行に必要な Functional Unit 名、6 番目の要素は命令の種類である。7、8 番目の要素にはその命令が書き込みを行うレジスタ番号、9〜11 番目の要素には読み出しを行うレジスタ番号が定義されている。プロセッサシミュレータは、命令をデコードすることにより識別子 (ADDU) を得た後、DEF ファイルを読み出すことによって、対応するレジスタ番号や Functional Unit 名などの情報を取得する。その後、識別子_IMPL(ADDU_IMPL) を呼び出し、命令を実行する。

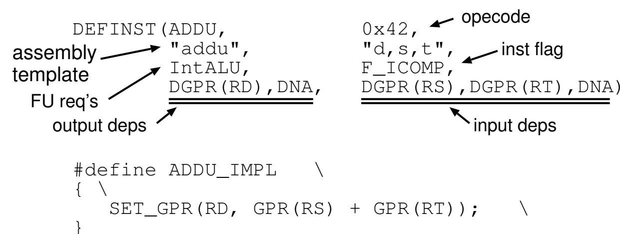


図 3 命令定義フォーマット

3.2 レ ジ ス タ

整数レジスタは r0 から r31 まで、浮動小数点単精度レジスタは f0 から f31 までである。浮動小数点レジスタは 2 つ繋ぎ合わせることにより倍精度での使用が可能である。また、PC(プログラムカウンタ)、HI、LO、FCC のレジスタを持つ。

3.3 アドレス空間

アドレス空間の割当を図 4 に示す。0x00000000〜0x003fffff のアドレスは、使用しない。0x00400000〜0x0fffffff のアドレスは、プログラムを格納する際に使用される。0x10000000〜0x7fffbfff のアドレスは、プログラム実行中に必要になったデータに割り当てられ、引数、環境変数などのデータは 0x7fffc000〜0x7fffffff に格納される。

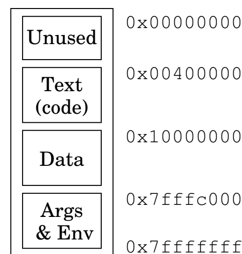


図 4　アドレス空間

3.4　sim-outorder

SimpleScalar は、性能、柔軟性、精度を実現するために、さまざまなプロセッサシミュレータを幅広く提供している。最も高性能なプロセッサシミュレータである sim-outorder は、out-of-order 実行、分岐予測などをサポートしたプロセッサの詳細なモデルである。sim-outorder の処理は 6 つのステージに分けられており、命令の実行はパイプライン処理される (図 5)。sim-outorder では、実行が終了するまで、以下の for 文によって各ステージの処理が繰り返される。

```
for (;;) {
  ruu_commit();
  ruu_writeback();
  lsq_refresh();
  ruu_issue();
  ruu_dispatch();
  ruu_fetch();
}
```

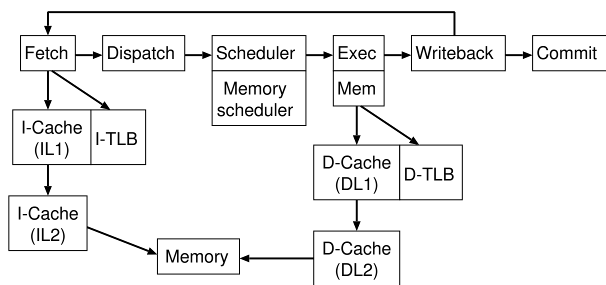


図 5　sim-outorder のパイプライン

Fetch Stage

命令幅分の命令を I-Cache または Memory から取り出し、IFQ に送る。ruu_fetch() に実装されている。

Dispatch Stage

IFQ から取り出した命令のデコード及び実行を in-order で行い、RUU(Register Update Unit)/LSQ(Load/Store Queue) に配置する。また、レジスタの依存関係などを考慮し、Execute Stage(`ruu_issue()`) に進める命令を readyq に投入する。`ruu_dispatch()` に実装されている。

Scheduler Stage

メモリアクセスアドレス、レジスタの依存関係などによって、readyq の順番を入れ替える。`ruu_issue()` 及び `lsq_refresh()` に実装されている。

Execute Stage

readyq から命令を取り出し、Functional Unit やメモリアクセスポートが確保できた命令に対しては、その遅延を計算をする。遅延時間により Writeback イベントの順序を決定し、eventq に投入する。`ruu_issue()` に実装されている。

Writeback Stage

eventq から命令を取り出し、分岐予測が失敗しているか否かを判断する。分岐予測が失敗していた場合には、元の正当な状態に戻す。また、その命令が Writeback されることによって、Execute Stage に進むことができるようになった命令を readyq に投入する。`ruu_writeback()` に実装されている。

Commit Stage

RUU/LSQ の先頭から writeback ステージまでの処理が終わった要素を取り出し、終了処理を行う。`ruu_commit()` に実装されている。

4.　ISIS-SimpleScalar

ISIS を用いて近年の out-of-order 実行、予測分岐などをサポートするような高性能なプロセッサを用いたシステムをターゲットとしたシミュレータを構築するには、これを新たに記述する必要がある。しかし、このようなプロセッサモデルを一から実装するためには、多大な時間と労力がかかる。そこで今回、SimpleScalar が提供するプロセッサシミュレータのうち最も高性能なプロセッサである sim-outorder を ISIS に対応させた ISIS-SimpleScalar の実装を行った。

4.1　ISIS への対応

ISIS は C++、SimpleScalar は C 言語で実装されているため、SimpleScalar のソースコードを ANSI 準拠の C++ 互換に書き直した。

4.2　マルチプロセッサシステムシミュレーションへの対応

オリジナルの sim-outorder では、命令は Dispatch Stage(`ruu_dispatch()`) において in-order で実行される。この際、メモリアクセスレイテンシ、Functional Unit 数などは一切考慮されていない。その後の Stage において、メモリアクセスレイテンシや Functional Unit 数、レジスタの依存関係などが考慮され、命令が out-of-order で issue、writeback され、in-order で commit される。そうすることにより、命令は out-of-order で実行されたように処理される。

バスなどの共有ネットワークを介して共有データにアクセスを行うマルチプロセッサシステムにおいては、共有データが各プロセッシングノードのキャッシュに格納

されることが考えられる。この場合、オリジナルの sim-outorder のように Dispatch Stage(`ruu_dispatch()`) を実行した瞬間にメモリの内容が書き換えられてしまうと、キャッシュ間のコンシステンシ制御に問題を起こし、挙動の正確なシミュレーションが困難になる。

また、システムによっては、プロセッサから発行されたアドレスとは別のアドレスにデータが割り当てられる場合もある。もし、命令の実行を Dispatch Stage (`ruu_dispatch()`) で行うことを維持するならば、このアドレス変換は sim-outorder 内で行わなければならない。その度にプロセッサのコア部分のプログラムを修正しなければならないのは、シミュレータとしての可搬性に欠ける。

4.2.1 動作の修正

オリジナルの sim-outorder では命令が in-order で実行されていたが、上記のような問題を解決するため、reservation station、reorder buffer などを実装し、out-of-order で実行されるようにした。また、マルチプロセッサシステムにおいてはプロセッサ外部にある共有メモリ領域へのアクセスが考えられるため、0x80000000〜0xffffffff のアドレスを、複数のプロセッサ間で共有される共有データに割り当てた。共有データへのアクセス要求はプロセッサ外部の ISIS ユニットと接続可能なポートに対して発行するようにした。また、共有メモリから読み出されたデータがプロセッサに転送されてくることに備え、プロセッサは外部と接続されているポートを常に監視するように実装した。

各 Stage での動作は次のように変更した。

Fetch Stage

命令幅分の命令を I-Cache または Memory から取り出し、IFQ に送る。`ruu_fetch()` に実装した。

Dispatch Stage

IFQ から取り出した命令のデコードを行い、すべての命令を RUU と reservation station に投入する。`ruu_dispatch()` に実装した。

Schedule Stage

reservation station に入っている命令のうち、

(1) Functional Unit を利用する命令は、以下の条件が揃った時点で、Functional Unit を占有し、execution buffer に投入する
- レジスタまたは reorder buffer から、値の読み出しが完了
- Functional Unit が利用可能

(2) local memory への Load 命令は、以下の条件が揃った時点で、ローカルメモリアクセスポートを占有し、execution buffer に投入する
- メモリアクセスするアドレスが決定
- 他のメモリアクセス命令との依存関係なし
- メモリアクセスポートが利用可能

(3) shared memory への Load 命令は、以下の条件が揃った時点で、外部メモリアクセスポートに共有メモリへの読み出し要求を発行し、execution buffer に投入する
- メモリアクセスするアドレスが決定
- 他のメモリアクセス命令との依存関係なし
- 外部メモリアクセスポートが利用可能

`ruu_issue()` に実装した。

Execute Stage

execution buffer に入った命令のうち、

(1) Functional Unit を占有している命令の Functional Unit 占有時間を 1 ずつ増やす

(2) local memory への Load 命令のメモリアクセスレイテンシを 1 ずつ増やす

機能を `ruu_writeback()` に実装した。

Writeback Stage

execution buffer に入った命令のうち、

(1) Functional Unit を利用する命令は、Functional Unit を占有する時間を経過した後、実際に計算を行い、Functional Unit を解放し、計算結果を reorder buffer に投入する

(2) local memory への Load 命令は、メモリアクセスレイテンシが実際に要するレイテンシに達したら、ローカルメモリからデータを読み出し、その値を reorder buffer に投入する

(3) shared memory への Load 命令は、外部メモリアクセスポートに共有メモリからの読み出されたデータが転送されてきたら、そのデータを reorder buffer に投入する

また、分岐予測が失敗しているか否かを判断し、分岐予測が失敗していた場合には、元の正当な状態に戻すように `ruu_writeback()` を実装した。

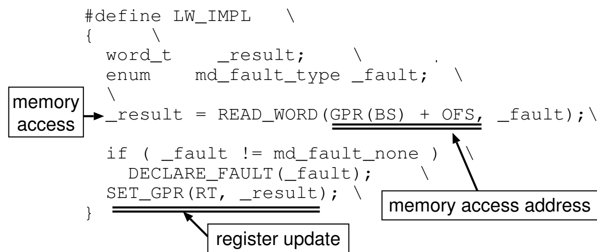
Commit Stage

RUU の先頭の命令の実行結果が reorder buffer に投入されたら、その値を用いてレジスタの更新を行い、RUU の先頭の命令を破棄する。その命令が Store 命令の場合、
- local memory への Store 命令ならば、ローカルメモリに値を書き込む
- shared memory への Store 命令ならば、外部メモリアクセスポートに共有メモリへの書き込み要求とデータを発行する

`ruu_commit()` に実装した。

4.2.2 命令定義の変更

各 Stage において上記のような動作を実現するために、DEF ファイル内の命令の定義を修正した。オリジナルの sim-outorder では、レジスタの読み出し、メモリアクセス、計算、レジスタへの書き込みなどを 1 つの命令の中で行っている。例えば、メモリから 8 バイトのデータを読み出す命令である LW は、以下のように定義されている。しかし、修正したプログラムでは、アドレスの計算、メモリアクセス、レジスタへの書き込みは、別のタイミングまたは別の Stage で実行される。そのため、LW は、アド



レス計算命令、メモリアクセス命令、レジスタ書き込み命令に分割し、DEF ファイル内に別々の命令として定義した。

4.2.3 クラス化

マルチプロセッサシステムでは、プロセッサを構成するモジュールがプロセッサ数分必要になる。各モジュールをクラス化することにより、プロセッサ分だけインスタンスを生成し利用できるように簡易化した。

5. 評 価

本章では、作成した ISIS-SimpleScalar の評価を行う。

5.1 シングルプロセッサシステムシミュレーション

オリジナルの sim-outorder と ISIS-SimpleScalar でのシミュレーションに要する時間の比較を行うため、図 6 のようなシングルプロセッサシステムをモデルとし、評価を行った。

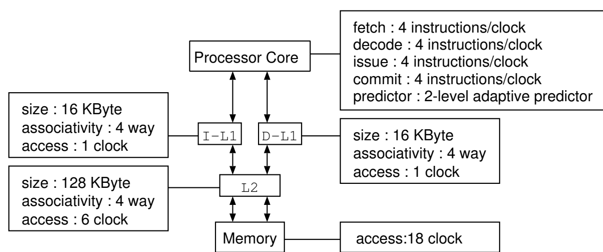


図 6 シングルプロセッサシステムシミュレータ

評価には、SimpleScalar ツールセットに添えられていた test-math、test-fmath、test-printf、test-llong、test-lswlr の 5 つのテストプログラムと、SPLASH2 ベンチマーク集から以下の 3 つのアプリケーションを使用した。

表 1 評価に使用したアプリケーション

FFT	高速フーリエ変換	2^{14}
LU	行列の LU 分解	128×128
RADIX	整数の基数ソート	131072 keys

5.1.1 シミュレーション速度

オリジナルの sim-outorder と ISIS-SimpleScalar のそれぞれにおいて、上記の 8 つのプログラムを実行した。表 2 にシミュレーションの速度と速度低下を示す。

表 2 シミュレーション速度の比較

	original sim-outorder (instrucitons/clock)	ISIS-SimpleScalar (instructions/clock)	速度低下 (倍)
test-math	189288	78494	2.41
test-fmath	42427	63552	0.67
test-printf	585057	78901	7.42
test-llong	18667	34202	0.55
test-lswlr	7152	9693	0.74
FFT	549047	48095	11.4
LU	558681	48252	11.6
RADIX	573903	34596	16.5

速度低下が 10 倍を超えているものもあるが、現在のコードはシミュレーション速度に重きを置いたものではない。現在のコードでは実行命令数が多いアプリケーションほど、シミュレーション速度が低下する傾向にある。

5.2 マルチプロセッサシステムシミュレーション

ISIS-SimpleScalar を用いたマルチプロセッサシステムのシミュレーション例として、図 7 のようなシミュレータを実装した。各 PU は図 6 のような構成となっているが、図 6 に記載されている memory や cache は、各プロセッサによって独自に利用されるローカルデータを格納する local cache、local memory として利用される。

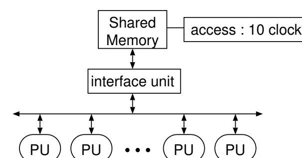


図 7 マルチプロセッサシステムシミュレータ

5.2.1 ネットワーク利用率と台数効果

図 7 に示したシステムにおいて、PU 数を 1、2、4、8、16 と変化させ、そのそれぞれにおいて表 1 の 3 つのアプリケーションを実行した。PU 数を増やすことによって共有バスがどの程度混雑したかを図 8 に、実行クロック数での性能向上がどの程度得られたかを図 9 に示す。図 8 の縦軸のネットワーク利用率は、単位時間当たりに共有バスが処理したアクセス数を示す値であり、バスの場合には複数のアクセスを同時に処理することができないため、その値が 1 を越えることはない。

各 PU がバス結合されたマルチプロセッサシステムでは、ある一定以上に PU 数を増やしても、バスの混雑により共有メモリアクセスレイテンシが増加してしまうため、性能向上が望めない。

ISIS-SimpleScalar を用いた評価の結果、図 8，9 においてその傾向が表われていることが分かる。

また、LU を実行をした場合には他のアプリケーションを実行した場合に比べて台数効果が小さい。これは図 8 から読み取れるように、LU を実行した際のネットワーク利用率が他のアプリケーションを実行したときに比べて高かったことが原因である。

図 10 に PU 数を 4 としたときの、全命令に占める共有

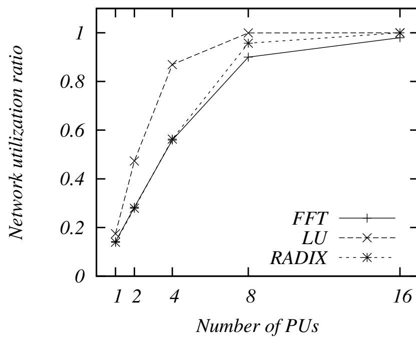


図 8　ネットワーク利用率の変化

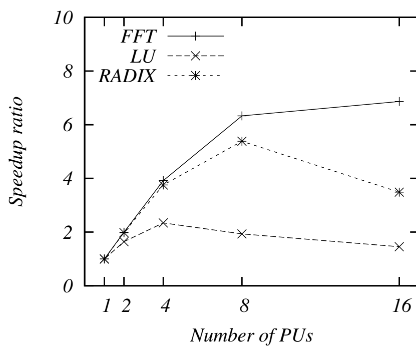


図 9　PU 数の違いによる実行クロック数の変化

メモリアクセス命令の割合を示した。図 10 より、LU は他のアプリケーションに比べ共有メモリにアクセスする命令の割合が高いことが読み取れ、このことが LU のネットワーク利用率を高めたと考えることができる。

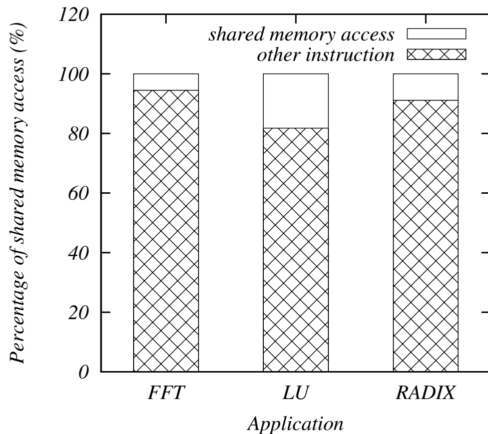


図 10　全命令数に占める共有メモリアクセス命令数の割合

5.2.2　シミュレーション時間

一般的に、シミュレーションする PU 数を増やすと、シミュレータ上で動作するユニット数も増えるため、シミュレーションに要する時間が増える傾向にある。図 11 において、PU 数を 1、2、4、8、16 とし、FFT、LU、RADIX を実行したときのシミュレーション時間を比較した。

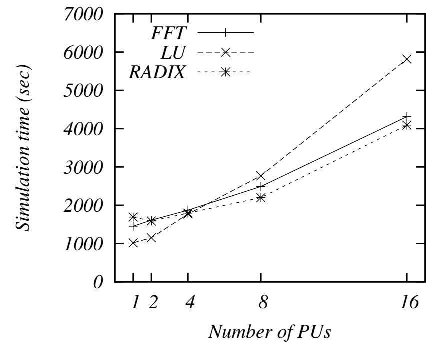


図 11　シミュレーション時間の変化

PU 数を増やすとシミュレーション時間が増加していることが分かる。特に大規模なマルチプロセッサシステムを想定したシミュレーションを行う場合には、シミュレーション時間を短縮する方法を検討する必要がある。

6. 結論および今後の課題

本稿では、既存の高性能なプロセッサシミュレータである SimpleScalar の sim-outorder をマルチプロセッサシステムシミュレータへ転用可能にし、また、ISIS に対応させることによって、高性能なプロセッサを搭載したマルチプロセッサシステムのシミュレータを実装し、評価を行った。

その結果、シミュレーション速度はオリジナルの sim-outorder と比較し、最大で約 16.5 倍の低下となったが、高性能なプロセッサモデルを使用したマルチプロセッサシステムのシミュレーションによる詳細な評価が可能となったことを示した。

今後の課題としては、以下の項目が挙げられる。

- シミュレーション時間の短縮
- ライブラリ化
- 公開

参　考　文　献

1) D. Burger, T. Austin, “The SimpleScalar Tool Set, Version 2.0”, 1997.
2) T. Austin, E. Larson, and D. Ernst, “SimpleScalar: An Infrastructure for Computer System Modeling”, IEEE Computer, 35(2), Feb 2002.
3) M. Rosenblum, S.A. Herrod, E. Witchel, and A. Gupta, “Complete Computer Simulation: The SimOS Approach”, IEEE Parallel and Distributed Technology, vol. 3, no. 4, Winter 1995.
4) C.J. Hughes, V.S. Pai, P. Ranganathan, and S.V. Adve, “Rsim: Simulating Shared-Memory Multiprocessors with ILP Processors”, IEEE Computer, 35(2), Feb 2002.
5) M.Wakabayashi, H.Amano, “Environment of Multiprocessor Simulator Developemnt”, Proc. of International Symposium on Parallel Architectures, Algorithms and Networks, pp,64-71, 2000.