修士論文

メニーコアプロセッサ構成の検討を目的とした高速トレースシミュレータの開発

指導教員　　津邑 公暁 准教授
松尾 啓志 教授

名古屋工業大学大学院 工学研究科
修士課程 創成シミュレーション工学専攻
平成 23 年度入学　23413575 番

山田 龍寛

平成 25 年 2 月 5 日

メニーコアプロセッサ構成の検討を目的とした高速トレースシミュレータの開発

山田 龍寛

内容梗概

ゲート遅延に対する配線遅延の相対的な増大から，動作周波数の向上だけではマイクロプロセッサの速度向上を見込めなくなってきた．また，集積回路の微細化に伴う消費電力や発熱量の増大から動作周波数自体の向上も困難になってきている．こうした中で，SIMD やスーパスカラなどの命令レベル並列性 (ILP) に基づく高速化手法が注目されてきた．しかし，多くのプログラムは明示的な ILP を持たない等の理由から，これらの手法にも限界がある．現在では，消費電力や発熱量の問題を解決しつつプロセッサあたりの処理能力を向上させるため，1 つの CPU に複数のコアを搭載したマルチコアプロセッサが広く普及している．

今後は集積度の向上に伴って，搭載するコア数をさらに増大させたメニーコアプロセッサが一般化すると予想されている．メニーコアプロセッサには高並列な処理性能と低消費電力化への期待が高まっているが，データ供給面の問題などにより多数のコアを有効活用することは困難である．こうした理由から並列処理による高速化は様々な研究が行われているが，代表的アプリケーションにおける並列化限界はあまり調査されておらず，それらのアプリケーションを効率よく実行できる理想的なプロセッサ構成を検討することが重要な課題となっている．

そこで本論文では，安定したデータ供給が可能なプロセッサ構成の実現可能性を検証するために，アプリケーションの実行トレースを採取可能なメニーコアトレースシミュレータを開発する．そのためには，メニーコアプロセッサの実現において重要となる配線遅延を考慮した構成方式を実現するために，キャッシュ構成やメモリ一貫性プロトコル等のデータ供給を効率的に行う方式を構築し検証する必要がある．加えて，コア数の増大に伴うメモリトラフィックの影響を調べるために，複数のコアやメモリを相互に結合し交信路を提供する相互結合網の様々な形状を，構築し検証する必要もある．そこで，これらの構成方式を検討するとともに，性能目標値を導出するための基本となるメニーコアプロセッサの構成を設計する．

加えて，メニーコア研究において，コア数の増大に伴うシミュレーション時間の増大が深刻な問題となっている．これを解決し，大規模なアプリケーションを現実的な時間で実行するために，シミュレータの並列実行による高速シミュレーションを可能にした。

これにより，SPLASH-2 ベンチマークのシミュレーション時間を 4 スレッド実行で平均 56.8%, 最大で 81.8%削減することに成功した.

メニーコアプロセッサ構成の検討を目的とした
高速トレースシミュレータの開発

目次

1 はじめに **1**

2 マルチコア／メニーコアプロセッサ **2**

2.1 研究背景 2

2.2 マルチコア／メニーコア間での実装の違い 3

2.2.1 コア数増大とキャッシュ構成の関係 4

2.2.2 キャッシュコヒーレンシプロトコル 5

2.2.3 結合網形状の検討 6

3 既存シミュレータの問題点とその解決方針 **9**

3.1 アーキテクチャシミュレータ 9

3.2 既存シミュレータとその問題点 10

3.3 研究計画概要 12

4 メニーコアトレースシミュレータの開発 **13**

4.1 アーキテクチャ設計 13

4.2 実行トレースの採取 15

4.3 動作モデル 16

4.3.1 シミュレータコアの起動と同期処理 16

4.3.2 キャッシュリクエスト 19

4.3.3 リクエストの衝突と優先順位 20

5 シミュレータの動作検証 **23**

5.1 動作環境 23

5.2 動作結果 23

5.3 考察 29

5.4 今後のメニーコア研究に向けたシミュレータの改良方針 31

6 シミュレーションの高速化 **33**

6.1 ソフトウェアシミュレーションの問題点 33

6.2 既存のシミュレーション高速化手法 34

6.3 並列度の自動調整 35
6.4 シミュレータの動作オプション 38
6.4.1 スレッドの生成 38
6.4.2 バリア 39
6.4.3 コアへのバインド 39

7 シミュレーション時間の評価 40

7.1 評価環境 40
7.2 動作オプションの選択 40
7.2.1 スレッドの生成 41
7.2.2 バリア 42
7.2.3 コアへのバインド 44
7.3 評価結果 46

8 おわりに 50

謝辞 51

著者発表論文 51

参考文献 52

1 はじめに

ゲート遅延が支配的であった2000年代初頭までは，集積回路の微細化による高クロック化により，マイクロプロセッサの高速化を実現してきた．しかし，それと同じくしてゲート遅延に対する配線遅延の相対的な増大や，集積回路の微細化に伴う消費電力および発熱量の増大といった問題から，マイクロプロセッサの動作周波数の向上は困難になってきている．こうした中で，SIMD やスーパスカラなどの命令レベル並列性（ILP: Instruction Level Parallelism）に基づく高速化手法が注目されてきた．しかし，多くのプログラムは明示的な ILP を持たないことや，ILP を抽出できる場合でもメモリスループットなどの資源的制約があることから，これらの手法にも限界がある．

このため現在では，消費電力や発熱量の問題を解決しつつプロセッサあたりの処理能力を向上させるため，1 つの CPU に複数のコアを搭載したマルチコアプロセッサが広く普及している．このため，並列化されていないプログラムを複数のコアを用いて高速化する一般的な手法として，スレッドレベル並列性（TLP: Thread Level Parallelism）に着目した様々な手法が研究されている．これらの手法は，プログラムを複数スレッドに割り当てられるよう分割し，それぞれのコアに割り当てることで高速化を図っている場合が多い．例えば，並列処理ライブラリを用いてプログラマが明示的にプログラムを並列化したり，自動並列化コンパイラ [1, 2] を用いてコンパイラによりプログラムを複数のコアに対して自動的に割り当てるような手法が挙げられる．しかし，そもそも並列性を持たず TLP を抽出することが難しいプログラムも存在することや，プログラマが明示的に並列処理プログラムを記述することは困難である場合が多いなど，様々な問題が存在している．このため，複数のコアを有効に利用してプログラム全体のスループットを向上させる高速化手法を検討する必要が出てきている．

さらに，今後プロセッサあたりのコア数をさらに増大させたメニーコアプロセッサが一般化すると予想されている [3, 4]．超並列処理により実効性能を飛躍的に改善するだけでなく、低速かつ定電圧動作を積極的に導入することにより、性能を担保しつつ大幅に消費電力を削減することが可能であるため，メニーコアプロセッサへ大きな期待が寄せられている．しかし，マルチコアプロセッサに対する既存の高速化手法を利用しても，一部のコアに処理を割り当てきれないという状況が発生することが想定されることや，データ供給面の問題なども存在し，多数のコアを有効に利用することは困難である．また，並列処理による高速化は様々な研究が行われているが，代表的アプリケーションにおける並列化限界はあまり検討されておらず，それらのアプリケーションを効率よく実行できる理想的な

プロセッサ構成を検討することが重要な課題となっている．

そこで本論文では，安定したデータ供給が可能なプロセッサ構成の実現可能性を検証するために，アプリケーションの実行トレースを採取可能なメニーコアトレースシミュレータを開発する．そのためには，メニーコアプロセッサの実現において重要となる配線遅延を考慮した構成方式を実現するために，キャッシュ構成やメモリ一貫性プロトコル等のデータ供給を効率的に行う方式を構築し検証する必要がある．加えて，コア数の増大に伴うメモリトラフィックの影響を調べるために，複数のコアやメモリを相互に結合し交信路を提供する相互結合網の様々な形状を，構築し検証する必要もある．そこで，これらの構成方式を検討するとともに，性能目標値を導出するための基本となるメニーコアプロセッサの構成を設計する．

また，マルチコア・メニーコアプロセッサ研究において，コア数の増大に伴うシミュレーション時間の増加が深刻な問題となっている．これを解決し，大規模なアプリケーションを現実的な時間でシミュレートするために，シミュレーション自体を高速に行う機能をメニーコアトレースシミュレータに追加実装する．

以下，2 章ではマルチコア／メニーコアプロセッサの現状について述べ，3 章では既存シミュレータの問題点を挙げた上で，その解決方針と研究計画を提案する．そして，4 章でメニーコアトレースシミュレータの開発について述べ，5 章で開発したシミュレータの動作を検証する．その後，6 章でシミュレーションの高速化について述べ，7 章でシミュレーション時間の評価結果と，それに対する考察を述べる．最後に，8 章で結論を述べる．

2　マルチコア／メニーコアプロセッサ

本章では，マルチコア／メニーコアプロセッサの現状と，複数コアを搭載する一般的なプロセッサ構成方式について述べる．

2.1　研究背景

現在では，消費電力や発熱量の問題を解決しつつプロセッサあたりの処理能力を向上させるために，1 つの CPU に複数のコアを搭載したマルチコアプロセッサが広く普及している．一般のパソコンやワークステーションに用いられる汎用マルチコアプロセッサの例として，Intel 社の Core i7[5] や IBM 社の Cell/B.E.[6]，Sun Microsystems 社の UltraSPARC T4[7] などが挙げられる．さらに，ネットワーク機器等で使用することを想定し 1 つのプロセッサに 64 個のコアを搭載した TILE64[8] などのメニーコアプロセッサも登場している．また，CPU や GPU（Graphics Processing Unit）のみならず，DSP

（Digital Signal Processor）などの組み込み向けプロセッサにまでマルチコア技術が幅広く普及してきた．このようにプロセッサのマルチコア化が進んでおり，複数のコアを利用してプログラム全体のスループットを向上させる高速化手法を検討する必要が出てきている．

今後は半導体のプロセスルールの縮小に伴い，単一プロセッサあたりに搭載されるコア数がさらに増大していくと予想される．近い将来，コア数は 10，100，1000 と規模を拡大していき，メニーコアプロセッサが一般化すると考えられる．このメニーコアプロセッサは，チップに多数のコアを集積してスレッドレベルの並列性を利用することで高並列な処理性能と低消費電力化の実現を狙うものである．

メニーコアプロセッサの実現方法にはいくつかのアプローチがある．その一つに TILE64 が採用しているタイルアーキテクチャを利用する方法が挙げられる．このようなタイルアーキテクチャでは，タイルと呼ばれる小さいサイズの機能ブロックを規則的に敷き詰めメッシュ状に結合されたネットワーク上で情報通信を行うことで配線遅延を軽減することができる．また，コア数を増やし回路規模を大きくしても動作速度を高速に保つことができる．しかし，独自のアーキテクチャとなるため既存の命令セットと互換性がないといった問題がある．他には，Intel 社の Core i7 に代表される汎用のチップマルチプロセッサで利用されている方法が挙げられる．チップマルチプロセッサでは，シンプルなプロセッサコアを多数搭載することで従来のプロセッサの設計を再利用し，設計を簡単にしながらスループットを向上させることができる．また，省電力効果を高めることもでき，配線遅延を考慮した設計にも有効である．

このように，どちらの実現方法においても共通して配線遅延を考慮している．配線遅延の問題は，現在よりさらに配線プロセスの微細化が進むことでより深刻化すると考えられるため，設計段階から配線遅延を十分に考慮して回路を設計する必要がある．そのため，高並列実行可能なアーキテクチャの実現に向けたプロセッサ構成の検討が重要になっている．

2.2　マルチコア／メニーコア間での実装の違い

データ供給の転送効率を向上させるキャッシュシステムと，複数のコアやメモリ間の交信路を提供する相互結合網のトポロジについて，マルチコアとメニーコアプロセッサの構成間における一般的な違いに着目しながら述べる．

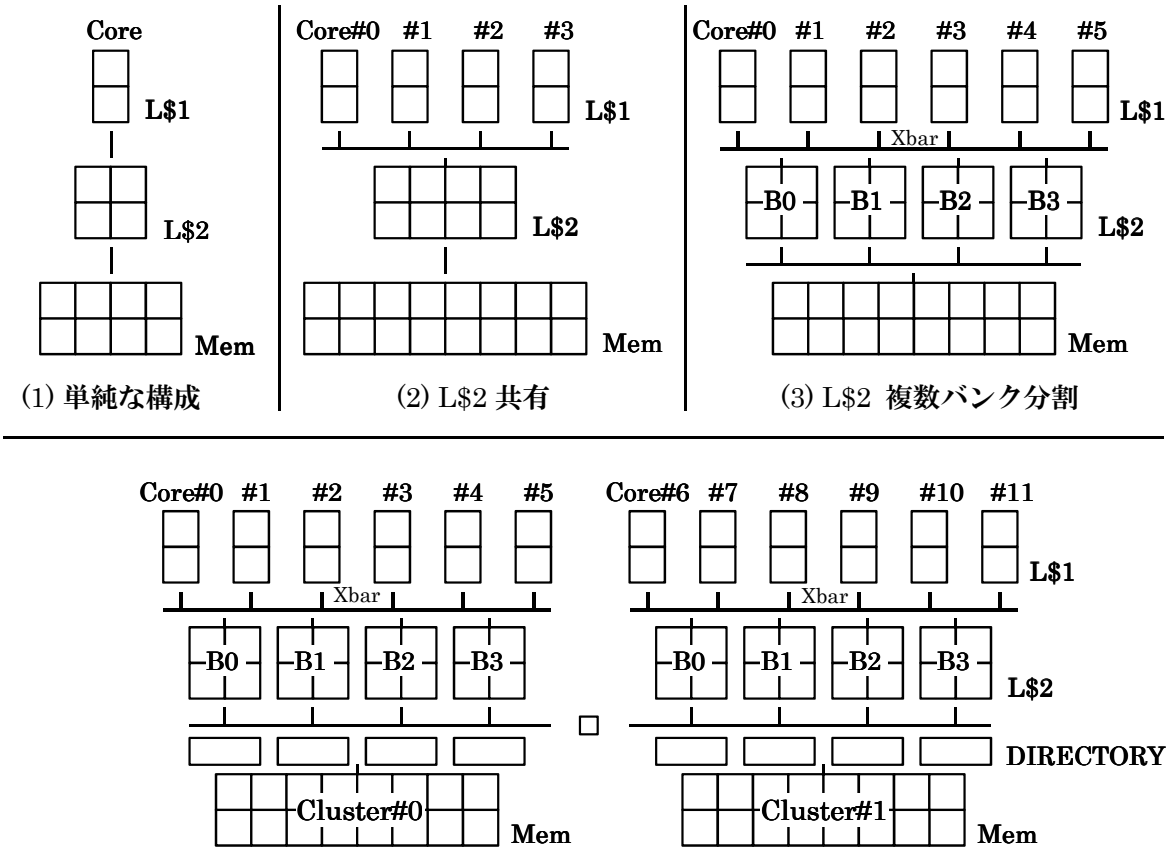


図 1: コア数の増大に伴うキャッシュの構成の様子

2.2.1 コア数増大とキャッシュ構成の関係

効率的なデータの安定供給のために，一般的なプロセッサではキャッシュシステムが用いられている．キャッシュシステムでは，プロセッサに搭載されるコア数の増大とそのキャッシュ構成に密接な関係性がある．ここで，コア数の増大に伴うキャッシュ構成の変化の様子を**図 1** に示す．

図 1 中の (1) は，シングルコアプロセッサにおける単純なキャッシュ構成を示している．キャッシュメモリは，データを取得・更新する際に主記憶装置やバスなどの低帯域を隠蔽し，処理装置と記憶装置の性能差を埋める．また，小容量のキャッシュメモリを何階層も重ねて持つことで，低速な主記憶装置に対するアクセスを削減し処理性能を大きく損なうことを避けてきた．

その後，マルチコアプロセッサが採用されるようになり，複数のコアから主記憶へのア

クセスが発生するようになった．そのため，シングルコア以上にキャッシュシステムによるメモリ帯域幅の確保が重要となっている．このようなマルチコアプロセッサには，図1中の (2) で示すように，L2 キャッシュの構成を変えず共有するものがある．しかし，並列実行を目的とした一般的なマルチコアプロセッサでは，2 次キャッシュを共有するだけの単純な構成では，各コアからの参照要求が頻繁に到達するようになるため，リクエストに対する応答が返ってくるまでの待ち時間が増大してしまい処理性能が低下してしまう．そこで，図 1 中の (3) で示すように，L2 キャッシュを複数のバンクに分割するものがある．L2 キャッシュをバンク分割しクロスバネットワークで接続することで，参照要求先のバンクが異なる限り複数コアからの要求を同時に受け付けることが可能になる．しかし，キャッシュのデータ一貫性を保持しなければならないため，データ管理の複雑度が増大することになる．

さらに，搭載されるコア数を増加させたメニーコアプロセッサでは，ハードウェア物量が増大してしまうため，バンク間でのクロスバネットワークの実現が困難になる．そこで，ハードウェア物量を抑えつつ，データの転送効率を維持するために図 1 中の (3) で示す構成を 1 つのクラスタとし，図 1 中の (4) で示すように複数のクラスタを接続するような構成をとる傾向がある．また，複数のコアやクラスタが備えるキャッシュでは，先ほど述べたようにデータの一貫性を管理する必要がある．そのような一貫性の管理には，スヌーピングやディレクトリベースの一貫性管理機構がよく用いられている．スヌーピングは各コア間の帯域幅が十分大きければ性能が良くなるが，全てのメモリアクセス要求を全体にブロードキャストする必要があるため，コア数が増えた場合も高い性能を出すためには，バスの帯域幅をより大きくしなければならなくなる．一方で，ディレクトリベースの一貫性管理機構はキャッシュとメモリの間にディレクトリ機構が存在することになるためレイテンシが増大する傾向があるが，ブロードキャストが不要となるため帯域幅が小さくても良いという利点がある．このため，多数のコアを搭載する大規模システムではディレクトリベースの一貫性管理機構を備えることが多い．

2.2.2　キャッシュコヒーレンシプロトコル

複数のコアやクラスタが備えるキャッシュでは，キャッシュとメモリ間におけるデータの一貫性を保持するためにキャッシュコヒーレンシプロトコルが採用されている．キャッシュコヒーレンシプロトコルは，キャッシュの内容に矛盾が生じないように，キャッシュやメモリへのリクエストを管理する役割を担っている．そのため，状態を管理するプロトコルに応じてトラフィック量が変化し，実際の帯域幅に影響を与えることになる．

このプロトコルには様々な種類が存在し，その性能とスケーラビリティは個々のプロト

コルごとに異なる．基本的なプロトコルとして知られる MSI プロトコルは，キャッシュライン状態を 3 つに分けて管理する．これらの状態は，キャッシュラインの内容が無効であることを示す Invalid ，キャッシュラインの内容が有効でメモリと一致していることを示す Shared，そして，キャッシュラインの内容が有効であるが当該キャッシュにのみ存在しメモリ上の値から変更されていることを示す Modified である．MSI プロトコルでは，データをキャッシュに保持しているのが自分だけであった場合でも，他のキャッシュが同一アドレスのデータを持っていないことを把握できないため，他の全てのキャッシュに対して Invalid 化の要求を送る必要がある．

そこで，MSI プロトコルを改良したものに MESI プロトコル [9] がある．MSI プロトコルにおける Shared の状態を，自分のキャッシュだけが有効の Exclusive の状態と他のキャッシュにも同一アドレスのデータが保持されている Shared の状態に分ける．このような 4 つの状態を採用することで，書き込もうとするキャッシュラインが Exclusive 状態の場合には，他のキャッシュに同一のデータが存在しないので，Invalid 化要求を行わずに書き込むことができる．

また，MSI プロトコルでは，他のキャッシュからの読み出し要求が，Modified 状態のキャッシュラインに届くと，書き換えられたデータの内容をメモリに書き戻す必要がある．そこで，MSI プロトコルに MESI とは異なる改良を加えた MOSI プロトコルがある．先に挙げた MSI プロトコルの 3 つの状態に，書き戻し責任を負う Owned の状態を追加して，書き戻しの頻度を減少させる．具体的には，Modified 状態のキャッシュラインに対して読み出し要求が届いた場合，主記憶への書き込みをせず，他のキャッシュにデータを供給するとともにキャッシュラインの状態を Owned に変更することで書き戻しの頻度を減らす．一方で，データを受け取った側のキャッシュラインは Shared 状態となる．Owned 状態のキャッシュラインは，キャッシュから追い出される時にはその値をメモリへ書き戻す必要があるが，他のキャッシュからアクセスされた場合にはメモリへ値を書き戻す必要はない．そのため，メモリへの書き込み頻度を減少させることができる．

さらに，基本となる MSI プロトコルに Exclusive と Owned の両方の状態を追加した MOESI プロトコルがある．マルチコアプロセッサのキャッシュでは，メインメモリへのアクセス時間と比較して，プロセッサ間のキャッシュライン転送時間が非常に短く高速に実行できるため，この MOESI プロトコルや，MESI プロトコルが採用されることが多い．

2.2.3　結合網形状の検討

コア間結合網では，データ転送性能とハードウェア物量の 2 つを考慮することが重要になる．これら 2 つはトレードオフの関係にあるため，システムに適したトポロジを選択す

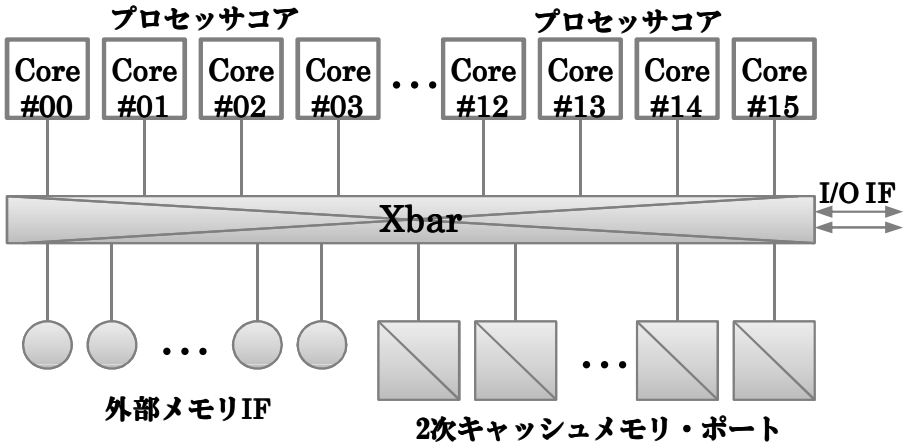


図 2: フラット型クロスバ結合網モデル

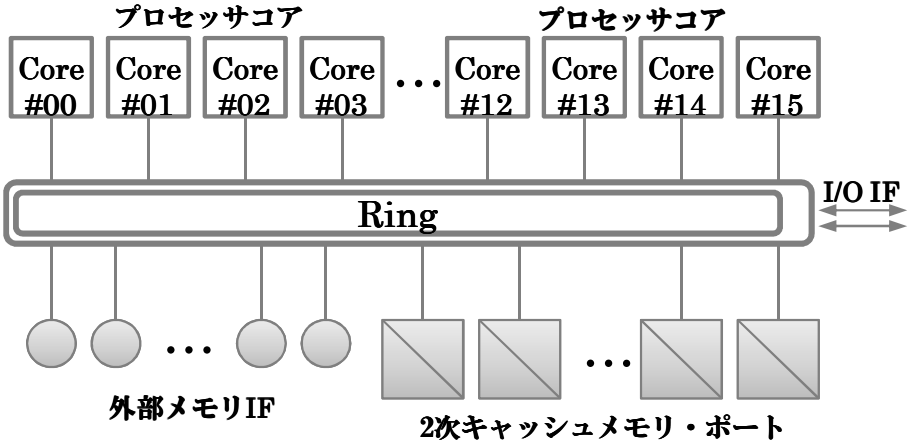


図 3: フラット型リング結合網モデル

る必要がある．このコア間結合網のトポロジとしては，様々な種類の形式が考えられる．この中で，最も転送性能が高くなる反面ハードウェア物量も大きいものの代表的な形式がクロスバ型結合網である．また，単純な構成でハードウェア物量が小さいものの代表的な形式がリング型結合網である．これら 2 種類の結合網はマルチコアプロセッサやメニーコアプロセッサシステムで広く用いられているため，本論文ではクロスバ型結合網とリング型結合網を基本としたコア間結合網に関して述べる．ここで，クロスバ型結合網，リング型結合網の両方とも，全てのコアを一つの結合網に繋いだフラット型および結合網を多段化した階層型の 2 種類が考えられる．そのため，今回とりあげる結合網形状としてはそれらを組み合わせた総計 4 種類の結合網形状が考えられる．

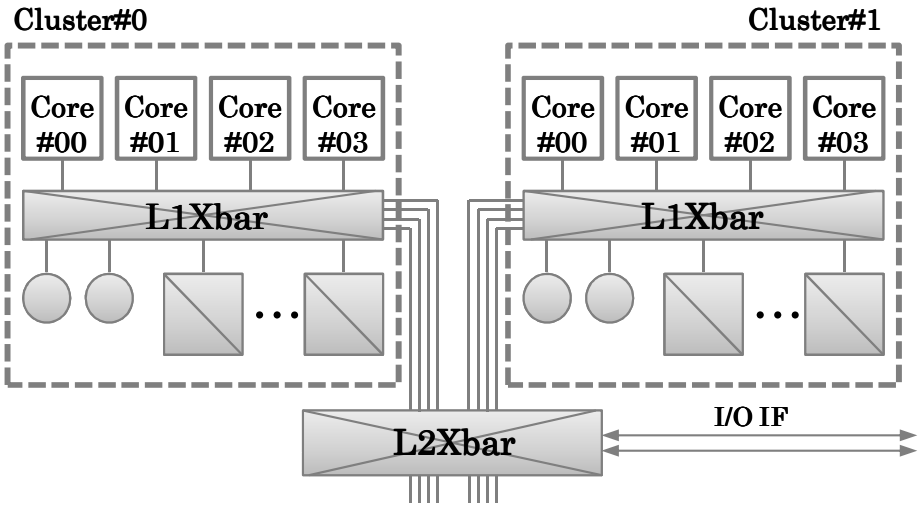


図 4: 階層型クロスバ結合網モデル

まず，フラット型クロスバ結合網モデルを**図 2** に示す．フラット型クロスバ結合網は，各プロセッサコアや 2 次キャッシュ，外部メモリインタフェース，I/O インタフェースなどの全ポートをフラットな完全結合クロスバスイッチで接続したモデルである．このモデルでは，全てのポート間で自由にデータが転送できるため，データ転送性能に最も優れるが，ハードウェア物量も最も大きい．

続いて，フラット型リング結合網モデルを**図 3** に示す．フラット型リング結合網モデルは，接続する機器の全てのポートをリンク結合網で接続する．そのため，通信パケットが全てのポートをホップするため，データを転送する際の中継数が増大し転送性能が著しく低下することが予測できる．このモデルでは，データ転送性能が最も低くなってしまうが，ハードウェア物量を最も小さく抑えることができる．

次に，階層型クロスバ結合網モデルを**図 4** に示す．階層型クロスバ結合網は，各ポートを階層化してクラスタ構成とし，各クラスタ内を完全結合クロスバスイッチで接続する．そして，各クラスタ間をさらに完全結合クロスバスイッチで接続している．このモデルでは，データ転送性能がフラット型クロスバ結合網よりも低くなるが，ハードウェア物量を抑えることができる．

最後に，階層型リング結合網モデルを**図 5** に示す．階層型リング結合網は，各ポートを階層化してクラスタ構成とし，各クラスタ内をリング結合網で接続する．そして，各クラスタ間をさらにリング結合網で接続している．このモデルでは，データ転送性能がフラット型リング結合の次に低くなってしまうが，ハードウェア物量を小さく抑えることができる．

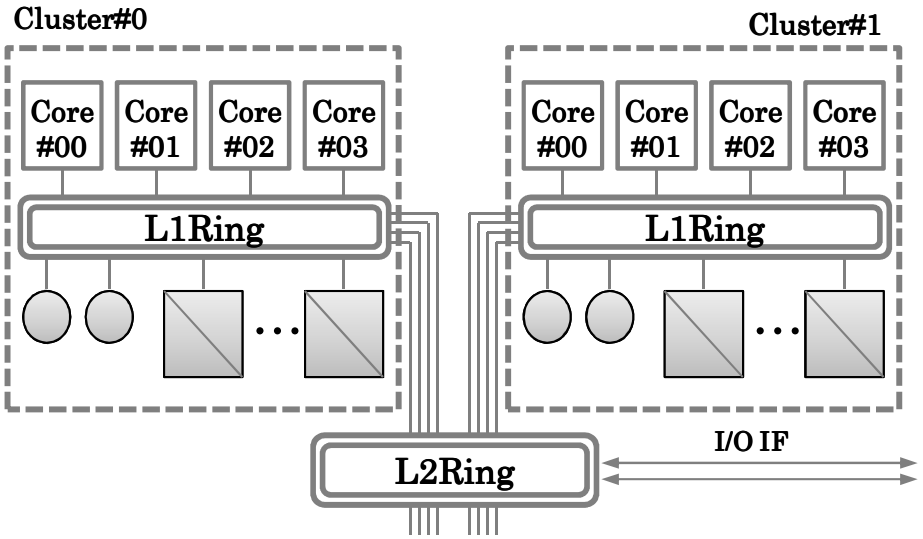


図 5: 階層型リング結合網モデル

一般的なマルチコア／メニーコアプロセッサは，以上で述べたキャッシュ構成や相互結合網形状を組み合わせて構成されている．そして，プロセッサの用途に合わせてこれらの構成を組み合わせることで，様々な処理に特化したプロセッサが開発されてきた．こうした理由から，本論文ではこれらのキャッシュ構成や相互結合網形状を基に高並列実行可能なアーキテクチャ構成を検討する．

3　既存シミュレータの問題点とその解決方針

本章では，アーキテクチャシミュレーションの重要性と，既存シミュレータの問題点および，それを解決するための方針と研究計画の概要について述べる．

3.1　アーキテクチャシミュレータ

マルチコアプロセッサアーキテクチャや，マルチコアプロセッサ時代のソフトウェアに関する研究・開発を効率的に行うためには，様々なアイディアを迅速に検証できる環境が必要となる．マルチコアプロセッサを実チップで制作することができれば，リアルな構成や実験環境を構築でき実際の環境に沿った研究が可能となる．しかし，実チップは非常に高価である上，制作に時間がかかるため実現が困難であることが多い．さらに，細かい構成やパラメータの変更を容易に行うことができないという問題もある．特に構成規模が大きくなるメニーコアプロセッサではこの問題点が顕著となる．そこで，コンピュータアーキテクチャの研究では，シミュレーションによる評価が可能なアーキテクチャシミュレー

タが重要な役割を担っている．

アーキテクチャシミュレータには大きく分けて2つの種類が存在している．そのひとつとして，FPGA（Field-Programmable Gate Array）を利用したシミュレータが挙げられる．FPGA シミュレータには，実チップによる制作に比べて安価に環境を構築できるだけでなく，開発期間を短縮できるという利点がある．しかし，配線を含めた基盤が一度組み上がってしまうと，構成の変更が必要な状況に陥った場合に最初から設計をやり直さざるを得なくなってしまう．また，シミュレーション中の内部状態の読み出しが難しい．こうした理由から，アーキテクチャの研究用途としては設計や構成方式を検討する場合，柔軟性に欠ける部分がある．

一方で，全てソフトウェアでシミュレーションするソフトウェアシミュレータがある．ソフトウェアシミュレータは，C 言語などを使って記述した一般的なソフトウェアであり，汎用コンピュータ上でも稼働させることができる．このシミュレータでは，プロセッサの設計が未確定の場合にも詳細部分を省略して全体の動作を大まかに模擬できるという利点があり，ハードウェア制約のない理想的な構成を手軽に実現できる．そのため，シミュレーションに多大な時間を要するという欠点はあるが，様々なパターンのプロセッサ構成方式を検討する場合には最も適している．

3.2　既存シミュレータとその問題点

マルチコアプロセッサシミュレータは現在までに様々なものが開発されている．FPGA を利用したシミュレータには，日本国内のプロジェクトとして ScalableCore[10] が提案・開発されている．FPGA を利用したシミュレータは，メニーコアプロセッサの動作を現実的な時間でシミュレーションすることを目的としており，プログラムに内在する並列性の活用によりソフトウェアシミュレータと比較してメニーコアアーキテクチャをより高速にシミュレーションすることを可能としている．ScalableCore では，小容量の FPGA で構成された1つのノードをタイル状に多数配置し，メッシュネットワークで接続するタイルアーキテクチャを採用している．また，ノード数に応じたシミュレーション速度のスケーラビリティを実現するために，仮想サイクルという概念を導入している．複数のクロックサイクルをかけて1仮想サイクルの動作を進め，1仮想サイクル中に対象アーキテクチャの動作およびそれに付随するユニット上の SRAM へのアクセスやユニット間通信・同期などの処理が行われる．このように仮想サイクルを用いることでシミュレーションを高速化できるが，シミュレート対象となるプロセッサ・ハードウェアの現実性を一部損なってしまっており，複雑なプロセッサモデルのシミュレーションに関しては信頼性が低い．

一方, ソフトウェアシミュレーションのひとつに, HP 研究所が開発している COTSon[11] がある．COTSon は，コンピューティングシステムを高速かつ正確に評価することを目的としたフルシステムのシミュレーションフレームワークで，機能シミュレーションとタイミングシミュレーションを組み合わせたシミュレーション環境となっている．このフレームワークでは，機能シミュレータが生成するトレースを元に，シングルコア上で動作するマルチスレッドアプリケーションの各スレッドをシミュレーション対象プロセッサの各コアにマップ・シミュレーションする．そして，タイミングシミュレータなどで構成されるバックエンド側でスレッド間の同期をとるというモデルを用いている．このモデルでは，既存のフルシステムシミュレータで動作するマルチスレッドアプリケーションをマルチコアプロセッサ上で実行した場合の性能を評価することが可能である．

また，フルシステムシミュレータの他に，アーキテクチャレベルのシミュレータが存在する．チップマルチプロセッサでは通信性能の限界がボトルネックとなりつつあり，性能効率と電力効率の良い相互結合網が求められている．電子機器はこれまでに性能要求に応えることができていたが，さらなる効率の向上には消費電力の限界値が問題となっている．そこで，ネットワークシミュレーションフレームワークの一つである OMNeT++を用いた PhoenixSim[12] が開発されている．PhoenixSim は，光通信ネットワークを用いるマルチコアプロセッサシステムのモデル化・解析のために，通信ネットワークを統合し実行するシミュレータである．従来のネットワークシミュレータと対照的に，電子に相当するものを持たない光通信における相互接続装置とネットワーク素子の測定基準および物理的な特徴を取得でき，エネルギー効率の良い高帯域な通信環境の構築を目的としている．

さらに, アーキテクチャレベルのシミュレータには, ハードウェア記述言語 SystemC[13] をベースとした NIRGAM[14] も存在する．NIRGAM は，ネットワークオンチップの研究において，様々なトポロジ上のルーティングアルゴリズムやアプリケーションの設計および実験の実質的なサポートを目的としている．そのため，拡張可能なモジュールである SystemC をベースとしており，ネットワークオンチップの設計で利用可能な様々なオプション機能を備えている．このシミュレータのオプションとしては，トポロジやスイッチング機構，ルーティング機構などがあり，新しいルーティングアルゴリズムを容易に実装できる．

他にも，ある特定のアーキテクチャをシミュレーションするのに特化したシミュレータとして，Shaksham[15] や MC-Sim[16] が存在する．Shaksham は，x86 系プロセッサのシミュレーションに特化している．初期世代から最新機まで歴代すべての x86 系のプロセッサの構成を組み替えて検証を行うことで，今後の新しい x86 系プロセッサの構成を探るこ

とを目的としている．また，MC-Sim は Cell/B.E. などに代表されるヘテロジニアスなマルチコアプロセッサのシミュレーションを正確に行うことができる．このシミュレータはヘテロジニアスマルチコア特有の，コプロセッサの構成やプロセッサ間のネットワーク構成を様々なものに変更し，シミュレーションすることができる．

このように，マルチコア／メニーコアを対象としたシミュレーション環境の研究が数多く行われているが，これら既存のシミュレータはいずれも共有メモリを前提として構築されている．しかし，メニーコアプロセッサで複数のコアを単一の主記憶装置へ接続してしまうと，メモリアクセスによるボトルネックが顕在化する危険性があるため，全コア共有のメモリシステムを想定することは現実的ではない．また，メニーコアプロセッサではメッセージパッシングなどの方式を採用する必要性が増す上，キャッシュシステムなどによるメモリ帯域幅の確保も重要になると考えられる．こうした理由から，既存のシミュレータのシミュレーションコア数を増大させるだけでは，メニーコアプロセッサのシミュレーションを行うことは難しい．

そこで本論文では，メニーコアプロセッサの理想的な構成を検討するために，様々な構成方式を検証可能なメニーコアシミュレータを開発する．プロセッサ構成の検討にあたって，2.2 節で述べたデータ供給方式と結合網形状に着目することで，配線遅延を考慮した高並列実行可能なアーキテクチャを模索していく．このような目的の下，プロセッサ構成の各方式の実現可能性を検証し，さらに各方式を組み合わせた様々な構成パターンを実現するために，本研究ではソフトウェアによるシミュレーションを採用する．

3.3 研究計画概要

本メニーコアプロセッサ研究の計画は大きく 3 つのステップに分けることができる．第 1 ステップでシミュレータを開発し，第 2 ステップで，開発したシミュレータを用いてメニーコアプロセッサの各構成方式の検証を行う．そして第 3 ステップで，高並列実行可能な並列アーキテクチャを模索する．これら 3 つのステップについて順に説明する．

まず第 1 ステップでは，メニーコアシミュレータを開発する．ハードウェア・アーキテクチャの検討にあたっては，性能目標値を導出する必要がある．そこで，基本となるメニーコアプロセッサの構成を設計し，代表的なアプリケーションを実行可能なシミュレータを開発する．そして，実行時におけるトレースを採取することで，メニーコアプロセッサ構成方式の諸検討に利用する．以下，これらの機能を備えたソフトウェアシミュレータを**メニーコアトレースシミュレータ（Manycore Traced Simlator）**と呼ぶ．

次の第 2 ステップでは，メニーコアプロセッサ構成方式の検証を行う．メニーコアプロ

セッサの実現において重要となる配線遅延を考慮して，キャッシュ構成やメモリ一貫性プロトコル等のデータ供給方式および，複数のコアやメモリを相互に結合し交信路を提供する相互結合網の様々な形状を構築する．また，メニーコアトレースシミュレータの実現のために基本構成だけでなく，各構成方式を組み合わせることで様々な構成パターンをシミュレートすることを目的とし，それらの実行トレースをそれぞれ採取する．

最後に第3ステップで，高並列実行を実現する最適なアーキテクチャを模索する．これまでのステップで得られた実行トレースの結果から，それぞれのメニーコアプロセッサ構成方式の比較・検討・考察を行う．また，それら構成方式において，単一プログラムを並列化して実行する場合の並列度の限界を調査する．多くのプログラムは潜在的な並列性を持っているが，抽出できる並列度は一般に高くない．そのため，多数のコアを有効に利用するためには単純な並列化のみならず他の高速化技術との組み合わせが必要になると考えられる．そこで，これまで提案されてきた他の高速化技術を組み込むことも視野に入れ，プロセッサ高速化技術の進むべき道筋を示すことが本研究の最終的な目標となる．

以上の3つのステップの中で，本論文では主に1つ目のメニーコアトレースシミュレータの開発について述べる．まずは，2.2節で述べたプロセッサ構成方式から，基本となるアーキテクチャを設計し，メニーコアトレースシミュレータで動作させるメニーコアプロセッサの基本構成を実装する．

4　メニーコアトレースシミュレータの開発

性能目標値を導出するためにメニーコアプロセッサの基本構成を設計し，その実行トレースを採取可能なメニーコアトレースシミュレータを開発する．

4.1　アーキテクチャ設計

データ供給方式および相互結合網の形状において，基本となるプロセッサ構成を検討する．キャッシュ構成については，プロセッサコア数を増大させることを考慮して，L2キャッシュを複数のバンクに分割したクラスタを複数備える構成とし，ディレクトリベースの一貫性管理機構を持たせる．また，最近のプロセッサでは，内蔵キャッシュを3階層とすることが一般的である．1次キャッシュをプロセッサコアに内蔵し，2次キャッシュをコアごとに内蔵するもしくは2コア程度で共有し，3次キャッシュを全コアで共有するといった構成をとることが多い．しかし，メニーコアプロセッサでは低消費電力化を目的とし，プロセッサコアの動作周波数を下げる傾向がある．そのため，高周波数のプロセッサと異なり，アクセス時間の観点からキャッシュの階層を増大させる必要性は小さい．ま

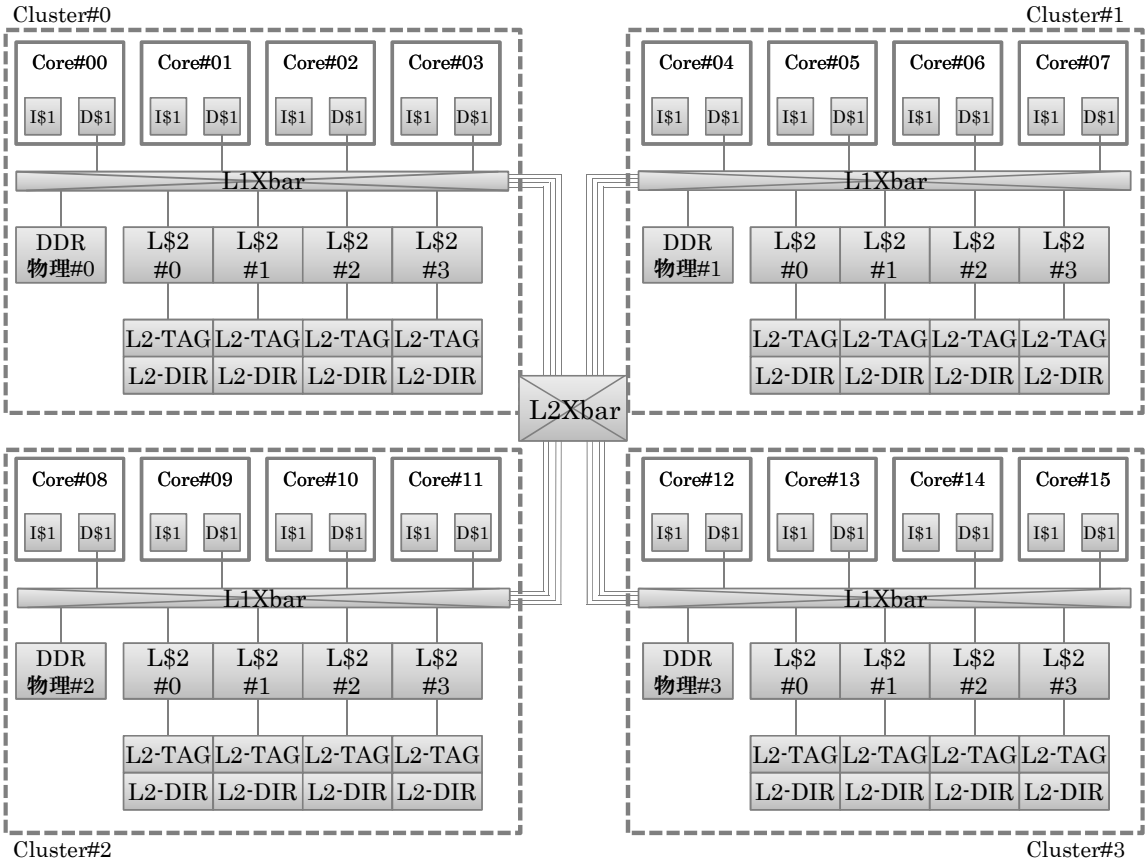


図 6: メニーコアトレースシミュレータのアーキテクチャ構成

た，キャッシュ階層を増大させた場合には，キャッシュミス時のオーバヘッドが大きくなり制御機構も複雑化する．そのため，今回の構成では 2 階層のキャッシュを採用する．

キャッシュコヒーレンシプロトコルは，各階層において別々のものを選択することができる．そこで，各クラスタの持つ 2 次キャッシュには，メインメモリへのアクセス回数を減少させるために，書き戻し頻度を削減できる MOESI プロトコルを採用する．また，各コアの持つ 1 次キャッシュには，一貫性管理の複雑度を抑えつつ効率的なキャッシュアクセスが可能な MESI プロトコルを採用する．一方で，相互結合網の形状には，2.2.3 項で述べた 3 種類の結合網形状の中で，データ転送性能，ハードウェア物量，消費電力のバランスが最も良いと考えられる階層型クロスバ結合網を選択する．

以上で述べた構成方式をもとに設計したアーキテクチャの構成図を**図 6** に示す．最小構成として 16 コア内蔵の 2 階層型クロスバ結合の cc-NUMA アーキテクチャを採用した．また，全体で 4 クラスタ構成となっており，1 クラスタに 4 つのコアが内蔵されるとする．そして，各コアと 2 次キャッシュ（L$2）がクロスバに接続され，各コアはローカルに命令

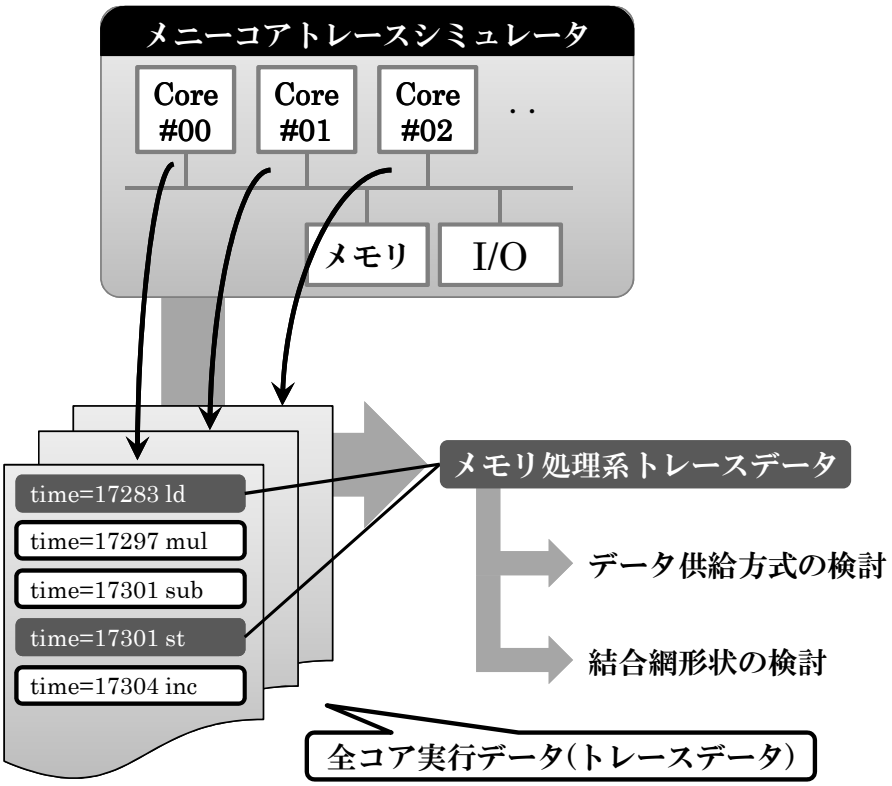


図 7: トレースデータ抽出フロー

キャッシュ（I$1）とデータキャッシュ（D$1）を内蔵している．また，主記憶装置 DDR はメモリの帯域幅を考慮して各クラスタに均等分割し，クロスバの 1 つに接続する．

ここで，図中の L2-TAG とは 2 次キャッシュの各バンクに付属するタグであり，キャッシュラインの現在の状態をラインごとに保持しており，これを利用してキャッシュのコヒーレンシを保証する．また，このタグは 1 次キャッシュのディレクトリ機構も担っており，同じクラスタ内に属するコアが持つ 1 次キャッシュに格納されている同一アドレスのデータの状態も保持している．L2-DIR は，2 次キャッシュを管理するディレクトリ機構であり，2 次キャッシュの各バンクに付属する．この機構では，2 次キャッシュの各ラインが保持するデータと同じデータを持つ，他クラスタの 2 次キャッシュのライン状態を保持している．

4.2　実行トレースの採取

メニーコアトレースシミュレータの開発では，代表的なアプリケーションを実行させてプログラム実行中のトレースデータを採取することを目的の一つとしている．そのため，シミュレータの基本的な機能としてシミュレーションのトレース機能を搭載している．こ

こで，メニーコアプロセッサのように大規模システムのトレースを採取する場合には，膨大なサイズのトレースデータが生成されてしまうことになる．そのため，今回のメニーコアトレースシミュレータでは，各コアで採取したトレースデータの中から，特にメモリ処理（ロード・ストア）だけを抽出する．このトレースデータの抽出フローを**図 7** に示す．メニーコアトレースシミュレータの各プロセッサコアがベンチマークアプリケーションを実行した際の全コア実行データを採取し，メモリ処理だけを抽出することでメモリ処理系のトレースデータを作成する．それを圧縮形式でファイルに書き込み，データ供給方式や相互結合網の形状における諸検討で統一的に利用する．

メモリ処理系のトレースデータにおけるトレースフォーマットは，メモリアクセスの種別，参照アドレス，アクセス時刻の情報のみを構成要素とする．そして，1 回のメモリアクセスを 1 レコードとし，複数のレコードを時間順にファイルに書き込む．具体的には，メモリアクセスの種別を 1 ビット，参照アドレスを 64 ビット，アクセス時刻をそのタイミングにおけるサイクル数の 64 ビットで表す．そのため，1 つのレコードを約 16 バイトで記述することが可能となる．このように，採取するトレースデータを限定することで，膨大なデータ量を大幅に削減できる．

4.3 動作モデル

次に，メニーコアトレースシミュレータの動作について説明する．

4.3.1 シミュレータコアの起動と同期処理

プログラムを複数のコアで並列実行するために，マルチスレッド化ライブラリが広く利用されている．マルチスレッド化ライブラリにはいくつかの種類が存在するが，いずれも OS がスレッドの生成からスケジューリングまでを行うという点においては共通している．しかし，現在のメニーコアトレースシミュレータはフルシステムシミュレータではないため，OS が存在せず，簡単なシステムコールをシミュレートすることしかできない．そのため，一般的に使われているマルチスレッド化ライブラリを使用したプログラムをシミュレータ上で実行できない．

そこで，メニーコアトレースシミュレータ向け専用 Pthread ライブラリを実装した．このライブラリは，広く利用されている Pthread ライブラリと同じ API を提供することで，シミュレータ上で実行するプログラムの書き換えが必要とならず，ユーザへの負担を増やすことはない．現在，専用 Pthread ライブラリが提供している関数を**表 1** に示し，それぞれの動作モデルについて概説する．

まず，スレッドを生成するための関数である pthread_create() が呼び出されると，引数

表 1: 専用 Pthread ライブラリが提供している関数

pthread_create()	スレッドの生成
pthread_join()	スレッドが終了するまで待機
pthread_mutex_init()	ロック用変数を初期化
pthread_mutex_lock()	ロックをかける
pthread_mutex_unlock()	ロックの解除
pthread_barrier_init()	バリア同期用変数を初期化
pthread_barrier_wait()	指定したスレッド数でバリア同期する

に指定されたスレッド ID に対応する番号のシミュレーションコアが起動する．その後，起動したコア上でスレッドのシミュレーションが開始される．なお，現在のメニーコアトレースシミュレータでは 1 つのシミュレーションコアにつき 1 つのスレッドを固定して実行する．その後, スレッドの実行が終了すると，pthread_join() が呼び出され，各シミュレーションコアを停止状態へ移行させる．ただし，シミュレーションコアを停止する前に，そのコアの 1 次キャッシュ上に最新の値（Modified）が存在する可能性があるため，1 次キャッシュを 2 次キャッシュへ書き戻す．また，所属しているクラスタ内のうち最後に停止状態に移行するシミュレーションコアは，1 次キャッシュの書き戻し後に 2 次キャッシュもメインメモリへ書き戻す．

また，全スレッドはメモリ領域を共有しているため，同じアドレスのデータに対して競合が発生する場合がある．こうしたスレッド間での競合を回避するためには，一般的に pthread_mutex_lock() と pthread_mutex_unlock() を用いて排他制御を行う．これらを利用してロックした際に，リソースを獲得できなかったコアは待ち状態へと移行する．こうしたコアの待ち状態を実現する方法には，スリープ状態に移行し CPU リソースを節約するような実装も考えられるが，簡単のためスピンロックで待ち状態を実現した．排他制御を行うためには，まずロックを管理するための変数であるロック変数を pthread_mutex_init() を用いて初期化する必要がある．そして, クリティカルセクションに入る前に pthread_mutex_lock() を使用する．ここで, 提供している pthread_mutex_lock() のフローチャートを**図 8** の (1) に示す．専用 Pthread ライブラリでは，ロックを管理するための変数（lock）がロック状態（LOCKED）ではない時だけアトミック命令を実行する，悲観的スヌーピングロック方式を採用している．これは，アトミック命令を頻繁に実行するとキャッシュコヒーレンシ要求が頻発し，メモリトラフィック量が爆発的に増加

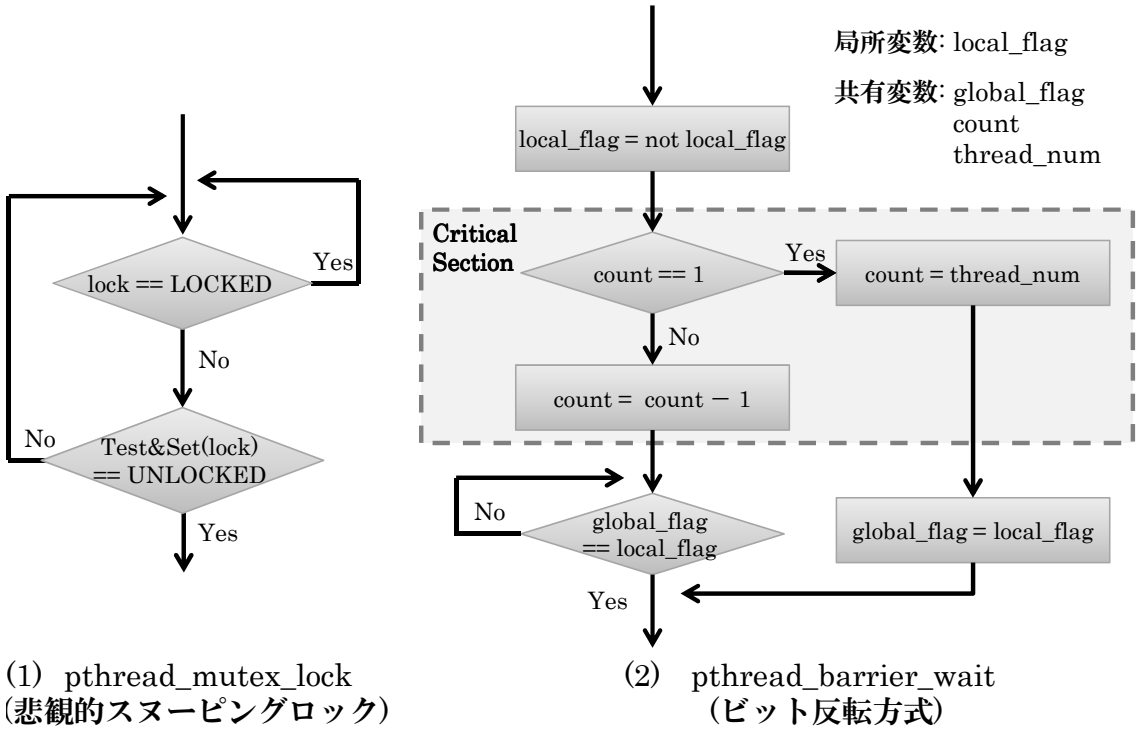


図 8: pthread_mutex_lock() と pthread_barrier_wait() のフローチャート

してしまうのを回避するためである．その後，クリティカルセクションを抜けるスレッドは pthread_mutex_unlock() を実行しロックを開放する．

最後に, 複数スレッドをあるタスクの完了時に同期させる場合には, pthread_barrier_wait() を使用する．提供している pthread_barrier_wait() のフローチャートを図 8 の (2) に示す．このバリア同期を使用する前に，まずはバリア変数を初期化する必要がある．この初期化には pthread_barrier_init() を使用し，同期させたいスレッドの数を記憶させる（thread_num）．その後，pthread_barrier_wait() に到達したスレッドは，ローカルフラグ（local_flag）を反転させ，バリアに到達したスレッド数をカウンタ（count）で把握する．もし，自分が同期をとる最後のスレッドではない，つまりカウンタの値が 1 では無い場合，カウンタをデクリメントする．その後，グローバルフラグ（global_flag）がローカルフラグ（local_flag）と一致するまでスピンし続ける．逆に，自分が同期をとる最後のスレッドだった場合，つまりカウンタの値が 1 の場合，カウンタの値をスレッド数（thread_num）で初期化し，グローバルフラグ（global_flag）をローカルフラグの値で上書きする．これにより，待機していた他のスレッドはバリアから抜ける事ができる．なお，共有変数 count を読み書きする際には，pthread_mutex_lock() と pthread_mutex_unlock() を利用して排

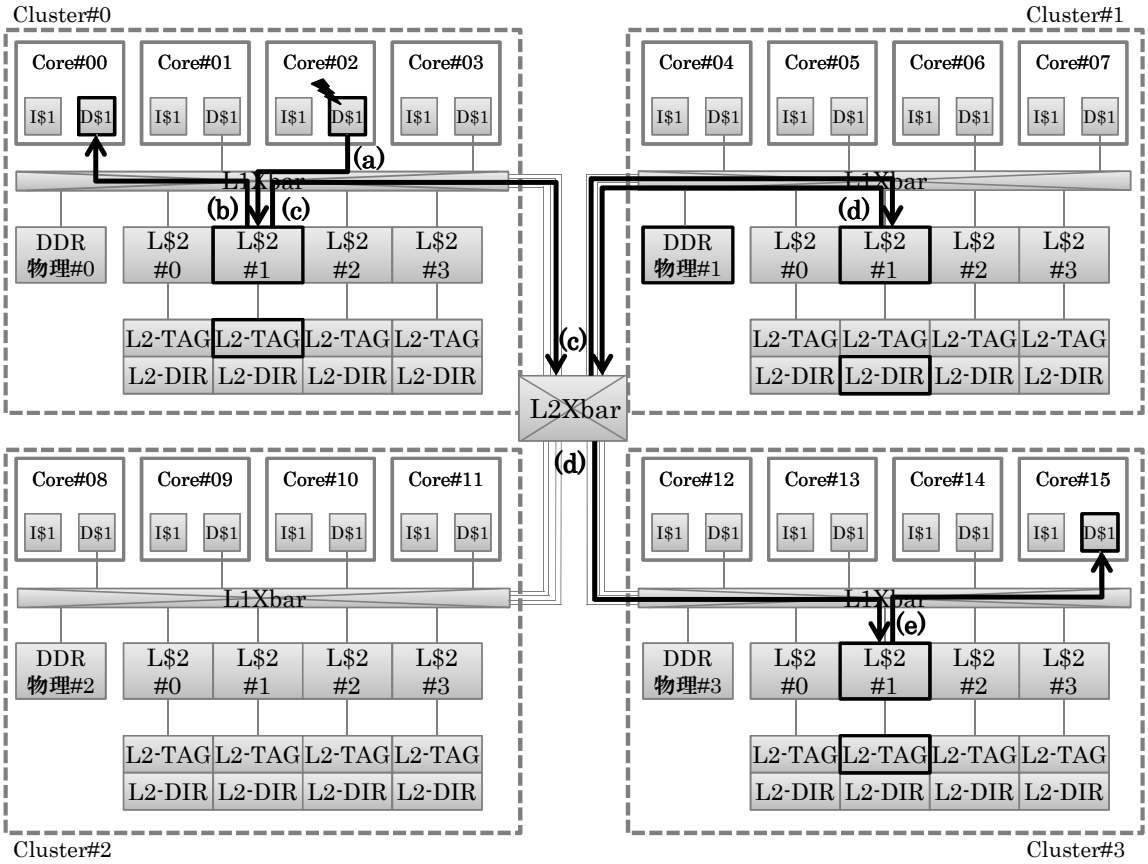


図 9: リクエストの流れ

他制御を行う.

4.3.2 キャッシュリクエスト

キャッシュリクエストが発生するとキャッシュコヒーレンスプロトコルが働き，各コアおよび各クラスタ間でリクエストが流れる．ここで，リクエストの流れる様子を**図 9** に示す．まず，あるコアがロードもしくはストア命令を発行した際，1 次キャッシュミスが発生するとデータを取得するために 2 次キャッシュに向けてリクエストを送信する必要がある．そのため，参照アドレスから要求するデータを担当するバンクを割り出し，該当バンクへキャッシュリクエストを送信する (a)．そして，2 次キャッシュの各バンクに付属される L2-TAG を参照することで，キャッシュラインの状態を知ることができる．また同時に，クラスタ内に属する他の 1 次キャッシュの状態を取得でき，必要があれば Invalid 化や Shared 化などのコヒーレンシ要求を該当コアに向けて送信する (b).

また，2 次キャッシュが Invalid 状態の場合には，メモリもしくは他のクラスタが持つ 2 次キャッシュからデータを取得する必要があるため，さらにキャッシュリクエストを伝

播させる必要がある．この時，キャッシュリクエストは該当データを保持する主記憶を持つクラスタへ送信することになる (c)．ここで，主記憶にデータを保持するクラスタにおいて，参照すべき同一番号のバンクを管理する L2-DIR は，要求データに対して全クラスタの 2 次キャッシュのライン状態を保持している．要求元以外のクラスタにもデータがキャッシュされていなければ主記憶アクセスが発生し，メモリからデータを取得できた時点で要求元クラスタへデータとともに Ack を返す．またこの際，L2-DIR における要求元クラスタのライン状態を Invalid から Exclusive に変更する．一方で，Ack を受け取ったクラスタ側では，受け取ったデータを 2 次キャッシュに格納し L2-TAG の状態を更新する．そして，データを 1 次キャッシュに伝搬する．

一方で，要求元以外のクラスタにデータがキャッシュされている場合には，Invalid 化や Shared 化などのコヒーレンシ要求およびデータの転送要求を該当クラスタへ送信する (d)．しかし，要求を受け取ったクラスタ側では，クラスタ内に属する 1 次キャッシュの状態を考慮しコヒーレンシ要求を送信しなければならない場合がある (e)．もし，要求データを持つ 1 次キャッシュが存在し，キャッシュラインの状態が Modified であるならばそのデータを 2 次キャッシュに書き戻す必要がある．ここで，1 次キャッシュは MESI プロトコルにより管理されているため，データ転送要求を受け取った際にはデータを伝播する必要が生じる．また，要求元のコアがストア命令を発行していた場合には，該当クラスタに Invalid 化要求が到達する．その際，クラスタに属する 1 次キャッシュのキャッシュラインも無効化するために，データをキャッシュしている 1 次キャッシュに Invalid 化要求を伝搬する必要も出てくる．

データ転送要求を受け取ったクラスタは，コヒーレンシ要求に応じて L2-TAG の状態を更新し要求データとともに Ack を返す．そして，Ack を受け取ったクラスタすなわち該当データを保持する主記憶を持つクラスタでは，変更した他クラスタの L2-TAG の状態を記憶するために L2-DIR を更新し，転送されてきたデータとともに要求元クラスタへ Ack を返す．以降は，主記憶アクセスが発生する場合と同様の処理を適用する．このように，全ての要求に Ack が返ってくると 1 つのキャッシュリクエストに対する処理が完了する．

4.3.3　リクエストの衝突と優先順位

各コアからのキャッシュリクエストが頻繁に飛び交うようになると，キャッシュリクエストやメモリリクエスト，コヒーレンシ要求の衝突が発生することになる．ここで，複数のコアから同時にキャッシュリクエストが発生した様子を**図 10** に示す．図 10 中では 2 つのコアから 2 次キャッシュの異なるバンクへ参照リクエストが発生している．この場合

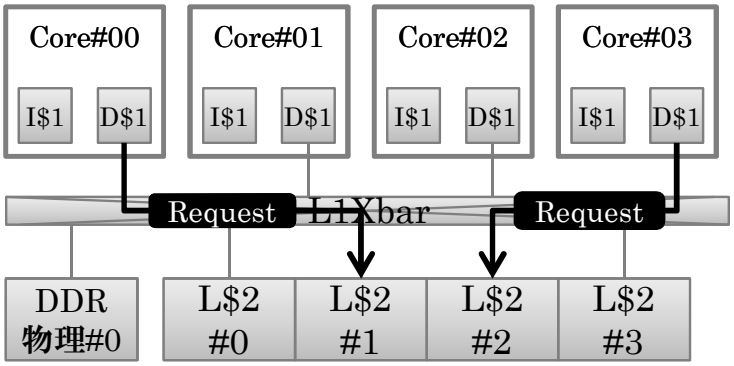


図 10: 異なるバンクへの参照リクエスト

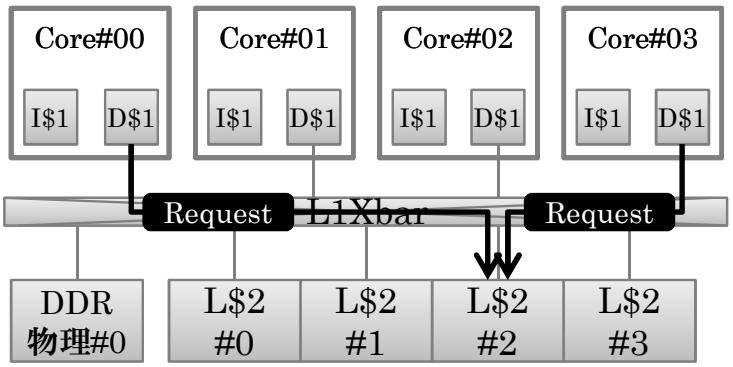


図 11: 同一バンクへの参照リクエスト

には，2.2.1 項で述べたように，複数コアからのリクエストを同時に受け付けることができ，並列して要求に応じることができる．次に，同時に同一バンクへキャッシュリクエストが発生した様子を**図 11** に示す．もし同一バンクへ参照リクエストが同時に発生した場合には，バンク内でキャッシュリクエストを順位付けし，優先されたリクエストから順に処理を進める必要がある．この際に，優先されたリクエストすなわちリソースを獲得できたリクエストは，該当バンクに対してロックを掛けることで他のリクエストを待機させ，キャッシュとメモリ間の一貫性を保持する．そして処理が終わり次第，ロックを解放することで次のリクエストに対する処理に移る．その後，待機していたリクエストは再び順序付けがなされ，リソースを獲得できた順から参照リクエストに対する処理が遂行される．ここで，2 次キャッシュへの参照リクエストだけでなく，1 次キャッシュへのコヒーレンシ要求やクラスタ間のリクエストの伝播時においても一貫性を保持するためにこうした順序付けが必要になる．

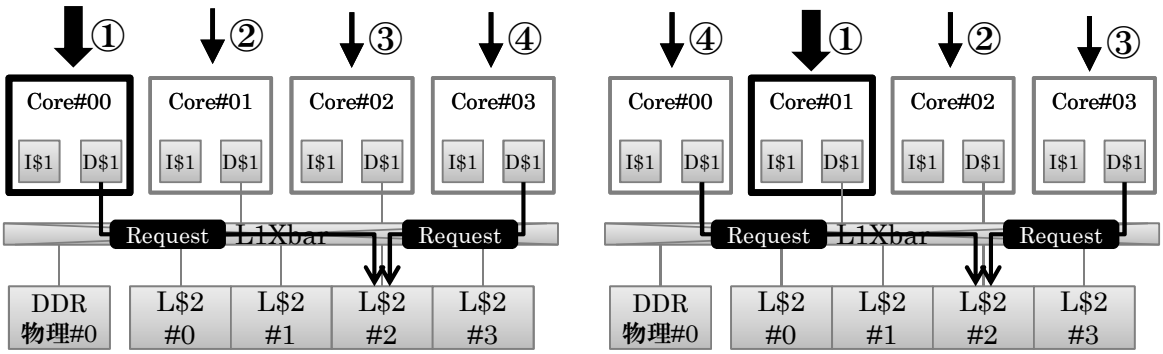


図 12: ラウンドロビン方式によるリクエスト優先順位の決定

さて，キャッシュリクエストの順序付けにはいくつかの方法が考えられる．最も簡単なものに，要求元のコア番号に応じて優先度を変える方法が挙げられる．例えば，2 つのコアから同時に参照リクエストが到達した場合には，コア番号の小さいリクエストを優先する手法が存在する．この方法では，複雑な機構を必要とせずにハードウェア物量を最小限に抑えることができるが，優先度の低いコアからの要求にリソースが回りにくくなるという欠点がある．そのため，複数のコアが頻繁にキャッシュリクエストを送信するような状況では，優先度の最も低いコアの参照リクエストはいつまでもリソースを獲得できず一種の飢餓状態になってしまう恐れがある．

次に，サイクル毎のラウンドロビン・スケジューリング方式により順序付けする方法が挙げられる．この手法では，処理待ちの参照リクエストに対し，サイクル毎に優先度を変更することで，コア番号依存の方式におけるリソーススタベーションの問題を解決できる．またこの手法は比較的実装が容易であるため，追加ハードウェア量も少なくすむという利点がある．しかし，各リクエストの転送速度の違いを考慮していないため，必ずしも最適な処理順序付けができるわけではない．

最後に，参照リクエストの要求順に応じて優先度を変える方法が挙げられる．この方法では，全てのコアにおけるロード・ストア命令の発行順に応じてリソースが割り与えられる．そのため，最もリソースを公平に利用することができる．しかし，要求順を記憶するためにリクエストキューを管理する機構が必要になるなど，追加ハードウェア量が増大してしまう．

開発するメニーコアトレースシミュレータでは，追加するハードウェア量やキャッシュリクエストの処理効率の両方を考慮して，サイクル毎のラウンドロビン・スケジューリ

ング方式を採用する．ラウンドロビン方式によるリクエスト優先順位の決定の様子を**図 12** に示す．この方式では，どのコアからの処理を優先するかサイクル毎に変更する．これは，シミュレーション実行開始時からの経過サイクル $Cycle$ をクラスタに属するコア数 $PENUM$ で割った余りの値を用いて優先度の最も高いコアを決め，そのコアから順にリクエストの有無を確認することで実現する．例えば，図 12(1) で示される状況では，$Cycle\%PENUM = 0$ であるため，コア#00 の優先度が最も高い．そのため，同じ時刻にコア#03 が，コア#00 と同じバンクへリクエストを送信していたとしても，コア#00 のリクエストが優先的に処理される．図 12(2) は，図 12(1) の次のサイクルにおけるリクエストの優先順位を表している．前のサイクルで最も優先度の高かったコアからのリクエストは，次のサイクルで優先順位が決定する場合には最も優先度が低くなる．

5　シミュレータの動作検証

開発したシミュレータ上でベンチマークプログラムを実行し，各プログラムの特徴を調査することで今後メニーコア研究を進めていくためのシミュレータとして妥当であるかを検証した．

5.1　動作環境

シミュレータは単命令発行の SPARC V9 アーキテクチャをベースとしている．評価に用いたパラメータを**表 2** に示す．プロセッサ構成は，4.1 節で述べたように，16 コア内蔵の 2 階層型クロスバ結合の cc-NUMA アーキテクチャとなっている．また，動作検証の初期段階であることを考慮して，データ管理の複雑度を抑えるために，データキャッシュの構成としてダイレクトマップ方式を採用している．動作検証対象のプログラムには，汎用ベンチマークプログラムである SPLASH-2 ベンチマーク [17] から FFT，LU，WATER-NSQUARED の 3 つを用いた．なお，本評価では 1 サイクル毎に全コアの動作およびキャッシュシステム，メモリの動作をシミュレートする方法をとっている．

5.2　動作結果

各プログラムを 1，2，4，8，16 スレッド実行でそれぞれシミュレートした．このとき，各コアには 1 スレッドずつ割り当てて実行している．SPLASH-2 FFT の実行結果を**表 3** に，LU の実行結果を**表 4** に，WATER-NSQUARED の実行結果を**表 5** に示す．表の各項目は左から順に，コア番号を示す pid，発行命令数を示す step，命令実行サイクル数を示す exec，I1 キャッシュミスにおける待ちサイクル数を示す I1wait，D1 キャッシュ

表 2: シミュレータ諸元

Processor	SPARC V9
number of cores	16 cores
number of clusters	4 clusters
issue width	single issue
issue order	in-order
register windows	8 sets
I1 cache	16 KBytes
line size	64 Bytes line
ways	4 ways
latency	1 cycle
D1 cache	16 KBytes
line size	64 Bytes line
ways	1 way
latency	1 cycle
D2 cache	16 MBytes
number of banks	4 banks
line size	64 Bytes line
ways	1 way
latency	8 cycles
Memory	256 MBytes
latency	32 cycles
Cache Coherency Protocol	
L1 cache	MESI
L2 cache	MOESI
Interconnect Network	2-Layer Cross Bar Topology
link latency	8 cycles

ミスにおける待ちサイクル数を示す D1wait，総実行サイクル数を示す cycle となっている．また，I1 キャッシュと D1 キャッシュのキャッシュヒット率を別途算出し，それぞれ I1hit， D1hit の欄に記載してある．

表 3: 実行サイクル数 (SPLASH-2 FFT)

	pid	step	exec	I1wait	D1wait	cycle	I1hit	D1hit
1thread	0	6769701	7608989	212257	150306	**7971552**	99.64%	99.27%
2thread	0	6593478	7385398	212240	95280	**7692918**	99.63%	99.57%
	1	182063	232907	2449	42386	277742	99.97%	92.54%
4thread	0	6505722	7273194	212016	70446	**7555656**	99.63%	99.72%
	1	94211	120721	2530	19811	143062	99.94%	95.94%
	2	94464	121001	2218	20310	143529	99.93%	96.38%
	3	94672	121401	2165	20055	143621	99.93%	96.41%
8thread	0	6463295	7219029	212224	70780	**7502033**	99.62%	99.74%
	1	52354	67178	2660	18117	87955	99.88%	96.07%
	2	50838	65553	2403	19946	87902	99.87%	95.29%
	3	52266	67155	2959	17816	87930	99.86%	96.25%
	4	50854	65881	2110	19937	87928	99.87%	96.03%
	5	50908	65796	2475	19655	87926	99.87%	95.41%
	6	50896	66441	2927	18556	87924	99.88%	95.22%
	7	51265	66383	2538	19001	87922	99.87%	95.97%
16thread	0	6451168	7203082	213352	78876	**7495310**	99.62%	99.74%
	1	37609	48008	3808	28501	80317	99.82%	94.65%
	2	36849	47225	3684	29355	80264	99.80%	94.91%
	3	37979	48951	3098	28213	80262	99.81%	94.94%
	4	34454	44252	2652	33356	80260	99.81%	94.77%
	5	35530	45618	2833	31807	80258	99.82%	93.62%
	6	35410	45319	2931	32072	80322	99.80%	94.67%
	7	33782	43554	3519	33247	80320	99.82%	93.28%
	8	33115	42915	2586	34817	80318	99.80%	94.10%
	9	35348	45414	2056	32860	80330	99.82%	94.77%
	10	34830	44942	2968	32418	80328	99.81%	94.45%
	11	32884	42949	2498	34879	80326	99.80%	93.03%
	12	34239	44345	2674	33305	80324	99.81%	93.23%
	13	34898	45144	3254	31924	80322	99.81%	94.58%
	14	33857	43842	2657	33821	80320	99.81%	93.14%
	15	35979	46836	3322	30160	80318	99.82%	93.64%

表 4: 実行サイクル数 (SPLASH-2 LU)

	pid	step	exec	I1wait	D1wait	cycle	I1hit	D1hit
1thread	0	539252692	910825937	22595	114662493	**1025511025**	100.00%	88.57%
2thread	0	287438805	554184781	22253	34099892	**588306926**	100.00%	93.73%
	1	254942231	372200660	2113	33757607	405960380	100.00%	93.12%
4thread	0	161216771	373503119	19939	11623690	**385146748**	100.00%	96.17%
	1	128781770	191785772	1956	10999801	202787529	100.00%	95.74%
	2	128743491	191766890	2455	11019042	202788387	100.00%	95.72%
	3	128578050	191552779	1994	11233874	202788647	100.00%	95.63%
8thread	0	98600660	283035073	20109	9153424	**292208606**	100.00%	94.90%
	1	65854434	100870541	1941	8970664	109843146	100.00%	93.08%
	2	66087903	101115738	3346	8725294	109844378	100.00%	93.31%
	3	65999746	101006738	2361	8835277	109844376	100.00%	93.11%
	4	65947082	100913380	2140	8928851	109844371	100.00%	93.10%
	5	66041780	101057457	2481	8784711	109844649	100.00%	93.30%
	6	65894607	100825044	2258	9017355	109844657	100.00%	93.13%
	7	65945922	100944190	2131	8898334	109844655	100.00%	93.27%
16thread	0	67485068	238466099	20362	3059593	**241546054**	100.00%	97.60%
	1	35154873	56744059	2207	2421051	59167317	100.00%	96.86%
	2	35167623	56781514	4774	2381027	59167315	100.00%	96.91%
	3	35094925	56716926	2511	2447876	59167313	100.00%	96.83%
	4	34971467	56653037	3015	2511256	59167308	100.00%	96.84%
	5	35003058	56642208	3068	2522422	59167698	100.00%	96.86%
	6	34933985	56572774	2941	2591981	59167696	100.00%	96.79%
	7	34854810	56446029	2357	2719308	59167694	100.00%	96.79%
	8	35002834	56630178	4132	2533382	59167692	100.00%	96.87%
	9	34789666	56377140	2960	2787597	59167697	100.00%	96.76%
	10	34977165	56568978	2339	2596376	59167693	100.00%	96.81%
	11	35076989	56683607	3096	2480989	59167692	100.00%	96.90%
	12	34851666	56511041	4097	2652553	59167691	100.00%	96.75%
	13	34936114	56639710	4257	2523723	59167690	100.00%	96.75%
	14	34955029	56664085	2957	2500645	59167687	100.00%	96.82%
	15	34949641	56585734	2430	2579521	59167685	100.00%	96.82%

表 5: 実行サイクル数 (SPLASH-2 WATER-NSQUARED)

	pid	step	exec	I1wait	D1wait	cycle	I1hit	D1hit
1thread	0	679195744	959654979	454990	20682780	**980792749**	99.99%	98.53%
2thread	0	363092069	502281305	283041	11767799	**514332145**	99.99%	98.54%
	1	348845587	498595554	11047	10453430	509060031	100.00%	98.59%
4thread	0	194743093	268492511	234982	8882438	**277609931**	99.99%	98.45%
	1	187450485	261678390	11479	7000127	268689996	100.00%	98.64%
	2	181113711	261549286	11544	7129164	268689994	100.00%	98.60%
	3	190609733	262176103	106537	6407418	268690058	99.99%	98.68%
8thread	0	105161763	145691771	274408	8310631	**154276810**	99.98%	97.64%
	1	96948648	133907700	10340	3748169	137666209	100.00%	98.65%
	2	97720431	134099708	12478	3554021	137666207	100.00%	98.68%
	3	97947968	134403543	10081	3252899	137666523	100.00%	98.75%
	4	93847440	133923736	11052	3731727	137666515	100.00%	98.56%
	5	95662825	134227345	11144	3428517	137667006	100.00%	98.69%
	6	97799575	134212051	63492	3391461	137667004	99.99%	98.70%
	7	92295345	133910016	11516	3745470	137667002	100.00%	98.62%
16thread	0	63339405	91016899	386948	12062364	**103466211**	99.95%	94.50%
	1	49518427	67737452	12402	2241754	69991608	100.00%	98.77%
	2	49296715	67743214	10424	2237968	69991606	100.00%	98.63%
	3	49570194	67806519	13543	2171542	69991604	100.00%	98.82%
	4	47512434	67746045	12488	2233063	69991596	100.00%	98.78%
	5	47211641	67266040	10914	2715023	69991977	100.00%	98.34%
	6	49117488	67808284	11487	2172204	69991975	100.00%	98.76%
	7	46479487	67315040	10324	2666609	69991973	100.00%	98.54%
	8	46611145	67584543	9416	2398012	69991971	100.00%	98.65%
	9	48332963	67788586	10713	2193041	69992340	100.00%	98.73%
	10	49034734	67410151	48495	2533692	69992338	99.99%	98.56%
	11	48928293	67270317	10848	2711171	69992336	100.00%	98.54%
	12	48840075	67173405	11760	2807169	69992334	100.00%	98.38%
	13	49464032	67674938	10331	2307069	69992338	100.00%	98.75%
	14	48740280	67231080	11114	2750142	69992336	100.00%	98.53%
	15	49449408	67644407	14642	2333285	69992334	100.00%	98.73%

プログラムはマスタ・ワーカ方式で実行され，マスタとなるメインスレッドもワーカを担っている．なお，メインスレッドを実行する pid0 のコアの総実行サイクル数 cycle がプログラムの実行に要した全体の実行サイクル数を表している．また，I1wait および D1wait は，キャッシュミスが発生し 2 次キャッシュやメモリへのリクエストを処理している間に発生した遅延サイクル数を表している．本シミュレータでは，先行するロード・ストア命令においてキャッシュミスが発生した場合でも，データに依存関係がない限り順に後続命令を発行している．そのため，キャッシュミスによる待ちサイクルを一部隠蔽することができている．また，要求データのキャッシュ状況に応じて，リクエストの処理にかかるサイクル数が変わってくるが，キャッシュミスによるペナルティが全体性能に与える影響を調べるために，I1wait および D1wait には 2 次キャッシュミス等における遅延サイクル数も含めている．

いずれのプログラムもスレッド数の増加に伴い，総実行サイクル数が削減されていく傾向にあり，台数効果を得られていることがわかる．まず FFT では，step や exec を比較するとスレッド数の増大に伴い演算処理を各スレッドで分担できていることがわかる．しかし，並列度を上げるにつれ総実行サイクル数の削減量が小さくなっていることも見てとれる．これは，実行スレッド数を増大させた場合には，D1 キャッシュに保持しているデータが他のコアからのコヒーレンシ要求によって無効化されやすくなり，D1 キャッシュのヒット率が低下しやすくなるからである．そのため，16 並列実行時には D1wait のサイクル数が大きく増加し，8 並列実行時とほとんど変わらないサイクル数となってしまっている．そのため，さらに並列度を上げた場合には総実行サイクル数が増加に転じる可能性がある．また，FFT はプログラム全体に対して並列実行している時間が短いため，並列化の恩恵を受けにくいと考えられる．

次に LU では，16 並列実行時の総実行サイクル数が逐次実行時の約 20%まで削減されている．また，step と exec から演算処理を効率良く分割できているだけでなく，特に D1wait が大きく削減されていることも注目すべき点である．現在のシミュレータでは 1 次キャッシュミスが発生してから要求するデータが 1 次キャッシュに届くまでの時間，つまり 2 次キャッシュミスが原因で発生するペナルティも含めて D1wait として計上している．そのため，D1wait が大きく削減されたのは，複数のコアが並列実行することでデータが 2 次キャッシュ上にキャッシュされていたため，2 次キャッシュミスが削減できたからだと考えられる．また，D1hit も 8 並列実行時には低下してしまっているが，基本的に並列度を上げるとヒット率が向上しているのが見てとれ，効率的に並列実行できているのがわかる．

最後に WATER-NSQUARED では，16 並列実行時の総実行サイクル数が逐次実行時

表 6: 実験環境

CPU	UltraSPARC T1
動作周波数	1.0GHz
1 次キャッシュ	24KB
2 次キャッシュ	3MB
OS	Solaris 10

の約 10%まで削減されており，LU よりも並列実行に向いているプログラムであることがわかる．LU 同様，演算処理を効率良く分割できている一方で，16 並列実行時の D1wait が 8 並列実行に比べ増加してしまっている．これは，16 並列実行時には，多くのコア間でデータの無効化要求が飛び交うため D1hit 率が低下したことが原因だと考えられる．そのため，現在のアーキテクチャ構成では，並列化によって WATER-NSQUARED をさらに高速化するのは難しいことがわかった．

これらの結果から，開発したシミュレータは，各プログラムを並列実行したときの特徴を調べることが可能であると確認できた．さらに，各サイクル数の推移からプログラムの並列度限界を調査することもできており，今後のメニーコア研究を進めるためのシミュレータとしての妥当性を確認できた．また今回得られた動作検証から，FFT のように，プログラムが複雑になると単純に並列化しただけでは高速化できないこと，キャッシュシステムがプロセッサ性能のボトルネックとなることが確認でき，高並列に実行可能なプロセッサ構成の検討が必要であることを改めて認識できた．

5.3 考察

5.2 節では，シミュレーションによって各プログラムを並列実行した際の特徴や並列度限界を調査可能であることを示すことで，開発したシミュレータが，今後メニーコアプロセッサを研究するためのシミュレータとして妥当であることを示した．次に，得られたシミュレーション結果自体が妥当なものであるか検証を行い，シミュレータの信頼性を示す．検証は，実機で同じベンチマークプログラムを実行した場合のシミュレーションにかかる時間の傾向と 5.2 節で得られた評価結果を比較することで行う．実験を行った環境を**表 6** に示す．開発したシミュレータが SPARC ベースであるため，UltraSPARC T1 を搭載した環境を選択した．また，シミュレータの各パラメータは 5.2 節の表 2 で示したものと同じである．

表 7: 実機上でベンチマークを実行するのに要した時間（ms）

ベンチマーク	1 並列	2 並列	4 並列	8 並列	16 並列
FFT	19.95	17.11	15.74	15.81	17.12
LU	9249.66	4863.67	2720.43	1729.34	1717.51
WATER-NSQUARED	23694.33	12216.75	6530.38	3770.50	3109.75

表 8: FFT の実行時間内訳

	1 並列	2 並列	4 並列	8 並列	16 並列
初期化を含めた全体の実行時間	19.95	17.11	15.74	15.81	17.12
初期化後の計算時間	7.16	3.70	2.16	1.72	1.73

表 9: FFT における初期化以降のシミュレーションサイクル数

	マスタ／ワーカ	step	exec	I1wait	D1wait	cycle
1thread	マスタ	3037490	3408478	79738	87970	**3576186**
2thread	マスタ	2870057	3196228	79591	49213	**3325032**
	ワーカ	171576	217887	1816	28869	248572
4thread	マスタ	2787244	3091354	79720	30311	**3201385**
	ワーカ	88367	112540	1598	11877	126015
8thread	マスタ	2745703	3038853	79798	32621	**3151272**
	ワーカ	46870	60369	1854	11944	74167
16thread	マスタ	2731607	3020755	80317	39200	**3140272**
	ワーカ	30353	39206	1933	20974	62113

各ベンチマークを実行した結果を**表 7** に示す．表の各項目は，左から順に，ベンチマークの種類とベンチマークを実行した並列度における実行時間を示しており，単位はミリ秒である．

LU，WATER-NSQUARED の結果を見ると，共に並列度の向上に伴い実行時間が短くなり，一方で実行時間の削減量が小さくなっている．これらは，表 4，表 5 で得られたサイクル数の推移と一致する．

一方 FFT の実行時間は，並列度の向上に伴い高速化しているものの，4 並列実行時を境に実行時間が増加に転じてしまっている．この原因を調査するため，FFT の実行時間

の内訳を計測した．その結果を**表 8** に示す．表の上段に，ベンチマークを実行してから終了するまでの実行時間を，下段に，初期化後の計算に要した時間を示している．この初期化とは，実行を開始してからスレッドを生成し，全スレッド間で同期をとりながらデータ全体を初期化して，各スレッドの担当する範囲のデータを取得するまでの処理を指している．表 8 から，計算時間は並列度の向上に伴い短くなっているが，全体の実行時間は増加してしまっており，これは初期化にかかる時間によるものであることがわかる．初期化処理中に行うスレッドの生成や同期処理は OS が担っているため，OS が存在する実機で実行した場合と OS の存在しないシミュレータ上で実行した場合では，初期化に要した時間を単純に比較することはできない．

そこで，シミュレーションによって得られた初期化以降のサイクル数と，初期化を除いた計算時間を比較することでシミュレーションの妥当性を考察する．得られたサイクル数を**表 9** に示す．表の見方は，5.2 節と同様だが，ワーカスレッドに関しては平均値を示している．表 9 から，初期化以降の計算に要したサイクル数は並列度の向上に伴い削減されているのがわかる．これは，表 8 で得られた計算時間の推移と一致する．そのため，並列実行によって得られたシミュレーション時間は妥当であると考えられる．

これらの結果から，シミュレーションで得られたサイクル数の推移と実機上で実行した場合のシミュレーション時間の推移は酷似しており，5.2 節で得られた結果は妥当なシミュレーション結果であると示せた．

5.4　今後のメニーコア研究に向けたシミュレータの改良方針

3.3 節で述べたように，本研究は 3 つのステップを段階的に踏むことで，高並列実行可能なメニーコアアーキテクチャを模索していく．第 1 ステップで開発したシミュレータを用いて，その後の研究を進めていくためにはメニーコアプロセッサの性能目標値を導出する必要がある．そのため，表 3，表 4 および表 5 で挙げた評価指標以外の結果も今後算出する必要がある．例えば，I1wait や D1wait は，キャッシュリクエストの処理に要したサイクル数でなく，1 次キャッシュミス時に待ちサイクルが発生した場合のペナルティを算出している．つまり，キャッシュリクエストの伝播時に発生する遅延サイクル数やデータ転送の時間を計上していない．しかし，キャッシュシステムや相互結合網の検討では，特にキャッシュリクエストの処理に要したサイクル数を詳細に評価する必要がある．そのために，キャッシュリクエストの処理に要したサイクル数の内訳について考察する．この際に考慮しなければならないことは，1 次キャッシュ，2 次キャッシュ，および主記憶のアクセスレイテンシとキャッシュリクエストの伝播時に発生する遅延サイクル数である．

こうしたキャッシュリクエストの伝播は以下のようなユニット間通信に分類できる．なお説明のために，要求元のクラスタをクラスタ A，要求データを保持する主記憶を持つクラスタをクラスタ B，要求元以外のクラスタをクラスタ C とする．

(1) クラスタ A の要求元 1 次キャッシュ → クラスタ A の 2 次キャッシュ
1 次キャッシュミス時に，属するクラスタの 2 次キャッシュへリクエストを送信する

(2) クラスタ A の 2 次キャッシュ → クラスタ A に属する他 1 次キャッシュ
要求元以外の 1 次キャッシュへコヒーレンシ要求を送信する

(3) クラスタ A の 2 次キャッシュ → クラスタ B の 2 次キャッシュ
要求データを保持する主記憶を持つクラスタへキャッシュリクエストを送信する

(4) クラスタ B の 2 次キャッシュ → クラスタ C の 2 次キャッシュ
要求元以外のクラスタへデータ転送要求とコヒーレンシ要求を送信する

(5) クラスタ C の 2 次キャッシュ → クラスタ C に属する 1 次キャッシュ
データ転送要求を受け取ったクラスタに属する 1 次キャッシュへコヒーレンシ要求を送信する

先に挙げたアクセスレイテンシに加えて，これら 5 つのキャッシュリクエストの伝播に要したサイクル数を算出することで，メニーコアプロセッサの性能をより詳細に評価することができると考えられる．他には，2 次キャッシュのヒット率などが算出すべき出力項目として挙げられる．

また，プロセッサの性能評価ではシミュレーションパラメータを変更した場合の性能の変化を調査することも重要である．表 2 で示した本シミュレータのパラメータにおいて，変更した際の影響を調査すべきものとしては，プロセッサコア数，クラスタ数，各キャッシュ容量，2 次キャッシュのバンク数などが挙げられる．ここで，プロセッサコア数およびクラスタ数は，現在最小構成となっているため，順に増大させていくべきである．特にプロセッサコア数は，今後 100 や 1000 と規模を拡大していくことが予測されるため，TILE64 のように少なくとも 64 コア構成程度にすることが望ましい．また，キャッシュ容量やバンク数は，単純に増大させることで性能が向上しやすい．キャッシュ容量は，サイズを大きくすることでヒット率が高くなり性能が向上するが，ルックアップ速度が低下する場合があり，適当なキャッシュサイズを考えなければならない．一方バンク構成では，複数のリクエストを同時に処理することで遅延サイクルを一部隠蔽できるため，バンク数を増大させることで性能の向上に繋がりやすい．しかし，同時に要求されるリクエスト数には限りがあるなど，性能向上においても限界値があり，また追加するハードウェア物量も考慮なければならない．そのため，キャッシュ容量やバンク数などの適切なパラメータ

を設定すべきである．

一方で，実行評価時に固定すべきパラメータもある．本研究では，キャッシュ構成や相互結合網に着目しているため，キャッシュおよび主記憶のアクセスレイテンシやキャッシュの連想度，そしてキャッシュのラインサイズなどは固定することが望ましい．また，レジスタウインドウ数や主記憶容量は，パラメータの変更による性能への影響が各構成方式の検討において重要性が低い．このように，評価指標やシミュレーションパラメータを考慮した実装が今後必要になると考えられる．

6　シミュレーションの高速化

今後のメニーコア研究を円滑に行うために，シミュレーションに要する時間を削減する機能をメニーコアトレースシミュレータに追加実装する．本章では，その高速化機能について述べる．

6.1　ソフトウェアシミュレーションの問題点

コンピュータアーキテクチャの研究において，アーキテクチャシミュレータは非常に大きな役割を担っている．しかし，3.1 節で述べたように，ソフトウェアによるアーキテクチャシミュレーションは評価に多大な時間を要することが知られている．

これまでにシングルコアプロセッサのシミュレーション評価に関しては，プログラムの一部分のみを切り出して評価し，その際の IPC を比較するという方法が採られてきた．しかし，マルチコアにおける並列アプリケーションの評価においては，プログラム全体の速度向上率が重要である．そのため，部分プログラムの IPC 比較は評価に適しておらず，シミュレーションそのものの高速化が求められている．さらに，複数のプロセッサコアを有するマルチコアやメニーコアのアーキテクチャシミュレーションを行う際には，コア数の増加に伴うシミュレーション時間の増加が深刻な問題となっており，メニーコアプロセッサ研究の障害になっている．

こうした理由から，これまでに様々なアーキテクチャシミュレーション高速化手法が研究されている．こうした高速化手法には，実マルチコア環境においてスレッド並列化や時間軸分割並列化などによりシミュレータを並列実行するものや，統計的手法を用いて詳細にシミュレーションを行う部分を限定するもの，計算再利用技術を利用して処理を省略するものなどがある．本論文で提案したメニーコアトレースシミュレータでもシミュレーションに膨大な時間がかかることが懸念されることから，こうした高速化手法をシミュレータに組み込んでいく必要がある．

6.2 既存のシミュレーション高速化手法

これまでに様々なアーキテクチャシミュレーションの高速化について研究が行われている．シミュレーションの高速化手法は，シミュレーション精度をある程度犠牲にして大幅な高速化を狙う手法と，精度を落とすことなく高速化を図る手法の 2 つに大別される．

まず，シミュレーション精度を犠牲にして高速化する手法では，詳細にシミュレーションする部分から，ある程度妥協してもよい部分を切り離し簡略化することで，全体のシミュレーション時間を削減する．例えば，メニーコアプロセッサのシミュレーションを，命令レベルシミュレーションと共有メモリシミュレーションに分離し，それぞれを簡易シミュレーションすることにより高速化する手法 [18] が提案されている．この手法は，アーキテクチャの初期設計段階を対象としており，厳密な精度ではなくとも大まかなアーキテクチャ構成や各構成要素のパラメータの絞り込みを迅速に行うことを最重要視している．しかし，シミュレーションを簡易にするために，「常に理想的なレイテンシで通信が可能である」などの仮定をしている．このため，メニーコアプロセッサのボトルネックのひとつになり得る，メモリトラフィック量などの影響をシミュレーションすることが困難である．

また，統計的手法を用いてプログラムの一部分のみを詳細にシミュレーションし標本抽出（サンプリング）することで高速化を図る手法として，SimFlex[19] と SimPoint[20] が挙げられる．SimFlex は，最初に評価対象のプログラム全体を簡易なシミュレーションで実行しながら，ある一定間隔毎にプロセッサ状態をチェックポイントとして記録していく．その後，記録したチェックポイントから一定区間を詳細なシミュレーションで実行し，先に得られた簡易シミュレーションの結果と統合して全体の評価結果を得る．一方で SimPoint はプログラム構造に着目し，詳細にシミュレーションを行う部分を抽出する．具体的には，Basic Block Vector（BBV）を利用することで，プログラムの振る舞いをキャプチャして，類似する部分を解析し，シミュレーションポイントをピックアップする．そして，シミュレーションポイントのみを詳細にシミュレーションすることで高速化を実現している．しかし，これらの手法では精度を犠牲にしても良い箇所と，そうでない箇所を明確に切り離していない．そのため，シミュレーションの際に重要視したい部分も区別無く簡易シミュレーションされてしまい，シミュレーション結果の信頼性が大きく損なわれてしまう可能性がある．

もう 1 つの手法である，精度を落とすことなく高速化を図る手法は，並列化や計算再利用などを利用して全体のシミュレーション時間を削減するもので，数多く提案されている．その代表的な成功例に FastSim[21] や BurstScalar[22] がある．FastSim はバイナリ

変換と計算再利用技術を利用することで，精度を落とさずに高速化を実現している．しかし，バイナリ変換を用いるため可搬性が犠牲になってしまっている．また，計算再利用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ため，保存しなければならない状態が膨大になるという問題点がある．それに対して BurstScalar は，バイナリ変換を利用せず計算再利用技術だけで高速化を実現しているため，可搬性を保っている．BurstScalar は，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，パイプラインのスケジューリングにも局所性が存在していることに着目している．たとえば，最内ループ実行時にはごく少数のパターンの命令スケジューリングが繰り返し行われており，そうした繰り返しを計算再利用技術によって省略することで高速化している．しかし，ユニプロセッサをシミュレートすることを前提に，シミュレータの一部が構築されているという問題点がある．

また，並列化による高速化手法のうち，時間軸方向にシミュレーションを分割して並列化する手法が古くから提案されている [23]．この手法では，分割点において各ノードのマシン状態が一致していなければ，誤ったシミュレーション結果を出力してしまう．そこで，各ノードは分割点まで論理シミュレーションを行い，メモリやレジスタ等の状態を予測し，分割点の前後の区間シミュレーションを重複して実行することでマシン状態を一致させる．また，分割点で状態が一致しなかった場合は，区間シミュレーションをやり直すことで精度を保証している．しかし，構成要素が多いメニーコアの場合，一致させなければならない状態が多く分割点での初期状態の予測が困難になると考えられる．

このほか，マルチプロセッサのシミュレーションを分散・並列処理によって高速化する手法も提案されており，WWT[24] や Shaman[25] では良好な台数効果が得られたと報告されている．また，分散処理による高速化だけでなく，分散処理と並列処理を組み合わせた場合の方がシミュレーション時間の削減に成功したという報告 [26] もある．

そこで，実マルチコア環境において，並列化することでシミュレーションを高速化させる機能を，4 章で開発したメニーコアトレースシミュレータに追加実装した．実マルチコア環境は現在ではごく一般的であるため，ユーザに環境構築などの負担をかけることはなく，手軽に高速化を実現できる．なお，今後この高速化させる機能を使用してシミュレーションを行うことを**高速実行**と呼ぶ．

6.3 並列度の自動調整

メニーコアトレースシミュレータは主に，コアのシミュレーション，各クラスタ内の 2 次キャッシュのシミュレーション，各クラスタに割り振られたメモリのシミュレーション

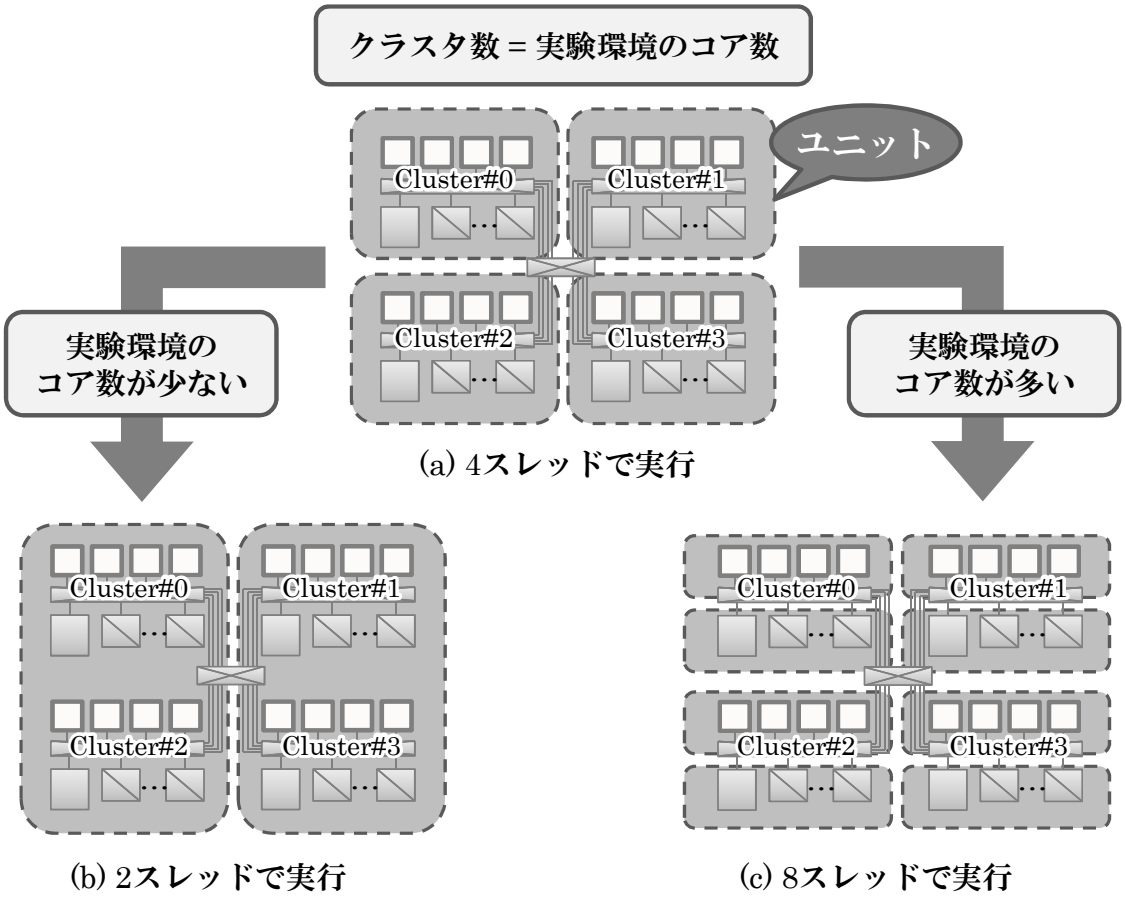


図 13: クラスタと並列化単位 unit の関係

の 3 種類の要素で構成されている．例えば，4.1 節で示したアーキテクチャ構成をシミュレートする際には，まずは 16 個のコアのシミュレーションを行う．その後，各クラスタに存在する計 4 つの 2 次キャッシュのシミュレーションを行い，最後に各クラスタに分割された計 4 つのメモリのシミュレーションを行っている．そこで，本メニーコアトレースシミュレータはこれらの要素をスレッド並列化することでシミュレーションの高速化を図る．しかし，これらの要素をただ単純にスレッド並列化するだけでは，実マシンのコア数に対して生成するスレッド数が多くなりすぎる場合があり，かえって時間がかかる可能性がある．そのため，ユーザは実験環境に合わせてシミュレータを高速実行させる際の最適なスレッド数やパラメータ値を探さなければならない．こうしたパラメータの変更は，メニーコアトレースシミュレータが本来目的としている，メニーコアプロセッサ構成の検討とは全く関係が無く，極力少ない労力で実現できることが望ましい．

そこで，実験環境に合わせてシミュレータ自身でスレッド数を調整する**並列度の自動調整機能**を開発したシミュレータに追加実装した．これにより，ユーザへの負担を増やすことなく，シミュレーションにかかる時間を削減することができる．具体的には，スレッド並列化において単一スレッドが担う処理単位をユニットと名付け，ユニット内で 3 種類のシミュレーション要素を呼び出す個数を変化させることで，並列度の調整を行う．

図 13 に，クラスタとユニットの関係を示す．まずは，図 13 の (a) に示すように 1 クラスタ分のシミュレーションを並列化の基本単位と定める．つまり，各ユニット内では 4 つのコアと 1 つの 2 次キャッシュ，1 つのメモリのシミュレーションのための処理を呼び出している．そして，実験環境のマシンのコア数に合わせてこのユニットの数を変化させる．例えば，実験環境が 4 コアを搭載したマシンだった場合，4 スレッドでシミュレータを並列実行するのが理想的であると考えられるため，並列度の調整を行う必要はない．もし，シミュレータのクラスタ数より実験環境のコア数が少ない場合は，4 スレッドで並列実行してしまうとスレッド数が多く低速化してしまう可能性がある．そこで，図 13 の (b) に示すように，ユニット内で複数のクラスタのシミュレーション，つまり 8 つのコアと 2 つの 2 次キャッシュ，2 つのメモリのシミュレーションを行うことで，スレッド数を減らし実験環境に合わせた並列度で高速実行する．一方で，クラスタ数より実験環境のコア数が多いような場合は，図 13 の (c) に示すように，ユニットから呼び出すシミュレーション要素を少なくする．図 13 の (c) の場合は，コアのシミュレーションを行うユニットと，2 次キャッシュ及びメモリのシミュレーションを行うユニットに細分化し，並列度を上げることで，実験環境のコアを有効に使い高速化を狙う．なお，シミュレータが並列度を決定する際に使用する，実験環境のコア数は，システムコールを利用することで取得可能である．

なお，1 つのユニットで複数のクラスタのシミュレーションを行う際には，本メニーコアトレースシミュレータでは，コア番号の小さいコアから使用していく実装となっているため，例えば，シミュレータ上でプログラムを 8 並列で実行する場合は，クラスタ 0 番（cluster#0）とクラスタ 1 番（cluster#1）内のコアしか使用されていない．そこで，図 13 の (b) に示すように，左側のユニットはクラスタ 0 番とクラスタ 2 番（cluster#2）をシミュレーションするようにシミュレーション要素を呼び出すことで各スレッド間の処理量をなるべく均等になるように分割する．

表 10: Pthread ライブラリと Solaris スレッドライブラリの主な関数対応表

Pthread	Solaris スレッド
pthread_create()	thr_create()
pthread_join()	thr_join()
pthread_mutex_init()	mutex_init()
pthread_mutex_lock()	mutex_lock()
pthread_mutex_unlock()	mutex_unlock()
pthread_barrier_init()	-
pthread_barrier_wait()	-

6.4 シミュレータの動作オプション

メニーコアトレースシミュレータを高速実行する際に，ユーザが各種パラメータを選択できるようにした．これにより，実験環境に適したシミュレータの運用ができる．

6.4.1 スレッドの生成

メニーコアトレースシミュレータの高速実行機能は，スレッド並列化によって実現している．ここで，スレッド並列化を実現するために利用できる，マルチスレッド化ライブラリにはいくつかの種類が存在している．POSIX 標準である汎用性の高い Pthread ライブラリが存在する一方で，特定の OS 上でしか動作しないかわりに，非常に高速なマルチスレッド化ライブラリも存在している．特に，代表的な Unix 系 OS である Solaris は，専用のマルチスレッド化ライブラリである Solaris スレッドライブラリを提供している．Solaris スレッドライブラリは，シンプルな機能しか提供していないがその分高速に動作するという特徴を持っている．また，Solaris はスレッドスケジューリングなどの実装が独特であるため，Solaris スレッドライブラリを利用した方が高速に実行できる場合がある．

そこで，実験環境に合わせてより高速に実行するために，Pthread ライブラリと Solaris スレッドライブラリ のどちらを利用してスレッドを生成するかオプションにてユーザが選択できるようにした．**表 10** に 2 つのスレッドライブラリで利用する主な関数の対応表を示す．

2 つのライブラリが提供している基本的な機能や関数にはほとんど差はないが，一部どちらかのライブラリでしか提供していない機能が存在する．例えば，バリア同期は Pthread ライブラリでしか提供されていない．しかし，これら 2 つのライブラリを併用することは可能なため，Solaris スレッドを利用してスレッドを生成してもバリア同期を実現する

pthread_barrier_wait() を利用することは可能である．そのため，ユーザがSolarisスレッドライブラリを選択した場合でも，次項で述べるバリアのオプションの選択肢が減ることはない．

6.4.2 バリア

本メニーコアトレースシミュレータは，並列実行する際に毎サイクル同期をとることで精度を落とすことなく詳細にシミュレーションを行っている．こうした複数のスレッドで同期をとるためには一般的に，汎用性の高い pthread_barrier_wait() が利用される．しかし，pthread_barrier_wait() を利用した場合、あるスレッドがバリアに到達してから一定期間内にすべてのスレッドが同じバリアに到達しなければ，すでに到達しているスレッドはスリープ状態に移行してしまう可能性がある．そのため，シミュレーション時間が増加してしまう可能性がある．

そこで，本メニーコアトレースシミュレータはスリープ状態に移行しないビジーウェイトバリアを提供している．このバリアは高速に実行できる一方で，共有変数にアクセスしながらスピンロックすることで全スレッドがバリアに到達するまでビジーウェイトするように実装しているため，CPU リソースを大量に消費してしまう．このため，実験環境で他のプロセスが動作している場合には，それらのプロセスの実行によって性能に大きな影響を受けてしまうと考えられる．なお，このビジーウェイトバリアには，4.3.1 項で示した図 8 の (2) と同じ方式を採用した．

ユーザは，実験環境である実マシンの運用方針に合わせて，どちらのバリア同期を利用するかオプションで選択することができる．たとえば，使用する CPU リソース量は考えず，シミュレーション結果をできるだけ早く取得したい場合には，ビジーウェイトなバリアを利用し，他のプロセスが動作している環境やシミュレータを一度に複数実行したい場合には，pthread_barrier_wait() を利用することができる．なお，ビジーウェイトバリアを選択した場合，内部でロックを利用しているが，そのロックにはSolarisスレッドライブラリ選択時には，Solaris スレッドライブラリの mutex_lock() を用いる．一方，Pthread ライブラリ選択時には Pthread ライブラリの pthread_mutex_lock() を用いる．

6.4.3 コアへのバインド

実マルチコアマシン上で，プログラムを並列処理させる場合に，スレッドを処理する実コアが変更されないようにすることで，スレッドとコアのマッピングが変化する際のオーバーヘッドを削減できるため，マルチコアプロセッサのシステムで最良のパフォーマンスを得ることができる．

しかし，実マシン上では他のプロセスも動作しているため，スレッドを実コアにバイン

表 11: 実験環境

	CPU	動作周波数	物理コア数	論理コア数	OS
マシン A	Core2 Ex. Quad	3.0GHz	4	4	Solaris10
マシン B	UltraSPARC T1	1.0GHz	8	32	Solaris10

ドさせるのは必ずしも有効であるとは言えない．また，異なるコアにバインドされたスレッド間で情報交換が頻発した際には，実コア間で通信が発生しシミュレーション時間が増加してしまう可能性もある．

そこで，コアにスレッドをバインドさせるかどうかユーザが任意で設定できるように，オプションで選択可能にした．なお，コアにスレッドをバインドさせるには，システムコールを利用する．

7　シミュレーション時間の評価

6 章で述べたシミュレーションの高速化機能を，開発したメニーコアトレースシミュレータに追加実装し，シミュレーションに要する時間を評価した．

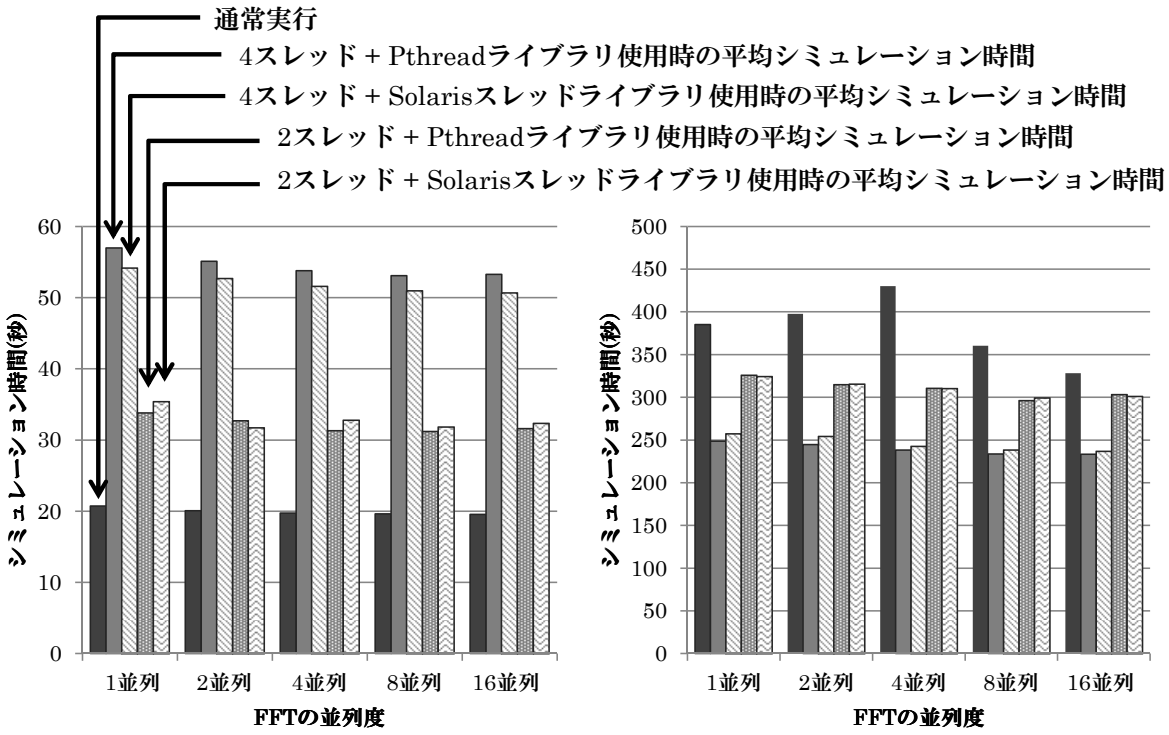
7.1　評価環境

4 章で開発したメニーコアトレースシミュレータを通常実行した場合と，高速実行した場合でシミュレーションに要する時間について評価した．シミュレータを実行した実験環境を**表 11** に示す．アーキテクチャやプロセッサコア数，同時に実行できるスレッド数が異なる 2 つの環境で評価した．

シミュレータのプロセッサ構成は，5 章で示したものと同じであり，16 コア構成で 1 クラスタに 4 つのコアを内蔵した 4 クラスタ構成となっている．なお，現在のメニーコアトレースシミュレータは，システムコールのシミュレーションを行う部分が実装途中であるため，クラスタ数以上のスレッド数に分割することができない．そのため，高速実行時のシミュレータの並列度は，2 スレッドと 4 スレッドで実行した場合をそれぞれ計測した．

7.2　動作オプションの選択

まずは，6.4 節で述べたシミュレータの動作オプションの設定によって，シミュレーション時間がどのように変化するかそれぞれ評価した．評価には，5 章で使用した SPLASH-2 ベンチマークから FFT を用いた．



(1) マシンAで実行した結果　　(2) マシンBで実行した結果

図 14: Pthread 使用時と Solaris スレッド使用時の平均シミュレーション時間

7.2.1　スレッドの生成

まずは，6.4.1 項で述べた，Pthread ライブラリを選択した場合と，Solaris スレッドライブラリを選択した場合のシミュレーション時間を比較した．スレッドの生成，バリア，コアへのバインドの 3 つのオプションにはそれぞれ 2 つの選択肢があるため，合計 8 つのオプションの組み合わせが存在する．その全組み合わせのうち，Pthread ライブラリを使用した場合の平均シミュレーション時間と，Solaris スレッドライブラリを使用した場合の平均シミュレーション時間を**図 14** に示す．縦軸はシミュレーション時間を表しており，単位は秒である．横軸はシミュレータ上で実行した FFT の並列度を表している．4 本あるグラフは左から，シミュレータを通常実行した場合，4 スレッド実行かつ Pthread ライブラリを使用した場合，4 スレッドかつ Solaris スレッドライブラリを使用した場合，2 スレッドかつ Pthread ライブラリを使用した場合，2 スレッドかつ Solaris スレッドライブラリを使用した場合の平均シミュレーション時間をそれぞれ表している．図 14 の (1) は

マシン A（Core2）で実行した場合の結果を，図 14 の (2) はマシン B（SPARC）で実行した場合の結果をそれぞれ示している．

評価結果から，マシン A（Core2）上で 4 スレッド実行した場合は，Pthread ライブラリ使用時より Solaris スレッドライブラリ使用時の方がシミュレーション時間を削減できている．その一方で，2 スレッド実行した場合には，Pthread ライブラリを使用した場合の方がシミュレーション時間を削減することができている．しかし，いずれの場合も通常実行時よりもシミュレーション時間が増大してしまっている．一方，マシン B（SPARC）上で実行した場合は，どちらのスレッド数で実行した場合でも通常実行時よりシミュレーション時間を削減できている．4 スレッド実行時は，Pthread ライブラリを使用した方が高速に実行可能であり，2 スレッド実行時は，ほぼ差はないが Pthread ライブラリ使用時の方が高速に実行可能である．

これらの結果から，スレッドライブラリの違いによってシミュレーション時間は想定していた程大幅に変化しないことがわかった．

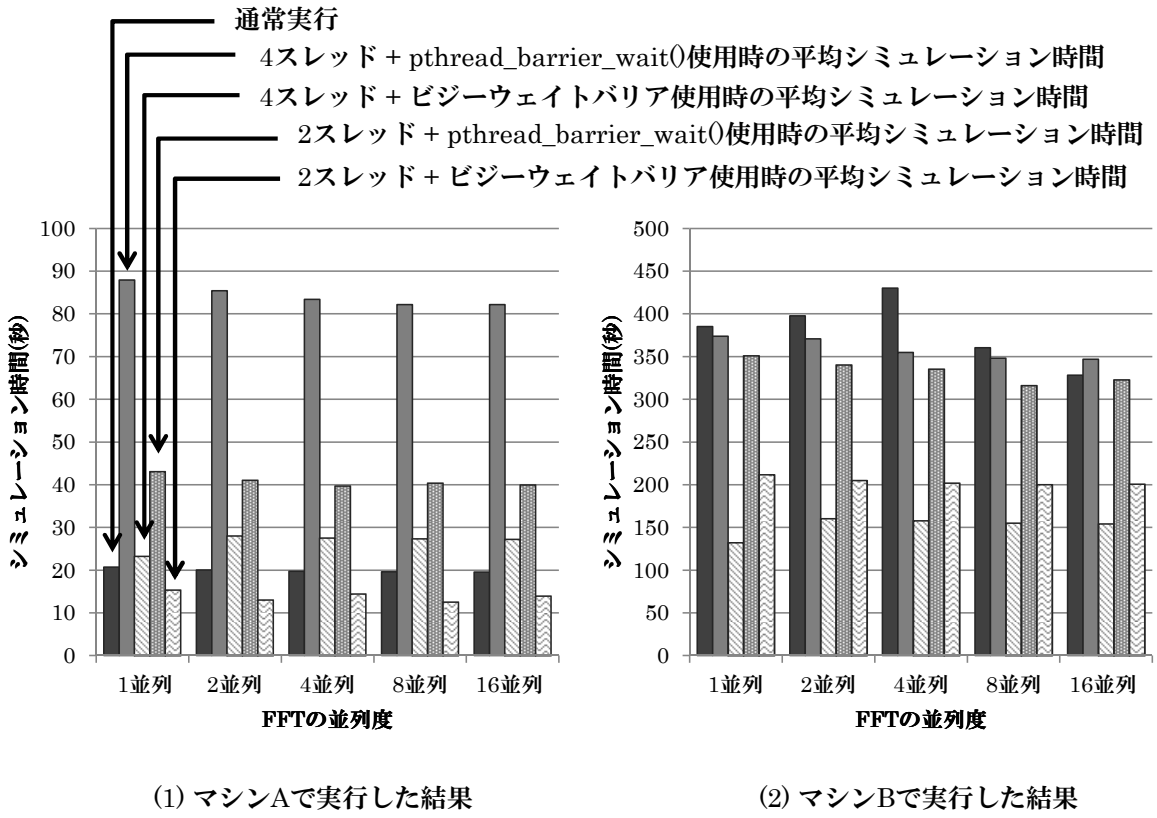
7.2.2 バリア

次に，6.4.2 項で述べた，pthread_barrier_wait() を使用した場合と，ビジーウェイトバリアを使用した場合のシミュレーション時間を比較した．7.2.1 項と同様，それぞれのバリアを使用した場合の平均シミュレーション時間を**図 15** に示す．グラフの見方も 7.2.1 項と同様である．4 本あるグラフは左から，通常実行，4 スレッド実行かつ pthread_barrier_wait() を使用した場合，4 スレッドかつビジーウェイトバリアを使用した場合，2 スレッドかつ pthread_barrier_wait() を使用した場合，2 スレッドかつビジーウェイトバリアを使用した場合の平均シミュレーション時間をそれぞれ表している．

評価結果からいずれの場合も，ビジーウェイトバリアを利用した場合の方が高速にシミュレーションができることがわかる．特に，マシン B（SPARC）上で 4 スレッド実行した場合には，平均で 60%近くシミュレーション時間を削減することに成功している．一方で，pthread_barrier_wait() を使用してマシン A（Core2）上で 4 スレッド実行した場合，シミュレーション時間が大幅に増大してしまっている．これは，バリア同期の際に各スレッドがスリープ状態に移行しているためだと考えられる．

しかし，図 15 の評価結果は，それぞれの実験環境で他のユーザが利用しているプロセスが存在しない状態で計測した結果である．そのため，CPU リソースを全て使うビジーウェイトバリアの方が pthread_barrier_wait() よりも高速にシミュレーションできたと考えられる．

そこで，同時に動作する他プロセスへの影響を調査するため，シミュレータを複数個同



(1) マシンAで実行した結果　　(2) マシンBで実行した結果

図 15: pthread_barrier_wait() 使用時とビジーウェイトバリア使用時の平均シミュレーション時間

時に実行した場合のシミュレーション時間を pthread_barrier_wait() を利用した場合とビジーウェイトバリアを利用した場合とで評価した．マシン A（Core2）上で実行した結果を**表 12** に示す．2 つの表のうち，上の表が pthread_barrier_wait() を使用した場合の結果，下の表がビジーウェイトバリアを使用した場合の結果をそれぞれ示している．各表はシミュレータを 4 スレッドで 1 つ実行した場合と，4 スレッドで 2 つ実行した場合のシミュレーション時間を表しており，単位は秒である．表 12 から，1 プロセスだけ実行する場合で比べるとビジーウェイトバリアの方が約 3 倍高速だが，プロセスを同時に 2 つ実行すると，シミュレーション時間が爆発的に増加してしまうことがわかる．これは，バリア同期の際に CPU リソースを大量に消費してしまうため，リソースの奪い合いが起きてしまったためだと考えられる．一方で，pthread_barrier_wait() 使用時は，プロセスを 2 つ実行した状態でも大幅な速度低下を引き起こすことはなかった．これは，バリア実行時にスリープ状態に移行したため，CPU リソースの奪い合いが起きなかったためであると考

表 12: シミュレータを複数実行した場合のシミュレーション時間（s）

pthread_barrier_wait() 使用時

	1 並列	2 並列	4 並列	8 並列	16 並列
4 スレッド，プロセス 1 つ	112.8	109.4	106.7	108.6	105.5
4 スレッド, プロセス 2 つ	141.2	135.1	133.9	131.5	132.1

ビジーウェイトバリア使用時

	1 並列	2 並列	4 並列	8 並列	16 並列
4 スレッド, プロセス 1 つ	30.9	32.7	29.2	32.0	29.1
4 スレッド, プロセス 2 つ	2076.6	1540.8	1022.6	1764.1	1798.8

えられる．

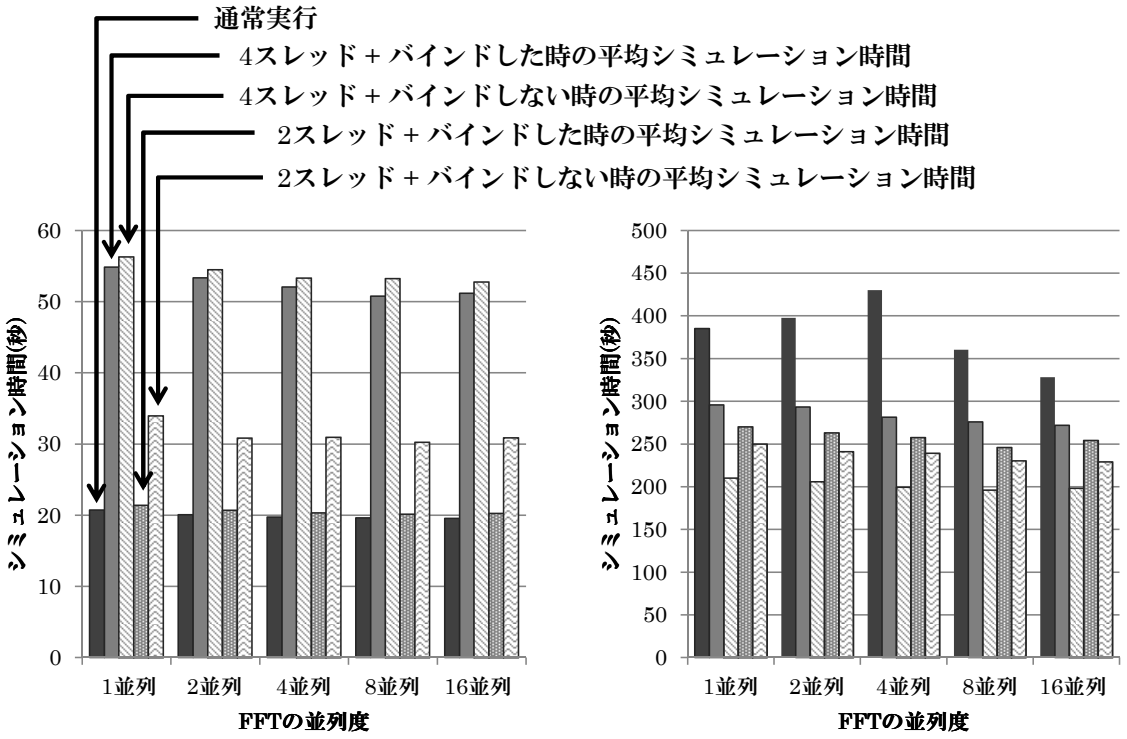
これらの結果から，どちらの実験環境でも他のユーザプロセスが実行されていない場合は，ビジーウェイトなバリアを利用した方が高速に実行できた．その一方で，CPU リソースを節約しながら実行を進める pthread_barrier_wait() を利用すると，予想以上に速度低下してしまうことがわかった．また，シミュレータの他に大きなプロセスも実行するような環境では，ビジーウェイトバリアが他プロセスへ与える影響が非常に大きいことも確認できた．

7.2.3 コアへのバインド

最後に，6.4.3 項で述べた，スレッドをコアにバインドさせた場合と，バインドさせない場合のシミュレーション時間を比較した．7.2.1 項と同様，コアへバインドした場合としなかった場合の平均シミュレーション時間を**図 16** に示す．グラフの見方も同様である．4 本あるグラフは左から，通常実行，4 スレッド実行かつコアへバインドした場合，4 スレッドかつバインドしなかった場合，2 スレッドかつコアへバインドした場合，2 スレッドかつバインドしなかった場合の平均シミュレーション時間をそれぞれ表している．

評価結果から，マシン A（Core2）上で 4 スレッド実行した場合は，どちらのスレッド数でもコアへバインドした方が高速に実行できている．特に 2 スレッド実行時は，バインドしない場合に比べバインドすると 50%近くのシミュレーション時間を削減できた．一方で，マシン B（SPARC）上で実行した場合はどちらのスレッド数でもバインドしない方が高速に実行できている．

これらの原因を調査するために，バインドした時としない時のコンテキストスイッチの回数を調べた．その結果を**表 13** に示す．2 つの表のうち，上側の表がマシン A（Core2）での実行結果，下側の表がマシン B（SPARC）での実行結果である．各表は，左からバ



(1) マシンAで実行した結果　　(2) マシンBで実行した結果

図 16: コアにバインドする場合としない場合の平均シミュレーション時間

インドの有無，4 スレッド実行時のコンテキストスイッチ回数，2 スレッド実行時のコンテキストスイッチ回数を示している．なお，他のオプションについては，スレッド生成には Pthread ライブラリを使用し，バリアはコンテキストスイッチ回数の影響が少なくなるように，スリープ状態に移行しないビジーウェイトバリアを選択した．

表 13 の上側の表から，マシン A（Core2）上で 2 スレッド実行した場合は，バインドした方がコンテキストスイッチ回数が少なく，高速にシミュレーションできている．さらに，4 スレッド実行時に比べ 2 スレッド実行時のコンテキストスイッチ回数が非常に少ないのがわかる．これは，マシン A（Core2）のコア数よりも少ないスレッド数で並列実行したため，OS 等の他のプロセスは余ったコア上で実行されたことで，シミュレータが他プロセスからの割り込みをほとんど受けなかったからだと考えられる．また，6.3 節で実験環境のマシンのコア数と同じ数のスレッドを生成するのが理想的だと考えていたが，これらの結果から，マシンのコア数より少なくなるようにスレッドを生成するべきだとわ

表 13: コンテキストスイッチ回数の比較

マシン A（Core2）上で実行した場合

	4 スレッド実行時	2 スレッド実行時
バインドした場合	3938 回	202 回
バインドしない場合	3675 回	273 回

マシン B（SPARC）上で実行した場合

	4 スレッド実行時	2 スレッド実行時
バインドした場合	6256 回	3347 回
バインドしない場合	2619 回	1947 回

かった.

一方で，マシン B（SPARC）上で実行した場合，バインドしない方がコンテキストスイッチ回数が少なく，高速にシミュレーションできている．これは，マシン B（SPARC）の CPU が 1 つの物理コアで 4 つのスレッドを同時に実行できる UltraSPARC T1 であるためだと考えられる．UltraSPARC T1 はメモリアクセスを隠蔽するために，クロックごとにスレッドを切り替えることで，同時マルチスレッディングを実現している．このスレッド切り替えは，ハードウェアレベルで行われるため OS によるコンテキストスイッチに比べ非常に高速である．しかし，バインドするオプションは，別々の物理コアへスレッドをバインドするように実装しているため，バインドした場合 UltraSPARC T1 の同時マルチスレッディングの恩恵を受けることができない．それに対して，バインドしない場合，OS によって同じ物理コア上に 4 つのスレッドを実行するようにスケジューリングされたため，同時マルチスレッディングによるメモリアクセスの隠蔽だけでなく，2 次キャッシュを共有しているためキャッシュミスの削減にも繋がり大幅な高速化に成功したと考えられる．さらに，ハードウェアによるスレッド切り替えが中心になるため，OS によるコンテキストスイッチ回数も減少したと考えられる.

これらの結果から，シングルスレッド性能の高いマシン A（Core2）上で実行する場合は，バインドした方が高速に実行できるが，並列処理性能の高いマシン B（SPARC）上で実行する場合は，バインドしない方が高速に実行できることがわかった.

7.3 評価結果

7.2 節の結果を踏まえて，各実験環境に適したオプションを選択し，汎用ベンチマークプログラムである SPLASH-2 から FFT，LU，WATER-NSQUARED の 3 つのシミュレー

ションにかかる時間を評価した．実験環境ごとに選択したオプションの組み合わせを**表14**に示す．表のスレッド数とはシミュレータを何スレッドで実行するかを表している．なお，7.2.3 項で示した結果は，マシン A（Core2）上で 4 スレッド実行する際はバインドした方が平均シミュレーション時間は短いというものだった．しかし，pthread_barrier_wait() を使用しかつバインドしない場合のシミュレーション時間が大幅に悪化していたため，バインドしなかったときの平均シミュレーション時間を押し上げており，ビジーウェイトバリアを使用する場合はバインドしない方が高速にシミュレーションできることから，バインドしないという選択に変更した．

まずは，マシン A（Core2）上で実行した場合のシミュレーション時間を**図17**に示す．図 17 中で各ベンチマークプログラムの結果を 3 本のグラフで示しているが，それぞれ左から順に

（**N**）　通常実行

（**4A**）　4 スレッド／ Solaris スレッド／ビジーウェイトバリア／バインドしない の組み合わせで実行

（**2A**）　2 スレッド／ Pthread ／ビジーウェイトバリア／バインドする の組み合わせで実行

が要したシミュレーション時間を示している．なお，通常実行時の結果を 1 として正規化している．

4 スレッド実行時（4A）は，WATER-NSQUARED の 8 並列と 16 並列を除いたすべてのシミュレーション時間が増加してしまっている．これは，7.2.3 項で述べたようにマシン A（Core2）の持つ物理コア数とシミュレータのスレッド数が同じため，OS や他のプロセスとの資源の奪い合いが発生してしまったためだと考えられる．しかし，並列実行する時間が長く，コア間やクラスタ間での通信が頻発する WATER-NSQUARED ではシミュレーション時間を削減できている．また，ベンチマークの並列度が大きいほどシミュレーション時間が短い傾向にあることがわかる．これは，1 並列時はコア 0 番が所属しているクラスタのシミュレーションしか行わず，並列度が大きくなるほど他のクラスタのシミュレーションに要する時間が増え，それを並列に実行できているためだと考えられる．ただし，FFT は全体に比べ並列に実行している期間が短いため，差がほとんど現れなかったと考えられる．一方で，2 スレッド実行時（2A）では，全プログラムのシミュレーション時間を削減できている．これにより，マシン A（Core2）上では（N）に比べ平均で 40.9%，最大 43.8%のシミュレーション時間の削減に成功した．

次に，マシン B（SPARC）上で実行した場合のシミュレーション時間を**図18**に示す．

表 14: 選択したオプション

実験環境	スレッド数	スレッド生成	バリア	コアへのバインド
マシン A	4 スレッド	Solaris スレッドライブラリ	ビジーウェイトバリア	バインドしない
	2 スレッド	Pthread ライブラリ	ビジーウェイトバリア	バインドする
マシン B	4 スレッド	Pthread ライブラリ	ビジーウェイトバリア	バインドしない
	2 スレッド	Pthread ライブラリ	ビジーウェイトバリア	バインドしない

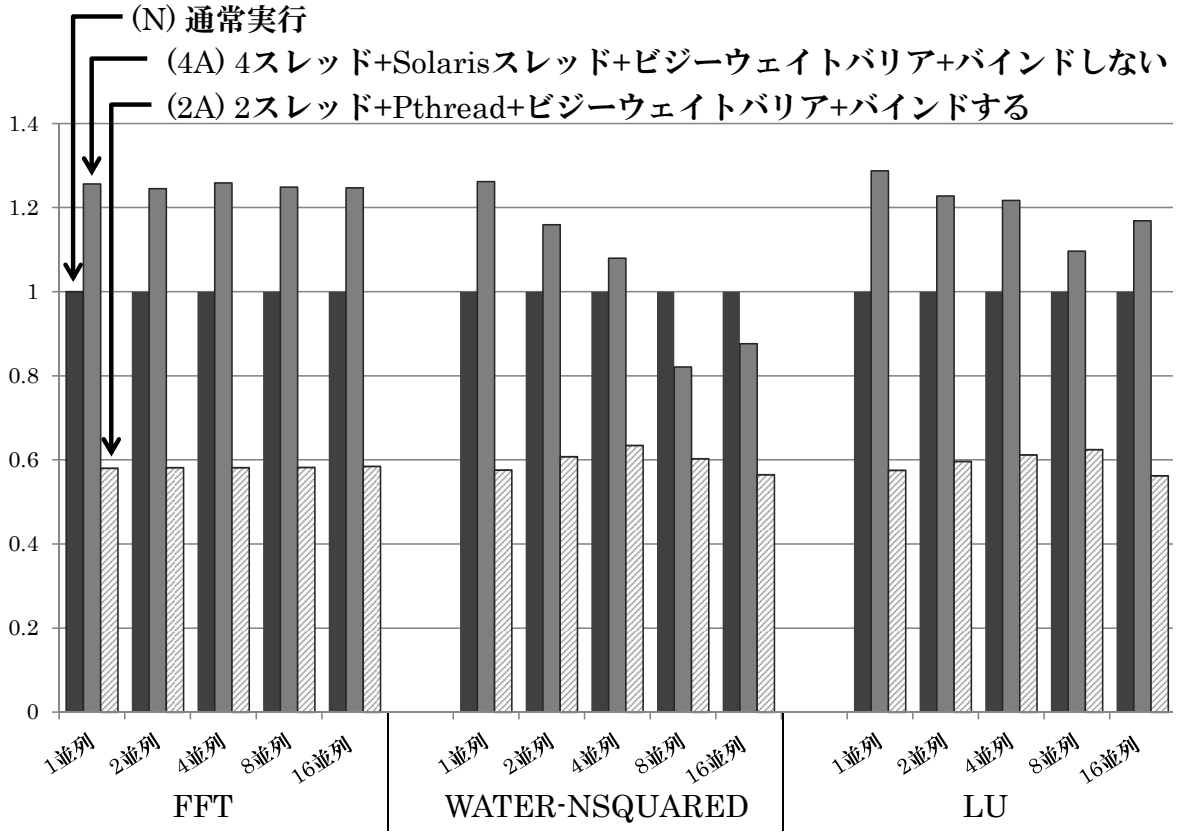


図 17: マシン A（Core2）で実行した場合のシミュレーション時間

図 18 中の 3 本のグラフは，左から

（**N**） 通常実行

（**4B**） 4 スレッド実行／ Pthread ／ビジーウェイトバリア／バインドしない の組み合わせで実行

（**2B**） 2 スレッド実行／ Pthread ／ビジーウェイトバリア／バインドの組み合わせで実行

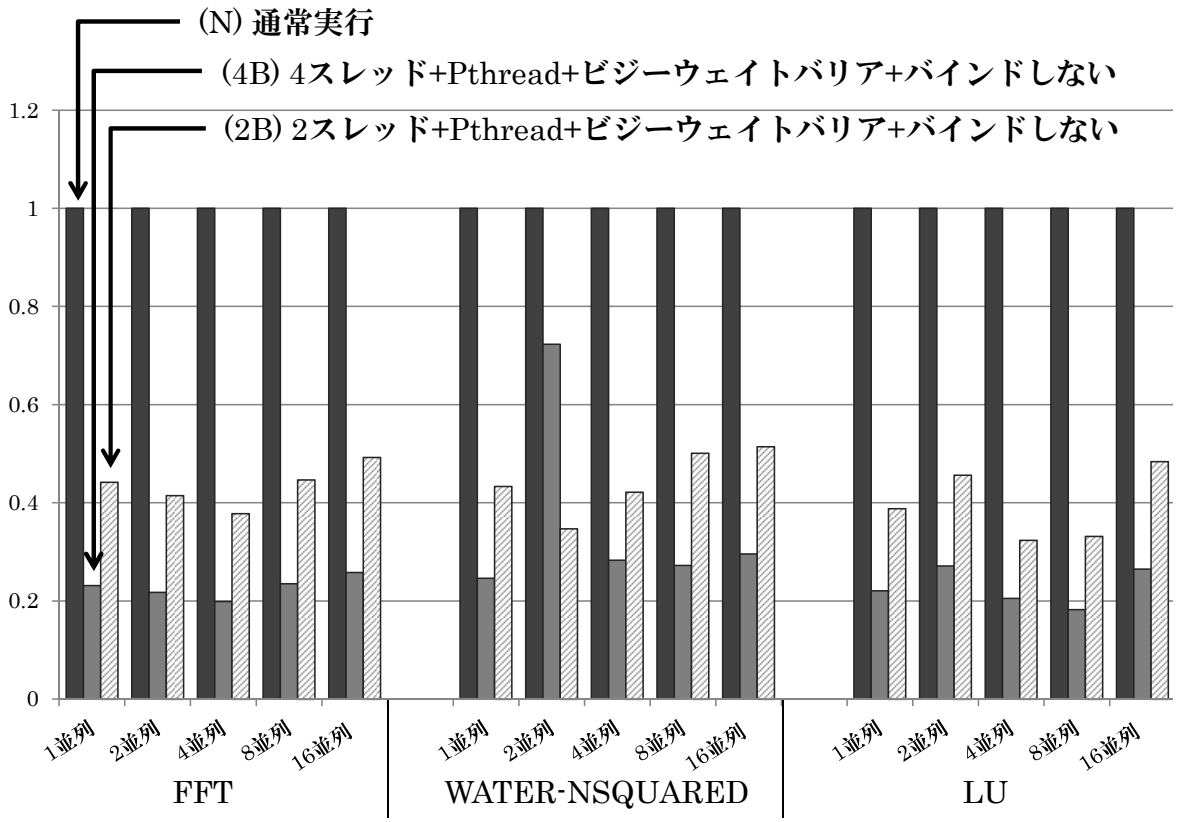


図 18: マシン B（SPARC）で実行した場合のシミュレーション時間

が要したシミュレーション時間を示しており，通常実行時を 1 として正規化している．

マシン B（SPARC）上で実行した場合は，スレッド数に関わらず全てのプログラムでシミュレーション時間の大幅な削減に成功した．（4B）では 4 スレッド実行しているため 4 倍の速度向上が上限だと予測していたが，ほとんどのプログラムでそれを上回っている．これは，7.2.3 項で述べたように，並列処理性能の高い UltraSPARC T1 で実行したため，メモリアクセスレイテンシの隠蔽や，キャッシュのヒット率が向上し，その結果大幅に高速化したためだと考えられる．これにより，4 スレッド実行時は平均で 72.6%，最大で 81.8%のシミュレーション時間の削減に成功した．

なお，マシン B（SPARC）は物理コアを 8 個，論理コアを 32 個持っているが，今回の評価では 4 スレッドと 2 スレッドでしか評価を行っていない．そのため，システムコールのシミュレーションを改良し，クラスタ数以上にスレッド並列化できればさらなる高速化を見込めると考えられる．また，その際のスレッド間の処理量がなるべく均等になるよう

に調整する方法を検討することなどが今後の課題である．

結果をまとめると，マシン A（Core2）上では 2 スレッド実行時に最も高速に実行でき，平均で 40.9%，最大 43.8%のシミュレーション時間の削減に成功した．一方，マシン B（SPARC）上では 4 スレッド実行時に最も高速に実行でき，平均で 72.6%，最大で 81.8%のシミュレーション時間の削減に成功した．

8　おわりに

本論文では，集積度の向上に伴い搭載するコア数を増大させたメニーコアプロセッサのアーキテクチャを検討するために，メニーコアトレースシミュレータを開発した．メニーコアトレースシミュレータでは，メニーコアプロセッサの実現においてボトルネックとなる配線遅延を考慮して，キャッシュ構成やメモリ一貫性プロトコル等のデータ供給方式および，複数のコアやメモリを相互に結合し交信路を提供する相互結合網の様々な形状を検討した．本論文では，性能目標値を導出するために基本となるメニーコアプロセッサの構成を設計し，代表的なアプリケーションを実行可能なシミュレータを実装した．加えて，今後のメニーコア研究を円滑に行うために，スレッド並列化によって高速にシミュレーションを行う機能をメニーコアトレースシミュレータに追加実装した．これにより，平均で 56.8%，最大で 81.8%のシミュレーション時間の削減に成功した．シミュレータ自身が並列度を自動で調整するため，ユーザへの負担を増やすことなく高速化を実現した．

本研究の今後の課題として，以下の 4 つが挙げられる．まず 1 つ目の課題として，開発したメニーコアトレースシミュレータの動作の正当性を向上させることが考えられる．本研究では，シミュレータの動作を確認するために，データキャッシュにダイレクトマップ方式を採用するなど実装を簡略化している部分がある．しかし，一般的なマルチコア・メニーコアプロセッサに搭載されるキャッシュシステムでは，データキャッシュのウェイ数を増やし転送効率を向上させた N-ウェイセットアソシアティブ方式が採られることが多い．そのため，こうした方式をメニーコアトレースシミュレータに対して組み込んでいく必要がある．また，動作可能なベンチマークプログラムを増やすとともに，各種の評価パラメータを変更して動作検証することでシミュレータの正当性を向上させる．

2 つ目の課題としては，メニーコアプロセッサ構成方式の検討が挙げられる．本論文では，基本構成によるシミュレーション実行までに留まっており，研究の全体計画における次の段階に進むために，基本構成以外の様々な構成パターンを構築してその実行トレースを採取する必要がある．また，得られた実行トレースの結果からメニーコアプロセッサ構成方式を比較・検討・考察し，単一プログラムの並列度限界を調査することで，効率的に

高並列実行可能なアーキテクチャを模索する．これにより，ハードウェア物量やデータ転送性能，消費電力において新たな知見が得られると考えられる．

3つ目の課題としては，メニーコアトレースシミュレータのシミュレーション実行のさらなる高速化が挙げられる．本論文では，単純なスレッド並列化で高速化を図ったが，シミュレータの高速化の研究は広く行われており，それらを組み込むことでさらに高速化させる．また，現在本シミュレータ上で動作するシステムコールは実装途中であるため，高速実行時のシミュレータの並列度をクラスタ数以上にすることができない．そのため，これらの改良も必要である．

最後の課題としては，メニーコアプロセッサに様々なプロセッサ高速化手法を組み合わせることが挙げられる．これまでに多数のコアの資源を有効に利用するための研究が行われており，それらの手法をメニーコアプロセッサに組み込んだ場合の効果を検証することで，今後の高速プロセッサ技術の進むべき道筋を示す．そのために，より複雑な環境を想定したメニーコアトレースシミュレータで動作を検証できるようにするのが今後の課題である．

謝辞

本研究のために，多大な御尽力を頂き，御指導を賜わった名古屋工業大学の松尾啓志教授，津邑公暁准教授，齋藤彰一准教授，松井俊浩准教授に深く感謝致します．また，本研究の際に多くの助言，協力をして頂いた松尾・津邑研究室，齋藤研究室および松井研究室の方々に感謝致します．加えて，卒業後も様々な助言や息抜きに誘って下さった池谷友基氏，稲葉崇文氏に深く感謝致します．

著者発表論文

報文

1. 山田 龍寛, 小田 遼亮, 池谷 友基, 津邑 公暁, 松尾 啓志, 中島 康彦: “命令区間の特徴を用いた自動メモ化プロセッサの再利用率向上手法”, 情処研報 (SWoPP2011), Vol.2011-ARC-196, No.1, pp.1–7 (2011).
2. 小田 遼亮, 山田 龍寛, 池谷 友基, 津邑 公暁, 松尾 啓志, 中島 康彦: “自動メモ化プロセッサの入力値エントリ統合による高速化”, 情処研報 (SWoPP2011), Vol.2011-ARC-196, No.2, pp.1–10 (2011).
3. 神村 和敬, 山田 龍寛, 小田 遼亮, 津邑 公暁, 松尾 啓志, 中島 康彦: “再利用対象区間の細分化による自動メモ化プロセッサの高速化”, 情処研報 (SWoPP2012), Vol.2012-

ARC-201, No.16, pp.1–8 (2012).

論文

1. Ryosuke ODA, Tatsuhiro YAMADA, Tomoki IKEGAYA, Tomoaki TSUMURA, Hiroshi MATSUO, Yasuhiko NAKASHIMA: “Input Entry Integration for an Auto-Memoization Processor”, Proc. 3rd Workshop on Ultra Performance and Dependable Acceleration Systems (UPDAS), pp.179–185 (2011).
2. Kazutaka KAMIMURA, Ryosuke ODA, Tatsuhiro YAMADA, Tomoaki TSUMURA, Hiroshi MATSUO, Yasuhiko NAKASHIMA: “A Speed-Up Technique for an Auto-Memoization Processor by Reusing Partial Results of Instruction Region”, Proc. 3rd Int’l. Conf. on Networking and Computing (ICNC), pp.49–57 (2012).
3. Tomoki IKEGAYA, Ryosuke ODA, Tatsuhiro YAMADA, Tomoaki TSUMURA, Hiroshi MATSUO, Yasuhiko NAKASHIMA: “A Hybrid Model of Speculative Execution and Scout Threading for Auto-Memoization Processor”, Proc. of Int’l. Symp. on System-on-Chip (SoC), pp.22–28 (2011).

参考文献

[1] Intel Corporation: *Product Brief: Intel C++ Compiler 11.0* (2008).

[2] SunMicrosystems: Sun Studio: C, C++ & Fortran Compilers and Tools, http://developers.sun.com/sunstudio/ (2009).

[3] Jason, H., Saurabh, D., Yatin, H., R., V. S., David, F., Gregory, R., David, J., Howard, W., Nitin, B., Gerhard, S., Fabric, P., Shailendra, J., Tiju, J., Satish, Y., Sraven, M., Praveen, S., Vasantha, E., Michael, K., Michael, R., Guido, D., Joerg, L., Matthias, G., Thomas, A., Kersten, H., Tor, L.-L., Sebastian, S., Shekhar, B., Vivek, D., der Wijngaart Rob F. Van and G., M. T.: A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS., *IEEE Int’l Solid-State Circuits Conf.(ISSCC) Digest of Technical Papers*, pp. 108–109 (2010).

[4] Seiler Larry and Carmean Doug and Sprangle Eric and Forsyth Tom and Dubey Pradeep and Junkins Stephen and Lake Adam and Cavin Robert and Espasa Roger and Grochowski Ed and Juan Toni and Abrash Michael and Sugerman Jeremy and Hanrahan Pat: Larrabee: A Many-Core x86 Architecture for Visual Computing, *IEEE Micro*, Vol. 29, No. 1, pp. 10–21 (2009).

[5] Intel Corp.: First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem), Intel Whitepaper (2008).

[6] Sony Computer Entertainment: *Cell Broadband Engine Architecture*, 1.01 edition (2006).

[7] Manish Shah and Robert T. Golla and Greg Grohoski and Paul J. Jordan and Jama Barreh and Jeffrey Brooks and Mark Greenberg and Gideon Levinsky and Mark Luttrell and Christopher Olson and Zeid Samoail and Matt Smittle and Thomas A. Ziaja: Sparc T4: A Dynamically Threaded Server-on-a-Chip, *IEEE Micro*, pp. 8–19 (2012).

[8] Tilera Corporation: *Product Brief: TILE64 Processor* (2007).

[9] Censier, L. M. and Feautrier, P.: A New Solution to Coherence Problems in Multicache Systems, *IEEE Trans. on Computers*, Vol. C-27, No. 12, pp. 1112–1118 (1978).

[10] 高前田伸也, 佐藤真平, 藤枝直輝, 三好健文, 吉瀬謙二: メニーコアアーキテクチャの HW 評価環境 ScalableCore システム, 情報処理学会論文誌コンピューティングシステム, Vol. 4, No. 1, pp. 24–42 (2011).

[11] Argollo, E., Falcon, A., Faraboschi, P., Monchiero, M. and Ortega, D.: COTSon: Infrastructure for Full System Simulation, *SIGOPS Operating Systems Review*, Vol. 43, No. 1, pp. 52–61 (2009).

[12] Johnnie Chan and Gilbert Hendry and Aleksandr Biberman and Keren Bergman and Luca P. Carloni: Phoenixsim: a Simulator for Physical-Layer Analysis of Chip-Scale Photonic Interconnection Networks, *Design, Automation and Test in Europe Conf. and Exhibition(DATE)*, pp. 691–696 (2010).

[13] Jose, S.(ed.): *TSystemC 2.0.1 Language Reference Manual Revision 1.0*, IEEE std. (2003).

[14] M.S.Gaur and B.M.Al-Hashimi and V.Laxmi and Navaneeth R and Naveen Choudhary and Lavina Jain and Mushtaq Ahmed and K.K.Paliwal and Varsha and Rekha and Vineetha: NIRGAM: A simulator for NoC Interconnect Routing and Applications' Modeling, *Design, Automation and Test in Europe Conf.(DATE)* (2007).

[15] Vasudeva Amol and Sharma Arvind Kumar and Kumar Ashish: Saksham: Customizable x86 Based Multi-Core Microprocessor Simulator., *Proc. 1st Int'l Conf.*

on Computational Intelligence, Communication Systems and Networks(CICSyN), pp. 220–225 (2009).

[16] Jason Cong and Karthik Gururaj and Guoling Han and Adam Kaplan and Mishali Naik and Glenn Reinman: MC-Sim: an efficient simulation tool for MP-SoC designs, *IEEE/ACM Int'l Conf. on Computer-Aided Design (ICCAD 08), November 10-13, 2008, San Jose, CA, USA*, pp. 364–371 (2008).

[17] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P. and Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proc. 22nd Annual Int'l. Symp. on Computer Architecture (ISCA'95)*, pp. 24–36 (1995).

[18] 中田尚, 三輪忍, 中村宏: NoC 型メニーコア設計のための高速キャッシュシミュレーション, 第 194 回 ARC・第 137 回 HPC 合同研究発表会 (HOKKE-20) (2012).

[19] Thomas F. Wenisch and Roland E. Wunderlich and Michael Ferdman and Anastasia Ailamaki and Babak Falsafi and James C. Hoe: SimFlex: Statistical Sampling of Computer System Simulation, *IEEE Micro*, Vol. 26, No. 4, pp. 18–31 (2006).

[20] Perelman Erez and Hamerly Greg and Van Biesbrouck Michael and Sherwood Timothy and Calder Brad: Using SimPoint for accurate and efficient simulation, *Proc. Int'l Conf. on Measurements and Modeling of Computer Systems, SIGMETRICS '03*, SIGMETRICS '03, New York, NY, USA, ACM, pp. 318–319 (2003).

[21] Schnarr Eric and Larus James R.: Fast out-of-order processor simulation using memoization, *Proc. 8th Int'l Conf on Architectural support for programming languages and operating systems*, ASPLOS VIII, New York, NY, USA, ACM, pp. 283–294 (1998).

[22] 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（ACS）, Vol. 45, No. SIG 6(ACS 6), pp. 54–65 (2004).

[23] 高崎透, 中田尚, 津邑公暁, 中島浩: 時間軸分割並列化による高速マイクロプロセッサシミュレーション, 情報処理学会論文誌：コンピューティングシステム, Vol. 46, No. SIG 12(ACS 11), pp. 84–97 (2005).

[24] Reinhardt Steven K. and Hill Mark D. and Larus James R. and Lebeck Alvin R. and Lewis James C. and Wood David A.: The Wisconsin Wind Tunnel: virtual prototyping of parallel computers, *Proc. Int'l Conf. on Measurements and Mod-*

eling of Computer Systems, SIGMETRICS '93, SIGMETRICS '93, ACM, pp. 48–60 (1993).

[25] Matsuo H. and Imafuku S. and Ohno K. and Nakashima H.: Shaman: A Distributed Simulator for Shared Memory Multiprocessors, *Proc. 10th IEEE Int'l Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems(MASCOTS)*, MASCOTS '02, Washington, DC, USA, IEEE Computer Society, pp. 347– (2002).

[26] Potuzak Tomas: Distributed-Parallel Road Traffic Simulator for Clusters of Multicore Computers, *Proc. 16th IEEE/ACM Int'l Symp. on Distributed Simulation and Real Time Applications(DS-RT)*, DS-RT '12, Washington, DC, USA, IEEE Computer Society, pp. 195–201 (2012).