Vol.2012-ARC-202 No.15
Vol.2012-HPC-137 No.15
2012/12/13

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

中田 尚[1,a)]　三輪 忍[1]　中村 宏[1]

概要：比較的小規模なコアを多数接続したメニーコアシステムは電力あたり性能に優れたシステムとして注目されている．特にコア間をオンチップネットワーク（Network on Chip: NoC）で接続した NoC 型メニーコアはコア数増大に適応可能なスケーラブルなメニーコアシステムであるといえる．しかし，NoC 型メニーコア設計においては設計パラメータが多岐にわたり，各パラメータを変えつつ，詳細なシミュレーションを行うためには膨大な時間が必要となり，効率的な設計の妨げになっている．そこで本研究では，NoC 型メニーコアの性能に与える影響が大きな共有キャッシュに注目し，そのシミュレーションをコア本体やネットワークのシミュレーションと分離する．これにより，共有キャッシュの挙動を高速にシミュレートし，性能予測を大幅に高速化することで，設計空間の大まかな絞り込みを実現する．

キーワード：NoC 型メニーコア，設計空間探索，共有キャッシュシミュレーション

1. はじめに

一般的なプロセッサでは 1 コアあたりの規模が大きくなるにつれ，性能は向上するがそれ以上に消費電力が増大することが知られている [1]．そのため，電力効率向上のためにはシンプルなコアを大量に接続したメニーコアが有望視されており，多くのメニーコアが登場してきている．多いものでは GPGPU のように 1000 コア以上を実装したシステムがすでに実用化されている．本研究でシミュレーション対象とする **NoC 型メニーコア**の例を**図 1** 示す．各コアは 1 次キャッシュと 2 次キャッシュを搭載するとともに，共有キャッシュを制御するディレクトリコントローラやネットワークに接続するためのルータを備えている．コア間はメッシュネットワークで接続されており，コア数の増加に対しても柔軟に対応することができる．

メニーコアシステムの利用方法は，現在は科学技術計算のように 1 つのシステムを 1 つのアプリケーションで専有するものが中心であるが，今後メニーコアがさらに普及するとマルチタスク処理のように複数のアプリケーションを 1 つのシステムで同時に実行することが予想される．このような状況では，実行するアプリケーションの組み合わせによる性能の変化を予測することが重要である．しかし，各アプリケーションに割り当てるコア数や，アプリケーションの組み合わせが多岐にわたるため，全てのパターンについて網羅的に厳密な性能予測を行うことは設計空間の膨大さから非現実的である．そこで，本研究ではメニーコアを構成する各要素の特徴を利用することにより，高速かつ的確な性能予測を実現し，メニーコアの設計最適化問題を解決する．

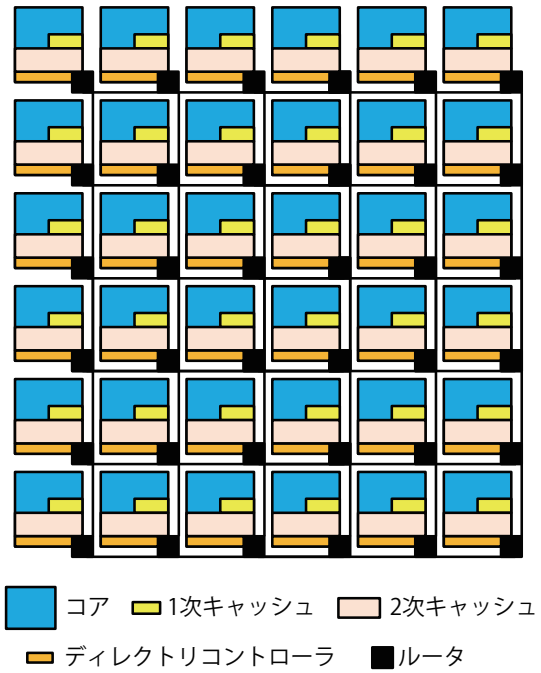

図 1　NoC 型メニーコア

[1] 東京大学
7-3-1 Hongo, Bunkyo, Tokyo 113–8656, Japan
[a)] nakada@hal.ipc.i.u-tokyo.ac.jp

メニーコアシミュレーションは大まかには，プロセッサコア自体のシミュレーション，共有キャッシュのシミュレーション，コア間ネットワークのシミュレーション，に大別される．プロセッサコアはシンプルなコアを利用するため，挙動の把握は容易であり，全体のシミュレーション時間に占めるコア自体のシミュレーション時間の割合は多くない．また，すでに多くの高速化の研究がありその成果を応用することもできる．ネットワークシミュレーションについては，最適な設計付近では混雑が発生することはまれであることに注目し，簡易的なシミュレーションで対応する．以上のことから，本研究ではキャッシュシミュレーションを重点的に高速化する．

本研究が対象とする初期設計の段階では，厳密な精度ではなくとも，大まかな絞り込みが迅速にできることが重要である．具体的には高速なシミュレーションにより最適な構成の候補となるいくつかのパターンを提示し，それらの中に正解が含まれていれば良いと考える．以上のことから，シミュレーション結果の絶対誤差よりも相対誤差を減らすことができれば，迅速かつ正確な設計空間の絞り込みが可能となる．

2. 関連研究

性能予測は設計開発に不可欠であり，これまでに非常に多くの研究がされてきた．本研究の対象とする NoC 型メニーコアは多数のプロセッサをネットワークで接続したものであり，これまでに提案されてきた各種高速シミュレーション技術を活用することは重要である．

本章では，命令の論理的動作のみをシミュレートする**命令レベルシミュレーション**，システム全体を詳細にシミュレートする**詳細シミュレーション**，共有メモリの動作を詳細にシミュレートする**共有メモリシミュレーション**について既存の高速化手法について概観する．

2.1 高速命令レベルシミュレーション

論理的動作のみをシミュレートする命令レベルシミュレーションは，マイクロプロセッサ開発にとって最も基本的なものである．また，論理的動作のみをシミュレートすれば良いため，高速性が強く要求される．

代表的な高速化手法としてダイナミックバイナリ変換がある．この高速命令レベルシ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，従来の命令インタプリタ方式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は，Shade[2] がある．Java や各種動的言語で広く用いられている JIT 技術もダイナミックバイナリ変換の 1 種といえる．

本稿ではこのような高速化技術の利用を想定し，命令レベルシミュレーションは十分高速に実行可能であると仮定する．

2.2 高速詳細シミュレーション

システム全体をシミュレートする詳細シミュレーションでは，命令レベルシミュレーションと比較して複雑な処理を行う必要があり，その実行速度も必然的に低下することとなる．また，シミュレーションの主な目的が性能の予測であるため，精度と速度のトレードオフがある．

精度を犠牲にして，高速化するものとして，SimPoint[3] がある．SimPoint では統計的手法を用いることにより，プログラムの実行をいくつかのフェーズに分類し，各フェーズに対して許容される誤差を達成するために必要な最小限の詳細シミュレーションを行い，その結果から全体の性能を予測する．

厳密な精度を保ったまま高速化を行う技術として，計算再利用を用いた手法がある [4]．この手法では詳細シミュレーションで出現するパターンを記録し，同一のパターンが繰り返されることを検出すると，前回の結果を再利用しその部分のシミュレーションを省略することにより，高速化を実現する．

また，シミュレーション区間を時間軸で分割することにより単一プロセッサの詳細シミュレーションを並列化する手法がある [5]．これらの手法では再利用や，分割の統合時にシミュレーション結果が正しいことを検証しており，結果の正当性が保証出来ない場合には自動的に詳細なシミュレーションが追加で実行される．

本研究では，おおまかな性能予測が目的であるため，このような詳細シミュレーションを行うことなく全体性能を予測する．

2.3 高速共有メモリシミュレーション

多数のコアを接続するためには，コア間のデータを共有する手段が必要不可欠である．特にメモリ空間を共有する共有メモリ型システムは古くから存在し，その高速シミュレーション技術も数多く研究されている．Shaman[6] では，共有メモリの挙動を決定するメモリアクセスは，全メモリアクセス中のごく一部であることに注目し，フィルタキャッシュと呼ばれる小規模なキャッシュを導入することにより，共有メモリシミュレータへの入力を大幅に削減する手法を提案している．

本研究ではメモリアクセスレイテンシの変化が命令の論理的動作には影響を与えないと仮定することにより，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を分離する．

3. NoC 型メニーコア

本章では本研究が対象とする NoC 型メニーコアのアー

表 1 対象アーキテクチャ

ネットワークトポロジ	メッシュ
ルータ	ワームホールルータ
ルーティング	次元順ルーティング
キャッシュ階層	2 階層
L1 キャッシュ	専有，ライトスルー
L2 キャッシュ	共有，ユニファイド
一貫性管理	ディレクトリ方式
プロセッサタイプ	インオーダ実行

キテクチャの概要について述べる．

3.1 想定するアーキテクチャ

本研究で対象とする NoC 型メニーコアのアーキテクチャを**表 1** に示す．

実行方式はインオーダー実行とする．メモリアクセス命令を除く全命令は 1 サイクルで実行完了し，分岐のペナルティは無視出来るものとする．したがって，論理的な実行内容が同じであれば，性能の差はキャッシュミスによって追加で必要となるサイクル数の差に等しい．

メインメモリはコアの数と同数のバンクに分割され，各コアに 1 バンクずつ接続されているものとする．キャッシュは 2 レベルとし，1 次キャッシュはプライベート，2 次キャッシュは共有とする．共有 2 次キャッシュの一貫性はディレクトリベースプロトコルとする．各コアはディレクトリを持ち，同一コアに接続されているメインメモリのバンクに相当するアドレス空間を管理する．

コア間ネットワークのルーティングは固定式であり，ネットワークが混雑していなければ，同一の通信パターンであれば通信のレイテンシは常に一定となる．

3.2 共有メモリ

対象とするアーキテクチャでは 2 次キャッシュを共有する構成とする．したがって，1 次キャッシュや 2 次キャッシュにヒットしている限りはコア間通信は発生しないが，2 次キャッシュミスが発生すると当該アドレスを管理するディレクトリコントローラに最新データが保管されている場所を問い合わせる．その結果，他のコアの 2 次キャッシュに最新データが保管されていた場合には，そのコアに当該データを要求元のコアに転送するようにリクエストする．どの 2 次キャッシュにも保管されていない場合には，メインメモリから取得する必要があるが，ディレクトリコントローラとメインメモリは同一間隔でバンク化されているため，必要なデータが保存されているメインメモリは必ずディレクトリが存在するコアと同じコアに接続されており，追加の通信は不要である．

また，書き込みアクセスが発生した場合には，他のコアが古いデータを保持していかどうかをチェックし，保持していたコアに対しては上記の処理に加えて，無効化要求を送信し，データの一貫性を保つ．

本研究では以上のような共有メモリシミュレーションを高速化することが目的である．

3.3 ネットワーク

前述のとおり，本研究ではネットワークは混雑せず，常に理想的なレイテンシで通信が可能であると仮定する．

具体的には，2 つのコア間の通信レイテンシは，各コアのネットワークインターフェースの遅延，ルータとリンクの遅延の合計となる．本研究では固定ルーティングを仮定しているため，1 回の通信で通過するルータとリンクの数はコア間のホップ数から求めることが出来る．例えば，本研究で仮定するメッシュ型ネットワークであれば，ホップ数はコア間のマンハッタン距離と等しく，送受信コアの場所がわかれば容易に求めることができる．

3.4 性能モデル

前節までの議論で NoC 型メニーコアの性能を予測するための指標を全て求めることができた．すなわち，総実行命令数とキャッシュのミスレイテンシの総和がプログラムの実行時間に等しい．各ミスレイテンシにはネットワークの遅延が含まれるが，これはリクエスト元のコア位置とリクエストアドレスから決定されるディレクトリの位置によって求められる．

以上により，共有キャッシュシミュレーションを高速に実行出来ればシステムの性能が求められることがわかる．

4. 高速キャッシュシミュレーション

本章では共有キャッシュシミュレーションの高速化手法について述べる．

詳細なシミュレーションが低速となる主な原因は，常に全コア間で同期を取っているためである．厳密にはあるコアがキャッシュアクセスした結果に応じて，他のコアにおけるその後の動作が変化する可能性があるため，詳細な解析には十分短い間隔での同期が必要不可欠である．しかし，高速シミュレーションのためにはこのような厳密な同期のコストは許容出来ない．

そこで，本研究ではキャッシュシミュレーションの結果によってプログラムの実行パターンは変化しないと仮定する．つまり，命令レベルシミュレーションを 1 度だけ行いその実行パターンを保存し，その結果を用いて共有キャッシュシミュレーションを行う．具体的にはある並列アプリケーションを指定されたコア数で実行したときの全コアのアクセストレースを取得し，その全順序関係はキャッシュシミュレーションによって変化しないと仮定する．

実行の流れを**図 2** に示す．まず，実行対象の並列アプリケーションの命令レベルシミュレーションを個別に実行し，アクセストレースを取得する．割り当てるコア数を変化さ

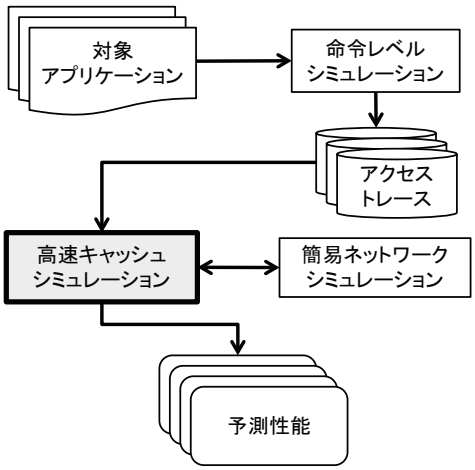

図 2 実行の流れ

せる場合にはそのパターンの数だけ命令レベルシミュレーションを繰り返す．次に，保存したアクセストレースを元にキャッシュシミュレーションを行う．複数アプリケーションの同時実行をシミュレーションする場合には，それぞれのアクセストレースを入力とする．キャッシュシミュレーションで必要となるネットワークシミュレーションは混雑が発生しないと仮定した簡易的なモデルを用いる．

以降では各シミュレータモジュールの詳細について順に述べる．

4.1 命令レベルシミュレーション

並列アプリケーションにおいてコア間の時刻同期はアプリケーションの振る舞いを決定するうえで重要であるが，キャッシュやネットワークのシミュレーションと分離した時点で厳密な時刻管理は不可能であり，なんらかの仮定を用いる必要がある．そこで命令レベルシミュレーションでは，全てのメモリアクセスは 1 サイクルで完了するものとする．つまり，コアはメインメモリと直結されており書き込んだデータは即座に他のコアから読み出し可能になることと等価である．

マルチコアの命令レベルシミュレーションを高速に行うためには，コアの切替間隔が重要なパラメータとなる．並列アプリケーションが正しく作成されていれば，各コアがどのようなタイミングで動作しようとも，実行結果の正当性が保証される．一方，コアの切替にはオーバヘッドが存在するため，シミュレーション高速化のためにはできる限り 1 つのコアを連続で長時間シミュレートすることが有効である．しかし，あまりにコアの切替頻度を少なくすると，コア間の情報伝搬速度が低下し同期処理のような双方向の依存関係が発生したときに，実際よりも長時間の同期待ちが発生する可能性がある．実際のシミュレータでは，内部構造にも依存するが，1000 命令程度の間隔でコアを切り替えることでオーバヘッドをほぼ隠蔽出来る．本研究では厳密な時刻同期を行わないとはいえ，1000 命令の同期間隔は許容出来ないと考え，より短い間隔でコアを切り替えることを検討する．

以上のように命令レベルシミュレーションを行い，そのアクセストレースを全て記録するし，後段のキャッシュシミュレーションへ供給する．

4.2 キャッシュシミュレーション

現実の共有メモリでは，同じアドレスに対して同時に書き込みが発生すると，どちらの書き込みが先に処理されるかはネットワーク遅延などにも影響される．しかし，このような処理はシミュレーションを複雑化させ，シミュレーション速度の低下を招く．

そこで，本研究では高速化のためにキャッシュへのアクセスはお互いに独立であると仮定する．つまり，複数のキャッシュアクセスが同時平行して発生する場合であっても，トレース上で先行するアクセスのシミュレーションを先に行い，そのシミュレーションが完了してから後続のアクセスの処理を開始する．これにより，特に 2 次キャッシュミスのようにレイテンシが長く，途中で他の処理と競合する可能性があるアクセスに対しても高速なシミュレーションを実現する．

また，命令レベルシミュレーションでは時刻情報が取得出来ないため，割り切った仮定をおく必要があったが，キャッシュシミュレーションではある程度の時刻情報が取得出来るため，時刻の扱いに対していくつかの方針を考えることができる．具体的には以下の 3 パターンが考えられる．

(1) 時刻管理を行わない
　キャッシュヒット/ミスに関わらず 1 サイクルで実行が完了すると仮定する．
(2) コア毎に時刻管理を行う
　キャッシュミスが発生するとコアの時刻をその時刻だけ遅らせる．
(3) アプリケーション毎に時刻管理を行う．
　キャッシュミスが発生するとコアの時刻を遅らせるが，アクセスの全順序関係を維持するために，他のコアの時刻も遅らせる．

これらの手法には一長一短があり，どの手法が最適であるかは明らかではない．

まず，時刻管理を行わない場合が最も単純であり，ネットワークシミュレーションも不要になるため，最もシミュレーション時間が短くなることが期待できる．この方法は単一の並列アプリケーションのみのシミュレーションであれば，ある程度の有効性は期待出来るが，複数の並列アプリケーションを実行する環境では問題が発生する．

例えば，2 つの並列アプリケーションが同時に実行されているときに，一方のプログラムでキャッシュミスが頻発し実行速度が低下すると，単位時間あたりのキャッシュア

クセス回数も減少する．その結果，キャッシュの使用量が減り，他方のアプリケーションが利用可能なキャッシュ容量が増加する．このように，並列アプリケーション間の相対的な実行速度は共有キャッシュ環境において重要な要素である．

次に，キャッシュミスレイテンシが取得出来ることを利用しコア毎に時刻管理を行う方法がある．これにより，キャッシュミスの発生状況が，コア毎のアクセス頻度に反映されるようになる．しかし，コア単位の時刻管理だけでは同一並列アプリケーションを実行するコア間での時刻がずれてしまい，本来であればあり得ないアクセスパターンが発生する危険性がある．

そこで，同一アプリケーションを実行するコア間で時刻の同期を取る方法が考えられる．単純にはキャッシュミスが発生した場合に，そのアクセスレイテンシを他のコアに対しても同時に適用することが考えられる．しかし，異なるバンクに属するアドレスに対してほぼ同時にキャッシュミスが発生した場合には，それらのキャッシュミスは並列に処理されると考えるべきであり，レイテンシの単純な合計では過剰な遅延となる可能性がある．

この問題を解決する方法の一つには，アクセス毎にレイテンシを反映させるのではなく，ある程度の時間間隔で同期を取ることが考えられる．これにより同期間隔内においてある程度の順序関係の逆転が発生するが，過剰な遅延の発生を防止することができる．もう一つの方法として，アクセスのアドレスをお互いにチェックし，同時に解決可能かどうかを判断する方法も考えられる．しかし，この方法ではチェックが必要なアクセスが煩雑に発生すると速度低下に直結する危険性があるため，慎重に検討する必要がある．

4.3 ネットワーク

ネットワークシミュレーションについては，3.3 節で述べたように，アクセス間の競合は発生しないと仮定しているので，アクセスアドレスから決定されるリクエスト先のコアと，アクセス元のコアのマンハッタン距離からホップ数を求め，そのホップ数を通信するのに必要なレイテンシをルータなどの遅延パラメータから求める．

5. 予備評価

本章では，提案する高速キャッシュシミュレータの予備評価として，それぞれのシミュレータを個別に実行した場合の実行時間やアクセストレースのサイズについて考察する．評価には Gems/Simics[7]，ベンチマークには NAS Parallel Benchmark[8] の LU を用いた．また，NoC 型メニーコアの構成を**表 2** に示す．ここで，ネットワークとルータの遅延は先頭フリットのみにかかる遅延であり，後続のフリットの処理は適切にパイプライン化されているものとする．

表 2 シミュレーション条件

L1 キャッシュ	16KB, 4way set-associative 1 cycle latency
L2 キャッシュ	256KB, 16way set-associative 4 cycle latency
メインメモリ	160 cycle latency
フリットサイズ	16Byte
ネットワーク インターフェース遅延	2 cycle
ルータ遅延	3 cycle
仮想チャンネル数	4
コア数	16
命令セット	SPARC
OS	Sun Solaris9

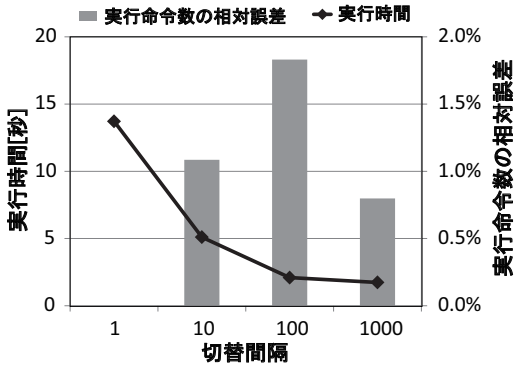

図 3 命令レベルシミュレーション

また，今回の予備評価では 1 つのみの並列アプリケーションを搭載する全 16 コアを用いて実行した．

5.1 命令レベルシミュレーション

並列アプリケーションの命令レベルシミュレーションにおける，コア切替間隔のオーバヘッドと実行命令数の変化を評価した．結果を**図 3** に示す．実行命令数は切替間隔が 1 命令の結果を基準とした相対誤差である．

この結果から切替間隔が 1000 命令程度あれば実行速度は十分飽和しており，それ以上長い切替間隔にする必要はないことがわかる．また，切替間隔の違いによる実行命令数の変化は最大で約 1.8%であることがわかった．

5.2 実行時間の比較

シミュレーション全体の実行時間を予測するために，各モジュールの実行時間を測定した．結果を**表 3** に示す．「実機」はシミュレータを用いずに直接実行，「命令レベル」は命令レベルシミュレーションをそれぞれのコア切替間隔で実行，「命令レベル＋全アクセストレース」は命令レベルに加えて全アクセスのトレースをファイルに保存した場合，「高速キャッシュ」は高速キャッシュシミュレーション，「詳細」は詳細シミュレーションを実行した場合のそれ

表 3　シミュレーション時間

	実行時間 [秒]	相対速度
実機	約 0.006	1
命令レベル (1000 命令切替)	1.7	280
命令レベル (1 命令切替)	14	2400
命令レベル＋全アクセストレース	(540)	-
高速キャッシュ	40	6700
詳細	244	40700

ぞれの実行時間である．相対速度は「実機」を 1 としたときの実行時間の比である．

各モジュールの実行時間は大きな差があるが，設計空間を探索することを想定すると，各モジュールの実行回数は同一ではない点に注意が必要である．例えば，一度アクセストレースを取得すれば，実行コア数などが変わらない限り同一のトレースを再利用することができる．また，「命令レベル＋全アクセストレース」の結果は命令レベルシミュレーションを行いながら，全アクセストレースを NFS 経由のファイルサーバに書き出した場合の結果である．I/O 性能の影響を大きく受けていることが予想されるので，参考程度とする．

前節の結果のとおり，命令レベルシミュレーションにおけるコア切替間隔による実行時間の差は大きいが，他のモジュールと比較すると十分無視出来る程度であり，1 命令切替を利用しても問題ないといえる．しかし，アクセストレースを取得しようとすると詳細シミュレーションよりも長い時間がかかってしまうことがわかった．これは Gems/Simics の標準機能を用いており，命令トレースには逆アセンブル結果がが付加されるなど非常に充実した内容であるが，本研究にとってはそのような情報は不要であるため，改善の余地は大いにあるといえる．またこのとき生成される出力ファイルは約 11GB であり，ファイルサイズの点からも出力形式の検討が必要であるといえる．

また，高速キャッシュシミュレーションの実行速度が詳細シミュレーションの約 6.1 倍程度にとどまっているが，高速な性能予測のためにはさらなる高速化が必要不可欠である．改善のためには，関連研究であげたような既存の高速化手法を注意深く検討し，本研究の目的と合致する高速化手法を取り入れていくことが有用であると考える．

6. まとめ

本稿では NoC 型メニーコア設計を効率化するため，性能予測に重要となる要素を検討しキャッシュシミュレーションの高速化が重要かつ効果的であるとの見通しを得た．命令レベルシミュレーションとキャッシュ，ネットワークシミュレーションを分離し，トレースベースの共有キャッシュシミュレーションを採用した．ネットワークシミュレーションは最適設計付近ではネットワークが混雑しないということに注目し，理想的なモデルを用いる．さらに，キャッシュシミュレーションではアクセス間の競合が発生しないと仮定し，共有キャッシュシミュレーションを単純化した．

予備評価の結果から，現在の構成には，アクセストレースの取得方法および高速キャッシュシミュレーションに改善が必要であることが明らかとなった．今後は関連研究を参考にこれらの改良を行い，NoC 型メニーコア設計の効率化に有用な高速キャッシュシミュレーション手法の完成を目指す．

謝辞　本研究は (株) 半導体理工学研究センターとの共同研究「NoC 型メニーコア SoC のアーキテクチャレベル設計最適化の研究」による．

参考文献

[1] Pollack, F. J.: New microarchitecture challenges in the coming generations of CMOS process technologies (keynote address)(abstract only), *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture, MICRO 32*, p. 2 (1999).

[2] Cmelik, B. and Keppel, D.: Shade: A Fast Instruction-Set Simulator for Execution Profiling, *ACM SIGMETRICS Performance Evaluation Review*, Vol. 22, No. 1, pp. 128–137 (1994).

[3] Sherwood, T., Perelman, E., Hamerly, G. and Calder, B.: Automatically characterizing large scale program behavior, *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems, ASPLOS-X*, pp. 45–57 (2002).

[4] 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，Vol. 45, No. SIG6(ACS 6), pp. 54–65 (2004).

[5] 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性能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. SIG12(ACS 11), pp. 84–97 (2005).

[6] Matsuo, H., Imafuku, S., Ohno, K. and Nakashima, H.: Shaman: a distributed simulator for shared memory multiprocessors, *Proceedings of 10th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems, MASCOTS 2002.*, pp. 347–355 (2002).

[7] Martin, M. M. K., Sorin, D. J., Beckmann, B. M., Marty, M. R., Xu, M., Alameldeen, A. R., Moore, K. E., Hill, M. D. and Wood, D. A.: Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset, *SIGARCH Comput. Archit. News*, Vol. 33, No. 4, pp. 92–99 (2005).

[8] Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V. and Weeratunga, S.: The NAS parallel benchmarks summary and preliminary results, *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pp. 158–165 (1991).