修士論文

メモ化とキャッシュプリフェッチの融合およびトレースシミュレータの開発によるメニーコアアーキテクチャの検討

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 22 年度入学　22413505 番

池谷 友基

平成 24 年 2 月 3 日

メモ化とキャッシュプリフェッチの融合およびトレースシミュレータの開発によるメニーコアアーキテクチャの検討

池谷 友基

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 を持たないことから，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．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ある．

そこで，本論文では 2 つのアプローチからメニーコア時代に向けた高並列実行可能なアーキテクチャを検討する．1 つ目は，計算再利用をハードウェアにより動的に適用する自動メモ化プロセッサの高速化である．計算再利用は，主に関数などの命令区間に対してその入力と出力の組を実行時に記憶しておき，再び同じ入力によりその命令区間が実行されようとした場合に，過去に記憶された出力を利用することで命令区間の実行自体を省略し，高速化を図る手法である．さらに，この自動メモ化プロセッサにおいて並列事前実行という投機的手法を用いることで，複数コアを有効活用する手法がこれまでに研究されてきた．コア数の増大に伴い，並列化手法を用いても一部のコアに処理を割り当てきれず，全てのコアを有効に利用できない状況が発生することが想定されるが，並列事前実行はそうしたコアの空き資源を有効に活用することができると考えられている．

本論文では，従来の自動メモ化プロセッサで並列事前実行を担っている投機実行コアに，計算再利用の効果が見込めない命令区間に対するキャッシュプリフェッチを行う一種のスカウトスレッドを実行させる手法を提案する．この手法では，並列事前実行を行う事前実行スレッドとスカウトスレッドを計算再利用の成功状況に応じて動的

に選択する．これにより，従来の再利用の効果を阻害することなくメモリアクセスレイテンシの一部を隠蔽し，自動メモ化プロセッサのさらなる高速化を図る．提案手法の有効性を検証するため，従来の自動メモ化プロセッサに提案手法を実装し，SPEC CPU95 FP ベンチマークでシミュレーションにより評価した．その結果，通常通り命令を実行するのと比較し，従来手法では最大 40.6%，平均 15.0%のサイクル数の削減であったのに対し，提案手法では最大 41.3%，平均 19.1%のサイクル数を削減し有効性を確認した．

2 つ目は，高並列に実行可能なアーキテクチャを検討するためのメニーコアトレースシミュレータの開発である．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るが，データ供給面の問題などにより多数のコアを有効に利用することは困難である．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．

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，代表的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．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形状における様々な構成方式を構築し検証する必要がある．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．シミュレータの動作を確認するために，行列積演算プログラムおよび SPLASH-2 ベンチマークでシミュレーションにより検証した．その結果，複数のコアによる台数効果が得られており，正しく動作していることを確認した．

メモ化とキャッシュプリフェッチの融合およびトレースシミュレータの開発によるメニーコアアーキテクチャの検討

目次

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，配線プロセスの微細化による高周波数化により，マイクロプロセッサの高速化を実現できた．しかし，数年前からは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．

一方現在では，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．そこで，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) に着目して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技術が研究されている．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事ができる．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し，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．また，今後プロセッサ当たりのコア数の増大に伴い，既存の手法を利用しても一部のコアに処理を割り当てきれず，全てのコアを有効に利用できないという状況が発生する事も想定される．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．

また，今後は集積度の向上に伴って，搭載するコア数を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．そこで，本論文では 2 つのアプローチからメニーコア時代に向けた高並列実行可能なアーキテクチャを検討する．

1 つ目は，計算再利用をハードウェアにより動的に適用する自動メモ化プロセッサの高速化である．プログラムの並列化による高速化手法とは別の概念として，計算再利用と呼ばれる従来とは着眼点の異なる高速化手法があり，これまでに計算再利用を用いた自動メモ化プロセッサに関する研究が行われている [3, 4]．メモ化 [5] とは，関数等の命令区間を計算再利用な形に変換する処理であり，命令区間の実行時にその入出

力を記憶しておくことで，同一入力による当該関数の再計算を省略し実行を高速化する手法である．このようなメモ化技術を用いた高速化に関する研究はソフトウェア分野では広く行われている [6, 7]．例えば，関数型言語 Haskell を拡張し，関数にメモ化を適用可能にした言語が提案されている [8]．しかし，この手法ではメモ化を適用する関数を明示的に指定する必要があるため，プログラマの負担が大きい．また，プログラムを再度コンパイルする必要があるため，ソースコードが提供されていないプログラムを高速化することはできない．さらに，特定のプログラミング言語による記述をプログラマに強制とするという問題もある．また，ソフトウェアによるメモ化はオーバヘッドが大きいため，メモ化が適用可能なプログラムでも，高速化が見込めるようなプログラムは特定のプログラムに限定される傾向がある．

これに対し，本研究で扱う自動メモ化プロセッサは，ハードウェアを用いることで既存のバイナリを変更することなく，動的に関数やループ等の命令区間を検出し，それら命令区間に対してメモ化を自動的に適用する．さらに，ループイタレーション等の命令区間のうち入力が単調変化するものに対し，過去の履歴から次の入力を予測し，得られた値を用いてその命令区間を別コアで予め実行しておくことで出力を生成・記憶する並列事前実行と呼ばれる機構を備えるモデルも提案されている．このモデルでは，予測が正しかった場合にメインコアによる当該イタレーションの実行が計算再利用により省略できる．

本論文では，従来の自動メモ化プロセッサで並列事前実行を担っている投機実行コアに，計算再利用の効果が見込めない命令区間に対するキャッシュプリフェッチを行う一種のスカウトスレッド [9] を実行させる手法を提案する．並列事前実行を行う事前実行スレッドとスカウトスレッドを計算再利用の成功状況に応じて動的に選択することで，従来の再利用の効果を阻害することなくメモリアクセスレイテンシの一部を隠蔽する．また，このスカウトスレッド実行は値予測に基づく情報を使用するため，命令区間の入力毎にアドレスストライドが異なるような場合にも効果が期待できる．

2 つ目は，高並列に実行可能なアーキテクチャを検討するためのメニーコアトレースシミュレータの開発である．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るが，データ供給面の問題などにより多数のコアを有効に利用することは困難である．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，代表的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．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形状における様々な構成方式を構築し検証する必要がある．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．

以下，2 章では既存の自動メモ化プロセッサの構成と投機実行による高速化手法である並列事前実行について述べる．3 章では既存の自動メモ化プロセッサの問題点を挙げ，その解決策として投機コアのスカウトスレッド実行による高速化手法について述べ，4 章で提案手法を評価する．また，5 章でメニーコアプロセッサ研究の現状について述べ，6 章でメニーコアプロセッサの構成方式を検討する．7 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，8 章でシミュレータの動作を検証する．最後に，9 章で結論を述べる．

2　メモ化と自動メモ化プロセッサ

本論文で取り扱う自動メモ化プロセッサについて，その高速化の方針と動作原理を概説する．

2.1　メモ化と計算再利用

計算再利用 (Computation Reuse) とは，主に関数などの命令区間に対してその入力と出力の組を実行時に記憶しておき，再び同じ入力によりその命令区間が実行されようとした場合に，過去に記憶された出力を利用することで命令区間の実行自体を省略し，高速化を図る手法である．既知の入力値に対して再び同じ区間を実行する際に，正しい出力値を得ることができ，入力値さえ一致すれば出力結果を検証する必要がないことが特長に挙げられる．また，それら命令区間に計算再利用を適用することを**メモ化 (Memoization)** と呼ぶ．

並列化が処理全体の総量自体は変化させず複数の単位処理を同時実行することにより高速化を図る手法であるのに対し，計算再利用は処理自体を省略することで高速化を図る手法であり，その着眼点は根本的に異なっている．また，計算再利用は並列化とは直交する概念であるため，並列化が有効でないプログラムでも効果が得られる可

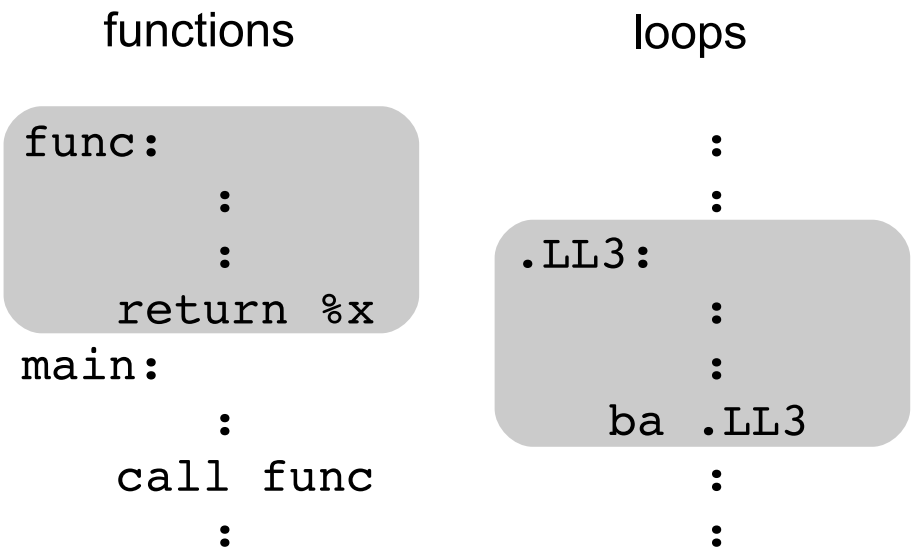

図 1: メモ化対象区間

能性があり，並列化とも併用可能であるという利点がある．

メモ化は元来，高速化のためのプログラミングテクニックであるが，本論文で扱う**自動メモ化プロセッサ (Auto-Memoization Processor)** は，既存バイナリを変更することなくメモ化実行可能なプロセッサである．自動メモ化プロセッサがメモ化対象とする関数およびループ区間のアセンブリプログラムの例を**図 1** に示す．関数の命令区間は call 命令によるジャンプ先の関数ラベルから当該関数の return 命令までであり，図中ではラベル func から return 命令までの範囲が一つの関数となる．一方で，ループの命令区間は後方分岐命令によるジャンプ先の分岐先ラベルから当該後方分岐命令までであり，図中ではラベル.LL3 から分岐命令 ba までの範囲が一つのループとなる．メモ化対象区間が関数の場合には，call 命令の検出から return 命令を検出するまでに出現した入出力セットを記憶する．一方で，メモ化対象区間がループの場合には，後方分岐命令によりジャンプした直後の命令から再び同一の後方分岐命令を検出するまでに出現した入出力セットを記憶する．ただし，後方分岐命令は必ずしもループ区間を構成するわけではないため，再び同一の後方分岐命令を検出した際に初めてループ区間を構成すると判断できる．そのため，1 回目のループイタレーションはそれをループ区間として認識することができない．

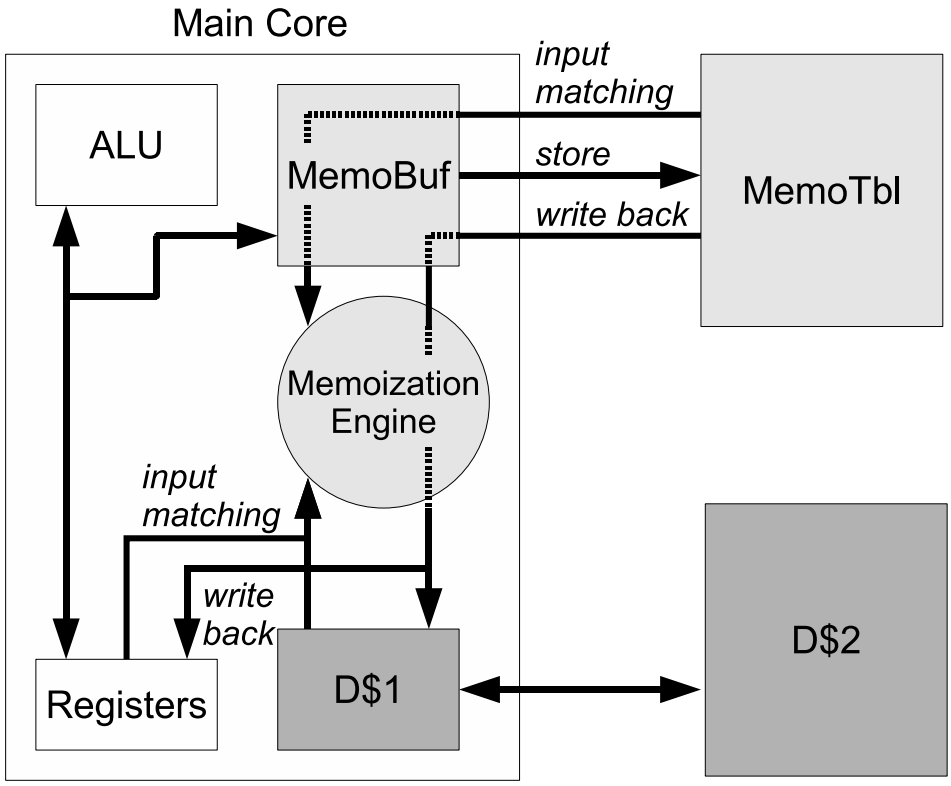

図 2: 自動メモ化プロセッサの構成

2.2 自動メモ化プロセッサ

自動メモ化プロセッサの構成を**図 2** に示す．自動メモ化プロセッサは，単命令発行の SPARC V8 をベースアーキテクチャとしており，コアの内部には一般的なプロセッサコアが持つ ALU，レジスタ (Registers)，1 次データキャッシュ(D$1) を持ち，コアの外部に 2 次データキャッシュ(D$2) を持つ．また，自動メモ化プロセッサが独自に持つ機構として，コアの内部に入出力を一時的に蓄えるバッファ(MemoBuf) とメモ化機構を管理するための制御ユニット (Memoization Engine) を持ち，コアの外部に命令区間の入出力セットを記憶するための表 (MemoTbl) を持つ．MemoTbl はサイズが大きく CPU コアからのアクセスコストが大きい．そのため，命令区間の入出力を MemoTbl に登録する際，MemoTbl に対して頻繁に参照を行うとオーバヘッドが大きくなってしまう．このオーバヘッドを軽減するため，作業用の書き込みバッファとして MemoTbl に比べてサイズの小さい MemoBuf が用いられ，各命令区間の実行終了時に MemoBuf の内容を一括して MemoTbl へと登録する．

まず，MemoBuf の構成を**図 3** に示す．MemoBuf は複数のエントリを持ち，1 エントリが 1 入出力セットに対応する．各エントリは，該当する命令区間を記憶する FLTbl

FLTbl idx.	SP	retOFs	read			write		
			#1	···	#n	#1	···	#n

図 3: MemoBuf の構成

idx，各命令区間の実行開始時のスタックポインタ SP，関数の戻り値とループの終端アドレスを記憶する retOFs，命令区間の入力セット Read，および出力セット Write の各フィールドからなる．命令区間の実行中に出現した入出力は，命令区間の実行を進めながら Read と Write フィールドに記憶されていく．入出力には参照したレジスタおよび主記憶アドレスの値を記憶する．そして，各命令区間の実行終了時に再利用エントリを一括して MemoTbl に登録する．

MemoBuf に複数のエントリが存在するのは，メインコアが実行中に呼び出した関数やループのネスト構造を保持するためである．MemoBuf はスタックのように扱われ，階層の低い方から順に使用される．そして，MemoBuf のどのエントリを使用しているかを判断するためにポインタを用いる．関数呼び出しや多重ループによってネストが増加すると，それに応じてポインタの値がインクリメントされる．逆に，関数呼び出しから戻った場合やループの実行が終了した時に，ポインタの値をデクリメントし階層を下げる．このようにして MemoBuf はコアが現在実行している関数やループのネスト構造を保持する．なお，命令区間の深さが MemoBuf の保持できる階層よりも深くなった場合には，最も外側つまり最上位の命令区間の入出力セットを記憶しているエントリの内容を削除し，空いたエントリに新しい命令区間の入出力セットを登録する．

次に，MemoTbl の構成を**図 4** に示す．MemoTbl は，命令区間の開始アドレスを記憶する **FLTbl**，入力値を記憶する **InTbl**，入力アドレスを記憶する **AddrTbl**，そして出力値を記憶する **OutTbl** の 4 つの表から構成されている．

FLTbl は，各再利用対象命令区間に対応する行を持っており，メモ化のためのフィールドおよび後述するオーバヘッドフィルタのためのフィールドを持っている．メモ化のためのフィールドには，関数およびループの別（F or L），命令区間開始アドレス（addr），また後述する並列事前実行の入力ストライド予測に用いるための直近の入力値セット（prev inputs）を記憶するフィールドがある．オーバヘッドフィルタのためのフィールドには，当該命令区間のサイクル数（S），過去の再利用に要した入力検索

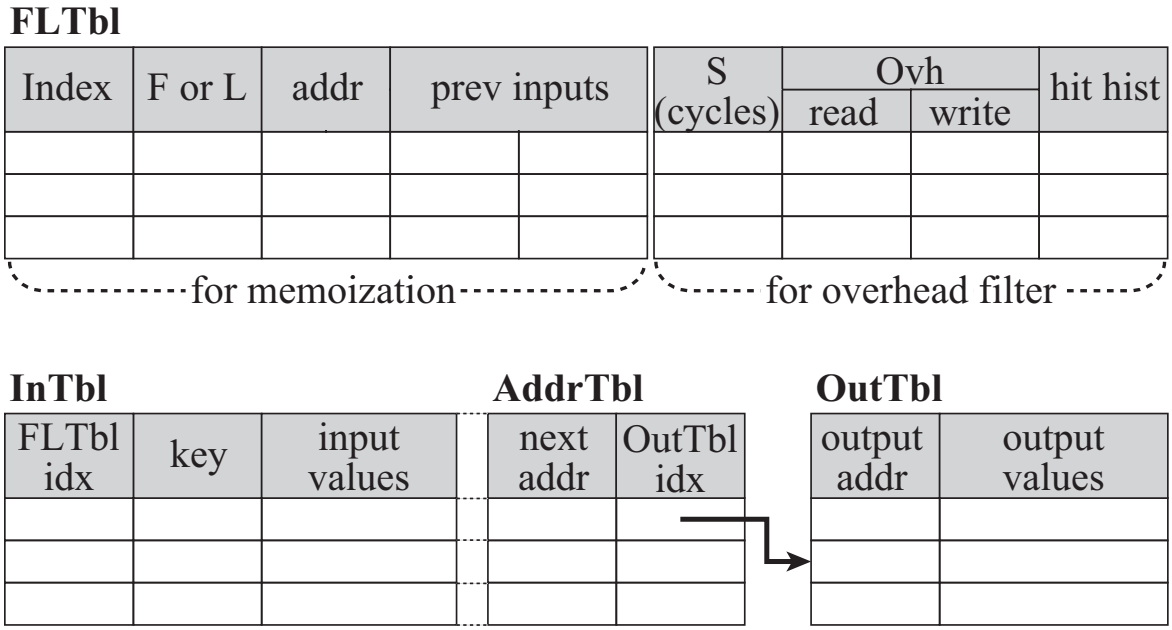

図 4: MemoTbl の構造

および出力書き戻しオーバヘッド（Ovh read/write），過去の再利用ヒット履歴（hit hist）が保持される．

InTbl は，命令区間の入力値を記憶する表である．各行は FLTbl の行番号 Index に対応する FLTbl idx を持ち，どの命令区間の入力値を記憶しているかがこの値により判別される．一般に命令区間内では，複数の入力が順に参照され使用されるが，ある入力の値が異なると同じ命令区間でもその次入力アドレスが変化する場合がある．これは，主記憶アドレス値自体が入力値として用いられる場合や，条件分岐の存在が原因である．つまりある命令区間の入力アドレスの列はその入力値によって分岐してゆくため，全入力パターンはツリー構造で表現できる．そこで，このツリー構造を管理するために，InTbl の各エントリは入力値を記憶する input values フィールドに加えて，当該エントリの親エントリを指す key フィールドを持つ．

AddrTbl は，命令区間の入力アドレスを記憶する表であり，上で述べた次入力アドレスを保持している．AddrTbl は InTbl と同数のエントリを持ち，同じ Index を持つ InTbl エントリの次入力アドレスを，next addr フィールドで記憶している．

OutTbl は，命令区間の出力を記憶する表であり，命令区間の出力列のアドレスおよび値を，output addr/output values フィールドで保持している．また，全ての入力の一致を確認した際に適切な OutTbl エントリを参照できるよう，入力列の末尾を構成する AddrTbl エントリは，対応する OutTbl エントリのエントリ番号を OutTbl idx

フィールドに保持している．

MemoTbl 検索手順の概要は以下の通りである．命令区間を検出すると，その命令区間の開始アドレス，および関数/ループの別の 2 つの情報を用いて FLTbl を検索し，その Index を得る．次に，その Index を FLTbl idx フィールドに持ち，当該命令区間のレジスタ入力を input values フィールドに格納しているエントリを InTbl から検索する．これにより得られたエントリが，これから検索しようとする入力ツリーのルートとなる．

ここでマッチしたエントリと同じ行番号を持つ AddrTbl を参照すると，次に参照すべき入力アドレスが格納されているため，この値をキャッシュから読み出すことで次入力値を得る．そして，この入力値を input values フィールドに持ち，かつ先にマッチした親エントリの番号を key フィールドに持つエントリを，再び InTbl から検索する．これを全ての入力の一致が確認されるまで繰り返し行う．

全ての入力の一致が確認できると，入力セットの終端を保持する InTbl エントリと同じインデクスを持つ AddrTbl エントリが，対応する出力を格納している OutTbl エントリへのポインタを OutTbl idx フィールドに保持しているため，これを用いて OutTbl を参照し，得られた出力のアドレス/値の組を全てレジスタおよびキャッシュに書き戻すことで命令区間の実行が省略される．

なお，InTbl は 連想検索が可能な 3 値 CAM(Content Addressable Memory) で実装することにより高速な入力値検索を実現している．また，MemoTbl の大きさは有限であるため，MemoTbl のエントリが溢れる前に不要なエントリを削除する必要がある．このエントリの削除アルゴリズムには LRU(Least Recently Used) 方式を用いている．

2.3 並列事前実行

マルチコア技術が幅広く普及してきており，複数のコアを利用した様々な高速化手法が研究されている．自動メモ化プロセッサでは，複数のコアを有効に利用する**並列事前実行 (Parallel Early Computation)** という仕組みを備えている．並列事前実行とは，過去の命令区間の入力に基づき，その命令区間を将来実行する際に用いられる入力をストライド予測 [10] を用いて予測し，該当区間をメインコアとは別のコアで予め実行しておくことである．以下，この並列事前実行を行う投機実行コアを **SpC(Speculative Core)** と呼ぶこととする．SpC は予測された入力を用いて得られた出力と，その入力を計算再利用可能な状態で MemoTbl へと登録する．メインコアは自身や SpC によって登録された MemoTbl のエントリを用いて再利用を適用することができる．これに

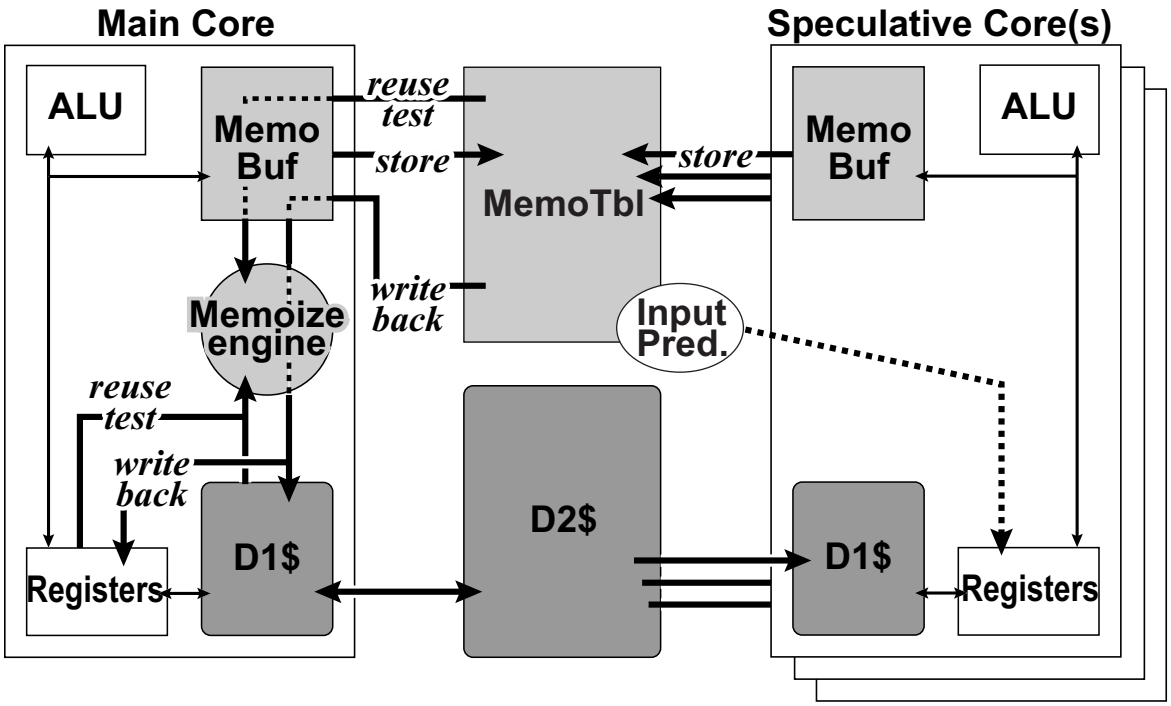

図 5: 並列事前実行機構

より，ループのように入力パラメータが単調に変化する場合など，過去の実行結果を利用しても計算再利用の効果が全く得られない命令区間に対しても，高速化を図ることができる．

この並列事前実行機構を備えた自動メモ化プロセッサのアーキテクチャを**図 5** に示す．各 SpC は，ALU，レジスタ (Registers)，1 次データキャッシュ (D$1)，MemoBuf を独立して持ち，MemoTbl と 2 次データキャッシュ (D$2) および主記憶は全てのコアで共有されるものとする．

このように，各コアが MemoBuf を持っているため，各命令区間の実行終了時にそれぞれのコアは，入出力情報を MemoBuf から MemoTbl へ独立して登録することができる．また，SpC は複数備えることができ，それぞれのコアが予測された入力に対する区間を並列に実行することができる．ここで，入力の予測が外れた場合でも，再利用を行うメインコアからは予測が外れたことが観測されないため，並列事前実行によるオーバヘッドは MemoTbl の検索に要するもの以外は生じない．ただし，エントリ数に制限のある MemoTbl に不必要なエントリが登録されることにより，有効なエントリが削除され，並列事前実行を行わない場合よりも性能が低下してしまう可能性はある．また，SpC が命令を事前に実行する際に，まだ 2 次データキャッシュ上に存在し

ない値をメインコアに先駆けて主記憶から読み出す場合がある．このような場合，並列事前実行はキャッシュプリフェッチ機構と同様の効果をもたらす．この場合，将来メインコアが当該主記憶アドレスの値を読み出す際に，主記憶にアクセスしなくても 2 次データキャッシュに当該アドレスの値が存在していることとなりキャッシュミスを削減できる．

この並列事前実行を適用するためには，過去の再利用エントリの情報から将来の入力を予測して，並列事前実行コアへ渡す必要がある．このため，入力を予測して SpC に渡すための小さなハードウェア (Input Pred) を MemoTbl に設けている．SpC が並列事前実行を開始するためには，まず直近に出現した 2 組の入力の差分に基づいてストライド予測が行われる．そして，予測された入力セットに基づき，SpC はメインコアと並列して当該命令区間の実行を開始する．そのため，図 4 で示したように，FLTbl では各命令区間に対して，ストライド予測に用いるために最近出現した 2 組の入力 (prev inputs) を保持している．

次に，SpC を用いた並列事前実行のタイミングチャートを**図 6** に示す．この例では，SpC が 3 台存在しており，ある命令区間に対してメインコアが入力値 4 で通常実行しているとする．また，それに並行して SpC ではストライド予測を用いて入力値 5,6,7 でそれぞれ実行する．

ここで，図 6 の (a) は最も効率良く再利用が適用できる場合を示している．この場合，メインコアが入力値 4 に対する処理を終え入力値 5,6,7 の実行に移ろうとしたとき，3 台の SpC でのそれぞれの入力に対する処理が完了しており，MemoTbl に入出力情報が既に登録されている．一方，図 6 の (b) は 3 台目の SpC3 で入力値 7 に対する処理が遅延してしまった場合の例である．この場合，キャッシュミスの発生などにより SpC の処理が遅延し，メインコアが SpC と同じ入力値による実行を開始してしまうことで，再利用が適用できなくなっている．

この問題を回避するため，自動メモ化プロセッサでは，SpC における入力値 5,6,7 に対する処理を，メインコアが入力値 4 に対する処理を実行するよりも早めに開始している．つまり，現在メインコアで実行中の入力よりもある程度先まで，SpC への入力割り当てを行っている．

また，メインコアと SpC では 2 次データキャッシュや主記憶が共有されている．このため，これらの共有領域に対して書き込みを行うと，他の SpC やメインコアがプログラムの実行を行う際にデータの不整合が生じてしまう．そこで，SpC では MemoBuf を主記憶書き込みの際のバッファとして扱い．コア間で共有しているデータに対する書

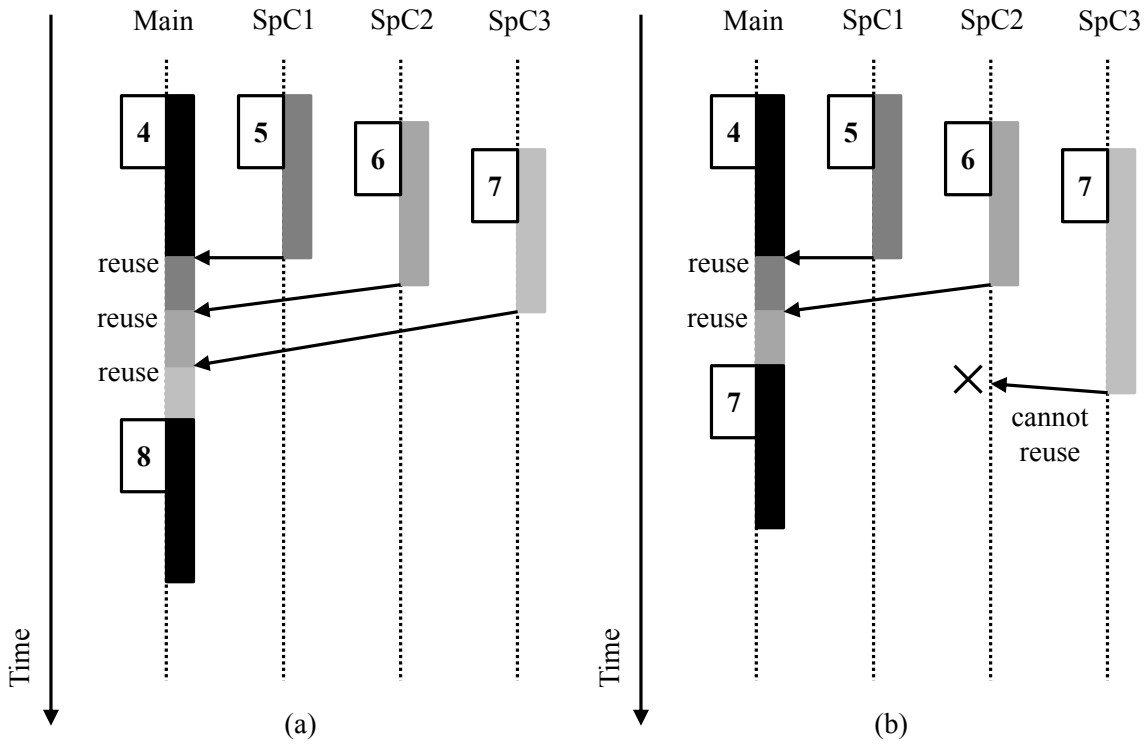

図 6: 並列事前実行の流れ

き込みを行わないようして，このようなデータにおける不整合の発生を回避している．

2.4 オーバヘッド評価機構

自動メモ化プロセッサにおいて，ある命令区間に対して計算再利用を適用するためには回避不可能なオーバヘッドが生じる．まず，再利用適用時の入力値検索の際に，メインコアのレジスタや主記憶の値と MemoTbl に登録されている値とを比較するための検索オーバヘッドがある．この検索オーバヘッドは入力値検索の成功・失敗に関わらず発生する．また，入力値検索が成功した際に，出力を MemoTbl からレジスタやキャッシュへ書き戻すための書き戻しオーバヘッドがある．入力値の検索オーバヘッドと出力の書き戻しオーバヘッドをあわせて**再利用オーバヘッド (Reuse Overhead)** と呼ぶ．

命令区間によっては，再利用オーバヘッドが大きく，計算再利用を行わずに実際に命令を実行した方が早く実行を終えることができる場合も存在する．その場合，計算再利用により性能が悪化するばかりか，必要としない入出力を MemoTbl に登録していることになり，MemoTbl が有効活用されない．そこで，自動メモ化プロセッサでは，

MemoTbl への無駄なアクセスを抑制する再利用オーバヘッド評価機構を備えている．再利用オーバヘッド評価機構を使用して，再利用オーバヘッドと計算再利用により高速化できる実行サイクル数を見積もり，計算再利用による効果が得られると判断した命令区間に対してのみ入力値検索や入出力セットの登録を行う．具体的には，命令区間の再利用により削減できるサイクル数と，その再利用に必要となるオーバヘッドについて概算を行う小さなハードウェアを MemoTbl に付加する．この機構を**オーバヘッドフィルタ (Overhead Filter)** と呼ぶ．

前述の並列事前実行では，SpC による投機実行の対象とする命令区間をいかに選択するかが重要である．そのため，図 4 で示したように，オーバヘッドフィルタ機構は命令区間を記憶する表である FLTbl の拡張として備えられている．FLTbl では，各命令区間に対して一定期間における再利用の状況をシフトレジスタ (図 4 中の hit hist.) を用いて記録し，これを用いてそれぞれの命令区間の再利用適応度を算出している．

ある命令区間について，最近の一定回数 T 回の再利用試行における再利用成功回数 M は上記シフトレジスタから得られる．この値と，当該命令区間の過去の省略サイクル数 S から，実際に削減できたサイクル数を

$$M \cdot (S - Ovh^R - Ovh^W) \tag{1}$$

として計算する．Ovh^R，Ovh^W はそれぞれ，過去の履歴より概算した，当該命令区間の MemoTbl 検索オーバヘッド，および MemoTbl からキャッシュ等への書き戻しオーバヘッドである．なお，S，Ovh^R，Ovh^W は，図 4 に示した FLTbl の S フィールド，および Ovh read/write フィールドからそれぞれ取得される．

また，再利用が適用できなかった場合でも，MemoTbl の検索オーバヘッドは存在する．このオーバヘッドは，

$$(T - M) \cdot Ovh^R \tag{2}$$

として計算できる．

ここで，発生したオーバヘッド (2) よりも，削減できたステップ数 (1) が大きいような命令区間は，再利用の効果が得られると考えられる．式 (1) から式 (2) を引いたものを ***Gain*** とすると，

$$Gain = M \cdot (S - Ovh^W) - T \cdot Ovh^R \tag{3}$$

となり，この $Gain$ が正値であれば再利用の効果があると判断できる．そこで，再利

用表に小さなハードウェアを付加することによってこれを計算し，再利用の効果が得られると判断された命令区間に対してのみ MemoTbl への登録および再利用を行っている．

3　投機実行コアによるスカウトスレッド実行

本章では，本論文で提案する投機実行コアによるスカウトスレッド実行モデルについて説明する．

3.1　スカウトスレッド実行

従来の自動メモ化プロセッサでは，再利用の適用によって却って性能が悪化してしまう可能性がある命令区間に対して，オーバヘッドフィルタ機構が MemoTbl への入出力情報の登録を抑制する．また，そのような性能低下の恐れのある命令区間を SpC による並列事前実行の対象から外すために，値予測に用いられる入力値セット (図 4 中 FLTbl の prev. inputs) の更新も中止している．

この，値予測に用いられる入力値セットは，当該区間の実行を開始する際に初期化され，MemoTbl へエントリを登録する際に，MemoBuf に記憶してきた値で更新される．値予測機構 (図 5 中 Input Pred.) はこの入力値セットを参照し，ストライド予測を用いて将来実行される入力値セットを予測し投機実行コアへと受け渡す．しかし，MemoTbl へのエントリの登録がオーバヘッドフィルタ機構によって中止されると，同時に値予測に用いる入力値セットも値が更新されなくなり初期値のままとなる．これにより，ストライド予測を用いて入力値の差分を取ることができなくなり，値予測は失敗する．値予測に失敗すると SpC は次に値予測が成功するまでの間，投機実行を適用することができない．以下，この SpC が投機実行を行えない状態を，SpC が**遊休状態**であると呼ぶことにする．

本論文では，この遊休状態である SpC においてキャッシュプリフェッチを行う一種のスカウトスレッドを実行させることで，メインコアによるメモリアクセスレイテンシの一部を隠蔽し，高速化を図る手法を提案する．値予測が成功する命令区間では従来モデルと同様に並列事前実行を行う事前実行スレッドを実行させ，値予測が失敗する，すなわち再利用の効果が見込めない命令区間ではスカウトスレッドを実行させる．このように再利用の効果を考慮しつつ，投機実行コアによるスレッド実行を動的に選択するモデルを提案する．

ここで，スカウト (斥候) スレッドとは，Sun Microsystems 社で開発が進められてい

たRockプロセッサ[11]で提案されている高速化手法の，スカウトスレッディングにおいて実行されるスレッドのことである．これは，本来の計算処理を行う通常実行スレッドに先行して必要となるデータをメモリからキャッシュに持ってくるという役目のスレッドで，理想通りに働けばメモリアクセスレイテンシを完全に隠蔽して，実行スレッドの処理時間を短縮することができる．しかし，スカウトスレッドが必要なデータをメモリから先取りしてキャッシュにロードし通常実行スレッドが高速化すると，前を走るスカウトスレッドに通常実行スレッドが追い付いてしまうため，理想的なスカウトスレッドは一般的に存在しないと言われている．

さて，並列事前実行における値予測では，入力アドレス自体がストライド予測の対象となる場合がある．そのため，スカウトスレッド実行にもこの値予測に基づく情報を用いることで，ある命令区間が持つ入力毎にアドレスのストライドが異なる場合にも効果が期待でき，メインコアが将来引き起こす可能性のある主記憶アクセスを，SpCが事前により多く処理することが可能となる．

また，キャッシュミスが削減されることで，限定的ではあるがメインコアにおける再利用率自体も向上する可能性がある．事前実行スレッドはメインコアに先駆けて命令区間を実行するため，一般にキャッシュミスを起こしやすい．そして事前実行スレッドにおいて発生したキャッシュミスにより，その実行結果をMemoTblに登録するタイミングが遅れるため，メインコアによる実行に追い付かれてしまうことで，図6(b)のように入力値予測が正しかった場合でもその結果を再利用できない状況を引き起こすことがある．ここで，再利用効果が見込めないループの内部に，再利用効果が得られるループが存在している場合を考える．このような場合には，提案手法を適用することで外側のループに対してはスカウトスレッドが実行され，内側のループに対しては事前実行スレッドが実行されることになる．上記のようなループの入れ子構造において，外側のループを先駆けてスカウトスレッド実行することによって，内側のループにおけるキャッシュミスの発生確率も低下し，事前実行スレッドの実行結果がメインコアの再利用に間に合わなくなる状況が既存モデルに比べて起りにくくなると考えられ，再利用率が向上する可能性がある．

上記のような入れ子構造を持つループの具体例を**図7**に示す．これは，SPEC CPU95 FPに含まれる107.mgridの部分コードであり，3つの3次元行列 R, V, U から $R = V - AU$ を計算するものである．従来モデルでは，この最内ループ(189行目)は並列事前実行 による再利用効果が得られているが，外側の2つのループ(187,188行目)に関しては入力数が多く再利用オーバヘッドが大きいため再利用対象となっていない．し

```
186 C
187       DO 600  I3=2,N-1
188       DO 600  I2=2,N-1
189       DO 600  I1=2,N-1
190 C R = V-AU
191 600 R(I1,I2,I3)=V(I1,I2,I3)
192      >  -A(0)*(U(I1,   I2,   I3   ))
193      >  -A(1)*(U(I1-1,I2,   I3   ) + U(I1+1,I2,   I3   )
194      >         + U(I1,   I2-1,I3   ) + U(I1,   I2+1,I3   )
195      >         + U(I1,   I2,   I3-1) + U(I1,   I2,   I3+1))
196      >  -A(2)*(U(I1-1,I2-1,I3   ) + U(I1+1,I2-1,I3   )
197      >         + U(I1-1,I2+1,I3   ) + U(I1+1,I2+1,I3   )
198      >         + U(I1,   I2-1,I3-1) + U(I1,   I2+1,I3-1)
199      >         + U(I1,   I2-1,I3+1) + U(I1,   I2+1,I3+1)
200      >         + U(I1-1,I2,   I3-1) + U(I1-1,I2,   I3+1)
201      >         + U(I1+1,I2,   I3-1) + U(I1+1,I2,   I3+1))
202      >  -A(3)*(U(I1-1,I2-1,I3-1) + U(I1+1,I2-1,I3-1)
203      >         + U(I1-1,I2+1,I3-1) + U(I1+1,I2+1,I3-1)
204      >         + U(I1-1,I2-1,I3+1) + U(I1+1,I2-1,I3+1)
205      >         + U(I1-1,I2+1,I3+1) + U(I1+1,I2+1,I3+1))
206 C
```

図 7: 107.mgrid のプログラムコード (部分)

かし，外側のループを最内ループの事前実行に先駆けてスカウトスレッド実行することにより，最内ループの直近のイタレーション，すなわち複数 SpC によって最初に事前実行される数回分のイタレーションに関してキャッシュミスが削減され，その実行結果の MemoTbl への登録タイミングが早まることで，計算再利用を適用できる場面が増加すると考えられる．

3.2 動作モデル

再利用の効果が見込めない区間である場合には，SpC に事前実行スレッドではなく，当該区間内に存在するロード命令を中心とした一部の命令のみを発行する一種のスカウトスレッドを実行させる．遊休状態である SpC にスカウトスレッドを実行させるためには，まず従来と同様に先行区間を投機実行するための入力値を得る必要がある．投機実行に必要な入力値は値予測が成功したときに取得できる．つまり，値予測の成功

率が上昇することで，SpC は投機実行を適用できる機会が増加することになる．先ほど述べたように，値予測が成功するためには値予測に用いる入力値セットが FLTbl に書き込まれていなければならない．そこで，オーバヘッドフィルタ機構が当該区間に対して再利用効果が見込めないと判断し再利用表への登録を中止しようとした場合でも，値予測に用いる入力値セットへの書き込みだけは中止せず，MemoBuf に記憶した値を用いて更新するようにする．これにより，SpC が投機実行を適用できる回数が増える．以下，説明のために，オーバヘッドフィルタ機構が再利用表への登録を中止する場合，つまり再利用効果が見込めない ($Gain$ が負値である) 場合のことを**フィルタの判定が真である**と呼ぶ．

上述したように，SpC による投機実行の機会が増えることにより，従来モデルと同様に事前実行スレッドを SpC に実行させるだけで，ある程度のデータプリフェッチの効果が見込める．しかし，この場合の投機実行は，フィルタの判定が真である時の情報をもとに行われるため，その実行結果が再利用できた場合にも性能の悪化に繋がる可能性が高い．よって，その実行結果は MemoTbl に登録する必要がなく，その結果を得るために事前実行する必要もないと考えられる．そこで，事前実行スレッドではなくスカウトスレッドを実行させることで，有限サイズの表である MemoTbl に不必要な再利用エントリが登録されることを避け，計算再利用の効果を阻害することなくキャッシュプリフェッチの効率を向上させることができる．

スカウトスレッドの実行対象となるのは，再利用効果が見込めない再利用対象区間である．そのため，このスカウトスレッドを実行する期間は比較的短いと考えられ，僅かな電力消費量の増加によって高速化が見込める．また，ロード命令によって取得されるデータは実行に用いることはないため，ローカルな 1 次キャッシュおよびレジスタに値を格納する必要はない．そのため，スカウトスレッドを実行する際に消費電力が増大するユニットはクロックと 2 次キャッシュだけである．また，ロード命令のみを発行することで ALU および MemoBuf を動作させる必要はなくなる．自動メモ化プロセッサにおいて，各ユニットのエネルギー消費量は既に知られており [12]，自動メモ化プロセッサ全体のエネルギー消費量に対して，クロックのエネルギー消費量の割合は約 26%であり，2 次キャッシュは約 15%である．よって，スカウトスレッドの実行によって増加するエネルギー消費量の割合は，

$$(0.26 + 0.15) \cdot (ST/Total) \cdot SpCs \tag{4}$$

として概ね計算できる．なお，ST，$Total$，$SpCs$ はそれぞれ，SpC におけるスカウト

スレッドの実行サイクル数，総実行サイクル数，SpC のコア数である．ここで，スカウトスレッドを実行できるのは，再利用対象区間の中で再利用の効果が見込めない命令区間であり，その命令区間においてロード命令のみを発行するため，ST は $Total$ に比べて小さい値となる．そのため，提案手法により増加するエネルギー消費量は少ないと考えられる．また，提案手法により 2 次キャッシュミスを削減し高速化が達成できれば，クロックと 2 次キャッシュのエネルギー消費量を抑えることができると考えられる．したがって，式 (4) は消費エネルギー量の最大値を表しており，実際に必要なエネルギー消費量はこれより少ないと予測される．

3.3 入力値セットの更新とスレッド役割の切り替え

前節までで述べた動作を実現するための，並列事前実行機構の実装モデルについて説明する．このモデルでは，まず SpC による投機実行の機会を増やすために，フィルタの判定が真である場合にも，MemoBuf に記憶してある当該区間の値を FLTbl 中の値予測に用いる入力値セットに書き込み，その値を更新するようにする．

次に，値予測が成功したことで投機実行することになった命令区間に対して，SpC に事前実行スレッドまたはスカウトスレッドのどちらを実行させるか判定する必要がある．この判定は，値予測に用いる入力値セットに値を書き込んだ際のオーバヘッドフィルタ機構の判定に依存する．つまり，フィルタの判定が真である場合にはスカウトスレッドを，フィルタの判定が偽である場合には従来通りに事前実行スレッドを実行させる．そこで，命令区間を管理する FLTbl の各行に新たに 1 ビットフラグのフィールドを追加し，オーバヘッドフィルタの判定を記憶する．ここでは，フィルタの判定が真である場合にそのフラグを 1 にセットする．すなわち，MemoTbl への登録時におけるオーバヘッドフィルタ機構の判定つまり $Gain$ の値に応じてフラグを操作する．値予測に成功し投機実行を開始する際に，当該区間のフラグを参照することで，いずれのスレッドを実行するかを動的に選択することが可能になる．

ここで，このスレッド切り替えのフローチャートを**図 8** に示す．ストライド予測を用いた値予測に失敗した場合は次に値予測が成功するまでの間，投機実行を適用することができない．したがって，提案手法を適用した場合でも SpC が遊休状態となる場合はまだ存在している．一方で，値予測に成功した場合は，オーバヘッドフィルタ機構の判定に応じて実行するスレッドの役割を切り替える．この時，FLTbl に追加したフィルタの判定を記憶するフラグが参照され，直前に行われた入力値セットの更新時におけるフィルタの判定，すなわち $Gain$ の値が正値か負値かにより事前実行スレッド

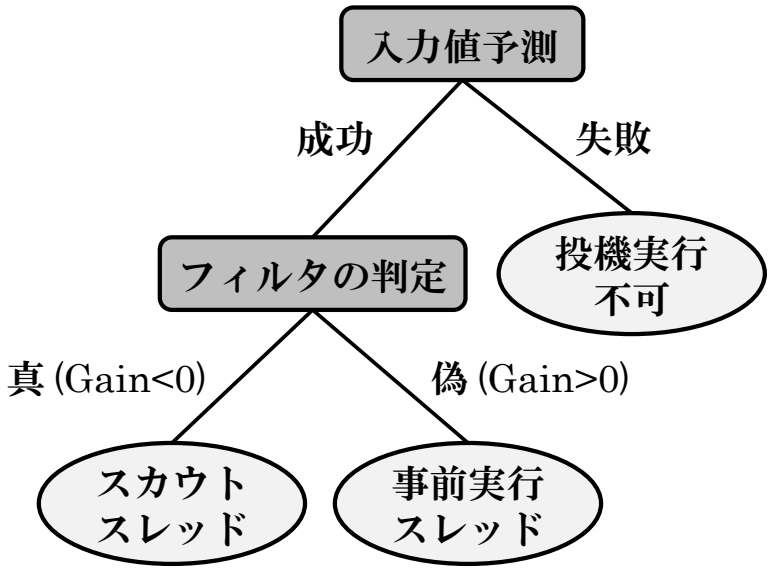

図 8: スレッド役割の切り替え

とスカウトスレッドを動的に切り替えて実行させる．

なお，事前実行スレッドとスカウトスレッドではその処理内容が異なる．事前実行スレッドでは計算再利用のために当該区間の入出力を MemoBuf に記憶し，実行終了時に MemoTbl に一括して登録しなければならない．そのため割り当てられた入力値セットを用いて当該区間の命令を発行する必要がある．一方，スカウトスレッドは演算等の命令を処理する必要はなくデータプリフェッチのみを行えば良い．そのため，SpC は投機実行している間，割り当てられたスレッドの役割に応じて処理内容を変えなければならない．また，スレッドの実行が開始された後は，割り当てられた命令区間の実行が終了するまでスレッドの役割を変えてはならない．しかし，フィルタの判定を記憶する FLTbl のフラグは，SpC が投機実行中であっても，その値がメインコアによって書き換えられる可能性がある．そのため，この値を投機実行開始時におけるスレッド割り当ての判別に用いることは可能であるが，各 SpC が実行すべき処理の管理に用いることはできない．そこで，SpC が現在，事前実行スレッドまたはスカウトスレッドのどちらを実行しているかを判断するために，各 SpC にスレッド判別用の 1 ビットフラグを新たに追加する．そして，投機実行開始時にこのフラグに対して，フィルタの判定を記憶しているフラグの値をセットする．これにより，投機実行中の各スレッドの動作を管理する．

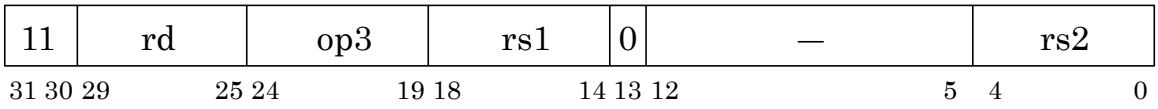

図 9: SPARC における命令セットのフォーマット (上位 2 ビットが ‘11’)

表 1: ロード命令群のオペコード部

Opcode	op3	Operation
LDUB	000001	Load Unsigned Byte
LDUH	000010	Load Unsigned Halfword
LDD	000011	Load Double Word
LDSB	001001	Load Signed Byte
LDSH	001010	Load Signed Halfword
LDF	100000	Load Floating-Point Register
LDDF	100011	Load Double Floating-Point Register
LDA	110000	Load Word from Alternate space

3.4 命令判別と区間終了判定

SpC にスカウトスレッドを実行させる場合には，その命令区間に存在するロード命令のみを実行するために，命令の種類を判別する必要がある．ここで，2.2 節で述べたように，自動メモ化プロセッサは SPARC V8 をベースアーキテクチャとしている．そのため，本プロセッサは単純な 32 ビット固定長命令を採っており，命令を判別するためのオペコードは 8 ビット固定長となっている．SPARC 命令セット [13] の仕様に基づくと，ロード命令群は命令の上位 2 ビットが ‘11’ である命令群に含まれている．この命令群に含まれる命令のフォーマットを**図 9** に示す．なお，図中の rd はディスティネーションレジスタ，また rs1，rs2 はソースレジスタの番号を表している．これらの命令群にはロード命令およびストア命令が含まれているが，オペコード部の下位 3 ビット目が ‘0’ であるか否かで，ロード命令群を判別可能である．基本的なロード命令の上位 2 ビットを除く残りのオペコード部を**表 1** に示す．このように，下位 3 ビット目は全て ‘0’ で統一されている．なお，ストア命令群では同ビットは 1 となっている．

投機実行区間に対してプログラムカウンタを順次動かしていき，命令判別を適用しロード命令のみを発行することでスカウトスレッド実行を実現する．ここで，当該区間が関数であった場合には，区間開始時に save 命令が，また区間終了時に restore 命

```
                :
1c1d4:  sethi  %hi(0x1d000), %o1
1c1d8:  sll  %l1, 2, %l0
1c1dc:  inc  %l1
1c1e0:  ld  [ %o1 ], %f3
1c1e4:  fdivs  %f3, %f4, %f2
1c1e8:  fstod  %f2, %f2
1c1ec:  fadds  %f4, %f4, %f4
1c1f0:  std  %f2, [ %fp + -8 ]
1c1f4:  ldd  [ %fp + -8 ], %o2
1c1f8:  mov  %o3, %o1
1c1fc:  st  %f4, [ %fp + -116 ]
1c200:  call  1c05c
1c204:  mov  %o2, %o0
1c208:  cmp  %l1, 0x19
1c20c:  sethi  %hi(0x1d000), %o3
1c210:  fadds  %f0, %f0, %f0
1c214:  ld  [ %o3 + 8 ], %f2
1c218:  fdivs  %f2, %f0, %f0
1c21c:  st  %f0, [ %l2 + %l0 ]
1c220:  ble  1c1d4
                :
```

図 10: サンプルプログラムコード

令が必ず存在している．SPARC アーキテクチャでは，レジスタウインドウ方式を採用しており，save および restore 命令によってこのレジスタウインドウを操作する．関数における入出力の受け渡しはこのレジスタウインドウを介して行われるため，SPARC V8 の仕様に準拠するために例外としてこの 2 つの命令も発行することにしている．

SpC が事前実行スレッドを実行する際，当該区間で新たに出現した関数やループはその入れ子構造を記憶し，内部を新たな対象区間として多重的に事前実行を適用している．例えば，**図 10** 中のループでは，アドレス $1c200$ で出現した call 命令により $1c05c$ にジャンプし，その関数に対しても投機実行を行う．一方で，スカウトスレッドを実行する場合には，当該区間で新たに出現した関数やループの入れ子構造を無視し，その内部は適用区間の対象外とする必要がある．この例ではループ内で関数呼び出しが行われることになるが，その関数の入力は区間開始地点 $1c1d4$ から $1c200$ の call 命令

までの命令を実行しなければ求めることができない．そのため，スカウトスレッド実行時に入れ子構造を保持するためには，当該区間の全ての命令を実行する必要がある．これは，ロード命令のみを発行する場合に比べコストが大きくなってしまう．よって，提案手法ではSpCにスカウトスレッドを実行させる場合には，投機実行開始時に定められた命令区間のみを実行対象としている．

命令区間の終了は，その命令区間が関数であればrestore命令の出現により検出できる．一方で，ループの場合には後方分岐命令が命令区間の終端となるが，多重ループなどでは命令区間内に分岐命令が複数存在してしまうことになる．従来手法では，後方分岐命令を実行する際に当該ループの終端であるかの判定を行っている．つまり，後方分岐命令を実行して初めて命令区間終了判定が行われるため，基本的にロード命令のみを発行するスカウトスレッド実行では，当該ループ区間の終了を検知することができない．そこで，提案手法では，命令判別の際にFLTblに登録されている命令区間の終端アドレスと現在のプログラムカウンタを比較することで，命令区間終了判定を行うことにする．このアドレスの値が一致していた場合にスカウトスレッドの実行を終了し，次の値予測を行う準備をする．これにより，関数およびループの該当区間でスカウトスレッドの実行を適用できる．

4　評価

以上で述べた拡張を既存の自動メモ化プロセッサに対して追加実装し，サイクルベースシミュレーションにより評価した．

4.1　評価環境

シミュレータは単命令発行のSPARC V8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．なお，キャッシュアクセスレイテンシや命令レイテンシはSPARC64-III[14]を，MemoTbl内のInTblに用いるCAMの構成はMOSAID社のDC182888[15]を，それぞれ参考にしている．また，評価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SPEC CPU95を用いた．

また，入力値検索のコストとして，レジスタとCAMを32バイト比較するのに9サイクル，メモリとCAMを32バイト比較するのに10サイクルを要するものとした．現在の一般的なアーキテクチャでは，CPUコア内部のクロック速度は，外部のメモリパスのクロック速度の10倍程度で動作している．そのため，このシミュレーションで設定した，レジスタやメモリと，CPUコア外部にあるMemoTblとの比較に要するサイ

表 2: シミュレータ諸元

MemoBuf	64 kBytes
MemoTbl CAM	128 kBytes
Comparison (register and CAM)	9 cycles/32Bytes
Comparison (Cache and CAM)	10 cycles/32Bytes
Write back (MemoTbl to Reg./Cache)	1 cycle/32Bytes
D1 cache	32 KBytes
line size	32 Bytes
ways	4 ways
latency	2 cycles
D2 cache	2 MBytes
line size	32 Bytes
ways	4 ways
latency	10 cycles
Memory	160 MBytes
latency	100 cycles
Register windows	4 sets
miss penalty	20 cycles/set

表 3: 削減サイクル数率 (SPEC CPU95 FP)

	Mean	Max
(P) 既存モデル	15.0%	40.6% (107.mgrid)
(S) 提案モデル	19.1%	41.3% (107.mgrid)

クル数は現実的な値となっている.

4.2 SPEC CPU95 FP

SPEC CPU95 FP (train) の 10 のプログラムを gcc-3.0.2 (-msupersparc -O2) によりコンパイルし，スタティックリンクにより生成したロードモジュールを用いて評価を行った．評価結果を**図 11** および**表 3** に示す.

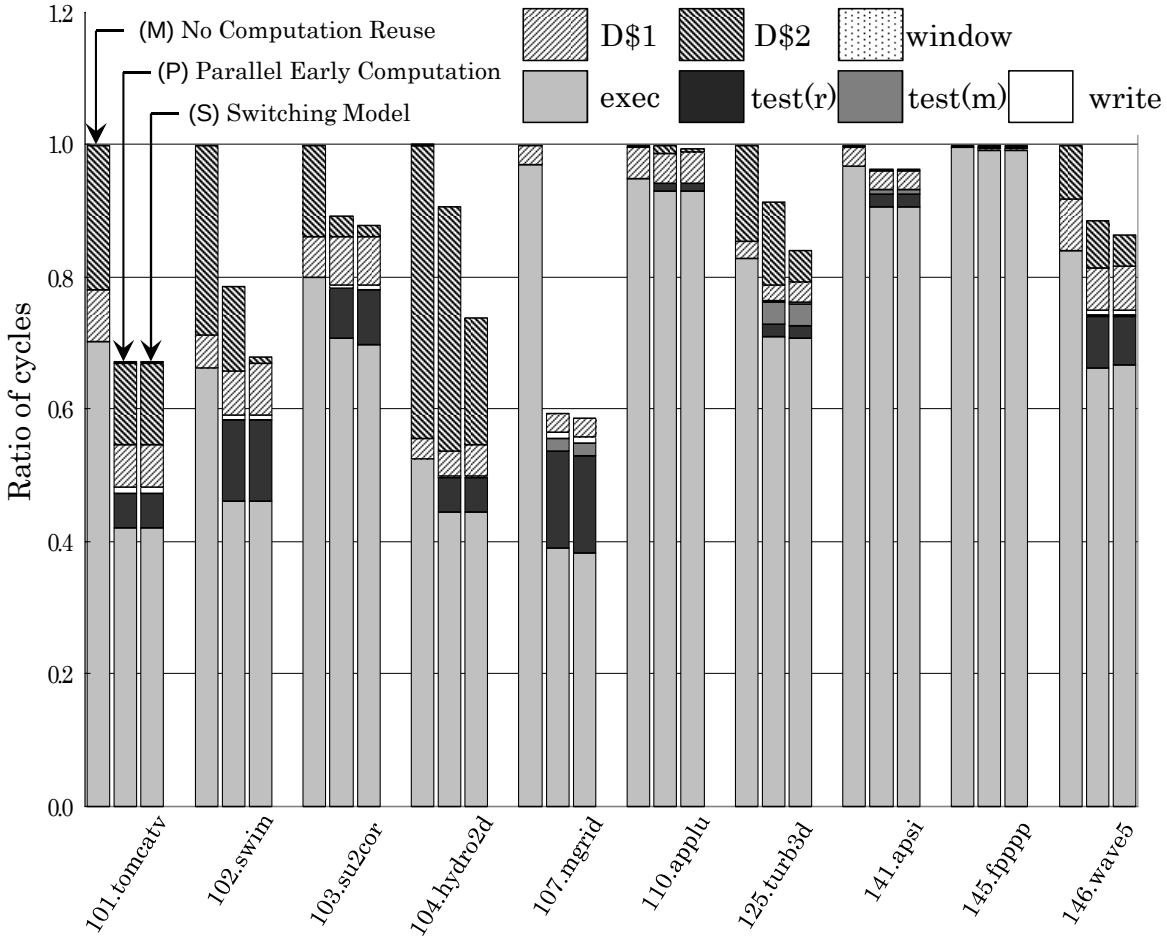

図 11: 実行サイクル数比（SPEC CPU95 FP）

図 11 中の凡例はサイクル数の内訳を示しており，exec は命令サイクル数，test(r), test(m) はそれぞれレジスタ/キャッシュと InTbl(CAM) との一致比較オーバヘッド，write は再利用成功時に発生する結果の書き戻しオーバヘッド，D$1, D$2, window はそれぞれ 1 次/2 次キャシュミスペナルティとレジスタウィンドウミスペナルティである．

評価は，再利用を行わないモデル，従来の並列事前実行モデル，提案モデルについて行った．なお，全てのモデルはメインコア 1 つに加え，SpC 3 つの合計 4 コア構成とした．図 11 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

(M) メモ化を行わないモデル

(P) 並列事前実行の従来モデル

(S) スカウトスレッドを実行する提案モデル

が要したサイクル数を表している．なお，各サイクル数は (M) を 1 とする正規化を行っ

表 4: 2 次キャッシュミスペナルティサイクルの削減比

	(P) 既存モデル	(S) 提案モデル
102.swim	55.1%	96.9%
104.hydro2d	16.7%	56.4%
125.turb3d	12.4%	67.1%
146.wave5	13.0%	42.6%
4 プログラム平均	24.3%	65.8%

ている．

提案手法 (S) は，概ね良い結果を示している．まず，102.swim，104.hydro2d，125.turb3d，146.wave5 では D$1 が増大し，D$2 が減少する傾向が見受けられる．つまり，従来では 2 次キャッシュミスによりメモリまでデータを取得しに行っていたものが，SpC がスカウトスレッドを実行したことにより，メインコアが当該区間を実行するときにはすでに共有の 2 次キャッシュにデータが格納されていたことがわかる．これら 4 つのプログラムにおける 2 次キャッシュミスペナルティサイクル削減率を**表 4** に示す．この結果から，提案手法によりメモリへのアクセスレイテンシが大幅に隠蔽できていることが分かる．SPEC95 FP 全体でも，2 次キャッシュミスの削減サイクル数は，従来モデルでは平均 31.9%であったのに対し，提案モデルでは平均 64.0%と大幅に改善された．またこれらのプログラムでは exec，test(r)，test(m)，write については既存手法である (P) から変化が見られない．このことから，スカウトスレッドの実行は従来の並列事前実行による計算再利用に対する効果を阻害することなく投機実行コアをより効率的に動作させることができていることが分かる．

図 11 より，総実行サイクル数で比較した場合も全体として性能向上していることが分かる．削減サイクル数率は，従来モデルの削減サイクル数が最大 40.6%，平均 15.0% であったのに対し，提案モデルでは最大 41.3%，平均 19.1% に，それぞれ改善された．

4.3 考察

107.mgrid，141.apsi，145.fpppp では，従来のモデルでも 2 次キャッシュミスがほとんど発生していないこともあり，提案手法の効果はあまり確認できなかった．しかし，これらのプログラムで再利用成功率を測定したところ，107.mgrid において約 1.9% 向上していることが確認できた．3.1 節で述べたように，107.mgrid には再利用区間の 1

つに 3 重ループが存在している．既存モデルでは，その最内ループの再利用率が大きく，外側 2 つのループは全く再利用できていないという傾向がある．そのため，外側のループに対してスカウトスレッドが実行され，最内ループにおけるキャッシュミスが削減されたことによって，再利用率を向上することができた．

103.su2cor および 146.wave5 でも再利用率に変化があり，r_step のサイクル数が変化しているが，再利用オーバヘッドを加味したサイクル数 (r_step，test(r)，test(m)，write の総和) ではほとんど変化が見られなかった．146.wave5 では r_step のサイクル数が増加し，再利用率が低下しているように見えるが，これは実行サイクル数が再利用オーバヘッドを僅かに上回っていた命令区間が，提案手法の適用により逆転してしまったものと考えられる．キャッシュミスが削減されたことで命令区間の実行サイクル数が減り，再利用を適用すると却って性能が悪化することになる．しかし，この場合には実行サイクル数と再利用オーバヘッドがほぼ等しくなるため，実行速度にはほとんど影響しない．そのため，実行サイクル数と再利用オーバヘッドの総和が変化せず，計算再利用の効果を大きく阻害しないことが確認できた．

101.tomcatv ではごく僅かながら D$1 が減少し，D$2 が増大しており，予測した効果と逆の結果となってしまっている．並列事前実行機構では，SpC の実行がキャッシュミスなどによって遅延してしまい，メインコアが再利用を適用する時に間に合わなかった場合を考慮して，現在メインコアが実行中の区間よりもある程度先まで入力値割り当てを行っている．スカウトスレッドを実行する場合は，入れ子構造を保持せず割り当てられた区間のみを投機実行の対象とするため，事前実行スレッドを実行する場合に比べて区間の長さが短い．そのため，スカウトスレッドは事前実行スレッドに比べて頻繁に実行されることになる．その際，より遠くまで入力値割り当てが行われるため，スカウトスレッドはメインコアが実行している地点より遠い区間を実行することになり，キャッシュラインの入れ替えが頻繁に起こることになる．101.tomcatv では，他のプログラムと比べてスカウトスレッドを実行する割合が大きかったため，必要なデータがメインコアの実行時点にはすでに 2 次キャッシュから追い出されてしまったと考えられる．

ここで，スカウトスレッドを実行することによる消費エネルギー増加について考察する．まず，既存モデル (P) において，単一 SpC の動作時間は総実行サイクル数の約 23%であった．提案モデル (S) ではこれに加え，スカウトスレッドを実行するサイクル数が総実行サイクル数の約 26%ほど増加している．ただし，スカウトスレッドを実行している間は，SpC 内の全ユニットを動作させる必要はなく，クロックとプリフェッ

チに必要なユニットのみで良い．そのため，電力消費量はさほど増大しないと考えられる．自動メモ化プロセッサにおける各ユニットのエネルギー消費量は 3.2 節で述べたように既に調査されており，これに基づきエネルギー消費量を概算したところ，プロセッサ全体で既存モデルに対し約 13%の増加で抑えられることがわかった．

さて，本論文ではここまで計算再利用技術に基づく自動メモ化プロセッサの並列事前実行機構に対し，遊休状態である投機実行コアにキャッシュプリフェッチを行う一種のスカウトスレッドを実行させることで，従来の再利用の効果を阻害することなくメモリアクセスレイテンシの一部を隠蔽した．また，一部のプログラムでは計算再利用率が向上することを確認した．このように，提案手法の適用によりマルチコアの資源をより有効に利用して高速化することができるという結果が得られた．しかし，自動メモ化プロセッサは比較的単純なアーキテクチャを想定して評価しており，現在主流となっているマルチコアプロセッサのような複雑な環境を想定してシミュレートすることが今後の課題に挙げられる．そこで，次の章ではマルチコアプロセッサ研究における現在の情勢について調査していく．

5　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既存・関連研究について述べる．

5.1　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[16] や IBM 社の Cell/B.E.[17]，Sun Microsystems 社の UltraSPARC T2[18] 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[19] 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．この流れを受けて，本論文では前章までで高速化手法の一つとして自動メモ化プロセッサの拡張を述べた．

今後は半導体のプロセスルールの縮小に伴い，単一プロセッサ当たりに搭載される

コア数がさらに増大していくと予想される．コア数は 10，100，1000 と規模を拡大していき，近い将来にはメニーコアプロセッサが一般化すると考えられている．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．

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が，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顕在化すると言われているため，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てプロセッサ構成の検討が重要になっている．

5.2 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

とつは，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し直さざるを得なくなってしまう．また，ハードウェア上で動作をシミュレートするため，構成する回路の設計が確定していない場合には動作させることができない．そのため，アーキテクチャの研究用途としては設計や構成方式の検討において柔軟性に欠ける部分がある．

もうひとつは，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挙げられる．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，汎用コンピュータで稼働させることができる．このシミュレータでは，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．

5.3 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[20] が提案・開発されている．FPGA 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．FPGA シミュレータの一種である 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，ノード数に対する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としては，HP 研究所が開発している COTSon[21] 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している．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システムの帯域幅と性能の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[22] 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 をベースとした NIRGAM[23] 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[24] 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．

このように，搭載されるコア数の増大に伴い，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．メニーコアプロセッサにおいて，

複数コアを単一の主記憶装置へ接続することは，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．そのため，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なく，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．しかし，上で述べた既存のシミュレータは共有メモリを前提として構築されているため，単純に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が難しい．

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，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，各方式の実現可能性の検証と構成方式の組み合わせによる様々な構成パターンの構築を実現するために，ソフトウェアによるシミュレーションを採用する．

6　メニーコアプロセッサ構成方式の実現可能性検証

本章では，本論文の提案となるメニーコアシミュレータの開発の概要およびメニーコアプロセッサ構成方式の検討について述べる．

6.1　研究概要

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．まず最初に，メニーコアシミュレータを開発することが挙げられる．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．また，実行トレースの採取に目的を絞ることで，高速なシミュレータを実現できる．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．

次に，メニーコアプロセッサ構成方式の検証が挙げられる．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形状における各構成方式を構築する．また，メニーコアトレースシ

ミュレータの実現のために構築した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構築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，高並列実行を実現する最適なアーキテクチャの模索が挙げられる．これまでに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．そうした知見を得るために，4 章までに高速化手法を提案してきた．このような高速化技術を組み込むことを視野に入れ，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．この開発におけるデータ供給方式に関しては，コア数の増大に伴うキャッシュ構成の関係性と，キャッシュとメモリ間の一貫性を保持するキャッシュコヒーレンシプロトコルについて検討する．一方で，相互結合網の形状に関しては，複数コアやメモリ間での交信路の構成の種類について検討する．そして，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．

6.2　データ供給方式の検討

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において，コア数の増大とキャッシュ構成の関係性およびキャッシュコヒーレンシプロトコルについて概説する．

6.2.1　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．コア数の増大に伴うキャッシュの構成の様子を**図 12** に示す．

図 12 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

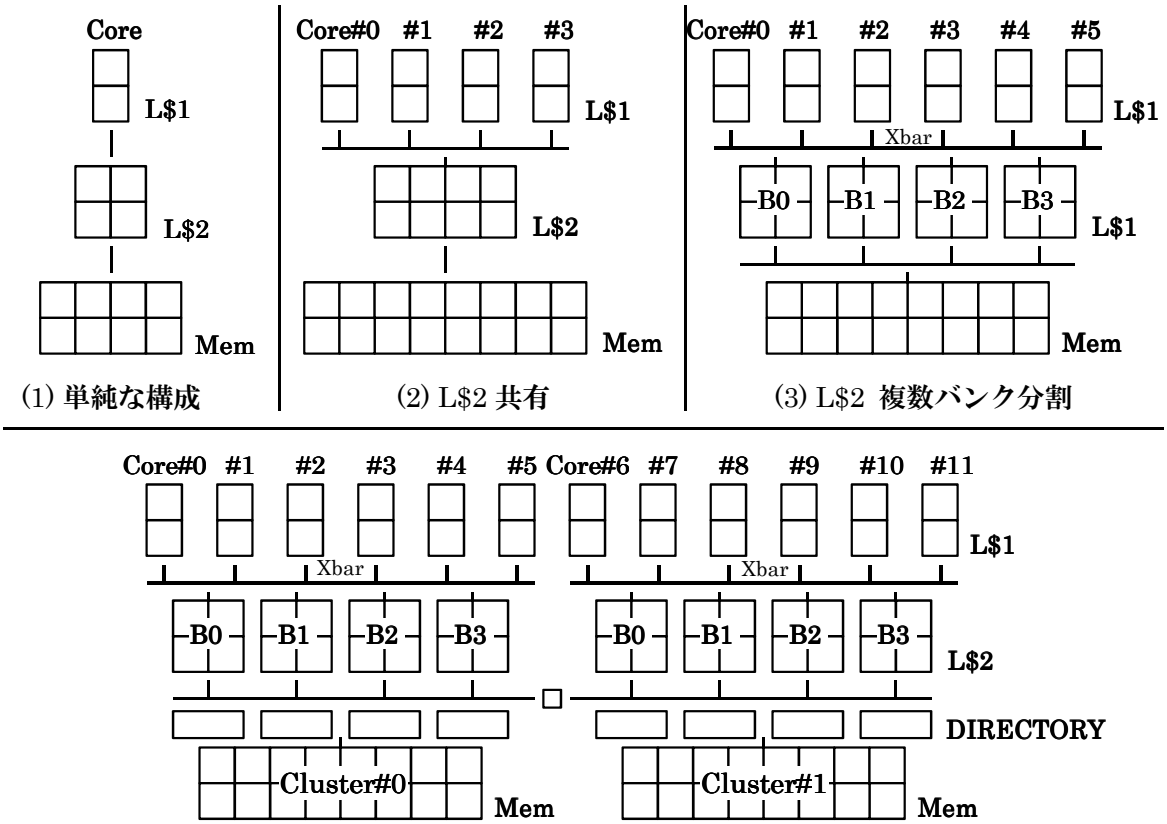

図 12: コア数の増大に伴うキャッシュの構成の様子

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 12 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．このモデルと同様に，自動メモ化プロセッサでは 4 つのコアで 1 つの 2 次キャッシュを共有する単純な構成をとっていた．自動メモ化プロセッサは，並列実行を目的とした一般的なマルチコアプロセッサと異なり，メインコアの動作をサポートするコアを備えているだけである．そのため，キャッシュシステムを複雑化する必要はなく単純な構成を採ることができていた．

一方で，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処理性能が低下してしまう．そこで，図 12 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などの問題により，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，図 12 中の (3) で示す構成を 1 つのクラスタとし，図 12 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とする傾向がある．また，メニーコアプロセッサなどの大規模なシステムでは，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管理機構がよく用いられている．スヌーピングは各コア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，コア数が増えると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．

6.2.2 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，メモリトラフィックの衝突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システムごとに異なる．この中で最も基本となる 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(無効)，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(共有)，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(変更) 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 [25] が提案されている．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(排他) 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(共有) 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 化要求を行わず書き込むことができる．

また，MSI プロトコルでは，Modified 状態のキャッシュラインのアドレスに他のキャッシュから読み出し要求が到達すると，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．MSI プロトコルの 3 つの状態に，書き戻し責任を負う Owned(所有) の状態を追加して，書き戻しの頻度を減少させる．そして，Modified 状態のキャッシュラインに対して読み出し要求が到達したときには，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．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．

6.3 結合網形状の検討

次に，相互結合網を構成するコア間結合網のトポロジについて概説する．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 つの観点が重要になる．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形式の代表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形式の代表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システムでも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検討する．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，検討対象としてはそれらを組み合わせた 4 種類の結合網形状が考えられる．し

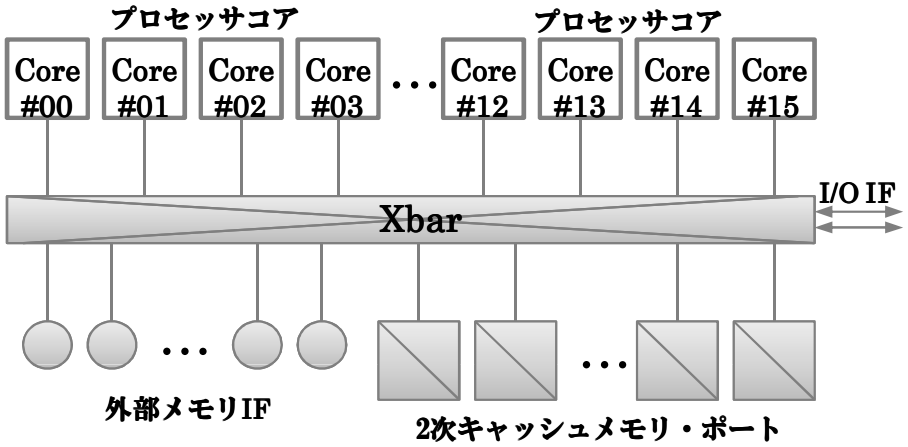

図 13: フラット型クロスバ結合網モデル

かし，フラット型のリング結合網では，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．そのため，フラット型リング結合網を除く 3 種類の結合網形状に関してのみ検討する．

まず，フラット型クロスバ結合網モデルを**図 13** 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 インタフェース用などの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．

次に，階層型クロスバ結合網モデルを**図 14** 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 15** 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．

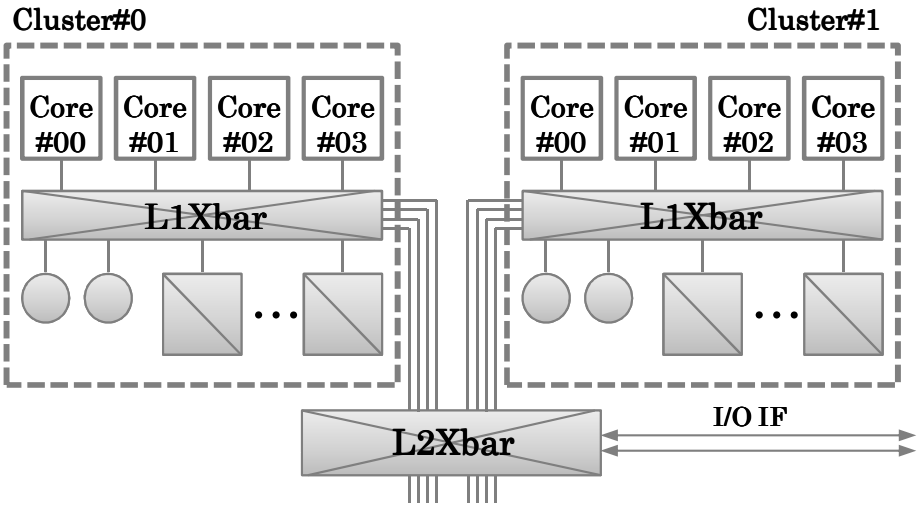

図 14: 階層型クロスバ結合網モデル

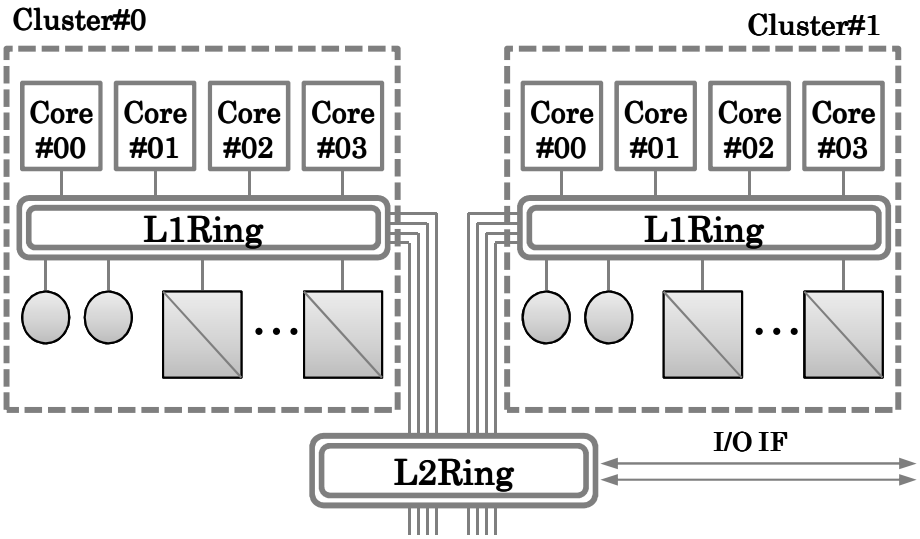

図 15: 階層型リング結合網モデル

7　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．

7.1 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．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．また，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．ここで，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，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 16** 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 アーキテクチャとなっている．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，キャッシュコヒーレンシプロトコルにおける現在の状態をキャッシュラインごとに保持している．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，同一アドレスのデータに対して，クラスタ内に属するコアが持つ 1 次キャッシュ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

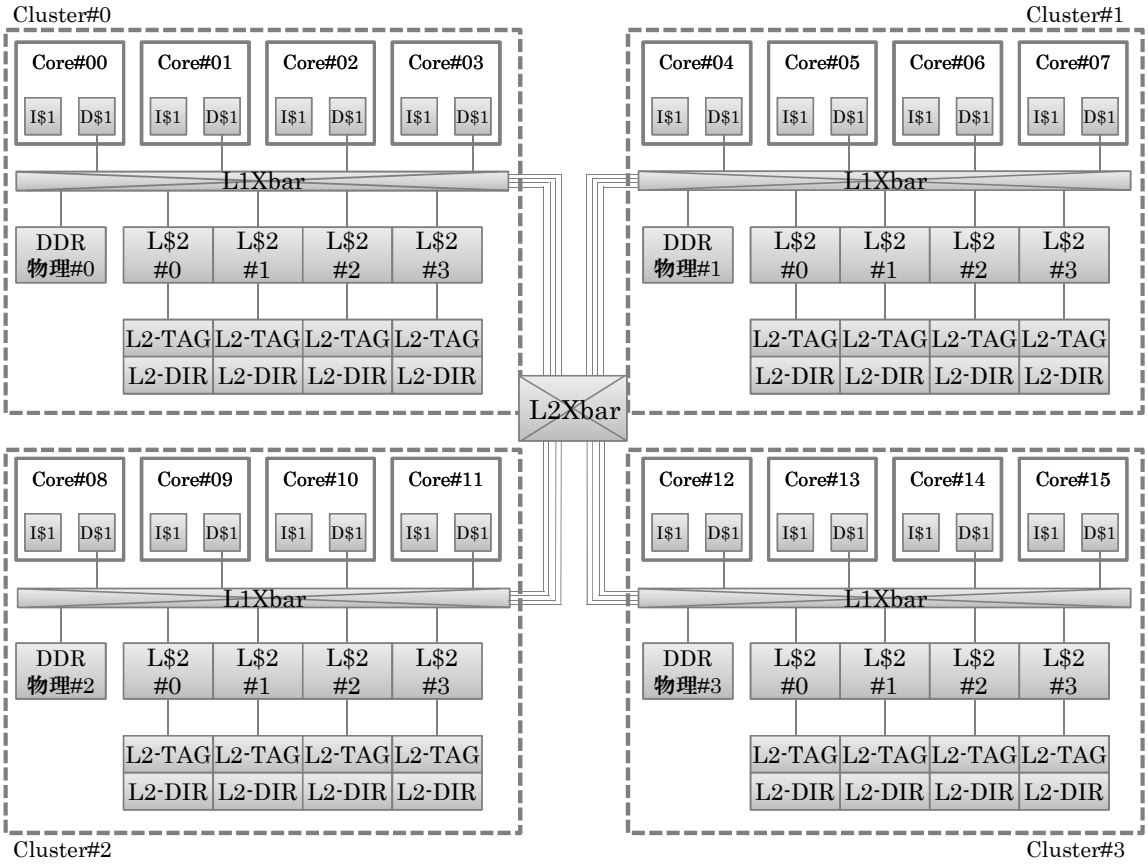

図 16: メニーコアトレースシミュレータのアーキテクチャ構成

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に対して，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．

7.2 動作モデル

次に，メニーコアトレースシミュレータにおける動作を説明する．

7.2.1 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シプロトコルが働き，各コアおよび各クラスタ間でメモリトラフィックが流れる．メモリトラフィックの流れる様子を**図 17** 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

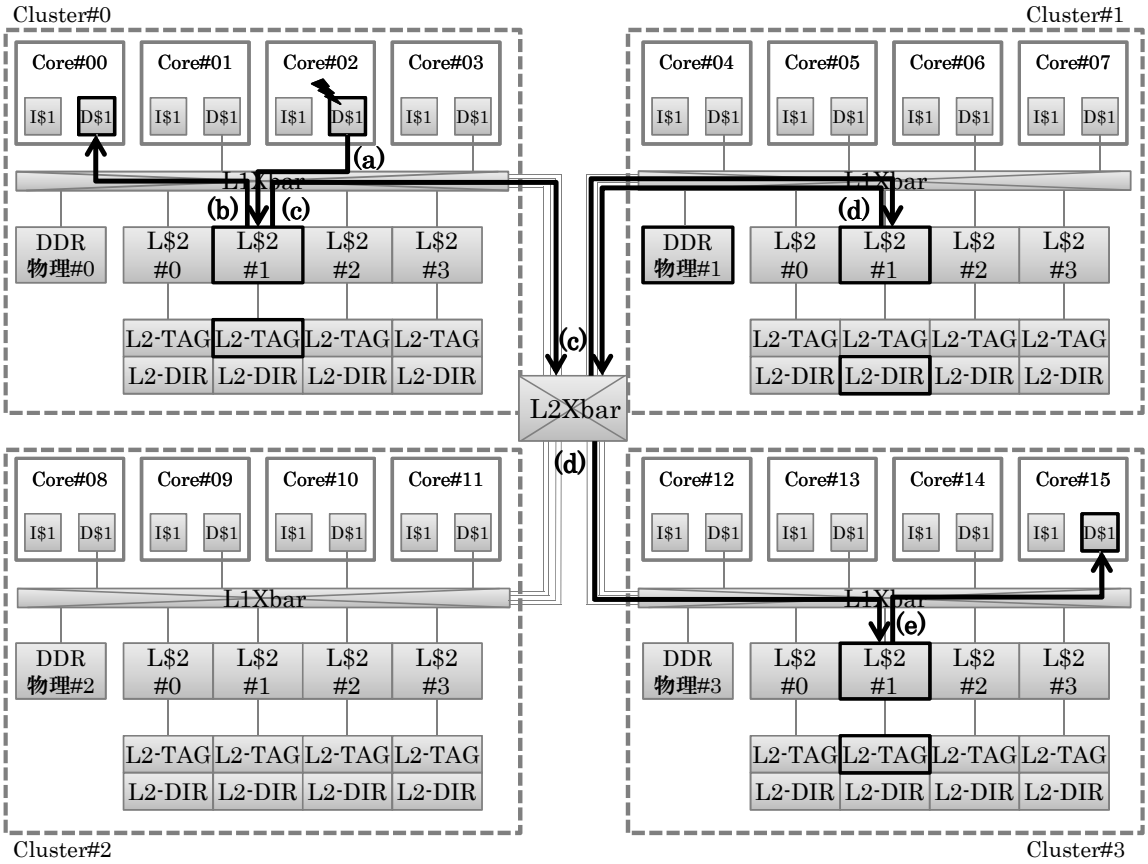

図 17: メモリトラフィックの流れ

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)．

ここで，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搬する必要がある．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)．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

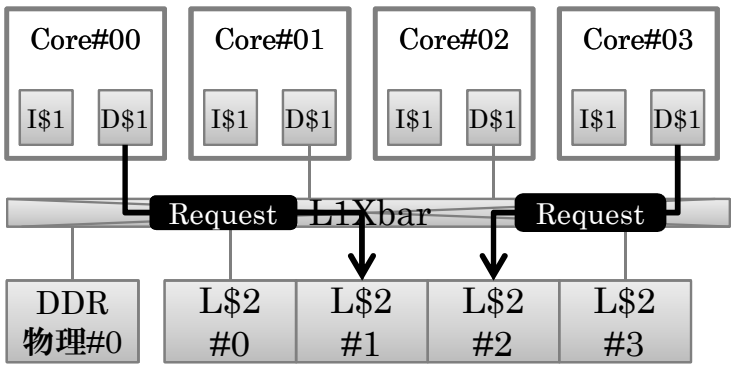

図 18: 異なるバンクへの参照要求

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．

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)．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搬する必要が生じる．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．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．

7.2.2 メモリトラフィックの衝突とリクエスト優先順位

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，メモリトラフィックの衝突が発生することになる．複数のコアから同時にキャッシュリクエストが発生

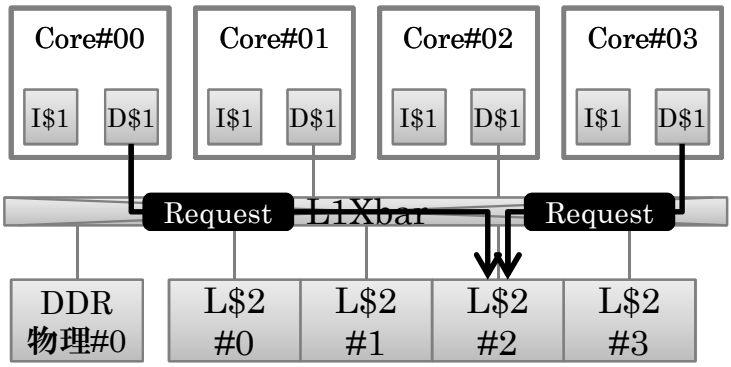

図 19: 同一バンクへの参照要求

した様子を**図 18** に示す．図 18 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．この場合には，6.2.1 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 19** に示す．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に対する処理を待機させ，キャッシュとメモリ間の一貫性を保持する．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．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要求が遂行される．ここで，2 次キャッシュへの参照要求だけでなく，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搬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．

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，優先度の低いコアの要求にリソースが回りにくくなるという欠点がある．複数のコアが頻繁にキャッシュリクエストを発生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

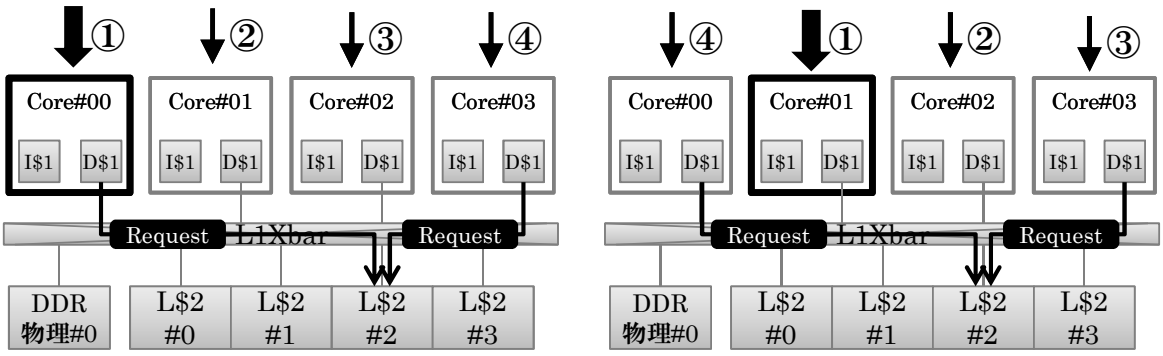

図 20: ラウンドロビン方式によるリクエスト優先順位の決定

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．これにより，ラウンドロビン方式は，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，各リクエストにおける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のタイムラインに応じてリソースが割り与えられる．これにより，バンクへのキャッシュアクセスの均一化を図ることができ，効率よくリクエストを処理できる．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 20** に示す．ラウンドロビン方式はコア番号依存方式を拡張したもので，図 20 中の (1) が，ベースとなるコア番号依存方式と同じ動作を表している．同一バンクへの参照リクエストが同時に発生した場合に，優先されるコアからの参照リクエストが到達しているかを確認し，最初に見つかったリクエストの処理を進める．ラウンドロビン方式では，このコアの優先度を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

余りの値を元にし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．図 20 の (2) は，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．

7.3 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．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ることになる．そのため，今回のメニーコアトレースシミュレータでは，各コアからの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．トレースデータの抽出フローを**図 21** 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．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 ビットで表すことで，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り，膨大なデータ量を大幅に削減できる．

7.4 シミュレータ実行の高速化

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，5.2 節で述べたように，ソフトウェアによるシミュレーションは評価において多大な時間を要することが知られている．特に複数のプロセッサコアを有するマルチ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．

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，以前から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

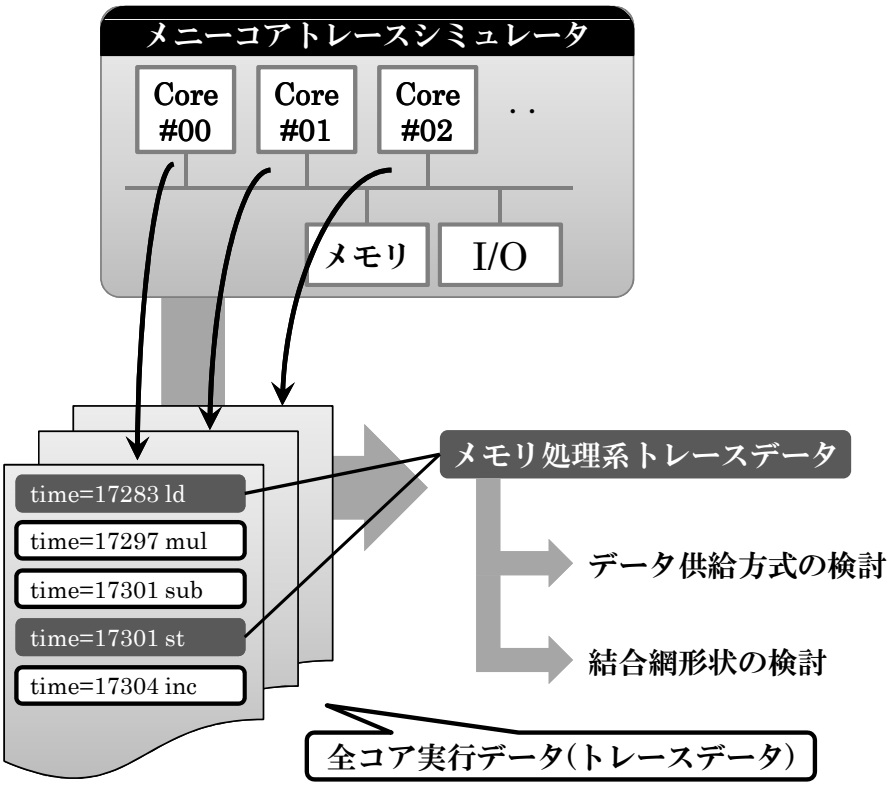

図 21: トレースデータ抽出フロー

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．

そのため，アーキテクチャシミュレーションの高速化に関して，高速で精度の高いシミュレーション手法についての研究が注目されている．このような手法には，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 [26] 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などがある．ここで，提案する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本シミュレータに組み込んでいく必要があると考えられる．

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，メニーコアトレースシミュレータをまず単純なスレッド並列化機構によって高速化する手法を考える．この手法では，1 サイクル毎に各コアの動作をシミュレートするのではなく，コア毎の一定サイクル数における実行を 1 つのタスクとし，スレッド並列化によって全コアの動作を並列にシミュレートする．また，本論文で述べられている計算再利用技術に基

づく高速化手法をシミュレート実行の高速化に適用する方法も考えられる．そのため，各コアのサイクル毎のシミュレート結果を記憶し，コア間の計算再利用により実行を省略するという手法を検討することが今後の課題に挙げられる．

8　動作検証

以上で述べたメニーコアトレースシミュレータを開発し，サイクルベースシミュレーションにより動作を検証した．

8.1　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 5** に示す．プロセッサ構成は，7.1 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，データキャッシュはデータ管理の複雑度を抑えるために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，行列積演算プログラムおよび汎用ベンチマークプログラムである SPLASH-2 ベンチマーク [27] の FFT を用いた．なお，シミュレータ実行の高速化のためのスレッド並列化機構は実装途中であるため，本評価では 1 サイクル毎に全コアの動作をシミュレートする方法をとっている．

8.2　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．行列積演算プログラムの実行結果を**表 6** に，SPLASH-2 FFT の実行結果を**表 7** に示す．表は左から順に，コア番号の pid，発行命令数の step，命令実行サイクル数の exec，I1 キャッシュミスにおける待ちサイクル数の I1wait，D1 キャッシュミスにおける待ちサイクル数の D1wait，総実行サイクル数の cycle を表している．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算出してあり，それぞれ I1hit， D1hit で表している．

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キャッシュ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

表5: 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	32 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

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

表 6: 実行サイクル数 (行列積演算プログラム)

	pid	step	exec	I1wait	D1wait	cycle	I1hit	D1hit
1thread	0	68013	132755	4631	2536	**139922**	99.87%	99.17%
2threads	0	43397	95974	4784	2675	**103433**	99.78%	98.69%
	1	25447	37986	213	590	38789	99.98%	99.28%
4threads	0	31087	77585	4761	2882	**85228**	99.68%	98.07%
	1	12735	19187	243	792	20222	99.96%	98.81%
	2	12735	19188	218	955	20361	99.96%	98.77%
	3	12735	19364	166	822	20352	99.96%	98.89%
8threads	0	25286	68754	4799	2979	**76532**	99.60%	97.44%
	1	6379	9787	218	706	10711	99.92%	98.04%
	2	6379	9787	209	946	10942	99.92%	98.04%
	3	6379	9904	164	818	10886	99.92%	98.04%
	4	6379	9787	189	1372	11348	99.92%	98.04%
	5	6379	9829	127	1333	11289	99.92%	98.04%
	6	6379	10126	196	958	11280	99.92%	98.04%
	7	6379	10152	95	1021	11268	99.92%	98.12%
16threads	0	22380	64347	4785	3239	**72371**	99.55%	96.88%
	1	3201	5086	225	785	6096	99.84%	96.43%
	2	3201	5087	176	863	6126	99.84%	96.43%
	3	3201	5087	109	1016	6212	99.84%	96.43%
	4	3201	5087	176	941	6204	99.84%	96.43%
	5	3201	5098	157	940	6195	99.84%	96.43%
	6	3201	5087	110	1196	6393	99.84%	96.43%
	7	3201	5097	154	1126	6377	99.84%	96.43%
	8	3201	5087	169	1122	6378	99.84%	96.43%
	9	3201	5114	149	1103	6366	99.84%	96.43%
	10	3201	5087	178	1263	6528	99.84%	96.43%
	11	3201	5087	185	1248	6520	99.84%	96.43%
	12	3201	5087	165	1369	6621	99.84%	96.43%
	13	3201	5087	189	1336	6612	99.84%	96.43%
	14	3201	5382	238	921	6541	99.84%	96.43%
	15	3201	5408	394	716	6518	99.84%	96.59%

表 7: 実行サイクル数 (SPLASH-2 FFT)

	pid	step	exec	I1wait	D1wait	cycle	I1hit	D1hit
1thread	0	350726	438370	17182	20018	**475570**	99.90%	98.67%
2threads	0	313760	389323	17684	15983	**422990**	99.88%	98.95%
	1	41025	54757	3050	6742	64549	99.83%	96.45%
4threads	0	294497	363736	17707	17509	**398952**	99.87%	98.65%
	1	21651	29050	3066	8674	40790	99.72%	89.88%
	2	21556	29288	2744	8702	40734	99.70%	90.13%
	3	21465	28947	2823	8962	40732	99.71%	89.87%
8threads	0	285901	352286	17541	19553	**389380**	99.87%	98.88%
	1	12761	17284	4142	10094	31520	99.45%	87.28%
	2	12677	17494	2619	11351	31464	99.49%	87.52%
	3	12939	17707	3101	10654	31462	99.50%	87.43%
	4	12661	17291	2875	11294	31460	99.49%	86.97%
	5	12523	17069	2847	11597	31513	99.48%	87.32%
	6	12212	16525	2613	12407	31545	99.49%	87.96%
	7	12407	17190	2849	11504	31543	99.48%	87.82%
16threads	0	283927	349389	18386	27770	**395545**	99.87%	98.94%
	1	10080	13458	5487	18555	37500	99.31%	83.66%
	2	9681	13223	3738	20483	37444	99.33%	84.48%
	3	9763	13333	3295	20814	37442	99.33%	84.23%
	4	8826	11975	5284	20181	37440	99.16%	82.19%
	5	8934	12179	4217	21042	37438	99.27%	85.44%
	6	9577	12921	3299	21216	37436	99.32%	84.51%
	7	8898	12194	4290	20950	37434	99.27%	85.13%
	8	9260	12671	3919	20842	37432	99.30%	83.31%
	9	9288	12703	2612	22115	37430	99.30%	83.42%
	10	8831	12148	3551	21729	37428	99.26%	85.58%
	11	8921	12437	3528	21461	37426	99.27%	85.47%
	12	8979	12153	3718	21553	37424	99.31%	86.23%
	13	9149	12519	6088	18815	37422	99.19%	82.39%
	14	8875	12380	2801	22239	37420	99.27%	84.32%
	15	8940	12534	3338	21546	37418	99.27%	85.14%

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を算出するために，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んでいる．

行列積演算プログラムでは，台数効果を得られていることがわかる．スレッド数の増大に伴い総実行サイクル数が減少しているだけでなく，並列実行時におけるワーカスレッドの実行サイクル数がスレッド数の増大に比例して減少する結果となっった．行列積演算は，比較的単純なプログラムであり，タスクを均等に分割して効率良く並列実行できることが知られている．予想通りの実行結果が得られたことから，シミュレータが正しく動作していることを確認できた．

一方で，SPLASH-2 の FFT では，8 並列実行までは並列化により高速化できているが，16 並列実行では 8 並列実行に比べて速度低下してしまっている．しかし，step や exec を比較してみると，スレッド数の増大に伴い効率的に演算処理を分割できていることがわかる．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い．そのため，16 並列実行時には D1wait のサイクル数が増加し，実行速度が低下してしまったと考えられる．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が改めて認識できた．

8.3 考察

今回のシミュレーション実行では，動作を検証することを目的としているため，実行結果の評価はあまり重要視していない．しかし，メニーコアプロセッサの性能目標値を導出するためには，表 6 および表 7 で挙げた評価指標以外の結果も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．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搬時に発生する遅延サイクル数である．キャッシュリクエストの伝搬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

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搬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．そのために，遅延サイクルを計上してキャッシュリクエストの Ack に付加することで，各コアの要求に要した遅延サイクル数をそれぞれ算出することが可能となる．他には，2 次キャッシュのヒット率やバンクにおけるリクエストの衝突回数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 5 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．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した場合に，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．

9 おわりに

本論文では，まず計算再利用技術に基づく自動メモ化プロセッサの並列事前実行機構に対し，遊休状態である投機コアにキャッシュプリフェッチを行う一種のスカウトスレッドを実行させることで，従来の再利用の効果を阻害することなくメモリアクセスレイテンシの一部を隠蔽する手法を提案した．また，そのキャッシュプリフェッチが，限定的ではあるが再利用率の向上への寄与が期待できることを述べた．

SPEC CPU95 FP ベンチマークを用いて評価した結果，提案手法がメインコアのメモリへのアクセスレイテンシを隠蔽し，2 次キャッシュミスを大幅に削減することを確認した．既存モデルでは最大 40.6%，平均 15.0%であった総実行サイクル数削減率が，提案するモデルでは最大 41.3%，平均 19.1%まで向上させることができた．また，107.mgrid においては僅かながら再利用率向上と，それに伴う実行サイクル数の削減も確認できた．比較的単純なアーキテクチャを想定して評価しているが，並列処理による高速化手法とメモ化を組み合わせることでさらなる高速化を図ることができるという知見が得られた．

また，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形状における各構成方式を検討した．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．

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，メニーコアトレースシミュレータの開発における，動作の正当性の向上が考えられる．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順に組み込んでいく必要がある．また，動作する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検証することで動作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を高速化することが挙げられる．多大な時間を要するソフトウェアシミュレーションの評価時間短縮のために，実マルチコア環境で並列実行可能な機構を追加し高速化を図る必要がある．そのために，一定サイクル数における実行を1つのタスクとし，スレッド並列化によって全コアの動作を並列にシミュレートした場合に，共有キャッシュの一貫性をどのように保持するか，またタスク間の各処理をどのように同期するかなどの検討が必要になる．

最後の課題としては，様々なプロセッサ高速化手法と組み合わせることが挙げられる．メニーコアプロセッサでは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いる．本論文で取り扱った自動メモ化プロセッサでは，計算再利用・並列事前実行・スカウトスレッドといった複数のコアを利用して高速化を図る機構を十分に備えている．本論文で提案したモデルの適用範囲や電力消費量の増加に関する詳細な調査を進め，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すること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．特に，研究に関して貴重な意見を下さった山田龍寛氏，小田遼亮氏に深く感謝致します．

著者発表論文

論文

1. 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 2011 (SoC2011), Tampere, Finland, pp.22-28(Nov. 2011)
2. Tomoki IKEGAYA, Tomoaki TSUMURA, Hiroshi MATSUO, Yasuhiko NAKASHIMA: “A Speed-up Technique for an Auto-Memoization Processor by Collectively Reusing Continuous Iterations”, Proc. 1st Int’l. Conf. on Networking and Computing (ICNC’10), Higashi-Hiroshima, Japan, pp.63-70, received the Best Paper Award (Nov. 2010)
3. 池谷 友基, 津邑 公暁, 松尾 啓志, 中島 康彦: “複数イタレーションの一括再利用による並列事前実行の高速化”, 情報処理学会論文誌 コンピューティングシステム (ACS), Vol.3, No.3, pp.31-41 (Sep. 2010)
4. 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における複数イタレーションの一括再利用”, 先進的計算基盤システムシンポジウム (SACSIS2010) 論文集, pp.149-156 (May. 2010)
5. Ryosuke ODA, Tatsuhiro YAMADA, Tomoki IKEGAYA, Tomoaki TSUMURA, Hiroshi MATSUO, Yasuhiko NAKASHIMA: “Input Entry Integration for an Auto-Memoization Processor”, Proc. of The 3rd Workshop on Ultra Performance and Dependable Acceleration Systems (UPDAS), Osaka, Japan, pp.179-185 (Nov. 2011)

報文

1. 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 (Jul. 2011)
2. 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 (Jul. 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] Tsumura, T., Suzuki, I., Ikeuchi, Y., Matsuo, H., Nakashima, H. and Nakashima, Y.: Design and Evaluation of an Auto-Memoization Processor, *Proc. Parallel and Distributed Computing and Networks*, pp. 245–250 (2007).

[4] 池谷友基, 津邑公暁, 松尾啓志, 中島康彦: 複数イタレーションの一括再利用による並列事前実行の高速化, 情報処理学会論文誌コンピューティングシステム（ACS）, Vol. 3, No. 3, pp. 31–43 (2010).

[5] Norvig, P.: *Paradigms of Artificial Intelligence Programming*, Morgan Kaufmann (1992).

[6] Swadi, K., Taha, W., Kiselyov, O. and Pasalic, E.: A Monadic Approach for Avoiding Code Duplication when Staging Memoized Functions, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '06)*, ACM Press (2006).

[7] Beame, P., Impagliazzo, R., Pitassi, T. and Segerlind, N.: Memoization and DPLL: Formula Caching Proof Systems, *Computational Complexity, Annual IEEE Conference on*, Vol. 0, p. 248 (2003).

[8] 森本武資, 岩崎英哉, 竹内郁雄: 枝刈り機構とメモ化機構をもつ言語, コンピュータソフトウェア, Vol. 21, No. 4, pp. 55–60 (2004).

[9] Tremblay, M. and Chaudhry, S.: A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread SPARC Processor, *ISSCC Dig. Tech. Papers*, pp. 82–83 (2008).

[10] Wang, K. and Franklin, M.: Highly Accurate Data Value Prediction Using Hybrid Predictors, *30th MICRO*, pp. 281–290 (1997).

[11] Chaudhry, S., Cypher, R., Ekman, M., Karlsson, M., Landin, A., Yip, S., Zeffer, H. and Tremblay, M.: Rock: A High-Performance Sparc CMT Processor, *IEEE Micro*, Vol. 29, No. 2, pp. 6–16 (2009).

[12] 島崎裕介, 津邑公暁, 中島浩, 松尾啓志, 中島康彦: 自動メモ化プロセッサにおける消費エネルギー制御, 情報処理学会論文誌コンピューティングシステム（ACS）,

Vol. 1, No. 2, pp. 1–11 (2008).

[13] Weaver, D. L. and Germond, T.(eds.): *The SPARC Architecture Manual Version 8*, Prentice-Hall, Inc. (1992).

[14] HAL Computer Systems/Fujitsu: *SPARC64-III User's Guide* (1998).

[15] MOSAID Technologies Inc.: *Feature Sheet: MOSAID Class-IC DC18288*, 1.3 edition (2003).

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

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

[18] Sun Microsystems, Inc.: *UltraSPARC T2 Supplement to the UltraSPARC Architecture 2007* (2007).

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

[20] 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).

[21] Argollo, E., Falcon, A., Faraboschi, P., Monchiero, M. and Ortega, D.: COTSon: Infrastructure for Full System Simulation, *SIGOPS Oper. Syst.Rev.*, pp. 52–61 (2009).

[22] Chan, J., Hendry, G., Biberman, A., Bergman, K. and Carloni, L. P.: Phoenixsim: a Simulator for Physical-Layer Analysis of Chip-Scale Photonic Interconnection Networks, *DATE 2010*, pp. 691–696 (2010).

[23] M.S.Gaur, B.M.Al-Hashimi, V.Laxmi, R, N., Choudhary, N., Jain, L., Ahmed, M., K.K.Paliwal, Varsha, Rekha and Vineetha: NIRGAM: A simulator for NoC Interconnect Routing and Applications' Modeling, *DATE 2007* (2007).

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

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

[26] 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).

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