Vol.2009-ARC-184 No.9
2009/8/4

CoreSymphony アーキテクチャの高効率化

若杉 祐太[†1] 坂口 嘉一[†2]
三好 健文[†1,†3] 吉瀬 謙二[†1]

我々はこれまでに，CMP の逐次性能を向上する CoreSymphony アーキテクチャを提案している．CoreSymphony は，発行幅の狭いプロセッサコアをいくつか協調動作させることで，より発行幅の大きな仮想コアを形成する技術である．本稿では，CoreSymphony の過去の実装を見直し，CoreSymphony アーキテクチャver.0.2 を定義する．これには，CoreSymphony のフロントエンドの分割を可能にするローカル命令キャッシュや，コア間通信を抑制し性能を向上するリーフノードステアリングといった重要な要素技術が含まれる．SPECint2006 を含むいくつかの整数ベンチマークにより評価した結果，CoreSymphony ver.0.2 は 4 コアの協調により，1 コア時と比較して 1.4 倍の IPC を達成した．

High-efficient implementation of CoreSymphony Architecture

Yuhta Wakasugi,[†1] Yoshito Sakaguchi,[†2]
Takefumi Miyoshi[†1,†3] and Kenji Kise[†1]

We previously proposed CoreSymphony architecture which improves the sequential performance of Chip Multi-Processors. CoreSymphony enables some narrow-issue cores to fuse into one wide-issue core. In this paper, we revise the past implementation of CoreSymphony and define “CoreSymphony architecture-ver.0.2”. This definition includes some important techniques such as Local instruction-cache or Leaf node steering. Local instruction-cache realize a dividable front-end of CoreSymphony. Leaf node steering reduces inter core communications and improves the performance. Our evaluation results using some integer benchmarks including SPECint2006 show that 4-way symphony achieves 40% higher IPC than an individual core.

1. はじめに

近年，1 チップに複数のコアを集積する CMP(Chip Multi-Processor) が主流のアーキテクチャとなっている．CMP はプログラムに内在するスレッドレベル並列性を利用し，複数のスレッドを複数のコアで並列実行することで性能向上を得る．チップあたりの搭載可能コア数は半導体技術の持続的な進歩により今後も増加する見通しである．

このような潮流を受けて Amdahl の法則が再び注目を集めている[1)]．本法則において，CMP による並列プログラムの性能向上は次式で表される．式中の f はプログラム中の全処理における並列化可能な処理の割合であり，n は同時実行可能なスレッド数である．

$$Speedup_{parallel}(f,n) = \frac{1}{(1-f)+f/n}$$

この式が注目を集める理由は，この式が CMP に対し重要な問題を提起するためである．それはプログラム中に存在する並列化不可能な処理 (逐次処理) が CMP の性能を制限するということである．例として，プログラム中の並列化可能な処理の割合が 90%であったとしよう．この場合，仮に 100 コアを使用して並列に処理したとしても，1 コア時に対する性能向上は 10 倍弱にとどまる．現状では，並列プログラム中の逐次処理をなくすことは難しい．**CMP においても逐次処理の高速化が依然重要な課題である**といえる．

CMP の逐次性能を高速化するアプローチとして，複数のコアを協調動作させることで，逐次実行能力の高い一つの仮想コアを作り出す技術[2)–5)] が存在する．我々が提案する CoreSymphony アーキテクチャ[6)] もそのひとつである．CoreSymphony は 2 命令発行の比較的軽量なコアを 2 コアないし 4 コア利用し，協調動作させることで，最大 8 命令発行の強力な仮想コアを形成するアーキテクチャ技術である．仮想コア上では従来のプログラムがネイティブで動作し，並列プログラム中の逐次処理の高速化を強力に支援する．コアの構成は動的に変更可能である．**図 1** に CoreSymphony の構成例を示す．左は各コアが独立に動作し，計 16 スレッドを実行可能な構成である．右は協調動作により，スレッドあたりの実行能力を変化させたひとつの構成例である．同時実行可能なスレッド数と各スレッドの処理能力を変

†1 東京工業大学 大学院情報理工学研究科
Graduate School of Information Science and Engineering, Tokyo Institute of Technology
†2 東京工業大学 工学部情報工学科
Department of Computer Science, Tokyo Institute of Technology
†3 独立行政法人 科学技術振興機構 CREST
CREST, Japan Science and Technology Agency

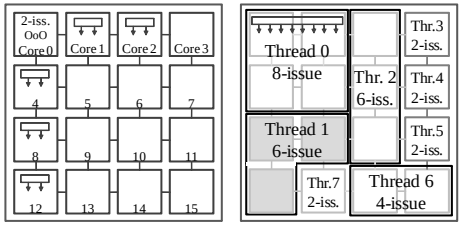

図 1 CoreSymphony アーキテクチャの構成例．16 スレッド実行可能な構成 (左)．8 スレッド実行可能な構成の 1 つ (右)．協調動作をおこなうコア数，協調対象のコアの配置にはある程度の柔軟性がある．

化させ，さまざまなタスクの状況に柔軟に対応することができる．

本稿では，我々が過去に提案した CoreSymphony アーキテクチャの実装[6](ver.0.1 とする) の効率化，未実装機能の追加をおこない，**CoreSymphony アーキテクチャver.0.2** を定義する．この定義は拡張されたローカル命令キャッシュや，CoreSymphony 向けの新しいステアリングアルゴリズムといった重要な要素技術を含む．

本稿の構成を示す．2 章では CoreSymphony のコンセプトや命令の実行モデルについて述べる．3 章では実装を詳細にまとめる．特に過去の実装から機能が拡張されたフロントエンドについて詳しく述べる．4 章では CoreSymphony 向けの新しいステアリングアルゴリズムを提案する．5 章では性能と HW 量の両面から CoreSymphony を評価し，6 章で関連研究をまとめる．最後に 7 章で結論と今後の課題を述べる．

2. CoreSymphony アーキテクチャの概要

2.1 アーキテクチャのコンセプト

CoreSymphony アーキテクチャは CMP における逐次性能の高速化という大目標に加え，次の 3 つの達成を目指す．

(1) コアの独立性の維持

CoreSymphony はメッシュ等のシンプルな結合網により接続された均質なコアを多数持つプロセッサへの適用を念頭におく．また，協調対象のコア数やコアの配置にはいくらかの柔軟性を与える．このため，コア間通信を必要とするモジュールを多数持つことや，クラスタ型アーキテクチャのフロントエンドのように，制御を集中化しておこなうことは望ましくない．CoreSymphony では，コア間通信は CMP に備わる結合網のみを利用し，制御は各コアに完全に分散する．このため，計算や制御情報，データを各コアで多重化する必要が生じ，実行効率および面積効率でいくらかの不利を被る．CoreSymphony ではこれらの不利を現実的な範囲に収めながら，各コアの独立性を維持することを重要な課題とする．

(2) アーキテクチャ技術の連続性の維持

コアの協調を実現するためには，従来のプロセッサコアに変更を加える必要がある．CoreSymphony ではベースアーキテクチャとしてアウトオブオーダを採用し，変更はなるべく小さいものに留める．これにより，従来のアーキテクチャ技術からの連続性，実現可能性を高める．

(3) バイナリの連続性の維持

関連研究 3) のように制御情報が命令に埋め込まれた特別な命令セット[7] を採用することは，協調により生じる各種制御の複雑さを緩和するために非常に有効な手段である．しかし，CoreSymphony では従来型の RISC 命令セットによる協調動作の実現を目指す．これは既存のコンパイラ最適化技術を流用するためとバイナリの連続性を維持するためである．

2.2 命令実行モデル

CoreSymphony では，命令を**フェッチブロック (FB)** と呼ぶ単位に分割して実行する．FB の最大長は協調動作中のコア数× 4 命令である．また，FB の長さは FB 中に 3 個以上の基本ブロックを含まないという条件によっても制限される．FB 中の命令はクラスタ型アーキテクチャのステアリングユニットに類似するステアリングモジュールにより分割され，協調動作中のコアに割り当てられる．これにより仮想的に発行幅の大きなスーパースカラを構成することが可能になる．

本モデルを実現するための課題は，フロントエンドを各コアに分割することである．アウトオブオーダのバックエンドはデータ駆動であるため，クラスタ型アーキテクチャのように小さな単位に分割することが比較的容易である．しかしながら，フロントエンドは本質的に制御駆動であるために制御の集中化を必要とする．このことは分割を困難にする．多くのクラスタ型アーキテクチャにおいてもフロントエンドの分割は達成されていない．

CoreSymphony では**ローカル命令キャッシュ**という，我々が提案する特別な命令キャッシュによりこの課題を解決する．ローカル命令キャッシュは FB 中の自コアにステアリングされた命令と FB の制御情報を保存するキャッシュである．ローカル命令キャッシュは従来型の命令キャッシュに加えてフロントエンドに実装する．2 コア協調時に (I0，I1，...，I7)[⋆1]の 8 命令からなる FB を実行する場合を例として，FB を分割実行する仕組みを**図 2** に示す．FB

⋆1 In はひとつの命令を表す．また，ある FB を (I0，I1，...In) と表すことがある．

Vol.2009-ARC-184 No.9
2009/8/4

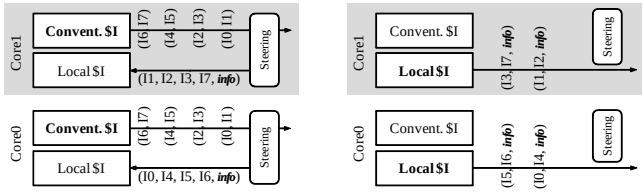

図 2 フェッチブロックを分割実行するための 2 つのフェーズ．(左) ローカル命令キャッシュ(Local $I) のエントリ構築フェーズ．(右) 分割実行フェーズ

の分割実行は 2 つのフェーズに分けられる．図 2(左) はローカル命令キャッシュのエントリ構築フェーズである．これはローカル命令キャッシュにミスする場合にのみおこなわれる．本フェーズでは従来型命令キャッシュから FB 中の全ての命令を読み出し，依存関係の解析とステアリングをおこなう．そしてローカル命令キャッシュの当該エントリに，自身にステアリングされた命令と依存の解析によって得られた各種の制御情報を格納する．図 2(右) に示す分割実行フェーズは，ローカル命令キャッシュにヒットする場合である．この場合，各コアは自身にステアリングされた命令のみをフェッチするためにフロントエンドは実質的に分割され，N 個のコアを利用することで最大 N 倍のスループットが得られる．この仕組みを実現するための詳細な実装を次章で述べる．

3. CoreSymphony アーキテクチャの実装

3.1 フロントエンドの実装

CoreSymphony アーキテクチャのフロントエンドの実装を詳細に示す．**図 3** は命令フェッチからディスパッチのブロック図である．以降各コンポーネントについて，従来のアウトオブオーダとの相違を中心にまとめる．

3.1.1 分岐予測器

CoreSymphony では協調動作中の全コアで分岐予測を多重化しておこなう．これは全てのコアが同様の分岐履歴を用いて同様の予測をすることを意味する．この実装では，協調により分岐履歴表のエントリを増加させることができない．しかし，全コアが同様のパスで命令をフェッチすることを容易にする．実際には，ある FB をフェッチするタイミングは各コアで異なる[⋆1]ために，全コアでフェッチパスを揃えるためには工夫が必要である．この手法

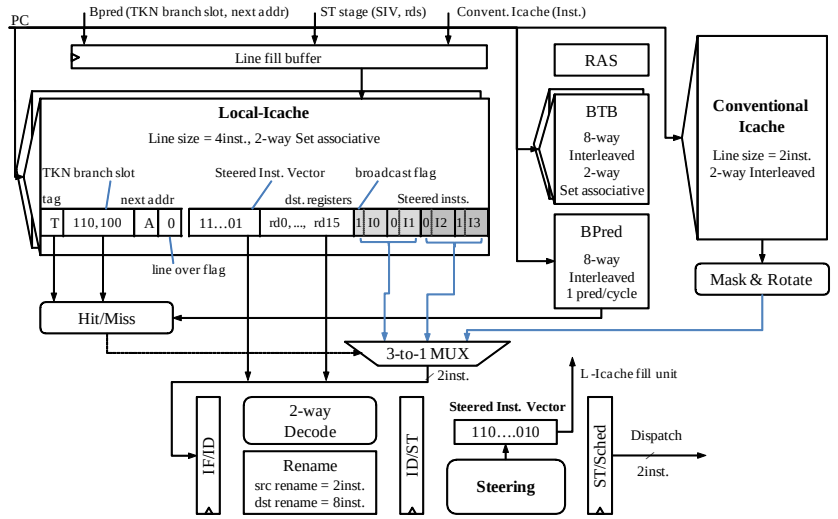

図 3 CoreSymphony アーキテクチャver.0.2 のフロントエンド．

については文献 6) を参照されたい．

分岐予測器自体は一般的なものを用いる．ただし，入力 PC から 8 命令先[⋆2]までの分岐予測を 1 サイクルでおこなう必要がある．そのために分岐先バッファ(BTB) は 8-way のインターリーブ構成をとる．また，分岐予測器のスループットは 1 予測/cycle である．

3.1.2 ローカル命令キャッシュ

ローカル命令キャッシュの 1 ラインは基本的に 1 つの FB に対応する．FB は最大 2 つの基本ブロックを含むため，実行トレースと捉えることもできる．よって，ローカル命令キャッシュはトレースキャッシュ[8] に近い構成をとる．ただし，分岐予測器のスループットは 1 予測/cycle であるために，ローカル命令キャッシュの Hit/Miss 判定には 2 サイクルを要する．各フィールドの役割を以下に示す．

- TKN branch slot (3 × 2 bit)
  当該 FB において各基本ブロックの先頭から何命令先の分岐が成立と予測されているかを示す．当該 FB に対する 2 回の分岐予測結果がそれぞれの値とマッチしたときのみ，

⋆1 キャッシュミスや物理レジスタの枯渇等でストールするタイミングがコアによって異なるため．

⋆2 4 コア協調時の最大の発行幅が 8 であるため．

ローカル命令キャッシュは Hit と判定される．

- next addr (32bit)
  次のブロックのフェッチアドレスを示す．
- line over flag (1bit)
  ステアリング結果によっては，1 つのコアに 4 を超える命令が割り当てられる．その場合にはこの flag を 1 にセットし，次のエントリに溢れた命令を格納する．
- Steered Instruction Vector (16bit)
  FB 中の自身に割り当てられている命令の位置を示すベクトル．N bit 目が 1 であることは，FB 中の N 番目の命令が割り当てられていることを示す．
- dst registers (6 × 16 bit)
  FB 中の全命令のデスティネーションレジスタの論理レジスタ番号を示す．自身に割り当てられていない命令のデスティネーションはデコードステージでマップ表を更新するために用いる．
- Steered insts ((32+1) × 4 bit)
  自身に割り当てられた命令を格納するフィールド．各命令には，生成された結果を協調動作中の各コアにブロードキャストするか否かを示す 1bit のフラグ (broadcast flag) が付与される．

以上を合計するとローカル命令キャッシュの 1 ラインは 283bit となる．命令部は 128bit であるため，制御情報が 55%を占めることになる．

3.1.3 デコードと依存解析

デコーダのスループットはベースのアウトオブオーダと同様 2 命令/cycle である．マップ表に関しては，ソースのリネームは 2 命令/cycle であるが，デスティネーションの登録は 8 命令/cycle のスループットを必要とする．これは現在のモデルでは，全てのコアで同一の規則の元でリネーミングをおこなう必要があるためである．ローカル命令キャッシュから読みだした FB 中の全命令のデスティネーションレジスタを用いてマップ表を更新する．

3.1.4 ステアリングユニット

ステアリングユニットの役割は，FB 中のどの命令を自身が実行するかを表す **Steered Instruction Vector(SIV)** を作成することである．ステアリングはローカル命令キャッシュにミスした場合のみおこない，作成された SIV はローカル命令キャッシュに格納される．次に同様の FB をフェッチする際にはローカル命令キャッシュに格納された SIV に従い，ステアリングされた命令のみを従来型命令キャッシュからフェッチする．この段階で初めてローカル命令キャッシュのエントリが完成する．すなわちローカル命令キャッシュに，ある FB のエントリが完成するためには **(1)SIV の作成**，**(2)SIV に基づき当該命令のみを読み出し**の 2 ステップを必要とする．SIV を用いる 2 段階の方法では，ステアリングステージに命令本体をバッファする必要がない．このため，FB 中の命令を逆順に解析するような，より高度でオーバーヘッドの大きいステアリングアルゴリズムの利用を容易にする．高度なステアリングの例として，次章でリーフノードステアリングを提案する．

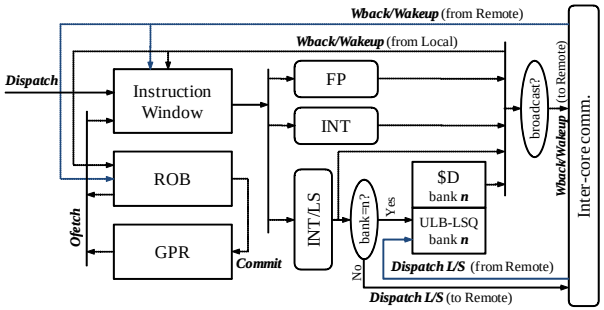

図 4 CoreSymphony アーキテクチャver.0.2 のバックエンド

3.2 バックエンドの実装

CoreSymphony のバックエンドの実装を詳細に示す．**図 4** はディスパッチからコミットのブロック図である．バックエンドの構成は ROB 方式のクラスタ型アーキテクチャにおける 1 つのクラスタと類似する．命令ウインドウと機能ユニットは各コアに分散するため，協調動作により発行幅の向上が可能になる．物理レジスタに関しては，今回の実装では多重化を選択する．このために物理レジスタのエントリは協調動作時の大きな命令ウインドウに耐えうるだけの数が必要になる．物理レジスタの分散方式の検討は今後の課題とする．

各コアで実行された命令の結果は物理レジスタのタグとともに，協調動作中の各コアにブロードキャストする．ただし，ある FB 中の命令 I_n と，同 FB 中の命令 $I_m (n < m)$ が同じデスティネーションレジスタを有し，なおかつ I_n の結果を利用する命令が全て I_n と同じコアにステアリングされている場合には，I_n の結果はブロードキャストする必要がない．この場合には I_n の結果のブロードキャストはフィルタリングされる．このフィルタリングは，他コアからのライトバック及び命令ウインドウのウェイクアップのポート数を削減する効果がある．このため，依存関係にある命令をなるべく同じコアにステアリングするアルゴリズムが重要となる．ポート数削減の効果は 5 章で評価する．

3.2.1 ロード/ストアユニットの実装

CoreSymphony では協調動作中の各コアのデータキャッシュを，実効アドレスでバンク分けされた一つの大きなデータキャッシュとして扱う．アドレスによるバンク分けにより，ロード/ストア命令のスケジューリングおよびメモリあいまい性除去を局所化できるという利点が生まれる．クラスタ型アーキテクチャでこのアプローチを採用する際には，メモリバンク予測器[9] によりロード/ストア命令がどのバンクにアクセスするかを予測し，当該バンクへのアクセスが可能なクラスタに命令をステアリングするという手法[10),11)] が用いられる．しかし，この手法ではバンク予測ミスの取扱いのためにバンク分けされたロード/ストアキュー同士を密に結合する必要がある．また，CoreSymphony ではある FB のステアリング結果は毎回同じであるという制約からこの手法を用いることは困難である．

そこで，CoreSymphony では Simha らによって提案された Unordered Late-Binding Load/Store Queue(ULB-LSQ)[12] を用いる．このロード/ストアキューではエントリの割り当てはアドレス計算後におこなわれる．このことは，キュー内のロード/ストア命令が Program order に並ばない事を意味する．しかし，ULB-LSQ ではロード/ ストア命令にプログラム順に割り当てられた通し番号と特殊な CAM を用いて，一般的なロード/ストアキューと同等の機能を提供する．CoreSymphony では ULB-LSQ によりデータキャッシュのバンク分けを実現する．Core x でアドレス計算がおこなわれたロード/ストア命令が Core y のデータキャッシュにアクセスする場合には，実効アドレスや命令タグ等を Core x から Core y へと送信する．そして，送信された情報を用いて Core y の ULB-LSQ のエントリが割り当てられる．このためのコア間通信にはオペランドの通信と同様のネットワークを用いる．

3.2.2 コア間通信機構

協調動作のためのコア間通信は次の 2 つの場合においてのみ生じる．

(1) コア間のオペランド通信

あるコアで実行された命令がブロードキャストを必要とする場合には，コア間のオペランド通信が発生する．オペランド通信は**図 5** に示すタイミングでおこなわれる．図 5 は Core x にステアリングされた命令 I0 に，Core y にステアリングされた命令 I1 が依存する場合である．オペランド通信は 2 フェーズに分けられる．まず，I0 は自身の実行が完了する 1 サイクル前に自身のタグを放送する．そして実行完了と同時に結果を放送する．結果の 1 サイクル前に到着したタグによって，I0 に依存する I1 のウェイクアップがおこなわれる．ここで I1 がセレクトされた場合には，I1 は続けて到着する I0 の結果を取り込んで実行ステージへ移行する．この仕組みを実現するには，ある命令の結果とタグが連続して送られて来ることを保証する必要がある．

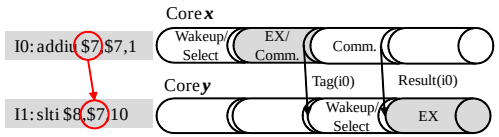

図 5 コア間オペランド通信のタイミング．コア間の通信レイテンシが 1 サイクル/ホップかつ，隣接コアに通信する場合．実際にはコア間レイテンシやコアの配置によってタイミングは変化する．

(2) 他コアの ULB-LSQ へのロード/ストア命令の割り当て

あるコアでアドレス計算をしたロード/ストア命令が他コアのデータキャッシュにアクセスする場合，ロード/ストア命令の他コアへの転送が発生する．この場合に送信されるデータは実効アドレス，命令のタグ，さらに命令の種類を表す 4bit 程度のタグである．ストア命令の場合にはこれに加えてストアデータが付与される．

コア間通信のためのハードウェアについて述べる．CoreSymphony ではコア間通信のためにスタティックネットワークを利用する．スタティックネットワークとはあらかじめ経路を固定し，サーキットスイッチング方式によりルーティングをおこなうネットワークである．通信前に経路を確保するため，パケットが衝突することはない．よって，協調動作の開始時に必要な経路を確保する．スタティックネットワークはダイナミックネットワークに比べ機能がシンプルであるために実装のコストが小さく，大量のデータを高速に転送する用途に向いている．このため，今後のオンチップネットワークに採用される可能性は十分にある．実際 Raw プロセッサ[13),14)] はダイナミックネットワークとは別にスタティックネットワークを有している．

CoreSymphony が必要とするバス幅について述べる．コア間通信のためのパケットは以下のようになる．CoreSymphony の 1 コアは 2 命令発行であるため，フィールドを 2 つもつ点に注意されたい．

```
{(type0, tag0, rslt0), (type1, tag1, rslt1), cfc}
```

まず，type0 と type1 はパケットのタイプ[⋆1]を示す 4bit のフィールドである．tag0 と tag1 は命令の物理レジスタ番号，rslt0 と rslt1 は命令の結果および分岐成立/不成立を示すフラ

⋆1 オペランド通信/ ロードストア命令の種類等．

グである．ある命令の結果とタグはパイプライン式に転送されるため，tag フィールドが指す命令と rslt フィールドが指す命令は異なる．cfc は 8bit 程度のカウンタで，コア間のアウトオブオーダースケジューリングの整合性を保つために利用される．cfc の詳しい役割は文献 6) を参照されたい．パケットの全長は物理レジスタのエントリ数が 64 の場合[⋆1]，94bit である．つまり，CoreSymphony では 1 方向あたり 100 ビット幅程度のスタティックネットワークを必要とする．ネットワークトポロジは規定しないが，CoreSymphony ver.0.2 ではメッシュを想定する．この場合，各コアを最短経路で接続するためにネットワークを何系統必要とするかは，協調動作可能な最大のコア数に依存する．協調可能コア数が 3 以下の場合にはネットワークは 1 系統で良い．しかし，4 コアを協調動作させるためには 2 系統のネットワークが必要になる．ネットワーク負荷を軽減する手法は今後の検討課題である．

4. リーフノードステアリングの提案

CoreSymphony におけるコア間のオペランド通信は，レイテンシや通信のためのハードウェア量の観点からオーバーヘッドが大きい．また，ある FB のステアリング結果は毎回同じであるという制約は，あるコアに負荷が集中する事態を招く可能性がある．これらの不利を抑えるために，CoreSymphony ではコア間通信の抑制と FB 内の負荷分散をバランスよく実現するステアリング方式が特に重要になる．本章では，CoreSymphony アーキテクチャの性能を引き出す**リーフノードステアリング**を提案し，その効率的な実装方法をまとめる．

4.1 提案するアルゴリズム

CoreSymphony の要求である，コア間通信の抑制と FB 内の負荷分散を両立することは困難な課題である．なぜならば，コア間通信を抑制するということは依存関係にある命令を同じコアにステアリングすることを意味し，これは FB 内における負荷の集中を招く．この相反する 2 つの要求を満たすために我々のアルゴリズムがとる選択は**「依存元となる命令を複数のコアに重複してステアリングすることを許す」**である．

例をあげてアルゴリズムの解説をおこなう．**図 6** は (I0, ..., I6) からなる FB を，2 種類の既存のアルゴリズム (Modulo-2, 依存ベース) と提案アルゴリズム (リーフノード) でステアリングした場合に得られる結果である．図中の矢印は命令間の依存関係を表す．

(1) Modulo-2 ステアリング

本アルゴリズムでは FB の先頭から 2 命令ごとに命令を区切り，ラウンドロビン式に各コアに割り当てる．非常に良い負荷分散が得られるが，図 6 中の破線で示すとおりコア間のオペランド通信が多くなってしまう．

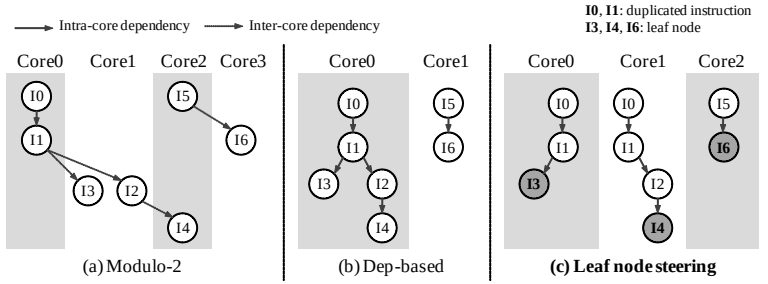

図 6 3 種のアルゴリズムによる，ある FB のステアリング．(a)Modulo-2 ステアリング，(b) 依存ベースステアリング，(c) リーフノードステアリング．

(2) 依存ベースステアリング

本アルゴリズムでは，ある命令は自身が依存する命令と同じコアにステアリングされる．依存する命令がない場合は負荷最小のコアに割り当てられる．依存する命令列を同じコアに割り当てるためコア間通信を抑えることができるが，負荷の集中を招く可能性がある．

(3) リーフノードステアリング

本アルゴリズムでは，命令ステアリングは FB 内におけるデータフローグラフのリーフ[⋆2]に着目しておこなわれる．まず，リーフの命令を各コアにラウンドロビンで割り当てる．そして，リーフ命令 I_x に対し先祖[⋆3]となる命令を全て命令 I_x と同じコアにステアリングする．図 6 の例では，リーフ命令は I_3，I_4，I_6 である．I_3 を例にとると，I_3 の先祖である I_0，I_1 が I_3 と同じコア 0 にステアリングされる．I_0，I_1 は I_4 の先祖でもあるため，コア 1 にも同時にステアリングされる．このアルゴリズムでは，依存ベースステアリング同様にコア間通信を抑えつつ，ある程度良い負荷分散が期待できる．ただし，命令の重複により負荷の総量は増加することに注意されたい．

4.2 アルゴリズムの予備評価

提案アルゴリズムの効果を予備評価する．コア間通信の抑制に対する評価の尺度として，

⋆1 この場合 tag フィールドの長さは 6bit である．

⋆2 子ノードを持たないノード．すなわち，FB 内にその命令に依存する命令が存在しない命令．

⋆3 DFG の根から当該ノードまでのパス上に存在するノード．図 6 では，I3 の先祖は I3，I1，I0 である．

表 1 3 種のアルゴリズムによる，FB の実行レイテンシ (Latency of FB)，負荷最大のコアにステアリングされた命令数 (# insts. in largest sub-block)，ステアリングされた命令数の総和 (Sum of # steered insts.) の平均値．A:Modulo-2 ステアリング，B:依存ベースステアリング，C:リーフノードステアリング．

benchmark	Latency of FB			# insts. in largest sub-block			Sum of # steered insts.		
	A	B	C	A	B	C	A	B	C
knight	6.60	3.72	3.51	3.61	6.22	5.40	12.95	12.96	14.18
queen	7.21	3.85	3.65	3.39	5.62	5.44	12.11	12.10	14.00
isomer	8.73	4.71	4.71	3.68	7.93	6.05	11.39	11.39	13.34
dijkstra	6.22	3.64	3.53	2.80	4.80	4.28	8.78	8.77	10.08
429.mcf	7.28	4.45	4.13	3.22	5.14	5.50	10.09	10.15	11.33
456.hmmer	7.35	4.34	3.91	3.29	5.10	5.80	9.71	9.71	10.47
462.libquantum	7.04	3.73	3.80	3.62	5.72	4.66	9.97	9.97	11.10
464.h264ref	8.41	4.86	4.43	3.46	6.15	5.89	11.05	11.05	12.62
Average	7.36	4.16	3.96	3.38	5.84	5.38	10.76	10.76	12.14

4 コア協調時の FB の実行レイテンシを用いる．これは全ての命令の実行サイクル数を 1 とし，同時に実行可能な命令数を無限とした時に FB の実行にかかるサイクル数である．ただし，コア間のオペランドバイパスに関しては所定のレイテンシがかかるとした．つまり，コア間通信が発生するほど FB の実行レイテンシが増加する．次に，負荷分散の尺度として負荷最大のコアにステアリングされた命令数[⋆1]を用いる．最後に命令の重複が発生する頻度を，各コアにステアリングされた命令数の総和により評価する．ベンチマークとして「C 言語による最新アルゴリズム辞典」[15] より 4 種のプログラムと SPEC2006 整数ベンチマークより 4 種を用いる．詳しい評価環境は 5.1 節を参照されたい．

表 1 に前節で紹介した 3 つのステアリングアルゴリズムにおける評価結果を示す．表中の値は実行中の全 FB に対する平均値である．まず，実行レイテンシを見ると提案手法は Modulo-2 よりも平均で 47%小さく，依存ベースよりも 5%小さい．コア間通信が抑えられていることがわかる．次に，負荷最大のコアにステアリングされた命令数では，提案手法は Modulo-2 よりも大きいが依存ベースよりも小さい．負荷分散の効果が表れていることがわかる．最後にステアリングされた命令数の総和をみると，重複してステアリングされる命令数は FB あたり，1.4 命令程度であることがわかる．これらのステアリングアルゴリズムが最終的な性能に与える影響は 5 章で示す．

4.3 アルゴリズムの実装

提案手法は FB 内の命令をプログラムの逆順に解析するため，少ないハードウェア量で実装するには工夫が必要である．本節では行列を用いる効率的な実装方法を示す．

ステアリングは 2 つのフェーズに分割しておこなう．1 つ目のフェーズでは先祖ノード行列 (Ancestor node matrix) と親ノードベクトル (Parent node vector) と呼ぶ，データフローグラフ解析のためのテーブルを構築する．2 つ目のフェーズでは構築したテーブルをもとに SIV を完成させる．それぞれのフェーズについて例を挙げて解説する．

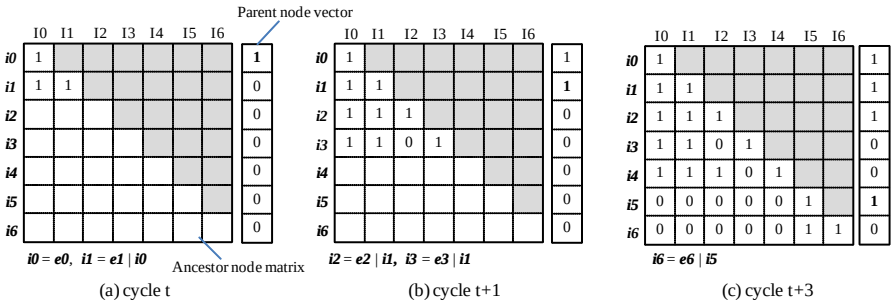

図 7 先祖ノード行列，親ノードベクトル構築フェーズ．実際の先祖ノード行列は 16 × 16 bit である．

(1) 先祖ノード行列，親ノードベクトル構築フェーズ

まず，先祖ノード行列とは FB 中の命令 I_x に対し命令 I_y が先祖であるか否かを示す 16 × 16 bit の行列[⋆2]である．すなわち先祖ノード行列の行 x は命令 I_x に対する先祖の集合を表す．これを命令 I_x の先祖ベクトルと呼び，i_x で表す．命令 I_x が命令 I_y と I_z に真のデータ依存をもつ場合，$i_x = e_x|i_y|i_z$ として計算できる．e_x は $x$bit 目のみ 1 である基本ベクトルを意味し，| はビット論理和演算を意味する．

次に，親ノードベクトルとは FB 中の命令が子ノード (その命令に依存する命令) を持つか否かを示す 16bit のレジスタである．レジスタの $x$bit 目が 1 であることは命令 I_x が子ノードを持つこと示す．

図 6 に示した FB を例に本フェーズの動作を解説する．**図 7** は本フェーズの動作を時間順に示したものである．命令は 2 命令/cycle のスループットでステアリングステージに訪れる．まず cycle t では (I0,I1) がステアリングステージに訪れる．I0 は依存する命令がないため，$i_0 = e_0$ である．続く I1 は I0 に依存するため，$i_1 = e_1|i_0 = 110...0$ と計算できる．I0 は I1 によって消費されたため I0 は親ノードである．親ノードベクトルの 0bit めを 1 にす

⋆1 例えば 16 命令の FB が各コアに 6，3，2，5 命令ずつステアリングされる場合，負荷最大コアの命令数は 6 である．

⋆2 16 は FB の最大長である．

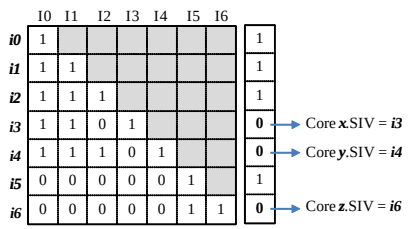

図 8 SIV 構築フェーズ

る．続く cycle t+1 では，(I2,I3) が訪れる．I2 は I1 に依存するため，$i_2 = e_2|i_1 = 1110...0$ である．同様に $i_3 = e_3|i_1 = 11010...0$ と計算できる．I1 は I2 と I3 によって消費されたため，I1 は親ノードである．親ノードベクトルの 1bit 目を 1 にする．以上のステップを続けると cycle t+3 には FB 中の全命令の解析が終了し，図 7(c) に示す先祖ノード行列と親ノードベクトルが得られる．

(2) SIV 構築フェーズ

本フェーズでは構築した先祖ノード行列をもとに SIV を完成させる．本ステアリングアルゴリズムでは SIV はリーフノードの先祖ベクトルと等しい．ここで，リーフノードとは子ノードを持たないノードのことであるから，親ノードベクトルが 0 であることによって特定できる．**図 8** は先の例における本フェーズの動作の様子である．親ノードベクトルの 3，4，6 bit 目が 0 であるため，命令 I3，I4，I6 はリーフノードである．先祖ノード行列の 3，4，6 行目を読み出し，ステアリング先の各コアに SIV として割りつける．

5. CoreSymphony アーキテクチャの評価

サイクルレベルシミュレーションにより，CoreSymphony アーキテクチャver.0.2 の性能を評価する．

5.1 評 価 環 境

評価環境を示す．シミュレータとして，独自開発のサイクルレベルシミュレータ SimMips/SS(SimMips/SuperScalar) を用いる．SimMips/SS は MIPS のシステムレベルシミュレータ SimMips[16] をサイクルアキュレートに拡張したものである．SimMips/SS は標準的なアウトオブオーダスーパースカラを模倣する．よって CoreSymphony のシミュレーションのために拡張を施した．SimMips/SS は命令セットとして MIPS32 を実装する．ただし，浮動小数点命令のアウトオブオーダースケジューリングはサポートしていない．浮動小数点命令はインオーダに実行される．以降の評価では整数ベンチマークのみを用いているが，浮動小数点命令を少量含むものが存在する．その場合には，IPC の計算に浮動小数点命令を含まない．

表 2 遅延分岐最適化をおこなう場合とおこなわない場合の比較．

Delayed-branch	(1)Binary size	(2)Executed insts.	(3)Execution cycle	(4)Rate of NOP inst.
yes	114 KB	369M	158M	0.2 %
no	122 KB	430M	153M	17.5 %

ベンチマークには「C 言語による最新アルゴリズム辞典」[15] より 5 百万命令程度の小規模な整数プログラムを 4 種[⋆1]と SPEC2006 整数ベンチマークより 4 種を用いる．SPEC ベンチマークは test データセットを用い，1 億命令をスキップした後の 1 千万命令を評価に使用する．ベンチマークプログラムのコンパイルには MIPS32 用に構築した gcc4.3.3 を利用する．ただし，遅延分岐最適化をおこなわないように gcc に手を加えている．これは，CoreSymphony 用のシミュレータが遅延分岐に対応していないためである．

遅延分岐最適化をスキップする影響を調べる．**表 2** は SPEC2006 より 462.libquantum を遅延分岐最適化をおこなってコンパイルした場合と，おこなわずにコンパイルした場合の比較である．遅延分岐最適化をおこなわない場合には遅延スロットには必ず NOP 命令が挿入される．そのため，バイナリサイズ (1) が 7%ほど増加する．また，NOP を含む実行命令数 (2) [⋆2]も 16.5%増加する．しかし，4 命令発行のスーパースカラ用に設定した SimMips/SS を用いて計測した実行サイクル数 (3) を見ると，遅延分岐最適化をスキップする影響は表れていない．アウトオブオーダプロセッサの場合，NOP 命令はフロントエンドでフィルタリングされ，命令スケジューリングに影響を与えないためである．遅延分岐最適化をしない方が実行サイクル数が短いのは，パイプラインを乱す原因となる Branch-Likely 命令[⋆3]が出現しなくなるためと考えられる．以上の結果から，遅延分岐最適化をスキップするコンパイラを利用することは，評価の妥当性に影響を与えないと考えられる．

評価に用いるパラメータは**表 3** の通りである．比較対象として用いるアウトオブオーダは (1)，(2) を発行幅と同じ値とし，各種バッファのエントリ数は協調動作時の CoreSymphony の総容量と同等とする．4 コア協調時の各バッファの総容量を () 内に併記する．

⋆1 騎士順列問題 (knight)，N-Queen 問題 (queen)，異性体の探索 (isomer)，ダイクストラのアルゴリズム (dijkstra) の 4 種である．
⋆2 以降の評価において IPC の計算に NOP は含めていない．
⋆3 分岐が不成立の場合にのみディレイスロットの命令実行をスキップする分岐命令．

表 3 評価に用いたパラメータ．全て 1 コアあたりの値．() 内は 4 コア協調時のトータルの実容量を示す．

Paramater	Value
(1) Fetch, Issue width	2
(2) Retire width	8
(3) Functional units	1 INT，1 INT/LS，1 FP
(4) Instruction window	20 entries (80 entries)
(5) Re-order buffer	80 entries (80 entries)
(6) Load/Store queue	20 entries (80 entries)
(7) Memory dependence prediction	LWT，512 entries (2K entries)
(8) Branch prediction	Bimodal，1K entries (1K entries)
(9) Branch target buffer	512 entries, 2-way (512 entries)
(10) Data-cache	4KB，2-way，2-port，blocking (16KB)
(11) Conventional-Icache	4KB, direct map (4KB)
(12) Local-Icache	4KB, 2-way (16KB)
(13) L2-cache	12 cycle hit-latency, 2MB，4-way，shared
(14) Main memory	0 cycle
(15) Network latency	1 cycle / hop

5.2 協調動作による性能の変化

図 9 は協調動作により発行幅を向上させた場合の IPC の変化をベンチマーク毎に示したものである．2-issue は 1 コア時の値を示し，4，6，8-issue はそれぞれ 2，3，4 コア協調時の値を示す．ステアリングアルゴリズムとして Modulo-2，依存ベース，リーフノードステアリングの 3 種を用いた．また，比較対象としてオリジナルのアウトオブオーダプロセッサの IPC を同時に示す．

ステアリングアルゴリズム別に結果を見ると，一部の例外を除いてリーフノードステアリングが最も高い IPC を示す．8 種のベンチマークの調和平均 (hmean) において，Modulo-2 ステアリングでは 4 コア協調時の IPC が 1 コア時の 1.23 倍，依存ベースステアリングでは 1.38 倍であったのに対し，リーフノードステアリングでは 1.40 倍の IPC を示した．協調の効果が最も大きく得られた isomer では，協調動作の効果は Modulo-2 で 2.25 倍，依存ベースで 2.43 倍，リーフノードで 2.56 倍となった．

ベースのアウトオブオーダと CoreSymphony(リーフノードステアリング) の比較により協調動作のオーバーヘッドが分かる．8 命令発行時の調和平均で CoreSymphony はアウトオブオーダに比べ 14.4%低い IPC を示した．

5.3 ライトバックのポート数と性能の関係

図 10 は他コアからのライトバックのポート数を変化させた時の IPC の変化である．ライトバックのポート数の狭幅化は，協調動作のための命令ウインドウや ROB の複雑度増

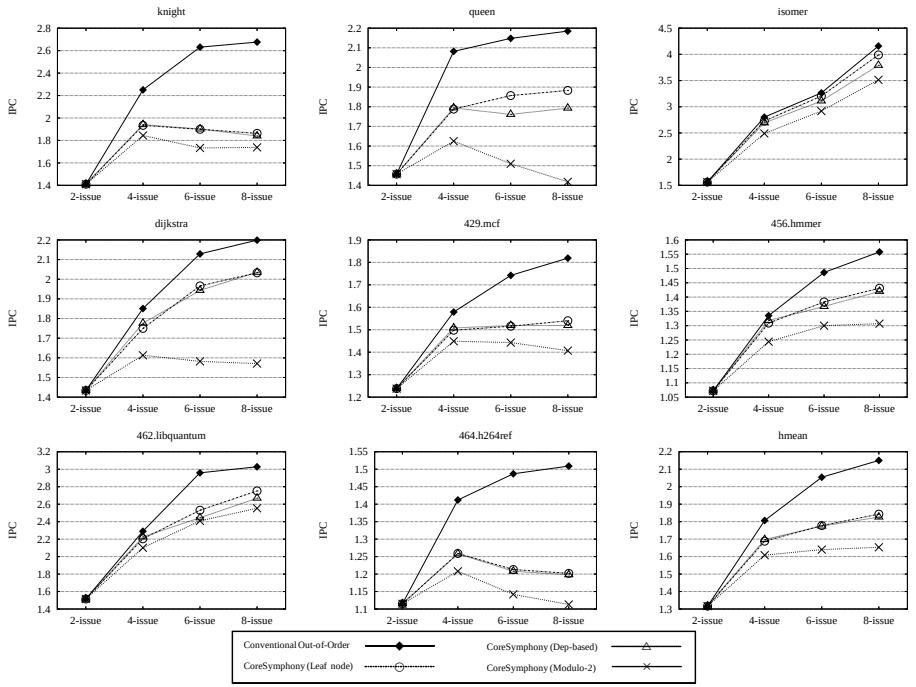

図 9 発行幅と性能の関係．CoreSymphony の 2N-issue は N コアの協調を意味する．

加を抑える効果がある．3 種のステアリングアルゴズムそれぞれについて，8 種のベンチマークの調和平均を示す．Modulo-2 ステアリングではライトバックのポート数を 3 とすると，4 コア協調時 (8-issue) の IPC 低下が 5.0%となり，3 コア以上の協調で性能向上が得られない．依存ベースステアリングでは 4 コア，3 ポートの場合の IPC 低下は 6.6%である．Modulo-2 同様，3 コア以上の協調で性能低下がみられる．これに対し，リーフノードステアリングでは 4 コア，3 ポートの場合の IPC 低下が 4.3%と 3 種のアルゴリズム中で最も小さく，3 コア，4 コアの協調でも性能向上が得られている．これは，リーフノードステアリングが最もコア間のオペランド通信を削減できるためであると考えられる．この結果から，リーフノードステアリングの導入により CoreSymphony のバックエンドの複雑度の軽減が期待できる．

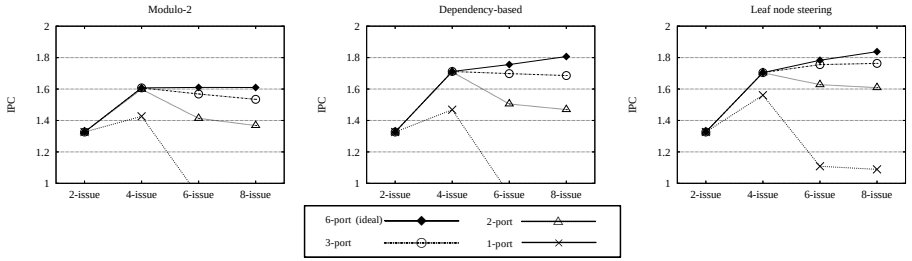

図 10 他コアからのライトバックのポート数と性能の関係.

5.4 命令キャッシュの構成と性能の関係

図 11 は 4 コア協調，リーフノードステアリングの CoreSymphony において，命令キャッシュの構成を変化させたときのフェッチ IPC とリタイア IPC の変化である．それぞれ，8 種のベンチマークにおける最大値，最小値，調和平均を示す．ローカル命令キャッシュと従来型命令キャッシュのエントリ数の割合を 1:2，3:2，8:1 と変化させた．それぞれ，命令キャッシュの合計容量[⋆1]がおよそ 1KB，2KB，4KB となるように設定した．評価の結果，フェッチ IPC，リタイア IPC ともローカル:従来型= 3:2 の場合が全ての場合で最も良い結果を示した．ローカル命令キャッシュはエントリ中に制御情報を多く含むために，従来型命令キャッシュよりも利用効率が悪い．しかし，従来型命令キャッシュは協調により実効エントリ数が増加しないのに対し，ローカル命令キャッシュは実効エントリ数が増加する．このことが，ローカル命令キャッシュに多く容量を割いた方がよいという結果の理由であると考えられる．

5.5 ハードウェアの複雑度

表 4 に CoreSymphony アーキテクチャver.0.2 のハードウェアの複雑度の見積もりを示す．5.3 節の結果からリモートのライトバックのポート数は 3 とした．比較対象として，典型的な 2-way アウトオブオーダと 8-way アウトオブオーダの値を併記する．CoreSymphony の複雑度はおおまかに述べて 2-way アウトオブオーダと 8-way アウトオブオーダの中間である．多くはベースとした 2-way アウトオブオーダの複雑度に準じているが，ROB やデスティネーションのリネーム部等，8-way と同等の複雑度を必要とする部分も存在する．これは協調時に物理レジスタを各コアで重複させるという仮定による．協調時に物理レジスタを各コアに分散させる方式の検討が今後最も重要な課題の一つであるといえる．

⋆1 ローカル命令キャッシュは制御情報も容量に含んで計算している．

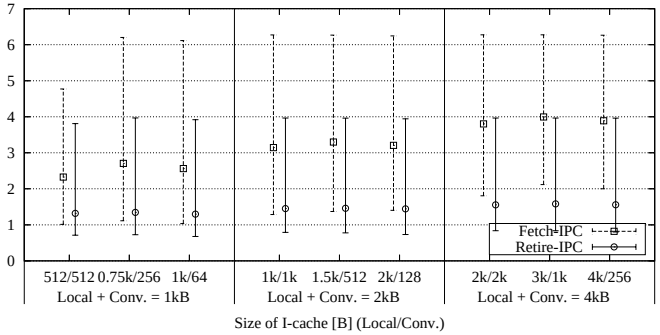

図 11 命令キャッシュの構成と性能の関係.

表 4 CoreSymphony アーキテクチャver.0.2 のハードウェアの複雑度の見積もり.

	2-way Out-of-Order	CoreSymphony	8-way Out-of-Order
Icache	2 Inst./cycle	2 Inst./cycle	8 Inst./cycle
Local-Icache	-	2 Inst./cycle	-
BTB	2-way interleave	8-way interleave	8-way interleave
Decoder	2-way	2-way	8-way
RMT	Src=4R, Dst=2R2W	Src=4R, Dst=8R8W	Src=16R, Dst=8R8W
Inst. Window	Disp/Wakeup/Issue=2/2/2	Disp/Wakeup/Issue=2/5/2	Disp/Wakeup/Issue=8/8/8
Phy. Register	4R/2W	4R/5W	16R/8W
ROB	Alloc/Commit=2/2	Alloc/Commit=8/8	Alloc/Commit=8/8
LSQ	Disp/Issue/Search=2/1/1	Disp/Issue/Search=2/1/1	Disp/Issue/Search=8/4/4
D-cache	2 port	2 port	8 port

6. 関 連 研 究

6.1 複数コアの協調により発行幅を増加させるもの

Ipek らが提案する Core Fusion[2] は，クラスタ型アーキテクチャをベースとしたコア協調機構である．2 命令発行のアウトオブオーダを最大 4 コア使用して，8 命令発行を実現する．Core Fusion は多くのクラスタ型アーキテクチャと同様，フロントエンドにおいて制御を一か所に集中しておこなう．そのため，FMU(Fetch Management Unit)，SMU(Steering Management Unit) といったコア間にまたがる制御モジュールを必要とし，融合の柔軟性，各コアの独立性が低い．

Tarjan らが提案する Federation[4] は隣接するスカラプロセッサを 2 コア使用し，2 命令発行のアウトオブオーダを構成する．優れた面積効率を有するが，協調動作には非常に密な

コア間通信を必要とするため，融合対象は隣接する特定の 2 コアに制限される．

Kim らが提案する Composable Lightweight Processors[3] はデータフロー型のプロセッサを対象としたコア協調機構である．2 命令発行の軽量なコアを最大 32 コア使用し，64 命令/cycle もの発行幅を実現する．非常にスケーラビリティの高いアーキテクチャであるが，命令セットはデータの依存関係が埋め込まれた EDGE アーキテクチャ[7] を利用する必要がある．また，性能の多くをコンパイラの高度な最適化に依存する．

6.2 命令の重複を許すステアリングアルゴリズム

Satish ら[17] は EPIC アーキテクチャ向けトレーススケジューリングアルゴリズムとして，命令の重複を許す手法を提案した．これは細粒度にクラスタ化された VLIW プロセッサ向けのアルゴリズムで，データフローグラフ中の依存ツリーを分割して各クラスタに割り当てる．依存ツリーの分割は，依存ツリーの根となる命令を複製することでおこなう．Aletà ら[18] も同様に，VLIW 向けに命令の複製をおこなう依存ツリーの分割手法を提案している．これらのアプローチは，データフローグラフの解析をコンパイル時におこなうことを想定している．そのため，アルゴリズムは高度で複雑である．現実的なハードウェアで動的に命令の複製を実現する必要があるリーフノードステアリングとは制約が異なる．

Aneesh ら[19] は，クラスタ型アーキテクチャのクラスタ間通信を削減する目的で，命令の複製をおこなうステアリング方式を提案している．この手法では，十数命令のウインドウ内の命令を Modulo-2 等の既存のアルゴリズムでステアリングした後，クラスタ間通信を発生する命令のコピーを追加でステアリングする．クラスタ間通信を発生する命令の検出方法として，ウインドウ内にのみ着目する Myopic Replication と，予測により既にディスパッチされた命令との依存も考慮する Look-ahead Replication が提案されている．本手法は，目的がクラスタ間通信を減らす事である点，動的にデータ依存解析をおこなう点でリーフノードステアリングと近い．ただし，既存のステアリングアルゴリズムへの拡張という位置づけであるため，リーフノードステアリングのようにウインドウ内におけるクラスタ間通信をゼロにすることは難しい．

7. おわりに

CoreSymphony は 2 命令発行のコアを 4 コアまで協調動作させることで最大 8 命令発行の仮想コアを構成するアーキテクチャ技術である．本稿では，我々が過去に提案した CoreSymphony アーキテクチャの実装の見直し，未実装機能の追加をおこない CoreSymphony アーキテクチャver.0.2 を定義した．この定義には，拡張されたローカル命令キャッシュやリーフノードステアリングといった，重要な要素技術が含まれる．拡張されたローカル命令キャッシュは，コア協調アーキテクチャにおける課題であるフロントエンドの分割を可能にする．リーフノードステアリングは，バックエンドにおけるコア間通信を削減し，性能向上とバックエンドのハードウェアの軽量化を可能にする．

SPEC2006 整数ベンチマークを含む 8 種のベンチマークによる評価の結果，CoreSymphony アーキテクチャver.0.2 は，4 コアの協調時に 1 コア時の 1.4 倍の IPC を達成した．8 命令発行の典型的なアウトオブオーダとの比較では，4 コア協調時の IPC 低下が 17.7%に留まることを示した．また，リーフノードステアリングによるバックエンドの軽量化についても議論し，リモートのコアからのライトバックのポート数を 6 から 3 に減少させても IPC 低下は 4.3%に留まることを確認した．

CoreSymphony ver.0.2 の実装では，物理レジスタを協調動作中の各コアで多重化するモデルを用いた．そのため，ハードウェア量のオーバーヘッドが大きい．今後は物理レジスタの分散方式を検討し，ハードウェアの軽量化をおこなう必要がある．また協調動作の効率を高め，さらなる性能向上を目指す．

参 考 文 献

1) Mark D. Hill and Michael R. Marty: Amdahl's law in the multicore era, *IEEE Computer*, 41(7), pp.33-38 (2008).
2) Engin Ipek, Meyrem Kirman, Nevin Kirman and Jose F. Martinez: Core Fusion: Accommodating Software Diversity in Chip Multiprocessors, In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA-2007)*, pp.186-197 (2007).
3) Changkyu Kim, Simha Sethumadhavan, M. S. Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger and Stephen W. Keckler: Composable Lightweight Processors, In *Proceedings of the 40th International Symposium on Microarchitecture (MICRO-2007)*, pp.381-394 (2007).
4) David Tarjan, Michael Boyer and Kevin Skadron: Federation: repurposing scalar cores for out-of-order instruction issue, In *Proceedings of the 45th annual Design Automation Conference (DAC-2008)*, pp.772-775 (2008).
5) Hongtao Zhong, Steven A. Lieberman and Scott A. Mahlke: Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications, In *Proceedings of the 13th International Conference on High-Performance Computer Architecture (HPCA-2007)*, pp.25-36 (2007).
6) 若杉 祐太, 吉瀬 謙二: メニーコアプロセッサに向けたシンプルで柔軟なコア融合機

構 CoreSymphony, 先進的計算基盤システムシンポジウム (SACSIS-2008), pp.411-419 (2008).
7) Burger, D., Keckler, S.W., McKinley, K.S., Dahlin, M., John, L.K., Lin, C., Moore, C.R., Burrill, J., McDonald, R.G. and Yoder, W: Scaling to the end of silicon with EDGE architectures, *IEEE Computer*, 37(7), pp.44-55 (2004).
8) Eric Rotenberg, Jim Smith and Steve Bennett: Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching, In *Proceedings of the 29th Annual International Symposium on Microarchitecture(MICRO-1996)*, pp.24-34 (1996).
9) Adi Yoaz, Mattan Erez,Ronny Ronen and Stephan Jourdan: Speculation Techniques for Improving Load Related Instruction Scheduling, In *Proceedings of the 26th International Symposium on Computer Architecture (ISCA-1999)*, pp.42-53 (1999).
10) Victor V. Zyuban and Peter M. Kogge: Inherently Lower-Power High-Performance Superscalar Architectures, *IEEE Transactions on Computers*, Vol.50, No.3, pp.268-285 (2001).
11) Rajeev Balasubramonian, Sandhya Dwarkadas and David H. Albonesi: Dynamically Managing the Communication-Parallelism Trade-off in Future Clustered Processors, In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA-2003)*, pp.275-286 (2003).
12) Simha Sethumadhavan, Franziska Roesner, Joel S. Emer, Doug Burger and Stephen W. Keckler: Late-binding: enabling unordered load-store queues, In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA-2007)*, pp.347-357 (2007).
13) David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III and Anant Agarwal: On-Chip Interconnection Architecture of the Tile Processor, *IEEE Micro*, 27(5), pp.15-31 (2007).
14) Tilera Corporation, Tile Processor Architecture, Technology Brief, http://www.tilera.com/technology/technology.php.
15) 奥村 晴彦: C 言語による最新アルゴリズム事典, 技術評論社 (1991).
16) 藤枝直輝, 渡邉伸平, 吉瀬謙二： SimMips: 教育・研究に有用な Linux が動く 5000 行の MIPS システムシミュレータ, コンピュータシステム・シンポジウム (ComSys2008) 論文集, pp.143-150 (2008).
17) Satish Narayanasamy, Hong Wang, Perry Wang, John Shen and Brad Calder: A Dependency Chain Clustered Microarchitecture, In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS-2005)*, pp.21-30 (2005).
18) Alex Aletà, Josep M. Codina, Antonio González and David R. Kaeli: Instruction replication for clustered microarchitectures, In *Proceedings of the the 36th Annual International Symposium on Microarchitecture (MICRO-2003)*, pp.326-338 (2003).
19) Aneesh Aggarwal and Manoj Franklin: Instruction Replication for Reducing Delays Due to Inter-PE Communication Latency, *IEEE Transactions on Computers vo.54, No.12*, pp.1496-1507 (2005).