

協調可能スーパースカラ CoreSymphony

若杉 祐太^{†1,*1} 坂口 嘉一^{†1} 吉瀬 謙二^{†1}

CMP の逐次性能の向上, および逐次性能と並列性能のバランスを目的として, スーパースカラの協調動作を実現する CoreSymphony アーキテクチャを提案する. 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, より発行幅の広い仮想コアを形成するアーキテクチャ技術である. 本論文では, 協調可能スーパースカラを実現するための課題を明確化する. そして, 各課題を克服するためにいくつかの要素技術を提案し, それらを組み合わせた CoreSymphony アーキテクチャを定義する. SPEC2006 ベンチマークによる評価の結果, 4 コアの協調により, 1 コア時と比較して 1.88 倍の IPC が得られることが分かった.

CoreSymphony: A Cooperative Superscalar Processor

YUHTA WAKASUGI,^{†1,*1} YOSHITO SAKAGUCHI^{†1}
and KENJI KISE^{†1}

This paper presents CoreSymphony, a cooperative superscalar processor architecture to improve sequential performance in Chip Multi-Processors. CoreSymphony enables some narrow-issue cores to fuse into one wide-issue core. In this paper, we clearly show the problems to be resolved for cooperative superscalar processor. Then, we propose some techniques to overcome the problems, and define “CoreSymphony architecture” which combines the techniques. Our evaluation result using SPEC2006 benchmark shows that 4-way symphony achieves 88% higher IPC than an individual core.

^{†1} 東京工業大学大学院情報理工学研究所

Graduate School of Information Science and Engineering, Tokyo Institute of Technology

^{*1} 現在, 株式会社ソニー・コンピュータエンタテインメント

Presently with Sony Computer Entertainment Inc.

1. はじめに

CMP (Chip Multi-Processor) 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, 複数スレッドを複数コアで並列実行することで性能向上を得るアプローチである. 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¹⁾.

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²⁾. 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は依然重要な課題であるといえる.

プロセッサの逐次実行能力を高める一般的な方法は発行幅を増加させることである. 本論文では, 複数個のスーパースカラ・プロセッサコアの協調により発行幅の広い仮想コアを形成する CoreSymphony アーキテクチャを提案する. CoreSymphony は 2 命令発行のアウトオブオーダーをベースとし, 最大で 4 コアの協調動作をハードウェアでサポートする. 4 コアの協調時には 8 命令発行の仮想コアを形成する. 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, 逐次処理の高速化を強力に支援する.

CoreSymphony を実装した CMP の構成を図 1 (a) に示す. 協調動作のためのハードウェアを実装した 4 つのコアと共有 L2 キャッシュを持つ. 協調動作のためのネットワークにより, コアどうしを接続する. 図 1 (b) は 2 コアの協調, (c) は 4 コアの協調により特定のスレッドの逐次性能を高めた構成である. コアの構成は動的に変更可能である. そのため, CoreSymphony を実装した CMP は逐次性能と並列性能を非常に高いバランスで両立させることが可能になる.

本研究の貢献を以下にまとめる:

- 次世代の CMP として非常に有望な協調可能スーパースカラの実現に向けて, マイクロアーキテクチャにおける課題を明確にする.

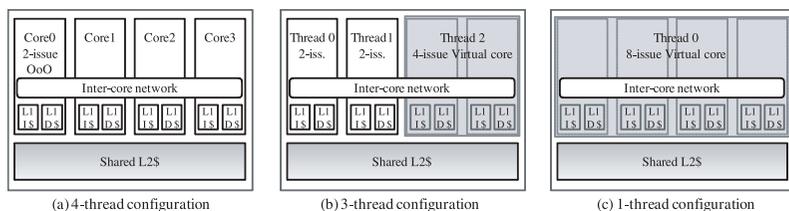


図 1 CoreSymphony アーキテクチャにおける実行時のさまざまな構成例。(a) 4 スレッド実行可能な構成。(b) 3 スレッド実行可能な構成例。(c) 4 コア協調による 1 スレッドを高速に実行可能な構成

Fig.1 Example of CoreSymphony configurations. –(a) 4-thread configuration. (b) One of the 3-thread configurations with 2-way symphony. (c) 1-thread configuration with 4-way symphony.

- それぞれの課題を克服するためにいくつかの要素技術を提案し、それらを組み合わせた CoreSymphony アーキテクチャを定義する。そして、協調可能スーパスカラの可能性を示す。

CoreSymphony は、発行幅の広いスーパスカラを複数個の軽量なスーパスカラに分割する技術であることとらえることもできる。そのため、本論文で提案する技術の多くは、発行幅の広いスーパスカラが必要とする複雑度の高いモジュールを、複数個の軽量なモジュールで置き換えるためのものである。よって、本研究によって得られる知見は面積効率と広い発行幅を両立させるスーパスカラの実現に向けて有用であるといえる。

本論文の構成を示す。2 章では関連研究についてまとめる。3 章では CoreSymphony の基本方針とランドデザインについてまとめる。4 章ではスーパスカラの協調動作を実現するにあたり、課題となる点を明確化する。5 章では CoreSymphony のマイクロアーキテクチャについて詳しく述べる。6 章ではシミュレーションにより CoreSymphony の性能を評価し、最後に 7 章で結論と今後の課題を述べる。

2. 関連研究

本章では、CoreSymphony の関連研究についてまとめる。

2.1 SMT アーキテクチャ

プロセッサの逐次処理性能と並列処理性能のバランスを変化させるという点において、同時マルチスレッディング (Simultaneous Multi-Threading: SMT) アーキテクチャ³⁾ と CoreSymphony は一部の目的を共有する。しかし、アプローチは大きく異なる。SMT は比較的大規模なコアをベースとし、単一コアで 1 つまたは複数のスレッドを実行する。一方、

CoreSymphony は 2 命令発行の小規模なコアの集合をベースとし、複数コアで 1 つまたは複数のスレッドを実行する。消費電力や設計の複雑度といった問題から、複雑度を抑えたコアを多数利用する方向へ向かいつつある現状では、CoreSymphony のアプローチは重要度の高い研究領域であるといえる。

2.2 クラスタ型アーキテクチャ

スーパスカラのバックエンドを複数のクラスタに分割するクラスタ型アーキテクチャ^{4),5)} は重要な関連研究の 1 つである。CoreSymphony との違いは、フロントエンドが集中化されていることである。CoreSymphony はスーパスカラをコアのレベルまで完全に分割し、制御の集中化を行わない。これにより、設計のモジュール化や不良コアの縮退といった CMP の利点を得ることを目指す。しかし、スーパスカラのフロントエンドは本質的に制御駆動であるため分割が難しい。そのうえ、フロントエンドには RMT (Register Map Table) のような非常に多ポートで遅延の大きいモジュールが存在する。フロントエンドの非集中化は CoreSymphony において最も挑戦的な課題の 1 つである。

2.3 コア融合アーキテクチャ

ここでは、複数個のコアの協調動作により、強力な仮想コアを構成するものをコア融合アーキテクチャと呼ぶ。CoreSymphony もコア融合アーキテクチャに属する。

Core Fusion⁶⁾ は CoreSymphony 同様、アウトオブオーダーをベースとしたコア融合機構である。2 命令発行のコアを最大 4 個融合させて 8 命令発行を実現する。Core Fusion はクラスタ型アーキテクチャと非常に近い構成をとる。そのため、多くのクラスタ型アーキテクチャと同様、ステアリング、リネームといったフロントエンドの制御を集中化して行う必要がある。

Anaphase^{7),8)} はコンパイラのサポートによりコア融合を実現する。逐次プログラムをコンパイル時に細粒度のマルチスレッドプログラムに分割し、複数コアで実行する。各コアはメモリアクセスのコヒーレンスを保つための特別な HW を備える。コンパイラのサポートを利用することで HW の変更を小規模に抑え、高い面積効率を実現している。これに対し、CoreSymphony はワークロードの状況に応じて適応的にコア融合を行うことを目標とするため、パイナリを変更することは望ましくない。よって、HW のみでのコア融合実現を目指す。

Composable Lightweight Processors⁹⁾ はデータフロー型の命令セット¹⁰⁾ を利用するプロセッサ、Voltron¹¹⁾ は VLIW 型のプロセッサを対象とするコア融合機構である。スーパスカラを対象とする CoreSymphony とは方向性が異なる。

3. CoreSymphony のグランドデザイン

本章では、CoreSymphony の基本方針について述べ、本研究が目標とする協調可能スーパスカラのグランドデザインを示す。

3.1 CoreSymphony の基本方針

CoreSymphony はスーパスカラの協調動作による逐次性能の向上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 は比較的軽量かつ均質なコアを多数搭載する CMP への適用を念頭におく。これは、現在のプロセッサアーキテクチャのトレンドを反映しており、また CoreSymphony はこのような CMP において最も効果を発揮する。この種の CMP では、設計時のモジュラリティや不良コアの縮退可能性を高めるという点でコアの独立性が重要な意味を持つ。そのため、CoreSymphony においては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、制御を集中化して行うことは望ましくない。CoreSymphony は、フロントエンドでいっさいのコア間通信を行わず、制御を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可能性が高い。CoreSymphony はこれらの不利を現実的な範囲に収めつつ、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、一般的なアウトオブオーダーを採用し、変更はなるべく小さいものにとどめる。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 9) 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¹⁰⁾を採用することは、協調動作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、およびバイナリの連続性を維持するためである。

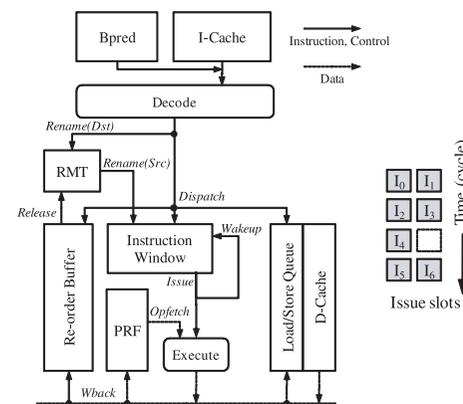


図 2 ベースのスーパスカラのブロック図

Fig. 2 Block diagram of baseline superscalar processor.

3.2 マイクロアーキテクチャの基本方針

3.2.1 ベースのスーパスカラの構成

まず、ベースとするスーパスカラの構成について述べる。図 2 は対象とするスーパスカラのブロック図である。構成方式として、物理レジスタファイル (PRF) をバックエンドで読み出す方式のアウトオブオーダー^{12),13)}を採用する。レジスタリネーミングのための RMT、コミットのインオーダー性を確保するための Re-order Buffer (ROB) を備える。Load-Store Queue (LSQ) はロード命令の投機発行を可能にする。命令供給部には標準的な分岐予測器 (Bpred) を備えるものとする。発行幅は、整数 2 命令/cycle、浮動小数点数 1 命令/cycle 程度を想定する。

3.2.2 目標とする協調可能スーパスカラの構成

次に、CoreSymphony が目標とする協調可能スーパスカラについて述べる。図 3 は協調動作を実現するスーパスカラ概念図である。複数個の 2 命令発行スーパスカラが協調動作し、あたかも発行幅の広い 1 つの仮想コアのように振る舞う。仮想コアは、命令レベル並列性を積極的に抽出し、逐次プログラムを高速化する。

協調可能スーパスカラを構成するに際し、マイクロアーキテクチャ上の基本方針を次に示す。

- 独立性の維持のため、フロントエンドは完全にコア単位に分割する。すなわち、データの授受および同期のためにコア間通信を行うモジュールをフロントエンドに持たない。

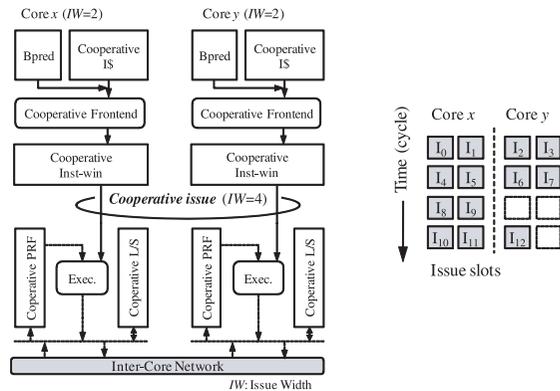


図3 協調動作を実現するスーパースカラ概念図. 2 コア協調時の例
Fig.3 Conceptual diagram of cooperative superscalar processor.

- 一次キャッシュや命令ウィンドウ, 物理レジスタといった各コアに属するモジュールは, アーキテクチャ技術により協調性を備え, 協調動作によってスレッドあたりのエントリ数を増加させる.

これらを同時に達成するのは非常に挑戦的な課題である. 一般的には, 協調動作の実現には積極的なコア間通信を必要とするが, コア間通信はコアの独立性を低下させる要因となりやすいためである. 次章では, これらの基本方針を達成しつつ協調動作が可能なスーパースカラを実現するうえで何が課題となるかを明確にする.

4. CoreSymphony の実現のための課題

本章では, 3章に示す方針に基づき, 協調可能スーパースカラを構築する際の課題を述べる. その際, 発行幅の広いスーパースカラ(協調時を想定)を複数個の発行幅の狭いスーパースカラに分割するという視点から説明を行う場合がある. また, 各課題に対して本論文で提案する要素技術が, どのようなアプローチをとるかを簡潔にまとめる.

4.1 課題1 命令フェッチ

フェッチステージにおける課題は次のとおり.

- 分散可能命令キャッシュを構築すること
協調により命令キャッシュの容量を増加させるためには, プログラムを各コアの命令キャッシュに分散して格納する必要がある.

- 後段でデータ依存関係を解決するために十分な情報を供給すること
フロントエンドでのデータの授受を行わないという方針から, リネームステージ以降に必要な依存関係の情報は自コアのフェッチステージから供給する必要がある.
- 協調中の全コアの制御フローを同期させること
CoreSymphony では, 協調動作中のコア群は1つのスーパースカラのように動作する. すなわち, すべてのコアは共通の制御フロー上の命令をフェッチする必要がある.

(a), (b)の課題に対して, 本論文ではローカル命令キャッシュを提案する. CoreSymphony では, 協調中のコア数 $\times 4$ 命令を最大長とする命令トレースを1単位としてフェッチ/ステアリングを行う. この命令トレースをフェッチブロック(以降FBと表記する)と呼ぶ. 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間の依存関係の解決に用いる制御情報を格納するステアリング済み命令トレースキャッシュである.

(c)に関しては, 分岐予測および, 投機ミスや例外に関する処理を全コアで多重化して実行することで対処する.

4.2 課題2 命令ステアリング

命令ステアリングは, クラスタ型アーキテクチャにおいて, どの命令をどのクラスタで実行するかを決定する操作である. CoreSymphony の場合, ステアリング先はクラスタではなくコアとなる. ステアリングステージにおける課題は次のとおり.

- CoreSymphony の性能を引き出すアルゴリズムの採用
CoreSymphony は, コア間のオペランドフォワードリングのレイテンシが大きい. そのため, 命令ステアリングにおいては依存関係にある命令を同じコアに割り当てることが重要となる. 一方で, あるコアに対する負荷の集中は命令ウィンドウにおけるSelectレイテンシの増加につながる. そのため, ステアリングステージにおいては, 依存関係にある命令をまとめる, 負荷を均等にする, という一見相反する要求を同時に満たすステアリングアルゴリズムを構築することが重要な課題となる.
また, CoreSymphony 特有の制約として次の条件も同時に満たさなければならない.
 - ステアリング済み命令トレースキャッシュを採用するため, ステアリングはトレース(FB)内で行う. また, あるトレースのステアリング結果は毎回同じである.
 - 後述する2-wayリネーミングのための制約として, FB内ではコア間の依存をいっさい発生させない.

この課題に対して, 本論文では複数の命令の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で, CoreSymphony の制約条件を満たしつつ, ある程度良い性能が得

られるリーフノードステアリングを提案する．

4.3 課題 3 レジスタリネーミング

リネームステージにおける課題は次のとおり．

- (a) 協調による物理レジスタ空間の拡大のサポート
協調により物理レジスタのエントリ数を増加させるためには，リネーム機構は対象とする物理レジスタ空間の拡大をサポートしなければならない．
- (b) 効率的な RMT の実装
一般的な 3 オペランド方式の RISC では，RMT のポート数は way 数の 4 倍にもなり遅延が大きい．さらに，RMT は命令間の依存関係を表すタグを一元管理するという性質上，何らかの形で集中化を必要とする．そのため，スーバスカラをベースとするコア融合アーキテクチャにとって RMT の分割は非常に困難な課題となる．さらに，CoreSymphony ではフロントエンドで通信を行わないという方針により，問題はますます困難になる．

(a) の課題に対して CoreSymphony は，ローカルの物理レジスタ空間とそれを管理するローカルな RMT をコアごとに持つという手法をとる．この手法は，コア数の増加にともない，物理レジスタ空間をスケラブルに拡大することができる反面，コア間にまたがる依存関係を表現できないという問題がある．そのため，CoreSymphony では，トレース (FB) 内ではコア間にまたがるデータ依存を発生させないという制約を設ける．これは命令ステアリングに関する制約である．これにより，コア間で発生する依存関係を FB 間のみ限定することが可能になる．コア間の依存関係は通常の RMT とは異なる特殊な RMT で別に管理する．このハードウェアは実装の工夫により軽量に作ることが可能で，(b) の課題を同時に解決することができる．この 2 種類の RMT を利用するリネーム機構を 2-way リネーミングと呼ぶ．

4.4 課題 4 物理レジスタの構成

物理レジスタにおける課題は次のとおり．

- (a) 分散可能物理レジスタを構築すること
協調により物理レジスタのエントリ数を増加させるためには，ステートを各コアの物理レジスタに分散して格納する必要がある．また，ポート数，エントリ数の面から，各コアが保有する物理レジスタは全体のサブセットとすることが望ましい．このような物理レジスタの分散は，クラスタ型アーキテクチャや Core Fusion において

実現されている．しかしそれは，ある論理レジスタに対してアーキテクチャステート^{*1}を与える物理レジスタがどのクラスタに属するかをフロントエンドで集中管理しているためである．よって，フロントエンドを分割する場合は，物理レジスタの分割はさらなる工夫が必要である．

フロントエンドを完全に分割する CoreSymphony では，それぞれの命令の結果を，明示的に各コアの物理レジスタに分散して格納することは難しい．そのため，あるコアで生成された値は全コアで共有する結果バスに放送される．そして，各コアはステート管理に基づくフィルタリングにより，必要な結果のみを自身の物理レジスタに取り込む．この際に，物理レジスタは必要な分だけが束縛され，自コアの物理レジスタ空間に新たなマッピングを作成する．この機構を CoreSymphony 向け物理レジスタ分散手法と呼ぶ．

4.5 課題 5 ロード/ストアユニット

ロード/ストアユニットにおける課題は次のとおり．

- (a) 分散可能データキャッシュおよび LSQ を構築すること
協調によりデータキャッシュや LSQ のエントリを増加させるために必要な課題である．この課題に対して，データキャッシュおよび LSQ を実効アドレスでバンク分けするアプローチ^{5),6)}は，LSQ 内のストアデータフォワーディングおよび，メモリあいまい性除去を局所化できるという点で都合が良い．しかし，各命令がアクセスするバンクは実効アドレスの計算後に判明する．このため，ステアリング時にメモリバンク予測¹⁵⁾を行う手法が採用されることが多いが，バンク予測ミスの取扱いが煩雑になるという欠点がある．

CoreSymphony では Simha らによって提案された Unordered Late-Binding Load/Store Queue (ULB-LSQ)¹⁶⁾を用いる．この LSQ はエントリの確保を実効アドレスの計算後に行うことができる．よって，バンク予測を行う必要がない．実効アドレスの計算後に，当該バンクを有するコアの ULB-LSQ のエントリを確保し命令をディスパッチする．リモートのコアへロード/ストア命令をディスパッチするためのコア間ネットワークを用意する．

4.6 課題 6 コミットの処理

コミットの処理における課題は次のとおり．

- (a) 投機ミスおよび例外に関する情報を共有すること
全コアが同一の制御フローをたどるためには，全コアが投機ミスに対して同一の対応

*1 命令の発行済みや完了済みに関係なく，各論理レジスタに対する最新の代入操作から構成されるステート¹⁴⁾．

を行う必要がある。

(b) インオーダーステート^{*1}を分散して管理すること

インオーダーステートは投機ミスおよび例外からの回復時にすべてのコアが必要とする。しかし、これを重複して管理することはレジスタファイルのHW量の面から現実的ではない。

(a) に関しては、CoreSymphony では分岐予測の正否等の投機ミスに関する情報、および発生した例外の種類を全コアのROBに重複して保存することで対処する。(b)のインオーダーステートの分散管理に関しては、検討が不十分であるため本論文では実装を見送る。暫定的な実装として、物理レジスタファイルとは別に、確定済みの値を格納する論理レジスタファイル(Logical Register File: LRF)を用意する。コミット時に物理レジスタファイルから全コアの論理レジスタファイルに値をコピーする。すなわち、インオーダーステートは全コアが重複して保有するものとする。

5. CoreSymphony アーキテクチャの実装

本章では、CoreSymphonyの実装の詳細をまとめる。特に、4章で示した課題を克服するための、重要な要素技術について述べる。

5.1 課題1 命令フェッチ：ローカル命令キャッシュの提案

命令フェッチステージのブロック図を図4に示す。従来型の命令キャッシュ(Conventional Icache)、分岐予測器、ステアリング済み命令トレースキャッシュであるローカル命令キャッシュ(Local-Icache)を備える。以降、ベースのアウトオブオーダーとの相違を中心に詳細を述べる。

5.1.1 分岐予測器

CoreSymphonyは、協調動作中の全コアで分岐予測を多重化して行う。これはすべてのコアが、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。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サイクルで行う必要がある。そのため分岐先バッファ(BTB)は8-wayのインタリーブ構成をとる。分岐予測器のスループットは1予

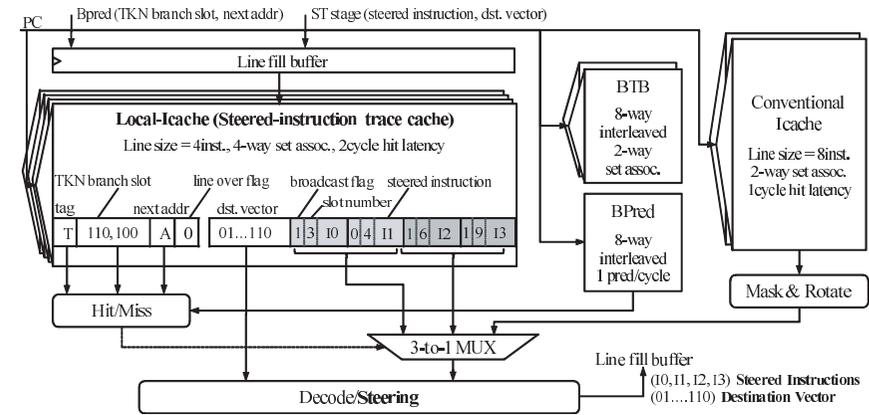


図4 CoreSymphonyの命令フェッチステージのブロック図

Fig. 4 Instruction fetch stage of CoreSymphony.

測/cycleである。

5.1.2 ローカル命令キャッシュ

CoreSymphonyは、協調中のコア数×4命令を最大長とする命令トレース(フェッチブロック:FB)を1単位としてフェッチ/ステアリングを行う。ローカル命令キャッシュは、FB中の自身にステアリングされた命令と、FB間の依存関係を表す情報を格納するトレースキャッシュである。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。ローカル命令キャッシュを用い、FBのフェッチを分割する仕組みを図5に示す。図5は2コア協調時に $(I_0, I_1, \dots, I_7)^{*3}$ の8命令からなるFBをフェッチする例である。FBを分割してフェッチするためには2つのフェーズを経る必要がある。図5(a)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のみ行われる。従来型命令キャッシュからFB中の全命令を読み出し、デコードとステアリングを行う。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各種の制御情報を書き戻す。図5(b)に示す協調フェッチフェーズは、ローカル命令キャッシュにヒットする場合である。

*1 非投機状態にある命令の、各論理レジスタに対する最新の代入操作によって構成されるステート¹⁴⁾。

*2 4コア協調時の最大の発行幅が8であるため。

*3 I_n は1つの命令を表す。また、あるFBを (I_0, I_1, \dots, I_n) と表す場合がある。

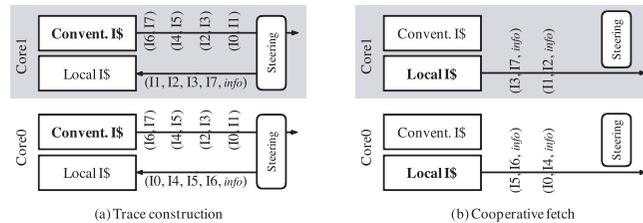


図5 ローカル命令キャッシュによる命令フェッチの分割。(a) ローカル命令キャッシュのエントリ構築フェーズ，(b) 協調フェッチフェーズ

Fig. 5 Cooperative instruction fetch mechanism with Local-Instruction cache. –(a) Trace construction phase, (b) Cooperative fetch phase.

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，フロントエンドのスループットは N コアで N 倍となる。

ローカル命令キャッシュは，自身にステアリングされた命令のみを保存するという性格から，協調動作によってスレッドあたりの実エントリ数を増加させることができる。しかし，トレースキャッシュと同様の理由で，エントリの利用効率が悪い。そのため，従来型命令キャッシュとローカル命令キャッシュのサイズの比は重要な設計パラメータである。これについては6章で評価する。

ローカル命令キャッシュのエントリは次に示すフィールドを持つ。すべてのフィールドを合計すると，ローカル命令キャッシュの1ラインは253 bitとなる。命令部は128 bitであるため，制御情報が占める割合は49%になる。これら制御情報を構成する125 bitのうち，TKN branch slotとnext addrが占める40 bitはトレースキャッシュを構成するために必要な情報である。残りの85 bitがローカル命令キャッシュに特有の情報である。これは，キャッシュラインの34%に相当する。すなわち，通常のコマンドキャッシュと比較して，ローカル命令キャッシュを採用することによるハードウェア量の増加は34%程度となる。一般に，プロセッサコアに占める1次命令キャッシュの割合は1割から2割程度である^{12),17),18)}。このため，ローカル命令キャッシュを採用することによるプロセッサ全体のハードウェア量の増加は3.4%~6.8%程度となる。このハードウェア量の増加は好ましくはないが，プロセッサ全体のコストに深刻な影響を与えるというわけではない。

- TKN branch slot (4×2 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。

- next addr (32 bit)
当該FBにおいて2つめの基本ブロックの開始アドレスを示す。TKN branch slotとnext addrフィールドが，当該FBに対する2回の分岐予測結果とマッチしたときのみ，ローカル命令キャッシュはヒットと判定される^{*1}。
- line over flag (1 bit)
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。
- destination vector (64 bit)
当該FB中の命令のデスティネーションレジスタの位置を示す，論理レジスタ数と同長のベクトル。 n ビット目が1であることは，論理レジスタ R_n に結果を書き込む命令がFB中に存在することを示す。
- steered instruction (32×4 bit)
自身に割り当てられた命令を格納するフィールド。
- broadcast flag (1×4 bit)
各命令が，協調中の各コアに結果を放送する必要があるか否かを示すフラグ。
- slot number (4×4 bit)
各命令が，当該FB中プログラム順で何番めに位置するかを示す。

5.2 課題2 命令ステアリング：リーフノードステアリングの提案

CoreSymphonyにおいて，コア間のオペランド通信はレイテンシの観点から性能に与えるオーバーヘッドが大きい。また，ステアリング済みトレースキャッシュを採用することにより，あるFBのステアリング結果を実行のたびに変わることができないという制約が生まれる。これは，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良く実現するステアリング方式が特に重要になる。本節では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，効率的な実装方法をまとめる。

5.2.1 提案するアルゴリズム

コア間通信の抑制とFB内の負荷分散を両立させ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事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。これらの相反する2つの要求を満

*1 分岐予測器のスループットは1予測/cycleであるため，ヒット/ミス判定には2サイクルを要する。

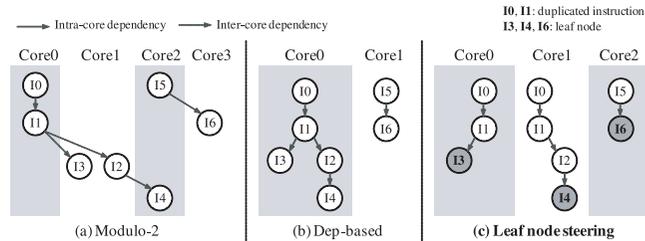


図 6 3 種のステアリングアルゴリズムによる、ある FB のステアリング結果
Fig. 6 Example of three steering algorithms.

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行う。図 6 は (I_0, \dots, I_6) 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ふ。

(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^{*1}に着目して行われる。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^{*2}となる命令をすべて、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、FB 内のコア間通信を発生させず、ある程度良い負荷分散が期待できる。FB 内のコア間通信がいっさい発生しないという特徴は、5.3 節で述べるリネームロジックの軽量化にも貢献する。ただし、命令の重複により負荷の総量は増加する。命令の重複度については 6 章で評価する。

5.2.2 リーフノードステアリングの実装

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。本項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行う。1 つめのフェーズでは、先祖ノード行列 (Ancestor-node matrix) とリーフノードベクトル (Leaf-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 つめのフェーズでは構築したテーブルをもとにステアリング結果を確定する。それぞれのフェーズについて例をあげて解説する。

(1) 先祖ノード行列、リーフノードベクトル構築フェーズ

まず、先祖ノード行列 ANC を定義する。先祖ノード行列は FB 中の命令 I_i に対し命令 I_j が先祖に含まれるか否かを示す 16×16 bit のビット行列^{*3}である。以降、先祖ノード行列の i 行 j 列の要素を $ANC[i][j]$ と表す。 $ANC[i][j]$ は以下のように定義される。 $ancestor(I_i)$ は I_i の先祖の集合を表す。

$$ANC[i][j] = \begin{cases} 1 & \text{if } I_j \in ancestor(I_i) \\ 0 & \text{others} \end{cases}$$

ANC の i 行めは I_i の先祖となる命令群の位置を表す。これを I_i の先祖ベクトルと呼び、 $ANC[i]$ で表す。命令 I_i が命令 I_j と I_k に真のデータ依存を持つ場合、 $ANC[i]$ は次のようにして計算できる。 e_i は基本ベクトル^{*4}を意味し、 $|$ はビット論理和演算を意味する。

$$ANC[i] = e_i | ANC[j] | ANC[k]$$

次に、リーフノードベクトル L を定義する。リーフノードベクトルは FB 中の命令 I_i がリーフノードであるかを示す 16 bit のベクトルである。リーフノードベクトルの i ビットめの要素 $L[i]$ は I_i がリーフノードであるか否かを表す。

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。先祖ノードベクトル、リーフノードベクトルは一部の要素のみを示し

*1 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。

*2 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、 I_3 の先祖は I_3, I_1, I_0 である。

*3 16 は FB の最大長である。

*4 i ビットめのみが 1 であるベクトル。

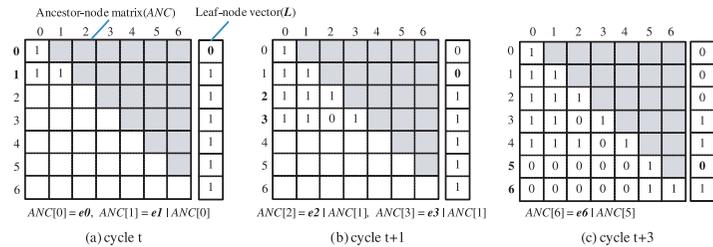


図 7 (1) 先祖ノード行列，リーフノードベクトル構築フェーズ
Fig. 7 (1) Construction of ancestor matrix and leaf-node vector.

ている．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．リーフノードベクトルは全要素を 1 で初期化しておく．まず cycle t では (I_0, I_1) がステアリングステージに訪れる． I_0 は依存する命令がないため， $ANC[0] = e_0$ である．続く I_1 は I_0 に依存するため， $ANC[1] = e_1 | ANC[0]$ と計算できる．ここで， I_0 は I_1 によって消費されたため I_0 はリーフノードではない．なぜならば，リーフノードとは DFG 上で子ノードを持たないノードであるためである．よって， $L[0] = 0$ とする．続く cycle t+1 では， (I_2, I_3) が訪れる． I_2 は I_1 に依存するため， $ANC[2] = e_2 | ANC[1]$ である．同様に $ANC[3] = e_3 | ANC[1]$ と計算できる． I_1 は I_2 と I_3 によって消費されたため， I_1 はリーフノードではない． $L[1] = 0$ とする．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リーフノードベクトルが得られる．

(2) ステアリング結果確定フェーズ

本フェーズでは構築した先祖ノード行列をもとにステアリング結果を確定する．本ステアリングアルゴリズムは，リーフノードに対する先祖を 1 つのまとまりとして，コアに割り当てる．すなわち， $L[i] = 1$ となる i について先祖ノード行列の第 i 行を読み出すことで，同じコアにステアリングする命令群が得られる．図 8 は先の例における本フェーズの動作の様子である．リーフノードベクトルの 3, 4, 6 bit 目が 1 であるため，命令 I_3, I_4, I_6 はリーフノードである．先祖ノード行列の第 3, 4, 6 行を読み出しステアリング結果とする．ステアリング結果に基づき，命令キューから自コアの命令のみを読み出してバックエンドに送ることでステアリングが完了する．

5.2.3 リーフノードステアリングの関連研究

命令を複製してステアリングするという手法がいくつか提案されている．Narayanasamy ら¹⁹⁾ 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

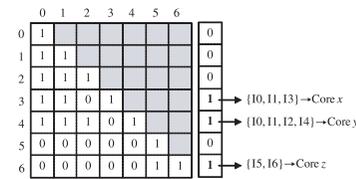


図 8 (2) ステアリング結果確定フェーズ
Fig. 8 (2) Steering with two tables.

重複を許す手法を提案している．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行う．Aletà ら²⁰⁾ も同様に，VLIW 向けに命令の複製を行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行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．

Aggarwal ら²¹⁾ 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，命令の複製を行うステアリング方式を提案している．この手法では，十数命令のウィンドウ内の命令を 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，ウィンドウ内のみ着目する Myopic Replication と，予測によりすで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ことである点，動的にデータ依存解析を行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ィンドウ内 (=FB 内) におけるクラスタ間通信をゼロにすることは困難である．リーフノードステアリングは FB 内のコア間通信をいっさい発生させない．この特徴は 5.3 節で述べるリネームロジックの軽量化に必要な不可欠である．

5.3 課題 3 レジスタリネーミング：2-way リネーミングの提案

レジスタリネーミングの主要ハードウェアである RMT は，非常にポート数が多く遅延の大きいモジュールである．本節では協調時の発行幅の広いスーパスカラが必要とする多ポートの RMT を，2 種類の小規模な RMT に置き換えるための 2-way リネーミングを提案する．本手法は，物理レジスタの分散に不可欠な物理レジスタ空間の拡大も同時に実現す

ることができる。

5.3.1 基本のアイデア

リネームステージの役割は、(1) 論理レジスタ番号から物理レジスタ番号への対応を与えること、(2) ある命令が依存する命令を表すタグを与えること、の2つである。一般的なアウトオブオーダーでは、タグ=物理レジスタ番号であるため、(1)と(2)は同義である。ここで、(2)のタグはその性質上グローバルな管理を必要とする。これにより、アウトオブオーダーにおけるRMTは集中化の必要が生じる。集中化はポート数が爆発する原因となる。しかし、(1)と(2)の役割は独立しているため、必ずしもタグ=物理レジスタ番号でなくてもよい。2-wayリネームはこの観察に基づく。

2-wayリネームの本質は、コア内の依存とコア間の依存を異なるタグで表現し、別々のテーブルで管理することにある。グローバルに管理する必要があるのはコア間の依存のみである。そこで、コア間の依存に関しては物理レジスタ番号をタグとして用いず、より圧縮された情報を用いる。グローバルな情報を管理するテーブルは全コアでコピーを持つ必要があるが、圧縮された情報を用いるため、ポート数を抑える効果が期待できる。ここで、コア内の依存を表すタグをLocal tag (Ltag)、コア間の依存を表すタグをGlobal tag (Gtag)と呼ぶ。あるリネーム前の論理レジスタ R_n に対してLtag、Gtagがとりうる値の集合は次のようになる。

$$Ltag: \{P_m : m = 0, 1, \dots, NP - 1\}$$

$$Gtag: \{R_{n,t} : t = 0, 1, \dots, NF\}$$

NP は物理レジスタの総数を、 NF はFB番号の最大値を表す。FB番号とはin-flightなFBにサイクリックに割り当てられる4ビット程度のタグである。すなわち、 FB_t^{*1} 中の命令のソースレジスタが R_n である場合、 R_n のGtagは $R_{n,t}$ で与えられる。Gtagは依存先の命令を特定せず、依存先のフェッチブロックのみを特定する。これは、コア間にまたがる依存関係を表現するための情報として十分である。なぜならば、あるFBにとってコア間通信を発生させる可能性があるデスティネーションレジスタは、各論理レジスタについてたかだか1個であるためである。このことは、リーフノードステアリングが、当該FB内で依存関係にある2命令を必ず同じコアにステアリングすることによって保証される。

*1 FB番号が t であるFBを FB_t と表す。

5.3.2 2-wayリネームの実装

LtagとGtagは別々のテーブルで管理される。Ltagを管理するテーブルをLocal RMT (LRMT)、Gtagを管理するテーブルをGlobal RMT (GRMT)と呼ぶ。LRMTは通常のRMTに相当するハードウェアである。よって、LRMTのポート数は基本的には2-wayアウトオブオーダーのRMTに準じる。ただし、各エントリを{物理レジスタ番号、生産者のFB番号}に拡張する。

一方、GRMTの構成は少々特殊である。ISAが規定する論理レジスタの本数を N 、パイプライン中に保持可能な最大FB数を M とすると、GRMTは $N \times M$ のビット行列として構成される。 i 行 j 列のビットは、 FB_j 中に論理レジスタ R_i をデスティネーションとする命令が存在するか否かを表す。すなわち、 $GRMT[i][j]$ は次のように定義される。 $dst(I_n)$ は命令 I_n のデスティネーションレジスタを表す。

$$GRMT[i][j] = \begin{cases} 1 & \text{if } \exists n (I_n \in FB_j \wedge dst(I_n) = R_i) \\ 0 & \text{others} \end{cases}$$

GRMTによるリネームは通常のRMTと同様、(a)デスティネーションの登録、(b)ソースのリネームの2ステップに分けられる。図9にGRMTの各ステップの動作を示す。(a)では各FBに付随するDestination vectorをGRMTのFB番号列に書き込む。Destination vectorはFB中の命令のデスティネーションレジスタの位置を示すベクトルで、各コアのローカル命令キャッシュにコピーが保存されているため、GRMTは全コアで同一に保たれる。(b)ではLRMTと同様に論理レジスタ番号でGRMTの行を読み出す。GRMTの第 x 行は、論理レジスタ R_x をデスティネーションとする命令を保持するFBのFB番号を与

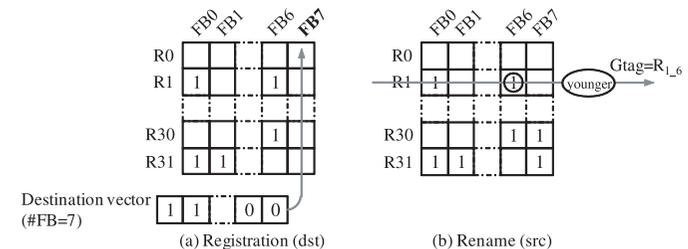


図9 GRMTの動作。(a)デスティネーションの登録、(b)ソースのリネーム

Fig. 9 Cooperative renaming with GRMT. (a) Registration of destination register, (b) Renaming of source register.

る．ここから図 9 中 younger で示すロジックにより，最も若い FB 番号 F_y を求める．結果， $G_{tag}=R_{x,y}$ としてタグ付けが完了する．GRMT のポート数について述べる．デスティネーションの登録には書き込みポートが 1 ポート（列方向の Write）必要である．ソースのリネームのための読み出しポート（行方向の Read）数は，命令のソースオペランドの個数とサイクルあたりにリネームする命令数の積となる．想定している CoreSymphony アーキテクチャでは，命令のソースオペランドの個数を 2，サイクルあたりにリネームする命令数を 2 としているため，ソースのリネームのための読み出しポート数は 4 となる．このように，GRMT は書き込みに 1 ポート，リネームのための読み出しに 4 ポートと比較的少ないポート数で構成できる．ただし，本論文では後述する CoreSymphony 向け物理レジスタ分散手法において GRMT を拡張し，3 ポートの読み出しポートを追加している．

2-way リネーミングでは Ltag と Gtag の 2 種類のタグが生成されるが，実際にスケジューリングに使用するのはどちらか一方である．LRMT の参照時に同時に読み出した FB 番号と，Gtag に含まれる FB 番号を比較し，FB 番号が若い（プログラム順で新しい）方のタグを採用する．採用されなかった方のタグは invalid とする．

5.3.3 2-way リネーミング向け命令ウィンドウの構成

命令スケジューリングには，一般的な連想方式の命令ウィンドウを用いる．ただし，Ltag と Gtag の 2 種類のタグを扱うために構成を変更する．図 10 に，CoreSymphony の命令ウィンドウの 1 エントリを示す．命令ウィンドウのスケジューラ部は次のエントリを持つ．左オペランドのフィールドを添え字 l ，右オペランドのフィールドを添え字 r で区別する．

- LR_l/LR_r
Ltag フィールドが Rdy あるいは invalid であることを示す 1 bit のフィールド．
- $Ltag_l/Ltag_r$
Ltag フィールド．CAM で構成される．
- GR_l/GR_r
Gtag フィールドが Rdy あるいは invalid であることを示す 1 bit のフィールド．
- $Gtag_l/Gtag_r$
Gtag フィールド．CAM で構成される．

Ltag フィールドは自コアで実行された先行命令によって wakeup される．Gtag フィールドは他コアで実行された命令によって wakeup される． LR_l ， LR_r ， GR_l ， GR_r の 4 つのフィールドがすべて 1 になるとき，エントリは発行可能となる．

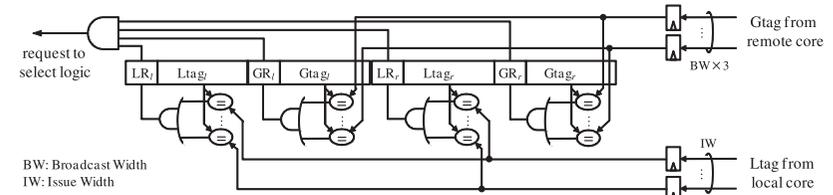


図 10 2-way リネーミング向け命令ウィンドウのスケジューラ部
Fig. 10 Scheduler part of cooperative instruction window.

5.3.4 2-way リネーミングの関連研究

2-way リネーミングは，FB 間の依存を，(1) のようなタグで表現することで RMT の軽量化を図る．ここでは，同様の表現方法を採用するいくつかの研究についてまとめる．

$$Gtag = \{ \text{論理レジスタ番号, サフィックス (FB 番号)} \} \quad (1)$$

Hopkins ら²²⁾ は，命令レベル並列プロセッサの軽量化と性能向上を目的として DIF (Dynamic Instruction Format) という手法を提案している．この手法は，狭帯域のアウトオブオーダー発行エンジンで命令グループの依存を解析し，スケジューリングが済んだ状態で DIF キャッシュという特別な命令キャッシュに保存する．DIF キャッシュにヒットする場合は，スケジューリング済みの命令グループが得られるので，広帯域な VLIW 発行エンジンで高速に実行する．この手法には，命令グループ間で物理レジスタのマッピングが異なるため，通常のリネーミングではグループ間の依存が正しく表せないという問題がある．これに対し，Hopkins らは，命令のタグを (1) と同様のフォーマットで表現する方法を提案している．(1) のフォーマットを利用する場合は，サフィックス部に適切なオフセットを加えることで，比較的容易に正しいタグが得られる．

Talpes ら²³⁾ が提案する EC (Execution Cache) も，DIF と同様にスケジューリング済み命令を特別なトレースキャッシュに保存することで，命令レベル並列プロセッサの消費電力低減を目指すものである．トレース間で物理レジスタのマッピングが異なる問題に対し，DIF と同様に (1) のフォーマットでタグを表現することで対処している．

これらの文献ではいずれも，物理レジスタの構成そのものを命令タグのフォーマットに合わせて変更している．物理レジスタは，各論理レジスタに対して数個^{*1}のプールを持つ構成となる．この実装は，物理レジスタの利用効率を著しく低下させ，結果として性能低下を引

*1 文献 23) では 4 個程度．

き起こす*1。

一方、2-way リネーミングでは性能低下の問題は発生しない。2-way リネーミングにおける Gtag は単なるタグであり、物理レジスタ番号ではない。Gtag のとりうる空間は、物理レジスタの構成とは無関係に拡大することができる。物理レジスタの構成も従来と同様のものを採用できるため、2-way リネーミングは性能低下の直接的な要因にはならない。その点において、タグと物理レジスタ番号を分けて考える 2-way リネーミングは、より進んだ実装であるといえる。

5.4 課題 4 物理レジスタの構成：CoreSymphony 向け物理レジスタ分散手法の提案

CoreSymphony において、あるコアで実行された命令の結果およびタグは協調動作中の全コアにブロードキャストされる。しかし、他コアからの結果をすべて自コアの物理レジスタに格納することは、物理レジスタのポート数やエントリ数の面から現実的でない。また、他コアからの結果が自コアの物理レジスタを確保することは、デッドロックという新たな問題を発生させる。本節では、これらの問題に対処する CoreSymphony 向け物理レジスタ分散手法について述べる。

5.4.1 コア間のオペランド通信とそのフィルタリング

まず、CoreSymphony におけるコア間のオペランド通信について述べる。コア間のオペランド通信には、ブロードキャストモデルを採用する。図 11 はコア x で実行された命令 I_0 の結果を、コア y の命令 I_1 が利用する場合のパイプラインの動作である。オペランド通信は 2 フェーズに分けられる。まず、コア x では I_0 が発行されるタイミングで I_0 の Gtag をブロードキャストする。その後、 I_0 の実行完了と同時に結果をブロードキャストする。一方コア y では、先に放送されてきたタグによって I_1 のウェイクアップが行われる。ここで、 I_1 がセレクトされた場合には、続けて到着する I_0 の結果をバイパスにより取り込んで実行

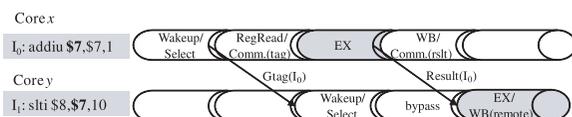


図 11 コア間のオペランド通信のタイミング。コア間レイテンシが 1 サイクルの場合

Fig. 11 Timing diagram of inter-core operand communication.

*1 極端な例では、プールが各論理レジスタについて 4 個だとすると、同じデスティネーションを持つ命令が 4 個連続でフェッチされるだけで物理レジスタは枯渇してしまう。文献 23) ではこの問題により、ベースモデルから最大で 19%も性能が低下することが示されている。

ステージへ移行する。

一般に、ブロードキャストモデルはハードウェアの複雑度に与える影響が大きい。1 サイクルあたりに他コアから到来する Gtag の数は、命令ウィンドウの Gtag フィールドを構成する CAM のサーチポート数に直結する。そのため、CoreSymphony では、各コアが 1 サイクルあたりに発生させるブロードキャストの数 (Broadcast Width: BW) を制限する。 BW は設計パラメータである。 BW を小さくすることは、HW の複雑度を抑える一方で、性能低下につながる。そこで、CoreSymphony は次の場合にブロードキャストをフィルタリングする。

- 当該命令が結果を生成しない場合 (ex. 分岐命令, ストア命令)。
- 当該命令が生成する結果が、同 FB 中の同じデスティネーションを持つ他の命令によって上書きされる場合*2。

これらの情報は事前に解析され、broadcast flag としてローカル命令キャッシュに保存されている。スケジューラの Select ロジックは、このフラグを加味して、ブロードキャストを発生させる命令がサイクルあたり BW 個以下になるように発行命令を選択する。典型的には BW は INT, FP ともに 1 命令/cycle とする。 BW が性能に与える影響は 6 章で評価する。

5.4.2 物理レジスタへの書き込みフィルタリング

$BW = 1$ としても、他コアから到来するオペランドは、4 コア時で最大 3 命令/cycle にもなる。そこで、物理レジスタへの書き込みをフィルタリングする。他コアで実行された命令の結果を、自コアの物理レジスタに書き込む必要があるのは次の場合に限定される。

- (a) 自コアの命令ウィンドウ中に存在する命令が、その結果を利用する場合。
- (b) これから自コアにディスパッチされる命令が、その結果を利用する場合。

すなわち、これらにあてはまらない場合は物理レジスタに書き込む必要はない。CoreSymphony では、これらにあてはまらない結果のブロードキャストを検出し、物理レジスタへの書き込みをフィルタリングすることで、物理レジスタのポート数およびエントリ数を現実的な範囲に抑えることを目指す。

まず、(a) の検出方法について示す。これは、先行して放送されてきた Gtag に依存する命令が命令ウィンドウ中に存在するか否かを調べることで検出する。このために、命令ウイ

*2 リーフノードステアリングは FB 内のコア間通信を発生させないため、上書きされる命令の結果が他コアで利用されることはない。

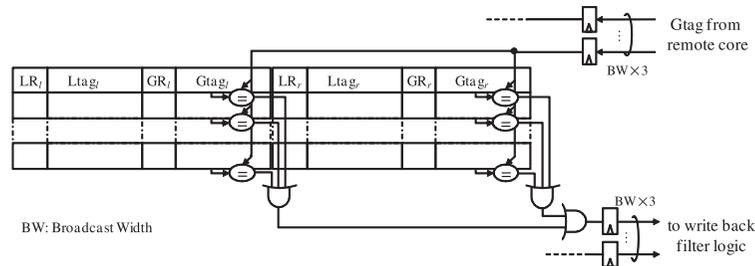


図 12 リモートの命令に依存する命令を検出するために変更を加えた命令ウィンドウのウェイクアップ部
Fig. 12 Modified instruction window to detect the instructions that depend on remote instructions.

ンドウのウェイクアップロジックを一部変更する．図 12 に変更を加えた命令ウィンドウのウェイクアップ部を示す．ソースの $Gtag$ を格納する CAM のマッチ線を引き出し，すべてのエントリについて OR をとることで，マッチするエントリ (= 放送されてきた命令の結果を利用するエントリ) の有無を検出する．

次に，(b) の検出方法について示す．これは，フィルタリング対象の命令がアーキテクチャステートに含まれるか否かを調べることで検出できる．なぜならば，すでにアーキテクチャステートでない結果は，投機ミスが起こらない限り，今後ディスパッチされる命令によって利用されることはないためである．対象命令がアーキテクチャステートを生成するか否かは，GRMT を利用することで容易に検出できる．対象命令のデスティネーションの論理レジスタ番号で GRMT の行を読み出すことで，その論理レジスタに対してアーキテクチャステートを与える FB 番号 FB_x を知ることができる． FB_x と対象命令の FB 番号を比較することで，対象命令がアーキテクチャステートか否かを検出できる．この機構のために，GRMT に $BW \times (NC - 1)$ と同数の読み出しポートを追加する必要がある． BW は各コアのプロードキャスト幅， NC は協調をサポートする最大のコア数である．典型的には， $BW = 1$ ， $NC = 4$ である．

最後に，リモートからの物理レジスタの書き込みについてまとめる．物理レジスタへの書き込みは (a) と (b) の検出によってフィルタリングされる．物理レジスタのリモート用の書き込みポートの数を Remote writeback Width (RW) と表す．書き込みが許諾された命令は，物理レジスタのエントリを確保し，LRMT にマッピングを登録する．そのため，LRMT に RW と同数の書き込みポートを追加する． RW は設計パラメータであり，典型的には $RW = 1$ である． RW が性能に与える影響は 6 章で評価する．

5.4.3 リモートのライトバックによるデッドロックの発生と対処

リモートの命令が自コアの物理レジスタに値を書き込む場合，その命令は自コアの物理レジスタをアウトオブオーダーに束縛する．物理レジスタのアウトオブオーダーな束縛はデッドロックという問題を引き起こす²⁴⁾．デッドロックは，書き込みを発生するリモートの命令 I_x が in-flight な命令中で最も古く，かつ物理レジスタが枯渇している場合に発生する．なぜならば， I_x に物理レジスタが割り当てられない限り，命令の実行は進まず物理レジスタが解放されることもないためである．

CoreSymphony は非常にシンプルな方式でこの問題を回避する．CoreSymphony はリモート命令の物理レジスタの束縛用に，あらかじめ物理レジスタのエントリを確保しておく．確保する物理レジスタの数を Number of Reserved Physical register (NRP) と呼ぶ． NRP は設計パラメータである．物理レジスタの空きエントリ数が NRP を下回ったときはフロントエンドをストールさせ，物理レジスタの束縛を止める． NRP によって物理レジスタの空きを確保していても，場合によってはリモートのライトバック時に物理レジスタに空きがない場合がある．この場合には物理レジスタの枯渇を引き起こした命令以降をフラッシュし，再実行する．すなわち， NRP の決定にはフロントエンドのスループット低下とパイプラインフラッシュの低減のトレードオフが存在する． NRP が性能に与える影響については 6 章で評価する．

5.5 課題 6 コミットの処理

5.5.1 コミットのタイミング

CoreSymphony は，コミットのインオーダー性を ROB によって保証する．ROB は FB 中のすべての命令について実行が正しく完了したことを示すフラグを保持する．あるコアで命令の実行が完了すると，例外の発生や投機ミスに関する情報が全コアにブロードキャストされる．ROB はこれらの情報を収集する．命令のコミットは FB 単位で行う．あるコアは，FB 中のすべての命令の実行完了を確認した後，命令のコミットを行う．ただし，コア間通信にはレイテンシが存在するため，通常，各コアでコミットのタイミングは同期させない．このことは次の場合に問題を引き起こす．

- (a) 投機ミスから未回復のコアより放送されてきた生産者の $Gtag$ が，投機ミスから回復済みのコア中の，ある消費者のソース $Gtag$ と一致する場合
この場合，生産者の $Gtag$ と消費者の $Gtag$ は，一致するにもかかわらず異なるステートを指している．そのため，消費者は誤ったオペランドを用いて実行を行ってしまう．

- (b) 分岐履歴更新タイミングの不整合により、各コアの分岐予測結果が異なり、各コアが別々の制御フローをたどる場合

CoreSymphony では、分岐履歴の更新は分岐命令のコミット時に行う。そのため、コミットのタイミングのずれは上記の理由により問題を引き起こす場合がある。

(a) に関しては、投機ミスからの回復時に再開のタイミングを全コアで同期させる方法や、投機ミスの数を記録する数ビットのカウンタを用意し、これを結果に付随して放送することで Gtag の不整合を検出するという方法²⁵⁾ が考えられる。本論文のシミュレーションモデルでは前者を採用して評価を行った。

(b) に関しては、分岐履歴の更新のみを自コア内で適切に待ち合わせればよい。たとえば、in-flight に保持できる FB 数が N の場合、 FB_t 中の分岐命令が分岐履歴を更新するタイミングを FB_{t+N} がフェッチされるタイミングと指定する。このようにすると、各コアでフェッチやコミットのタイミングがずれたとしても、ある FB をフェッチする際の分岐履歴は、全コアで同一のものとなる。よって、制御フローのずれも生じない。

5.5.2 インオーダーステートの管理と投機ミスからの回復

命令のコミットにおける主な操作として、インオーダーステートの更新がある。しかし、現時点ではインオーダーステートを分散管理する手法は十分な検討が行われていない。そこで、本論文では暫定的な手法として、インオーダーステートのみを保存する Logical Register File (LRF) と呼ぶ HW を用意する。LRF は論理レジスタと同数のエントリを持つ。あるコアにおける命令 I_x のコミットは次のように行われる。 I_x が自身の物理レジスタにマッピングを持つ場合、ROB に保存される物理レジスタ番号を利用して物理レジスタを読み出し、すべてのコアの LRF に値をコピーする。そして、物理レジスタを解放する。この手法は、単純であるがコア間ネットワークおよび LRF のポート数に与える影響が非常に大きく現実的ではない。

インオーダーステートの分散管理の可能性について考察する。コミット時に、結果を他コアにコピーする必要があるのは、次にその結果が新たなステートで上書きされるまでの間に、投機ミスが起こる場合のみである。また、ある論理レジスタに対するインオーダーステートは、全コアが保持する必要はなく、少なくとも 1 つのコアに存在すればよい。これらの観察から、コミット時には自身の LRF にのみ値をコピーし（各コアはインオーダーステートの一部を持つ）、投機ミス時に各コアの LRF を合成し、インオーダーステートを形成するというアプローチが考えられる。このアプローチは LRF のポート数を 2-way OoO 相当まで軽量化できるものの、インオーダーステートの合成方法に課題が残る。インオーダーステートの分散

は今後の検討項目とする。

5.6 各提案手法のまとめ

CoreSymphony アーキテクチャを実現するために、これまでに以下に示す様々な要素技術を提案、あるいは引用した。これらの要素技術は、現実的な実現コストで 4 章で述べた課題を克服するためのものである。

- ローカル命令キャッシュ
効果 命令キャッシュの分割を実現する。
実現コスト 2 命令/cycle のスループットを持つ、トレースキャッシュに準じるハードウェア。
- リーフノードステアリング
効果 FB 内のコア間通信をいっさい発生させず、かつある程度良い負荷分散を実現する。
実現コスト 16 ビット × 16 ワード、4R2W の RAM に相当するハードウェア等。
- 2-way リネーミング
効果 物理レジスタ空間の拡大と RMT のポート数削減。
実現コスト 6R2W の RAM (LRMT), 4R1W の RAM (GRMT) に相当するハードウェア。スケジューラの Wakeup ロジックの変更 (図 10)。
- CoreSymphony 向け物理レジスタ分散手法
効果 物理レジスタのエントリ分散による物理レジスタファイルのハードウェア量削減。
実現コスト LRMT に 1W ポートの追加。GRMT に 3R ポートの追加。物理レジスタファイルに 1W ポートの追加 ($BW = 1$, $RW = 1$ の場合)。BW を制限するためのスケジューラの Select ロジックの変更。スケジューラの Wakeup ロジックの変更 (図 12)。
- ULB-LSQ (文献 16) より引用
効果 データキャッシュと LSQ の分割を実現する。
実現コスト 特殊な CAM 構造とフロー制御のための機構を有する、2-way OoO の LSQ 相当のハードウェア。ロードストア命令の転送のためのコア間ネットワーク。

5.7 全体構成

本章で提案した要素技術をまとめた CoreSymphony の全体構成を図 13 に示す。影付きのモジュールはベースのスーパスカラ (図 2) に追加、あるいは大きな変更を加えたモジュールである。各モジュールの入出力ポートには、表 1 と対応する番号を併記する。表 1

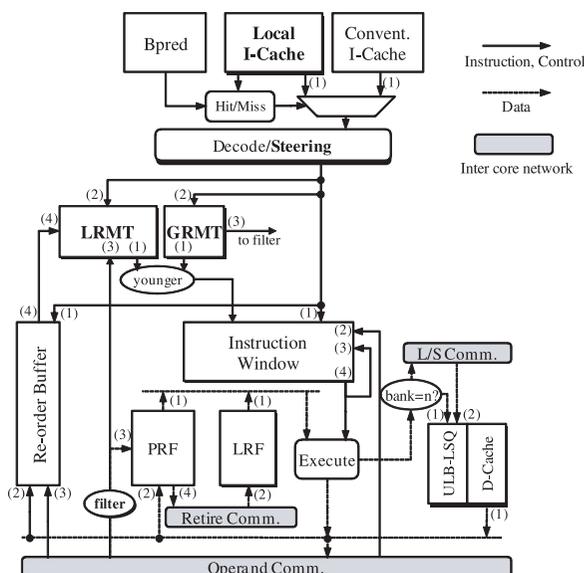


図 13 CoreSymphony アーキテクチャのブロック図
Fig. 13 Block diagram of CoreSymphony architecture.

は、CoreSymphony の HW の複雑度をまとめたものである。比較用に典型的な 2-way アウトオブオーダと 8-way アウトオブオーダ (4 コア協調時に相当) の値を併記した。BW および RW は設計パラメータで、典型的には $BW = 1$, $RW = 1$ である。表 1 によれば、CoreSymphony のコア複雑度はおおよそ、2-way から 4-way のアウトオブオーダ相当であるが、LRF および ROB に 8-way 相当の複雑度の部分が存在する。これは、インオーダー状態の分散が検討不十分であることによる。インオーダー状態の分散は今後の課題である。

6. CoreSymphony アーキテクチャの評価

本章ではシミュレーションにより CoreSymphony の性能を評価する。

6.1 評価環境

評価環境を示す。シミュレータとして、独自開発の実行駆動サイクルレベルシミュレータ SimMips/SS (SimMips/SuperScalar) を用いる。SimMips/SS は Linux の動作する MIPS

表 1 CoreSymphony の 1 コアの HW 複雑度。() 内の数字は図 13 に対応する
Table 1 Hardware complexity of CoreSymphony.

Module	function	CoreSymphony	2-way OoO	8-way OoO
I-cache	(1) fetch	2 inst/cycle	2 inst/cycle	8 inst/cycle
Local I-cache	(1) fetch	2 inst/cycle	-	-
Decoder	Decode	2 inst/cycle	2 inst/cycle	8 inst/cycle
Steering Unit	Steering	2 inst/cycle	-	-
LRMT	(1) Rename(Src)	4 Read	4 Read	16 Read
	(2) Rename(Dst, Local)	2 Write	2 Write	8 Write
	(3) Rename(Dst, Remote)	RW Write	-	-
	(4) Release	2 Read	2 Read	8 Read
GRMT	(1) Rename(Src)	4 Read	-	-
	(2) Rename(Dst)	1 Write	-	-
	(3) State check	3BW Read	-	-
Inst window	(1) Dispatch	2 inst/cycle	2 inst/cycle	8 inst/cycle
	(2) Wakeup(Ltag CAM)	2 Search	2 Search	8 Search
	(3) Wakeup(Gtag CAM)	3BW Search	-	-
	(4) Issue	2 inst/cycle	2 inst/cycle	8 inst/cycle
ROB	(1) Allocate	8 entry/cycle	2 entry/cycle	8 entry/cycle
	(2) Update exec-flag(Local)	2 Write	2 Write	8 Write
	(3) Update exec-flag(Remote)	6 Write	-	-
	(4) Release	8 entry/cycle	2 entry/cycle	8 entry/cycle
PRF	(1) Opfetch	4 Read	4 Read	16 Read
	(2) Writeback(Local)	2 Write	2 Write	8 Write
	(3) Writeback(Remote)	RW Write	-	-
	(4) Commit	2 Read	-	-
LRF	(1) Opfetch	4 Read	-	-
	(2) Commit	8 Write	-	-
ULB-LSQ	Allocate	-	2 inst/cycle	8 inst/cycle
	(1) Dispatch(Local)	1 inst/cycle	1 inst/cycle	4 inst/cycle
	(2) Dispatch(Remote)	1 inst/cycle	-	-
D-cache	Issue	1 inst/cycle	1 inst/cycle	4 inst/cycle
	(1) Load/Store	1 inst/cycle	1 inst/cycle	4 inst/cycle

BW: Broadcast Width, RW: Remote writeback Width

システムレベルシミュレータ SimMips²⁶⁾ 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。

独自開発のシミュレータを用いる理由は、忠実なシミュレーションを行うためである。たとえば、有名なプロセッサシミュレータである SimpleScalar²⁷⁾ は、命令のエミュレーション

をフェッチ時にインオーダーで行う．そのため，マイクロアーキテクチャに問題があり不正な命令実行を行った場合でもシミュレーションは正しく完了してしまう．CoreSymphony は，一般的なアウトオブオーダーのアーキテクチャをほぼ全域にわたって変更する．また，ULB-LSQの採用により，命令がフェッチされたコアと別のコアで実行される場合さえ存在する．このようなマイクロアーキテクチャをシミュレーションする場合には，命令を実際のモデルと同じタイミングで実行することが，シミュレーションの正確性を確保するうえで重要である．我々のシミュレータは命令を実際のモデルと同じタイミングで実行する．なおかつ，1 命令ごとの実行結果を機能レベルシミュレータの実行結果と比較することでシミュレーションの正確性を確保している．

ベンチマークには SPEC2006 ベンチマークより INT5 種類 (403.gcc , 429.mcf , 456.hmmmer , 462.libquantum , 464.h264ref) と FP5 種類 (433.milc , 444.namd , 453.povray , 470.lbm , 482.sphinx3) を用いる．データセットには train を用い，1 G 命令をスキップ後の 100 M 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 Buildroot²⁸⁾ release 2009.08 を使用し，MIPS32 用に構築した gcc4.3.3 (最適化オプション-O2) を用いる．ただし，遅延分岐最適化を行わないように gcc に変更を加えている．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．

評価に用いるプロセッサのパラメータは表 2 のとおりである．各命令のレイテンシと発行間隔については MIPS R10000¹³⁾ と同様に設定した．

6.2 協調動作による性能の変化

図 14 は協調動作により発行幅を増加させた場合の IPC の変化をベンチマークごとに示したものである．2, 4, 8-issue はそれぞれ 1, 2, 4 コアの協調動作を意味する．比較対象として，標準的な単一コアデザインのアウトオブオーダーの IPC を同時に示す．

比較対象に用いる標準的なアウトオブオーダーについては，表 2 の (2) の値を発行幅と同じ値とし，その他のバッファやキャッシュのエントリ数は CoreSymphony の協調動作時の実容量と同等とした．たとえば，CoreSymphony は協調動作により一次データキャッシュの容量を纯粹に増加させることができる．4 コア協調時にスレッドあたりの実容量は 64KB である．そのため，4 コアの比較対象である 8-way アウトオブオーダーは 64KB の一次データキャッシュを持つと設定する．ただし，レイテンシは CoreSymphony における 16KB のキャッシュと同等に設定する．このように，図 14 のアウトオブオーダーの IPC は協調のオーバヘッドを排した，理想値としての性格を持つ．

結果を見ると，CoreSymphony は協調動作を行うコア数の増加にともない，IPC が向上

表 2 評価に用いたプロセッサのパラメータ

Table 2 Baseline processor configuration.

(1) Pipeline	IF, ID, Steering, Rename, Sched, RF, EX, WB, Commit
(2) Pipeline width	2 for fetch, decode, issue. 8 for commit.
(3) Functional units	2 iALU, 1 LD/ST, 1 fpALU
(4) Instruction window	24/24 entries for INT/FP
(5) Physical register	48/48 entries for INT/FP, 18/18 entries <i>NRP</i>
(6) Load/Store queue	96 entries ULB-LSQ* ¹
(7) Memory disambiguation	LWT, 1 k entries
(8) Branch prediction	Bimode, 6 bit GHR, 1 K entries PHT
(9) Branch target buffer	1 k entries, 2-way
(10) L1 D-cache	16 KB, 2-way, 1 cycle hit-latency, non-blocking
(11) L1 I-cache (Conventional)	8 KB, 2-way, 1 cycle hit-latency
(12) L1 I-cache (Local)	8 KB, 4-way, 2 cycle hit-latency
(13) Shared L2 cache	2 MB, 4-way, 10 cycle hit-latency
(14) Main memory	100 cycle
(15) Inter core latency	1 cycle
(16) Broadcast Width(<i>BW</i>)	1/1 for INT/FP
(17) Remote writeback Width(<i>RW</i>)	1/1 for INT/FP

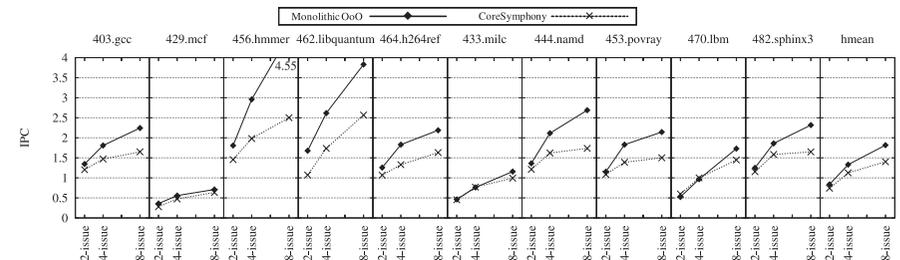


図 14 協調動作による発行幅の増加と IPC の関係
Fig. 14 Performance evaluation: issue width versus IPC.

することが分かる．1 コア時からの性能向上という観点では，10 種のベンチマークの調和平均 (hmean) において，2 コアの協調で 1.51 倍，4 コアの協調で 1.88 倍の性能を示す．最も性能向上の幅が大きかった lbm では，2 コアで 1.68 倍，4 コアで 2.46 倍という IPC を

*1 ULB-LSQ のエントリ数がコアあたり 96 と多いのは，ULB-LSQ に特有のデッドロック問題を回避するためである．実際には，フロー制御等の実装により，性能へほとんど影響を与えずにエントリ数を一般的な LSQ (この場合 24 エントリ程度) と同等以下に抑えることが可能である¹⁶⁾．

達成した．

単一コアのアウトオブオーダーとの比較では，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 は単一コアのアウトオブオーダーに比べ，23.3%低い IPC を示した．

6.3 性能解析

ここでは，CoreSymphony のより詳細な性能評価と，各設計パラメータが性能に与える影響の解析を行う．

6.3.1 協調動作によって得られる効果と性能に与える影響

コアの協調動作によって得られる効果のうち，性能に良い影響を与えるものは，1 次キャッシュの増加と，発行幅の増加*1である．本項ではそれら 2 つの効果が性能に与える影響を示す．

図 15 は，キャッシュ容量増加の効果のみに絞って IPC を測定した結果である．コア数が 1, 2, 4 コアの 3 つの場合に，ローカル命令キャッシュと 1 次データキャッシュの容量を変化させた．各キャッシュ容量は，1, 2, 4 コア協調時のキャッシュの総容量と同等である．結果を見ると，おもに整数ベンチマークにおいて，キャッシュ容量増加による性能向上は小さいが，h264ref や namd, povray, sphinx3 の 4 つのベンチマークでは顕著な性能向上がみとれる．すべてのベンチマークの平均では 4 コア時に，キャッシュ容量をローカル 32KB/データ 64KB とすると IPC が 1.54，ローカル 8KB/データ 16KB とすると 1.45 と，キャッシュ容量増加により 6.2%の性能向上が得られている．

一方，図 16 は，発行幅増加の効果のみに絞って IPC を測定した結果である．キャッシュ容量が 8/16KB, 16/32KB, 32/64KB の 3 つの場合に，コア数を 1, 2, 4 コアと変化した．評価の結果，povray と sphinx3 で 4 コア時の性能が 2 コア時と比べて落ち込む場合があるものの，ほぼすべての場合において発行幅増加による有意な性能向上がみとれる．特に，整数ベンチマークではキャッシュ容量増加の効果は薄かったにもかかわらず，発行幅増加により大きな性能向上が得られている．

協調による発行幅の増加はキャッシュ容量の増加に比べ，実現のために HW 複雑度を与える影響が大きい．しかし，得られる性能向上も大きいことから，協調によって性能向上を得る場合には発行幅の増加は不可欠な機能であるといえる．

*1 命令ウィンドウ等の資源の増加も含む．

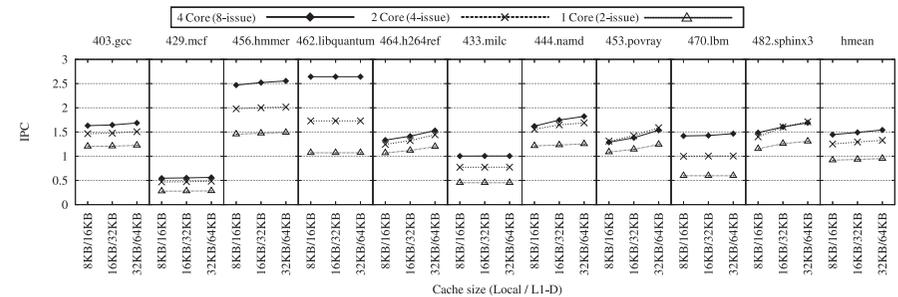


図 15 発行幅を一定とした場合のキャッシュ容量増加の効果の見積り
Fig. 15 Performance evaluation: Cache sizes versus IPC (with fixed issue width).

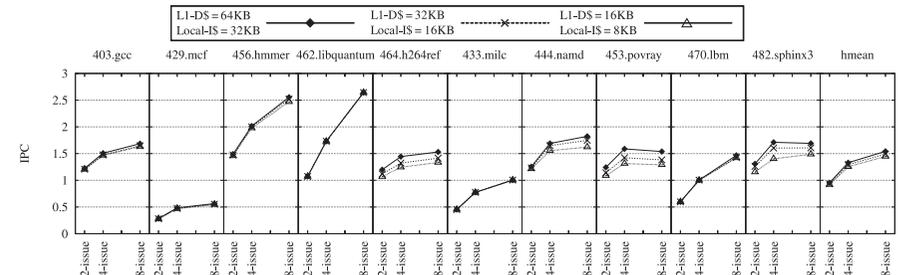


図 16 キャッシュ容量を一定とした場合の発行幅増加の効果の見積り
Fig. 16 Performance evaluation: Issue width versus IPC (with fixed cache sizes).

6.3.2 命令キャッシュの構成と性能の関係

図 17 は，各協調コア数について，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 (=実効 IPC) をまとめたものである．それぞれ，10 種のベンチマークにおける最大，最少，調和平均を示している．従来型命令キャッシュとローカル命令キャッシュについて，命令部の容量比が 4:1, 1:1, 1:4 かつ，制御部も含む合計の容量が 30KB 程度になるようにサイズを調整した．評価の結果，協調コア数が多いときほど，ローカル命令キャッシュに多く容量を割いたほうが性能が向上することが分かった．従来型命令キャッシュは協調によりスレッドあたりの実エンタリ数を増加させることができない．これに対し，ローカル命令キャッシュは実エンタリ数を増加させることができる．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ほうが良いという結果の理由であると考えられる．

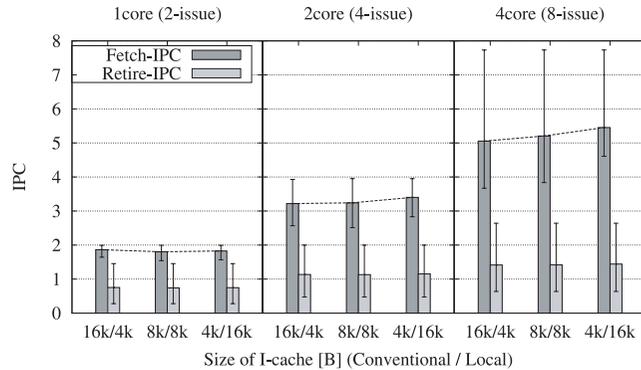


図 17 命令キャッシュの構成と性能の関係

Fig. 17 Fetch and retire IPC with some instruction-cache configurations.

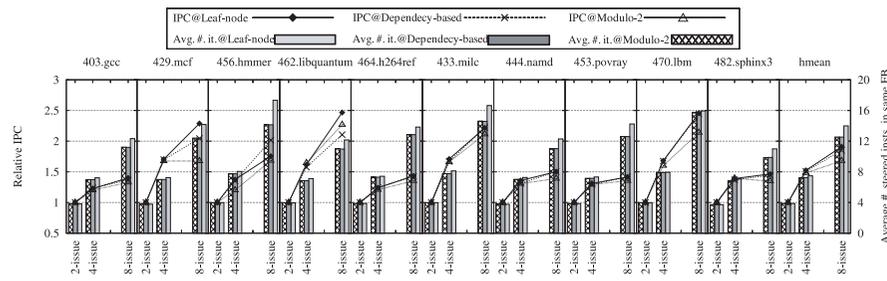


図 18 ステアリングアルゴリズムと性能の関係 (左軸), 命令の重複を含む FB のサイズの平均値 (右軸)

Fig. 18 Steering algorithm vs. performance. –Relative IPC based on 1-core, Leaf-node (left axis), Average number of steered instructions in same FB (right axis).

6.3.3 ステアリングアルゴリズムと性能の関係

図 18 は、ステアリングアルゴリズムと性能の関係をまとめたものである。リーフノードステアリング、依存ベースステアリング、Modulo-2 ステアリングの 3 種について^{*1}、リーフノードステアリングの 1 コア時に対する相対 IPC を測定した。同時に、コピーしてステアリングされた命令を含む、FB 内の命令数の平均値を右軸に示す。

*1 依存ベースステアリングと Modulo-2 ステアリングは、FB 内でコア間通信を発生させる可能性がある。そのため、2-way リネーミングが適用できない。これらのアルゴリズムを利用する場合、リネームロジックを理想化して性能を測定した。

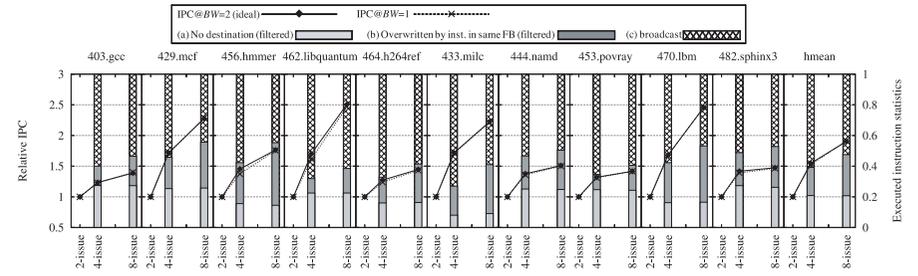


図 19 コアあたりのブロードキャストの幅 (BW) と性能の関係 (左軸), 実行された命令に関する統計情報 (右軸)
Fig. 19 Broadcast Width (BW) sensitivity analysis. –Relative IPC based on 1-core, BW = 1 (left axis), Statistics of executed instruction (right axis).

結果を見ると、hmmr という例外はあるもの、ほとんどのベンチマークでリーフノードステアリングが最も良い性能を示した。全ベンチマークの調和平均では、4 コア時に、依存ベースと比べて 3.2%、Modulo-2 と比べて 13.7%良い IPC が得られた。また、命令の重複による実効命令数の増加は 2 コア時に 3.2%、4 コア時でも 11.6%程度であることが分かった。実際にはリーフノードステアリングは、性能向上もさることながら、FB 内のコア間通信を完全になくすることができるというメリットの方が大きい。11.6%程度の実効命令数増加で、このメリット (+性能向上) を得られたのは良い結果であるといえる。

6.3.4 ブロードキャストの幅と性能の関係

図 19 は、1 サイクルあたりにブロードキャストを行うことができる命令の数 (BW) と性能の関係をまとめたものである。典型的な値である BW = 1 と、理想的な値である BW = 2 の 2 種類の設定で、BW = 1 の 1 コア時に対する相対 IPC を測定した。同時に、実行された命令に関する統計情報を示す。(a) はレジスタへの書き込みを行わない命令、(b) は同 FB 内かつ同コアの他の命令によって値が上書きされる命令、(c) はそれ以外のブロードキャストされる命令の割合である。(a) と (b) の場合にブロードキャストをフィルタできる。命令の統計情報は、協調中の各コアについて集計し平均をとった。

結果を見ると、BW を理想化してもほとんど性能は変化しないことが分かる。この結果は、命令の統計情報を見ることで納得ができる。ブロードキャストがフィルタされる命令の割合は、2 コア時には 4 割弱、4 コア時には 5 割弱にも及ぶ。特に、(b) によってフィルタされる命令の割合が協調コア数が増えるほど増加しており、非常に良い傾向を示す。これには、リーフノードステアリングの特徴である、依存関係にある命令を同じコアにまとめる効

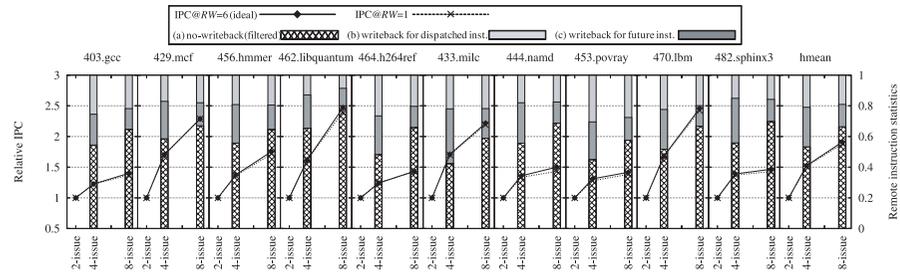


図 20 物理レジスタのリモートからのライトバックのポート数 (RW) と性能の関係 (左軸), ブロードキャストされた命令に関する統計情報 (右軸)

Fig. 20 Remote writeback Width(RW) sensitivity analysis. –Relative IPC based on 1-core, $RW = 1$ (left axis), Statistics of broadcast instruction (right axis).

果が寄与していると考えられる。協調コア数の増加により FB が長くなるほど、リーフノードステアリングの効果は大きくなる。

結論として BW は 1 命令/cycle で十分であり, 協調するコアの数に依存しないパラメータであることが分かった。

6.3.5 リモートのライトバックのポート数と性能の関係

図 20 は, 物理レジスタのリモートからのライトバックのポート数 (RW) と性能の関係をまとめたものである。典型的な値である $RW = 1$ と, 理想的な値である $RW = 6$ の 2 種類の設定で, $RW = 1$ の 1 コア時に対する相対 IPC を測定した。同時に, ブロードキャストされた命令に関する統計情報を示す。(a) は物理レジスタへのライトバックがフィルタされた命令, (b) は命令ウィンドウ内の命令のためにライトバックを行った命令, (c) はこれからディスパッチされるであろう命令のためにライトバックを行った命令である。命令の統計情報は, 協調中の各コアについて集計し平均をとった。

結果を見ると, hmmmer や libquantum, namd といった IPC の大きなベンチマークで, RW を狭幅化した影響がわずかに表れているものの, 総合的にはほとんど影響がないことが分かる。命令の統計情報を見ると, ライトバックがフィルタされる命令は 2 コア時に 5 割強, 4 コア時には 6 割強にも及ぶ。特に, 協調動作するコア数が増えるにつれ (b) の割合が減少し, フィルタできる命令数増加に寄与している。

結論として, RW は 1 ポートで十分であることが分かった。

6.3.6 物理レジスタの構成と性能の関係

次に, リモートからの束縛用に確保する物理レジスタ数 NRP が性能に与える影響を評

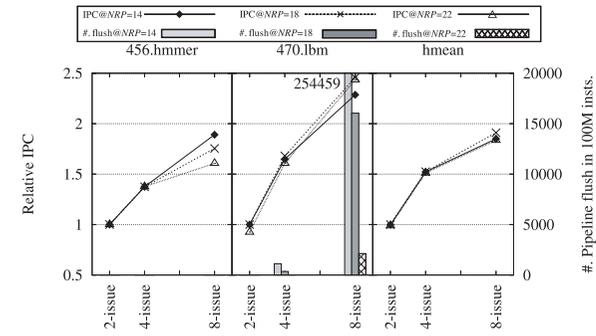


図 21 リモート命令用に確保する物理レジスタ数 (NRP) と性能の関係 (左軸), 物理レジスタの枯渇によるパイプラインフラッシュの発生回数 (右軸)

Fig. 21 Number of Reserved Physical register (NRP) sensitivity analysis. –Relative IPC based on 1core, $NRP=18$ (left axis), The number of pipeline flush caused by physical register depletion (right axis).

価する。物理レジスタ数を INT/FP ともに 48 エントリとし, NRP を 14, 18, 22 と変化させた。図 21 は $NRP = 18$ の 1 コア時に対する相対 IPC をまとめたものである。10 種のベンチマークのうち, 特徴的な傾向を示した hmmmer と lbm, および全ベンチマークの調和平均を示す。同時に, 100 M 命令を実行する間に発生したパイプラインフラッシュのうち, リモート命令の物理レジスタ束縛時における物理レジスタ枯渇に起因するものの回数を示す。

結果を見ると hmmmer と lbm では逆の傾向を示していることが分かる。hmmmer は NRP を少なく設定した方が良い IPC が得られ, lbm は NRP を大きめに設定した方が IPC が高い。これは, hmmmer が NRP の不足によるパイプラインフラッシュをまったく発生させないベンチマークであるのに対し, lbm は非常に多く発生するという, プログラムの性質の違いによるものである。基本的に NRP は小さく設定するほど, ディスパッチのストールが起こりにくくなるため, 性能が向上する。しかし, lbm のように, リモートからの束縛時に物理レジスタ不足を引き起こしやすいベンチマークでは, パイプラインフラッシュによるオーバーヘッドがディスパッチのスループット向上によるメリットを上回る。

このトレードオフの存在は, CoreSymphony のさらなる性能向上の可能性を示す。図 21 から分かるように, 最適な NRP はアプリケーションによって異なる。プログラム中の実行フェーズによっても異なると考えられる。2 によって, NRP を動的に変更することでさら

なる性能向上が得られる可能性が高い。

7. まとめと今後の課題

本論文では, CMP の逐次処理能力の向上および逐次処理能力と並列処理能力のバランスを目的として, CoreSymphony アーキテクチャを提案した。CoreSymphony は, 複数個のスーパスカラを協調動作させることで, 発行幅の広い仮想コアを形成するアーキテクチャ技術である。スーパスカラの協調動作にはいくつかの解決すべき課題が存在する。本研究では, それらの課題を明確にし, 解決手法として以下の要素技術を提案した:

- 命令キャッシュの分割を実現するローカル命令キャッシュ。
- コア間通信の抑制と負荷分散を両立するリーフノードステアリング。
- 物理レジスタ空間の拡大と RMT の軽量化を実現する 2-way リネーミング。
- 物理レジスタのエントリ分散とポート数の抑制を実現する CoreSymphony 向け物理レジスタ分散手法。

これらの要素技術により, 一部の未検討部分に例外はあるものの, 現実的なハードウェア増加量でスーパスカラの協調動作が実現できることを示した。シミュレーションによる性能評価の結果, 4 コアの協調動作時に, 1 コア時と比較して平均 1.88 倍, 最大 2.46 倍の IPC が得られることが分かった。

今後の課題として次のことがあげられる:

- インオーダー状態の分散の実現。
- より詳細なハードウェア量の見積り。
- NRP の動的最適化等による, さらなる性能向上の達成。

参 考 文 献

- 1) Held, J., Bautista, J. and Koehl, S.: From a Few Cores to Many: A Tera-Scale Computing Research Overview, white paper, Intel. http://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf
- 2) Hill, M.D. and Marty, M.R.: Amdahl's law in the multicore era, *IEEE Computer*, Vol.41, No.7, pp.33–38 (2008).
- 3) Tullsen, D., Eggers, S. and Levy, H.: Simultaneous multithreading: Maximizing on-chip parallelism, *Proc. 22nd International Symposium on Computer Architecture (ISCA-1995)*, pp.392–403 (1995).
- 4) Palacharla, S., Jouppi, N. and Smith, J.: Complexity-Effective Superscalar Processors, *Proc. 24th International Symposium on Computer Architecture (ISCA-1997)*, pp.206–218 (1997).
- 5) Zyuban, V.V. and Kogge, P.M.: Inherently Lower-Power High-Performance Superscalar Architectures, *IEEE Trans. Comput.*, Vol.50, No.3, pp.268–285 (2001).
- 6) Ipek, E., Kirman, M., Kirman, N. and Martinez, J.F.: Core Fusion: Accommodating Software Diversity in Chip Multiprocessors, *Proc. 34th International Symposium on Computer Architecture (ISCA-2007)*, pp.186–197 (2007).
- 7) Madriles, C., López, P., Codina, J.M., Gibert, E., Latorre, F., Martínez, A., Martínez, R. and González, A.: Boosting single-thread performance in multi-core systems through fine-grain multi-threading, *Proc. 36th International Symposium on Computer Architecture (ISCA-2009)*, pp.474–483 (2009).
- 8) Madriles, C., Lopez, P., Codina, J.M., Gibert, E., Latorre, F., Martinez, A., Martinez, R. and Gonzalez, A.: Anaphase: A Fine-Grain Thread Decomposition Scheme for Speculative Multithreading, *Proc. 18th International Conference on Parallel Architectures and Compilation Techniques (PACT-2009)*, pp.15–25 (2009).
- 9) Kim, C., Sethumadhavan, S., Govindan, M.S., Ranganathan, N., Gulati, D., Burger, D. and Keckler, S.W.: Composable Lightweight Processors, *Proc. 40th International Symposium on Microarchitecture (MICRO-2007)*, pp.381–394 (2007).
- 10) Burger, D., Keckler, S., McKinley, K., Dahlin, M., John, L., Lin, C., Moore, C., Burrill, J., McDonald, R. and Yoder, W.: Scaling to the end of silicon with EDGE architectures, *IEEE Computer*, Vol.37, No.7, pp.44–55 (2004).
- 11) Zhong, H., Lieberman, S.A. and Mahlke, S.A.: Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications, *Proc. 13th International Conference on High-Performance Computer Architecture (HPCA-2007)*, pp.25–36 (2007).
- 12) Kessler, R.E.: The Alpha 21264 Microprocessor, *IEEE Micro*, Vol.19, No.2, pp.24–36 (1999).
- 13) Yeager, K.C.: The MIPS R10000 Superscalar Microprocessor, *IEEE Micro*, Vol.16, No.2, pp.28–40 (1996).
- 14) Johnson, M.: *Superscalar Microprocessor Design*, Prentice hall, Englewood Cliffs, N.J. (1990).
- 15) Yoaz, A., Erez, M., Ronen, R. and Jourdan, S.: Speculation Techniques for Improving Load Related Instruction Scheduling, *Proc. 26th International Symposium on Computer Architecture (ISCA-1999)*, pp.42–53 (1999).
- 16) Sethumadhavan, S., Roesner, F., Emer, J.S., Burger, D. and Keckler, S.W.: Late-binding: enabling unordered load-store queues, *Proc. 34th International Symposium on Computer Architecture (ISCA-2007)*, pp.347–357 (2007).
- 17) Gerosa, G., Curtis, S., D'Addeo, M., Jiang, B., Kuttanna, B., Merchant, F., Patel, B., Taufique, M. and Samarchi, H.: A Sub-1W to 2W Low-Power IA Pro-

cessor for Mobile Internet Devices and Ultra-Mobile PCs in 45nm Hi-k Metal Gate CMOS, *Proc. 2008 IEEE International Solid-State Circuits Conference (ISSCC-2008)* (2008).

- 18) Golden, M., Arekapudi, S., Dabney, G., Haertel, M., Hale, S., Herlinger, L., Kim, Y., McGrath, K., Palisetti, V. and Singh, M.: A 2.6 GHz Dual-Core 64 bx86 Microprocessor with DDR2 Memory Support, *Proc. 2006 IEEE International Solid-State Circuits Conference (ISSCC-2006)*, pp.325–332 (2006).
- 19) Narayanasamy, S., Wang, H., Wang, P., Shen, J. and Calder, B.: A Dependency Chain Clustered Microarchitecture, *Proc. 19th International Parallel and Distributed Processing Symposium (IPDPS-2005)*, pp.21b–21b (2005).
- 20) Aleta, A., Codina, J., Gonzalez, A. and Kaeli, D.: Instruction replication for clustered microarchitectures, *Proc. 36th International Symposium on Microarchitecture (MICRO-2003)*, pp.326–335 (2003).
- 21) Aggarwal, A. and Franklin, M.: Instruction replication for reducing delays due to inter-PE communication latency, *IEEE Trans. Comput.*, Vol.54, No.12, pp.1496–1507 (2005).
- 22) Hopkins, M.E. and Nair, R.: Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups, *Proc. 24th International Symposium on Computer Architecture (ISCA-1997)*, pp.12–25 (1997).
- 23) Talpes, E. and Marculescu, D.: Execution cache-based microarchitecture power-efficient superscalar processors, *IEEE Trans. VLSI*, Vol.13, No.1, pp.14–26 (2005).
- 24) Monreal, T., Gonzalez, A., Valero, M., Gonzalez, J. and Vinals, V.: Delaying physical register allocation through virtual-physical registers, *Proc. 32nd Annual International Symposium on Microarchitecture (MICRO-1999)*, pp.186–192 (1999).
- 25) 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).
- 26) 藤枝直輝, 渡邊伸平, 吉瀬謙二: 教育・研究に有用な MIPS システムシミュレータ SimMips, 情報処理学会論文誌, Vol.50, No.11, pp.2665–2676 (2009).
- 27) SimpleScalar LLC: SimpleScalar version 3.0. <http://www.simplescalar.com/>

28) Andersen, E.: Buildroot. <http://buildroot.uclibc.org/>

(平成 22 年 1 月 26 日受付)

(平成 22 年 5 月 19 日採録)



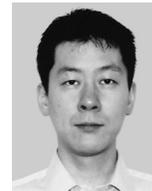
若杉 祐太 (正会員)

1985 年生。2008 年東京工業大学工学部情報工学科卒業。2010 年同大学大学院情報理工学研究科修了。現在、株式会社ソニー・コンピュータエンタテインメント。プロセッサアーキテクチャ、組み込みシステムに興味を持つ。



坂口 嘉一 (学生会員)

1987 年生。2010 年東京工業大学工学部情報工学科卒業。同年 4 月より同大学大学院情報理工学研究科計算工学専攻修士課程。計算機アーキテクチャの研究に従事。



吉瀬 謙二 (正会員)

1995 年名古屋大学工学部電子工学科卒業。2000 年東京大学大学院情報工学専攻博士課程修了。博士(工学)。同年電気通信大学大学院情報システム学研究科助手。2006 年東京工業大学大学院情報理工学研究科講師。計算機アーキテクチャ、メニーコアプロセッサアーキテクチャ、並列処理に関する研究に従事。電子情報通信学会, IEEE-CS, ACM 各会員。