Vol.2011-ARC-195 No.9
Vol.2011-OS-117 No.9
2011/4/13

CoreSymphonyの実現に向けた高性能フロントエンドアーキテクチャ

永塚 智之[†1]　坂口 嘉一[†2]
松村 貴之[†2]　吉瀬 謙二[†2]

CoreSymphony は，複数の発行幅の狭いコアを協調動作させることで 1 つの発行幅の広い仮想コアを形成し，逐次処理性能を向上させるアーキテクチャ技術である．CoreSymphony は高い逐次処理性能を達成することができる．しかし，CoreSymphony の初期実装では，フロントエンド部に最大 4 コア協調時を想定した複雑な分岐予測機構が存在する．

本稿では，従来のフロントエンドを見直し，ハードウェア複雑性の削減とさらなる高性能化を試みる．SPEC2006 ベンチマークを用いて評価を行った結果，提案するフロントエンドは，4 コア協調時において 8KB の分岐予測器を用いた場合に，平均 7.0%の IPC の向上を達成した．

A High Performance Front-End Architecture for Implementing CoreSymphony

Tomoyuki Nagatsuka,[†1] Yoshito Sakaguchi,[†2]
Takayuki Matsumura[†2] and Kenji Kise[†2]

CoreSymphony is an architecture that improves sequential performance by fusing some narrow-issue cores into one wide-issue core. CoreSymphony achieves high sequential performance. But, in initial implementation of CoreSymphony, the front-end has a complicated branch prediction mechanism for 4-way symphony.

In this paper, we review this front-end architecture, and attempt to reduce hardware complexity and improve performance. On the proposed front-end, our evaluation result using SPEC2006 benchmarks shows that the proposed front-end achieves 7.0% higher IPC than the conventional one on 4-way fusion with 8KB predictor's hardware budget.

1. はじめに

CMP（Chip Multiprocessor）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図るアプローチである．チップ当りの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)]．Amdahl の法則によれば，プログラム中に存在する並列不可能な処理（逐次処理）が CMP の性能を制限する可能性がある．例えば，全処理の内の 90%が並列化可能なプログラムがあるとする．本法則によれば，このプログラムを仮に 100 コアを使用して並列実行しても，1 コア時に対する性能向上は約 9.2 倍にとどまる．このように，10%の逐次部分が存在する場合，コア数を増やしても 1 コア比で 10 倍以上の性能向上は見込めない．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．そこで我々は，複数個のプロセッサコアが協調することで，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[3)] を提案している．本アーキテクチャでは，2 命令発行の out-of-order コアが最大で 4 コア協調することにより，8 命令発行の仮想的なコアを形成できる．

CoreSymphony の初期実装におけるフロントエンドでは，4 コア協調に対応するため，各コアに現在の PC から 8 命令先までの分岐予測を行うための機構が必要である．この分岐予測器の複雑性の削減は，CoreSymphony を実現可能なものにするための課題の一つである．

本稿では，この課題を解決すべく，分岐予測機構の改良を含む CoreSymphony 向けのフロントエンドを提案する．CoreSymphony は，トレースキャッシュ[4)] をベースとした，ローカル命令キャッシュを備えている．そこで，トレースキャッシュ向けの分岐予測機構を CoreSymphony に適用する．また，新たな分岐予測機構に合わせ，トレースに相当するフェッチブロック（FB）の構成法にも変更を加える．これらの改良により，ハードウェア複雑性の削減と，従来の CoreSymphony が達成した性能向上と比較してさらなる性能向上を実現する．

本稿の構成は以下の通りである．2 章では CoreSymphony アーキテクチャの概要を述べる．3 章では関連研究についてまとめる．4 章では従来のフロントエンドの動作とその問題点について述べる．5 章ではこの問題を解決したフロントエンドを提案する．6 章では両者の間で評価し比較を行う．最後に 7 章でまとめる．

2. CoreSymphony アーキテクチャ

2.1 CoreSymphony の基本方針

以下では，複数のコアが協調して 1 つの強力なコアを構成できるアーキテクチャ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の 1 つである．2 命

†1 東京工業大学 工学部情報工学科
Department of Computer Science, Tokyo Institute of Technology
†2 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

Vol.2011-ARC-195 No.9
Vol.2011-OS-117 No.9
2011/4/13

令発行の out-of-order コアをベースとして，最大で 4 つのコアが協調して動作できる．以下の要因により，協調動作時の高い逐次性能が達成される．

発行幅の拡大 各コアの実行ユニットは，クラスタ型の実行ユニットにおける 1 つのクラスタとして機能する．そのため，発行幅が協調コア数に比例して増加する．

L1 データキャッシュのマルチバンク化 協調動作により，データキャッシュの容量が 1 コア分の（協調コア数）倍に増加する．このとき，各コアのデータキャッシュは，アドレスによってインターリーブ化されたキャッシュのバンク 1 つとして機能する．

命令キャッシュの分散 CoreSymphony には従来型の命令キャッシュに加え，ローカル命令キャッシュを備えている．協調動作時，ローカル命令キャッシュに各コアの実行に必要な命令だけを格納することで，実質的な命令キャッシュ容量・帯域が増加する．

また，CoreSymphony では協調動作による逐次性能の向上に加え，次の 3 つの達成を目指す．

コアの独立性の維持 比較的軽量かつ均質なコアを多く搭載する CMP への適用を念頭におく．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，制御を集中化して行うことは望ましくない．CoreSymphony は，フロントエンドで一切のコア間通信を行わず，制御を各コアに分散する．

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．この変更をなるべく小さいものに留めることで，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．

バイナリの連続性の維持 従来型の RISC 命令セットによる，協調動作の実現を目指し，既存のコンパイラ最適化技術の流用やバイナリの連続性を維持する．

2.2 OS との連携

コアの融合（fuse）と分離（split）は，ワークロードの状況に応じて OS が実行時に動的に行う．逐次部分を多く含むセクションを実行する場合には，複数のコアを融合し，逐次処理を高速化する．逆に，十分に並列化されたセクションを実行する場合には，協調動作中のコアを分離し，並列処理を高速化する．

以上のことを効率的に行うアルゴリズムを考案することは，CoreSymphony の性能を最大限に引き出すための重要な課題である．

3. 関 連 研 究

本章では関連研究として，コア融合アーキテクチャの一つである Core Fusion[5] のフロントエンドの動作についてまとめる．

フロントエンドの制御 Core Fusion のフロントエンドの制御は，Fetch Management Unit（FMU）と呼ぶ集中的な機構により行われる．FMU は全コアから命令フェッチや分岐予測に関する情報を受け取り，必要な情報を各コアに通信する．

命令キャッシュ 各コアは命令を自コアの命令キャッシュから 2 命令ずつ独立にフェッチする．4 コア協調時には協調コア全体で 8 命令がフェッチされる．各コアで実行される命令は Core 0 に最も古い命令が割り当てられるように PC により整列される．Taken である分岐命令の飛び先からの命令フェッチ，又は，分岐予測ミスからの回復直後の命令フェッチでは，必ずしも Core 0 に整列されない．その場合には，余分な命令が割り当てられているコア番号の小さいコアは命令フェッチをスキップし，次の命令フェッチから，Core 0 に整列した命令フェッチを開始する．

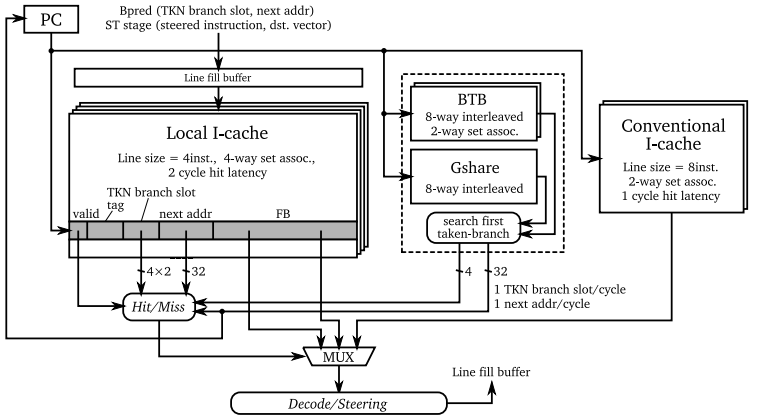

図 1 CoreSymphony における従来のシングルコアのフロントエンド．

分岐予測器 分岐予測では自身に割り当てられた命令のみを予測する．よって，分岐予測器や BTB の実効エントリ数は協調コア数により倍加させることが可能である．分岐予測ミスをあるコアが検出した場合には，そのコアが正しい PC を FMU に送信する．FMU は送られてきた PC から最も古い PC を選択し，全コアと通信することで，分岐予測ミスからの回復を行う．

グローバル分岐履歴 自コアの命令のみで GHR の更新を行っても，過去の分岐履歴との相関を利用することが出来ないため，GHR は全コアが同一のものを所有する．全コアで同じ更新が行われる必要があるため，更新は FMU により行われる．FMU は全コアの分岐予測結果を受け取り，その結果から新しい GHR を全コアに送信する．また，FMU は非投機的な GHR も持ち，分岐予測ミス時にはそれを使用して GHR の回復を行う．

このフロントエンドの動作は，複雑性を抑えつつ協調コア数の増加と共に命令キャッシュと分岐予測器のエントリ数を増加させることが可能である点で優れている．しかし，FMU のような集中的な機構が必要となる．CoreSymphony ではコアの独立性の維持という観点から，フロントエンドでの通信を行わない．そのため，CoreFusion とは異なる方法が必要となる．

4. 従来のフロントエンド

本章では，従来の CoreSymphony におけるフロントエンドの動作について述べ，その問題点を整理する．

フロントエンドのブロック図を**図 1** に示す．フロントエンドは主にローカル命令キャッシュ（Local I-cache），従来型命令キャッシュ（Conventional I-cache），分岐予測器，BTB からなる．

Vol.2011-ARC-195 No.9
Vol.2011-OS-117 No.9
2011/4/13

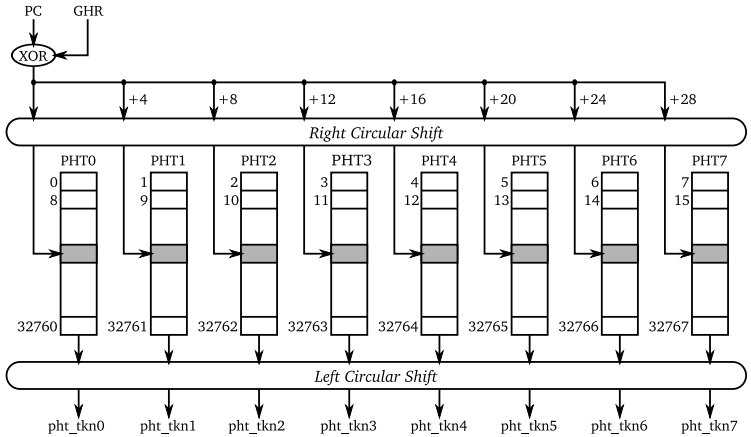

図 2 連続する 8 命令の分岐予測を行う Gshare．テーブルをインターリーブ化することで，最大 8 命令の分岐予測を 1 サイクルで行うことが可能である．8 命令の予測では同一の GHR が使用される．

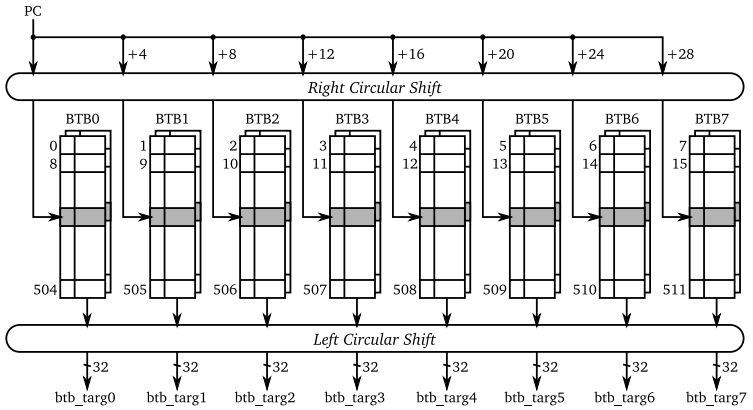

図 3 連続する 8 命令の分岐先アドレス予測を行う BTB．テーブルをインターリーブ化することで，最大 8 命令の分岐先アドレス予測を行うことが可能である．

4.1 ローカル命令キャッシュ

CoreSymphony は，単一プログラムの命令列をいくつかに分割し，複数コアにステアリングすることで各命令を並列に実行する．ステアリングの単位となる命令列をフェッチブロック（FB）と呼ぶ．ローカル命令キャッシュはこの FB 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．

FB は，協調動作中のコア数 ×2 命令に達するか Taken と予測される分岐命令までの命令ブロックを 1 単位とし，このブロックが 2 つ含まれたものと定義される．よって，FB の最大長は協調動作中のコア数 ×4 命令である．この定義では Not-Taken である分岐命令は FB 中にいくつ含まれても良い．

トレースキャッシュを構成するための情報として，ローカル命令キャッシュのライン中には TKN branch slot（4bit × 2）と next addr（32bit）を含む．この情報と後述する分岐予測の結果との比較により，Hit/Miss 判定を行う．

ミス時には，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/ステアリングを行い，ローカル命令キャッシュの該当エントリに，自身にステアリングされた命令と各種制御情報を書き戻す．

4.2 分岐予測器

4.2.1 構　　成

FB の最大長は，協調コア数を n とすると $4n$ 命令となる．そのため，n コア協調時の仮想的な $2n$ 命令発行スーパースカラでは，理想的には 2 サイクルで 1 つの FB がフェッチされる必要がある．命令フェッチに 3 サイクル以上かけることは望ましくない．

4.1 章で述べた通り，FB は 2 つの命令ブロックにより構成されている．分岐予測ではこの命令ブロックの終端位置（TKN branch slot）と次の命令ブロックの開始アドレス（next addr）を 1 サイクルで決定する．これらの予測は，分岐予測器と BTB をインターリーブ化することで複数の命令を同時に予測できるようにし，それらの結果から生成する．命令ブロックの最大長は 8 命令なので，テーブルを 8 個に分割し，連続する 8 命令分の予測を行う．

インターリーブ化された分岐予測器（Gshare[6]）と BTB のブロック図を**図 2**，**図 3** に示す．これらは，それぞれ現在の PC から最大 8 命令分の分岐予測と分岐先アドレス予測を，8 個に分かれたテーブルに別々に割り当てることにより 1 サイクルで行う．

4.2.2 動　作　例

4 コア協調時を仮定して，その動作例を述べる．

まず，現在の PC から 8 命令分の分岐予測と分岐先アドレス予測を行う．その結果から，最初に現れる Taken と予測される分岐命令のスロット番号（TKN branch slot）とその分岐先アドレス（next addr）を決定する．その後，next addr を PC にセットし，次のサイクルで分岐先アドレスから始まる 8 命令についてもう一度，同様の予測を行う．そうして得られた 2 つの TKN branch slot と 1 回目の予測で得られた next addr について，ローカル命令キャッシュのタグが一致したラインと比較し，Hit/Miss 判定を行う．よって，命令フェッチには 2 サイクルを要する．

4.3 問　題　点

以上で述べた従来のフロントエンドには 2 つの問題点が存在する．

複雑なハードウェア　8 命令先までの分岐予測が必要とされるのは，サポートする協調コア数の最大である 4 コア協調時の仮想的な 8 命令発行のスーパースカラを形成している場合のみである．例えば，1 コアの場合は 2 命令分の予測しか必要としない．分岐予測器と BTB の出力は図 2，図 3 が示す通り，分岐予測器が 1bit × 8，BTB が 32bit × 8

Vol.2011-ARC-195 No.9
Vol.2011-OS-117 No.9
2011/4/13

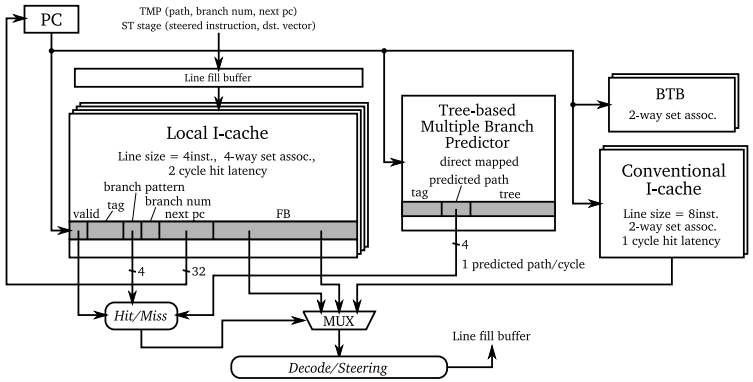

図 4 CoreSymphony における提案するシングルコアのフロントエンド.

であり，1 コアのときには 198bit の出力が使用されないことになる．CoreSymphony は CMP を対象としており，このように，1 つのコアのハードウェア複雑度が，ベースとなるコアと比較して著しく増加することは望ましくない．

グローバル分岐履歴の取扱い 本来の Gshare の動作であれば，GHR を以前の分岐予測結果により投機的に更新して，分岐予測を行う必要がある．複数命令の分岐予測を 1 サイクルで行うに当たって，先行する命令の分岐予測結果が得られてから，連続して 8 回テーブルを参照することは，クリティカルパスの問題から現実的ではない．そこで，8 命令分の分岐予測は同一の GHR を使用して並列に行われる．しかし，実際には正しい GHR を用いて予測される分岐命令は，同時に予測される命令の中で，最初に現れる分岐命令のみである．そのため，2 つ目以降の分岐命令の予測精度は低下する．この影響は，協調コア数が増えることによって FB の最大長が長くなる程，顕著になる．6 章でその影響を示す．

5. 提案するフロントエンド

ローカル命令キャッシュはトレースキャッシュである．典型的なトレースキャッシュを用いたフロントエンドでは，トレースキャッシュの Hit/Miss 判定のための特別な予測器[7)–9)]が用いられる．そこで，CoreSymphony においても，こういった予測器を導入することで前述の問題点を解決し，フロントエンドの複雑性の削減とさらなる高性能化を試みる．

提案するフロントエンドのブロック図を**図 4** に示す．予測器には Tree-Based Multiple Branch Predictor[7)]（TMP）を用いる．TMP の適用に伴い，フロントエンドのアーキテクチャを変更する．

5.1 新たな FB の定義

FB は命令トレースであるが，命令トレースの一般的な定義では，トレースは，最大長に達するか規定の基本ブロック数を含んだ位置で終端する[4)]．しかし，この定義ではトレースの終端位置が曖昧になりやすく，同じ命令を含むトレースのパターンが増加する．これにより，トレースキャッシュの利用効率が悪化し，ヒット率が低下する．

そこで，文献 10) を参考に，最大長を設定した基本ブロック単位で FB を構成することにする．これにより FB の終端は，分岐命令であるか，または，ある分岐命令の分岐先アドレスからブロックサイズごとに区切られた位置であることが保証される．これにより，命令トレースの境界が明瞭になり，ローカル命令キャッシュの利用効率が改善される．

コア数に応じた FB 中に含まれるこの命令ブロックの最大長と個数はパラメータである．本稿では，以下に示す 4 通りの構成を考える．

linear 4 命令に達するか分岐命令までの命令ブロックを 1 単位とし，この命令ブロックが協調コア数分含まれたものを FB とする．

fix1 協調動作中のコア数 × 4 命令に達するか分岐命令に出会うまでの命令ブロックを 1 単位とし，この命令ブロックが 1 つ含まれたものを FB とする．

fix2 協調動作中のコア数 × 2 命令に達するか分岐命令に出会うまでの命令ブロックを 1 単位とし，この命令ブロックが 2 つ含まれたものを FB とする．

fix4 協調動作中のコア数 × 1 命令に達するか分岐命令に出会うまでの命令ブロックを 1 単位とし，この命令ブロックが 4 つ含まれたものを FB とする．

いずれの定義も，FB の最大長は従来のものと同じである．ローカル命令キャッシュのラインサイズは従来と同じ 4 命令である．

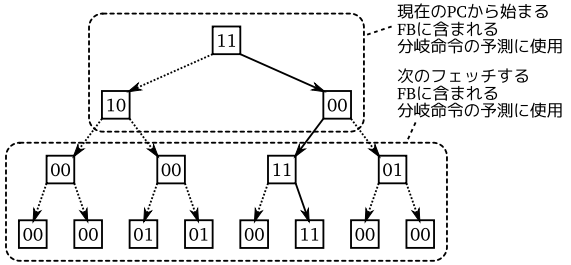

図 5 TMP の 1 つの tree を 2 つの FB の予測に使用する例．tree の上半分を分岐予測結果として使用する．残りの下半分は次の命令フェッチのために使用する．これにより，テーブル中のタグの割合を抑えつつ，tree 中に使用されない部分が減少し，Tree-PHT の利用効率が改善される．

5.2 分岐予測器

TMP による複数分岐予測では，次にフェッチすべきローカル命令キャッシュのライン中に現れる分岐命令の分岐方向のパターンを 0，1 で表したもの（predicted path）が予測結果として出力される．まず，TMP にいくつ先までの分岐命令を予測させるかを決定する．

fix1，fix2，linear（1–3 コア時）を定義とする FB は分岐命令が 1–3 個しか含まれないため，TMP はそれだけの分岐命令が 1 サイクルで予測できれば十分である．TMP で用いられる Tree-PHT はタグ付きのテーブルである．そのため，一度に予測できる分岐命令数を減らすことで実際に分岐予測のための情報が記録される tree フィールドが小さくなると，テーブル全体に占めるタグの割合が増加し，予測器が非効率なものとなる．例えば，2 命令先までの分岐命令の予測が可能な Tree-PHT のタグの割合は 60%程である．よって，TMP

は FB 中に含まれる分岐命令の数によらず，4 つ先まで予測できるものとする．

しかし，4 つ先までの予測が出来るにも関わらずその内の一部しか使用されないのは，tree 中に使用されない部分が存在することになるため，予測精度を低下させることになる．そこで，1 つの tree で複数の FB をフェッチするための手法を提案する．

提案する tree の使用方法を**図 5** に示す．これは fix2，linear（2 コア時）のように FB に 2 つ分岐命令を含む場合に，1 つの tree を，分岐命令を最大 2 つ含む FB を 2 つフェッチするために使用する例である．この tree を持つエントリで予測される分岐パターンは (TNTT) である．この予測結果が得られた後の最初の命令フェッチでは，前半の 2bit を用いた (TN**) でローカル命令キャッシュの Hit/Miss 判定を行い，FB をフェッチする．次の命令フェッチでは，TMP を参照せずに，以前の予測結果の後半の 2bit を用いた (TT**) を使用する．fix1，linear（1 コア時）のように FB 中に 1 つしか分岐命令が含まれない場合では，同様の方法で 1 回の予測結果を 4 回に分けて使用する．

この変更は TMP のハードウェア複雑度に影響を与えない．同じ tree では連続してフェッチされる FB についての予測しか行われない．そのため，tree の更新では，命令のリタイアを観測して得られた 4bit の分岐パターンに基づいて，従来のものと同様に行えばよいからである．従来の TMP から変更するのは，TMP のエントリの更新に用いる分岐パターンが 1 つの FB 内に閉じているか複数の FB に跨がっているかということと，分岐予測によって得られた 4bit の予測結果をどのように使用するかのみである．

この手法は，一度に予測する複数の FB に含まれる分岐命令の合計が 4 となる必要があるため，分岐命令を 3 個含むことができる linear（3 コア時）には適用出来ない．この場合は特別扱いし，予測結果の残り 1bit は捨てることにする．これによる，linear（3 コア時）の性能低下は 6 章で示す．

5.3 フロントエンドの変更

5.3.1 BTB

従来のアーキテクチャにおいて最も福雑なハードウェアである BTB について，ハードウェア複雑性の削減を考える．

前述の通り，提案する手法では FB のフェッチにおいて各分岐命令の分岐先アドレスは考慮せず，Taken/Not-Taken のパターンのみでローカル命令キャッシュの Hit/Miss 判定を行う．よって，FB 中の全ての分岐命令の分岐先アドレスは必要としないため，次回の命令フェッチのために，最後に出現する分岐命令の分岐先アドレスが分かれば十分である．これは BTB で予測する必要はなく，ローカル命令キャッシュのライン中に含めることで実現できる．

この変更により，ローカル命令キャッシュから FB をフェッチする際には BTB は必要としない．

5.3.2 Partial Matching

トレースキャッシュの Hit/Miss 判定に関して，分岐予測結果の Partial Matching[4),11)] をどのように扱うかは性能向上を考える上で重要な問題である．典型的なトレースキャッシュを用いたプロセッサでは，Partial Matching を行うことで性能が改善することが知られている．CoreSymphony ではローカル命令キャッシュにミスしたときに，各コアが独立に従来型命令キャッシュから命令をフェッチし，FB を構築する．そのため，Partial Matching を行うことでローカル命令キャッシュにヒットさせてしまうと，FB がローカル命令キャッシュへ格納される頻度が低下し，ヒット率に悪影響を与える．

よって，本手法では Partial Matching を行わない．

5.3.3 ローカル命令キャッシュ

提案する分岐予測機構が必要とする，ローカル命令キャッシュのライン中のトレースキャッシュとしての情報に関する変更をまとめる．

branch pattern 従来の TKN branch slot と next addr に代わり，その FB の辿る 1–4bit の分岐パターン（branch pattern）を含める．命令フェッチでは，これと TMP の予測結果である predicted path とを比較し，Hit/Miss 判定を行う．このフィールドの bit 数は FB の定義による．

branch num TMP で用いる GHR の投機的な更新のために，ライン中に，FB 中に含まれる分岐命令数（branch num）を含める．FB がフェッチされたとき，GHR を branch num だけシフトし，branch pattern を挿入することで更新する．

next pc この FB がフェッチされた後の PC（next pc）を含める．Hit/Miss 判定によりフェッチされる FB が決定されたらこのフィールドを PC にセットする．

5.4 問題点の解決

従来のフロントエンドが抱えていた問題がどのように解決されたかをまとめる．

複雑なハードウェア 提案するフロントエンドでは，BTB はローカル命令キャッシュにミスしたときにのみ使用されるため，従来より出力幅を抑えることが可能である．従来型命令キャッシュのフェッチ幅は 2 命令なので，BTB は 1 サイクルに 2 命令の分岐先アドレス予測が行えれば十分である．BTB のハードウェア複雑度を 8 命令発行のプロセッサに相当するものから 2 命令発行のプロセッサに相当するものに削減することが可能となった

グローバル分岐履歴の取扱い TMP は内部に分岐履歴の投機的な更新を含んだ予測が可能となっている．よって，従来のように本来と異なる分岐履歴を使用して予測を行うことはない．FB 中の分岐命令数に依存しない分岐予測精度を得ることが可能である．

6. 評　　価

本章では従来のフロントエンドと提案するフロントエンドの性能を評価する．

6.1 評 価 環 境

評価には，C++で記述されたサイクルレベルシミュレータを用いる．これは MIPS システムレベルシミュレータ SimMips[12)] を CoreSymphony のシミュレーション用に拡張したものである．

ベンチマークは SPEC2006 ベンチマークから INT6 種類（401.bzip2，403.gcc，429.mcf，445.gobmk，456.hmmer，464.h264ref）を用い，データセットには test を使用する．シミュレーションでは 1G 命令スキップ後の 100M 命令を評価に使用する．

シミュレーションの主なパラメータを**表 1** に示す．

6.2 従来のフロントエンドと提案するフロントエンドの性能比較

6.2.1 分岐予測精度

図 6(a) に，協調コア数ごとの分岐予測精度を示す．

従来の分岐予測器では協調コア数の増加と共に予測精度が低下することが分かる．これは FB の長さが長くなることで，4.3 章で述べたように，本来とは異なる GHR を用いて予測

Vol.2011-ARC-195 No.9
Vol.2011-OS-117 No.9
2011/4/13

表 1 シミュレーションにおける主要なパラメータ.

L1-I$ (Local)	16KB, 4-way set assoc., 2 cycle
L1-I$ (Convent.)	8KB, 2-way set assoc., 1 cycle
L1-D$	16KB, 2-way set assoc., 1 cycle
Shared L2$	2MB, 4-way set assoc., 10 cycle
Bpred	8KB
BTB	8KB, 2-way set assoc.
GHR	6bit
Instruction Window	24/24 entries (INT/FP)
PRF	32/32 entries (INT/FP)
NRP[3)]	12/12 entries (INT/FP)
LSQ	96 entries
ROB	96 entries
LWT	1K entries
Main Memory	100 cycle

される分岐命令の個数が増加するためであると考えられる．

逆に fix1，fix2，fix4 はコア数の増加と共に分岐予測精度が向上する．これは，複数分岐予測器の特性によるものである．複数分岐予測器では基本的に，トレースキャッシュの 1 つのラインをフェッチするために 1 つのエントリが使用される．そのため，トレースキャッシュのラインサイズが大きい場合には，プログラムはより少数のラインに分割され，複数分岐予測器のエントリ数を減少させても，競合の発生頻度が抑えることが可能である．CoreSymphony ではコア数により FB の最大長が長くなるため，協調コア数が増加すると共に，複数分岐予測器の予測精度は向上する．linear は 5.2 章で提案した方法により，1–2 コアで 4 コアと同じ予測精度が得られている．3 コア協調時は特別扱いをしたため，linear は 3 コア時で分岐予測精度が低下する．

コア融合アーキテクチャの目的は，逐次処理性能と並列処理性能の両立である．高い逐次処理性能が求められる場合にはコアを融合し，高い並列処理性能が求められる場合にはコアを分離する．つまり，4 コア協調時には高い IPC が期待される．しかし，従来のフロントエンドではコア数が増加するごとに分岐予測精度が低下し，この目的が達成できない．一方，提案手法では，linear は 3 コア以外での分岐予測精度の低下が無く，fix1，fix2，fix4 はコア数の増加に応じて分岐予測精度が向上するため，この目的が達成できる．

6.2.2 ローカル命令キャッシュのヒット率

図 6(b) に，協調コア数ごとのローカル命令キャッシュのヒット率を示す．

fix1，fix2，fix4 はコア数によらず，ほぼ一定のヒット率となるが，分岐命令を多く含む構成ほど，ヒット率が低下している．一方，linear はコア数が増加すると共にヒット率が低下する．このことから FB 中に多くの分岐命令を含む構成はローカル命令キャッシュのヒット率が低くなる．これは分岐命令を多く含むことが出来るようになった結果，同一の命令を含む命令トレースのパターンが増加してしまったためであると考えられる．

6.2.3 FB 中に含まれる平均命令数

図 6(c) に，協調コア数ごとのフェッチされた FB 中に含まれる平均命令数を示す．4.1 章，

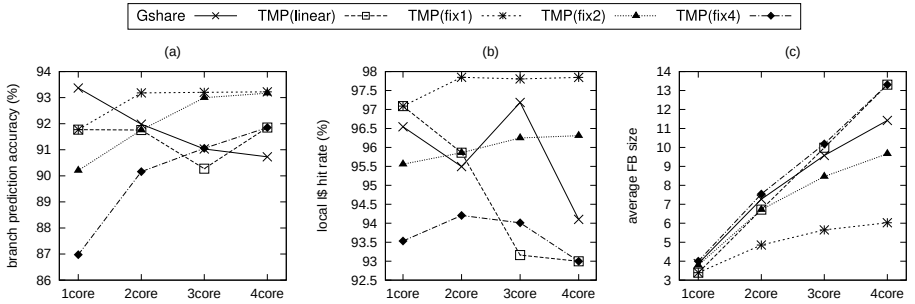

図 6 8KB の分岐予測器を用いた場合の協調コア数による性能の変化．(a) 分岐予測精度．(b) ローカル命令キャッシュのヒット率．(c) FB に含まれる平均命令数.

5.1 章で示した通り，FB の最大長は協調動作中のコア数 ×4 命令である．

fix4，linear，Gshare，fix2，fix1 の順に長い FB を形成することが可能であることが分かる．分岐命令を多く含むことが可能な構成は長い FB を形成することが可能である．

FB により多くの命令を含む構成は高い命令供給能力を持つため，高い性能が期待できる．しかし，前節で述べたように，これらの構成は低いローカル命令キャッシュのヒット率を示す．

6.2.4 IPC

以上のことをまとめると，分岐予測精度はコア数が少ない場合には Gshare が有利であり，コア数が多い場合には TMP が有利となる．ローカル命令キャッシュのヒット率は，FB 中に含むことが出来る分岐命令数が多くなる程，低くなる傾向があるが，逆に，FB の平均長は分岐命令を多く含むことが出来る方が長くなる傾向がある．

図 7 は各ベンチマークにおける協調コア数ごとの IPC，**図 8** は分岐予測器のサイズによる IPC の変化である．これらから，以上のことが性能に与える影響を総合的に見ると，fix2 が全てのコア数において平均的に高い性能を示すことが分かる．

linear はコア数によらない分岐予測精度を得ることが可能であり，平均的に長い FB を形成することが可能である．しかし，図 6(b) によると，linear はコア数が増加するとローカル命令キャッシュのヒット率が低下し，4 コア協調時では fix2 より低い IPC を示した．

TMP を用いることで多くの分岐命令の予測が容易に行えるようになったため，FB 中により多くの分岐命令を含む fix4 のような構成が可能となり，1 つ当たりの FB が長くすることが出来た．しかし，図 8 から分かるように，この構成はローカル命令キャッシュのヒット率の低下により，高い IPC 性能を出すことが出来なかったと考えられる．

fix1 は分岐予測精度，ローカル命令キャッシュのヒット率共に高い性能を示したが，図 6(c) が示すように，FB に含まれる命令数が少なく，2–4 コア時での IPC は最も低い数値を示した．

最も高い IPC を示した fix2 について，ベンチマークごとの性能を図 7 で見てみると，fix2 は 4 コア協調時において，bzip2，gcc，mcf の 3 種のベンチマークで Gshare より高い IPC となっている．特に，最も性能向上が見られる bzip2 においては IPC が 35%向上しており，

Vol.2011-ARC-195 No.9
Vol.2011-OS-117 No.9
2011/4/13

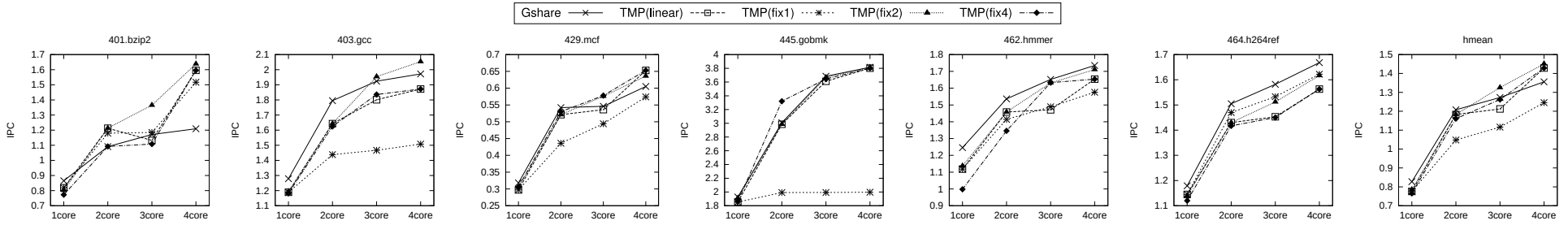

図 7 8KB の分岐予測器を用いた場合の協調コア数による IPC の変化

全ベンチマークの調和平均（hmean）では，IPC が 7.0%向上している．

6.3 BTB のハードウェア量が IPC に与える影響

従来のフロントエンドにおける BTB は 1 サイクルに 8 命令の分岐先アドレス予測を行う必要があった．一方，提案するフロントエンドにおける BTB は 1 サイクルに 2 命令の分岐先アドレス予測が行えれば十分であり，BTB のハードウェア複雑性を削減することが出来た．

従来のフロントエンドと提案するフロントエンドにおいて，BTB のハードウェア量が IPC に与える影響を考察する．BTB のハードウェア量による IPC の変化を**図 9** に示す．

BTB のハードウェア量を 8KB から 256B へ変化させたときの性能低下の度合いを調べる．従来のフロントエンドでは，1–4 コア時のそれぞれの場合において，6.0%，9.0%，9.8%，11%の IPC の低下が見られる．一方，提案するフロントエンドでは，FB の定義を fix2 としたときには，1–4 コア時のそれぞれの場合において，3.1%，4.7%，6.4%，7.3%の IPC の低下が見られる．

このことから，提案するフロントエンドは BTB のハードウェア量を小さくしても，性能に与える影響が小さいことが分かる．この理由として，提案するフロントエンドでは，BTB はローカル命令キャッシュにミスした場合のみ使用されることが考えられる．

7. おわりに

本稿では，実現困難であった CoreSymphony アーキテクチャのフロントエンドを見直し，現実的なフロントエンドを提案した．

従来のフロントエンドでは，最大 4 コア協調時に備えた複雑な分岐予測機構を持ち，1 サイクルで最大 8 命令の分岐予測と分岐先アドレス予測を行う必要があった．CoreSymphony は CMP を対象とするため，各コアのハードウェア複雑度が，ベースとなるコアに比べて著しく増加することは望ましくない．提案するフロントエンドでは，複数分岐予測器の一つである Tree-based Multiple Branch Predictor を用いることで分岐予測にかかる負担を軽減し，分岐先アドレス予測も 1 サイクルに 2 命令の予測が行えれば十分な構成とした．それに伴い，ローカル命令キャッシュの構成にも変更を加え，BTB はローカル命令キャッシュにミスした場合にのみ使用されるようにした．そのため，BTB のハードウェア量を小さくしたときの性能の低下を，従来に比べて低減させることが可能となった．

また，従来のフロントエンドには，グローバル分岐履歴の取扱いの簡略化により，協調コア数の増加と共に分岐予測精度が低下するという問題があった．コア融合アーキテクチャの目的は逐次処理性能と並列処理性能の両立である．高い逐次処理性能を求められる場合にはコアを融合し，高い並列処理性能を求められる場合にはコアを分離する．したがって，協調コア数が多い状況では高い逐次処理性能が必要とされる．従来のフロントエンドにおける，協調コア数の増加による分岐予測精度の低下は，この目的と反していた．提案するフロントエンドでは，協調コア数の増加と共に分岐予測精度が向上するため，この目的を達成することが可能である．

評価の結果，1 コア時と 2 コア協調時では IPC の低下が見られたものの，3-4 コア協調時には従来のフロントエンドより高い IPC を達成することが出来た．4 コア協調時において 8KB 程度の予測器を用いた場合に，IPC が平均 7.0%，最大 35%向上した．

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 (2006).
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) 若杉祐太，坂口嘉一，吉瀬謙二：協調可能スーパースカラ CoreSymphony，情報処理学会論文誌， Vol.3, No.3, pp.67–87 (2010).
4) Rotenberg, E., Bennett, S. and Smith, J.E.: Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching, *MICRO 29: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pp.24–34 (1996).
5) Ipek, E., Kirman, M., Kirman, N. and Martinez, J.F.: Core Fusion: Accommodat-

Vol.2011-ARC-195 No.9
Vol.2011-OS-117 No.9
2011/4/13

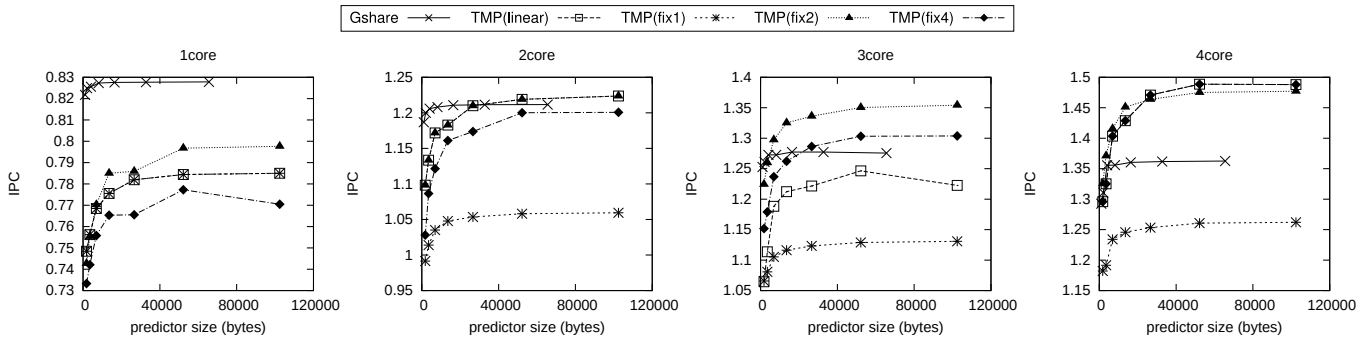

図 8 予測器のハードウェア量による IPC の変化

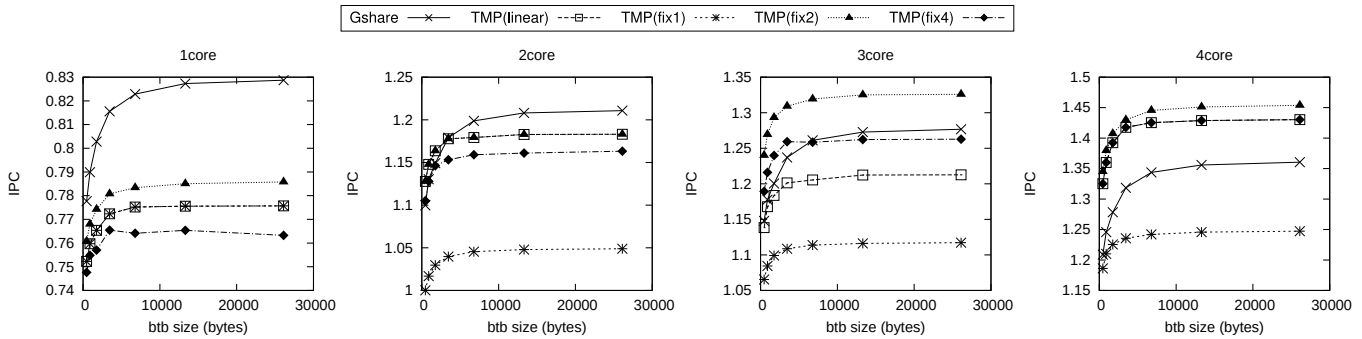

図 9 BTB のハードウェア量による IPC の変化

ing Software Diversity in Chip Multiprocessors, *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, pp.186–197 (2007).
6) McFarling, S.: Combining Branch Predictors, WRL Technical Note TN-36 (1993).
7) Rakvic, R., Black, B. and Shen, J.P.: Completion Time Multiple Branch Prediction for Enhancing Trace Cache Performance, *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp.47–58 (2000).
8) Jacobson, Q., Rotenberg, E. and Smith, J.E.: Path-Based Next Trace Prediction, *MICRO 30: Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp.14–23 (1997).
9) Moure, J.C., Benítez, D., Rexachs, D.I. and Luque, E.: Wide and Efficient Trace Prediction using the Local Trace Predictor, *ICS '06: Proceedings of the 20th Annual International Conference on Supercomputing*, pp.55–65 (2006).
10) Black, B., Rychlik, B. and Shen, J.P.: The Block-based Trace Cache, *ISCA '99: Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp.196–207 (1999).
11) Friendly, D.H., Patel, S.J. and Patt, Y.N.: Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism, *MICRO 30: Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp.24–33 (1997).
12) 藤枝直輝，渡邉伸平，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).