卒業論文

題目

マルチコアプロセッサに対応する高速な検証フレームワークの構築

指導教員

佐々木 敬泰助教

2016年

三重大学 工学部 情報工学科
コンピュータアーキテクチャ研究室

萱室 高樹 (412821)

内容梗概

近年，高性能，低消費電力の両立がプロセッサに求められている．その解決策の一つとして，異なる構成のコアを複数搭載するヘテロジニアスマルチコアプロセッサが注目されているが，各コア毎に個別の設計・検証が必要となるため，従来の方式に比べ，設計・検証コストが増大する問題がある．これに対し，当研究室ではヘテロジニアスマルチコアプロセッサの自動生成ツールセットとして FabHetero を提案している．これは，各コアやキャッシュ，バスの自動生成によって，ヘテロジニアスマルチコアプロセッサの設計コストの削減を目指すものである．また，仮想プロセッサとの検証フレームワークによって，生成したプロセッサを検証することで，検証コストの削減を図っている．しかし，現在の検証フレームワークでは，ロード，ストア動作の不整合，キャッシュコヒーレンシの破壊によって，マルチコアプロセッサの検証が不可能となっている．また，マルチコアプロセッサの並列処理性能を活用するマルチスレッドプログラムの実行には，OS の実行を伴うフルシステムシミュレーションが必要となり，実行時間が大幅に増加するため，検証コストの削減が達成されていない．

そこで，本研究では既存の検証フレームワークに，内部スケジューラによるスレッド管理機構，ロード値のバイパス機構，システムコール用キャッシュアクセス機構，といった 3 つの拡張を行うことで，マルチコアプロセッサ及びマルチスレッドプログラムに対応する高速な検証フレームワークを提案し，検証コストの削減を目指す．また，評価ではプログラムの直接実行を行う提案フレームワークの有効性を検証する．既存のマルチコアプロセッサシミュレータと比較して，シミュレーション時間の平均 77%, 最大 96%の削減を実現した．

Abstract

Currently, multi-core and multi-thread are widely used as one of the efficient computing methods. In addition, heterogeneous multi-core processor (HMP) is promising technique for achieving both high computing performance and low energy consumption by providing a suitable core for characteristic of each executing program. However, design effort for HMP is multiplied by the number of kinds of each of components because HMP must be designed and verified each core, cache and shared bus system. To solve this problem, FabHetero have been proposed as a tool-set of automatic HMP generation to reduce design efforts for HMP. Furthermore, verification cost of generated HMP is reduced by co-simulation framework with virtual processor. Nevertheress, the current framework is not enough to support the verification of multi-core processor design because of the order mismatch of load and store operation and cache coherency problem. In addition, the framework requires to execute operating system (OS) to use multi-threaded program. However, this method causes to increase exection time. Therefore, this paper proposes a design framework to support strong and rapid verification for multi-core processor with multi-threaded program. Proposed framework is composed of the following three extentions; cache access mechanism for systemcall execution, bypassing loaded value from the verified processor to virtual processor, and thread management using internal scheduler. Proposed framework achieves the reduction of verification time by 96% in maximum and 70% in average compared with the conventional method.

目 次

図 目 次

表 目 次

1　はじめに

近年，プロセッサに高性能化が要求されており，その解決策の一つとして複数のコアを用いて並列処理を行うことで処理性能を向上させるマルチコアプロセッサが広く普及している．加えて，その並列処理性能を活用するために，マルチスレッドによるアプリケーションの並列化も広く行われている．また，更なる高性能化手法として，異なる構成のコアを複数搭載するヘテロジニアスマルチコアプロセッサが注目されている．

これは，処理性能や消費電力など特性の異なるコアを搭載することによって，最適なリソースでアプリケーションの実行を行うものである．同じ構成のコアを複数搭載するホモジニアスマルチコアプロセッサに比べ，アプリケーションの特性に適応した実行が可能となるため，高性能，低消費電力を両立することが可能である．しかし，各コア毎に個別の設計が必要となり，それらに付随するキャッシュやバスの設計も複雑なものとなる．そのため，ホモジニアスマルチコアプロセッサと比べ，設計や検証コストが増大する問題がある．

この問題に対し，ヘテロジニアスマルチコアプロセッサの自動生成ツールセットとして FabHetero が提案されている．これは，各コアやキャッシュ，バスの自動生成を行うことによって，ヘテロジニアスマルチコアプ

ロセッサの設計コストの削減を目指すものである．また，FabHetero は，各モジュールを拡張し，ユーザーが任意の機能を追加することができる．このような独自拡張を行う場合には，生成したプロセッサが正しく動作しているか検証する必要があるが，ヘテロジニアス構成によって，その検証コストは非常に大きい．そこで，FabHetero では，仮想プロセッサとの検証フレームワークを用いることで，検証コストの削減を行っている．しかし，現在の検証フレームワークでは，ロード，ストア動作の不整合，キャッシュコヒーレンシの破壊によって，マルチコアプロセッサの検証が不可能である．また，マルチスレッドプログラムの実行を行う場合には，フレームワーク上で OS を実行するフルシステムシミュレーションが必要となる．この方式では，OS によるオーバーヘッドが大きく，シミュレーション時間が増大する．そのため，マルチコアプロセッサの検証コストが削減できていない．

そこで，本研究では既存の検証フレームワークに 3 つの拡張，すなわち，ロード値のバイパス機構，キャッシュへのアクセス機構，内部スケジューラによるスレッド管理を行うことで，マルチコアプロセッサ及びマルチスレッドプログラムに対応する高速な検証フレームワークを提案，実装する．詳細については，第 5 章で述べる．提案フレームワークによっ

て，マルチコアプロセッサの検証やマルチスレッドプログラムを用いた検証のコストを削減することが可能となる．

本論文は次のように構成される．まず，次章でマルチコアプロセッサ，マルチスレッドプログラムの概要，第 3 章で FabHetero 及び検証フレームワークやその問題点について述べる．第 4 章で既存のマルチコアプロセッサシミュレータについて述べ，第 5 章で提案手法とその実装である検証フレームワークの拡張について述べる．第 6 章では，提案手法の有効性を評価する．

2　マルチコアプロセッサ

本章では，マルチコアプロセッサやその構成要素について述べる．また，マルチコアプロセッサの性能を活用するマルチスレッドプログラムについて述べる．

2.1　ホモジニアスマルチコアプロセッサ

ホモジニアスマルチコアプロセッサとは図 2.1 のように，同じ構成のコアを複数搭載するマルチコアプロセッサである．各コアの構成は同じであるため，あるコアの設計データを流用でき設計・検証は容易である．各コアが独立して動作し，複数のアプリケーションを同時に実行すること

で，高い処理性能を持つ．しかし，アプリケーションの特性によって要求されるプロセッサ性能は異なるため，処理性能不足による処理時間の増加や過剰な処理性能による消費電力の増加という問題がある．

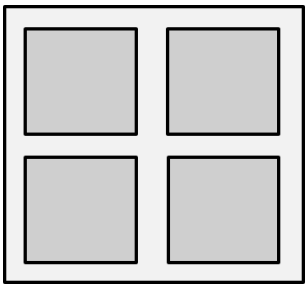

図 2.1: ホモジニアスマルチコアプロセッサ

2.2 ヘテロジニアスマルチコアプロセッサ

ヘテロジニアスマルチコアプロセッサとは図2.2のように，異なる構成のコアを複数搭載するマルチコアプロセッサである．処理性能や消費電力など各コアの特性と実行するアプリケーションの特性を考慮し，最適なコアで実行することで，高性能と低消費電力を両立することが可能である．しかし，各コアの構成が異なることから，各コア毎に個別の設計・検証が必要となるため，ホモジニアスマルチコアプロセッサと比べ，設計・検証コストが増加する問題がある．

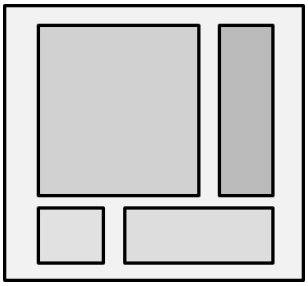

図 2.2: ヘテロジニアスマルチコアプロセッサ

2.3 スーパスカラ型プロセッサ

スーパスカラ型プロセッサとは，同時に複数の命令を実行可能なスカラ型のプロセッサを指す．スカラ型とは，1 命令が 1 つのデータに対応する処理方式のことである．スーパスカラは主に 2 つの要素から構成される．以降でその要素について説明する．

2.3.1 パイプライン処理

パイプラインとは，複数の段階に分割して命令を実行することで，処理性能を向上させる技術である．例として，図 2.3 に 5 段のパイプラインを示す．左から順に，実行する命令の取得，解析，実行，メモリアクセス，結果の書き戻しとなっている．各段階毎に異なる命令の処理を行うことができるため，単位時間あたりにより多くの命令を実行できる．スーパスカラでは，パイプラインを複数搭載することで，より多くの命令を同時実行している．

図 2.3: パイプライン処理

2.3.2 アウトオブオーダー実行

前述の通り，スーパスカラでは複数のパイプラインによって同時に複数命令を実行する．しかし，先行命令の実行結果を後続命令が使用するなど，命令間には依存関係が存在する場合がある．これによって，同時実行できずプロセッサの処理効率が低下してしまう．そこで，依存関係の無い命令から実行を行うことで処理性能を向上させる方式がある．この方式は，プログラムの記述順と異なる順序で実行を行うことから，アウトオブオーダー実行と呼ばれ，スーパスカラプロセッサでは広く採用されている．

2.4 キャッシュ

2.4.1 概要

プロセッサはメモリへアクセスすることで，実行する命令の取得や命令の処理を行うが，プロセッサとメインメモリの動作速度は大きく異なるため，性能面でのボトルネックとなる．この問題を解消するために，図 2.4 のように，動作速度や容量の異なるメモリを階層的に接続する構成が

広く用いられている．このメモリのことをキャッシュメモリと呼ぶ．

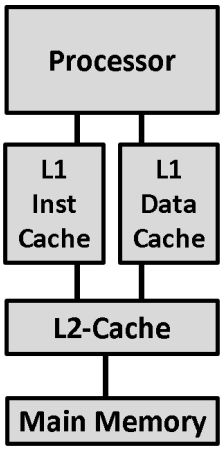

図 2.4: キャッシュメモリ

2.4.2 キャッシュコヒーレンシ

マルチコアプロセッサは複数のコアを持つため，図 2.5 のように，各コア毎に個別にキャッシュを搭載する場合がある．このとき，同じデータに対し各コアが異なる動作を要求した場合，各キャッシュは独立して動作するため，データの整合性が崩れてしまう．そこで，キャッシュコヒーレンシと呼ばれる機構によって，各コア間のデータに一貫性を持たせる．一般に，この機構はキャッシュ間の通信によって実装されている．

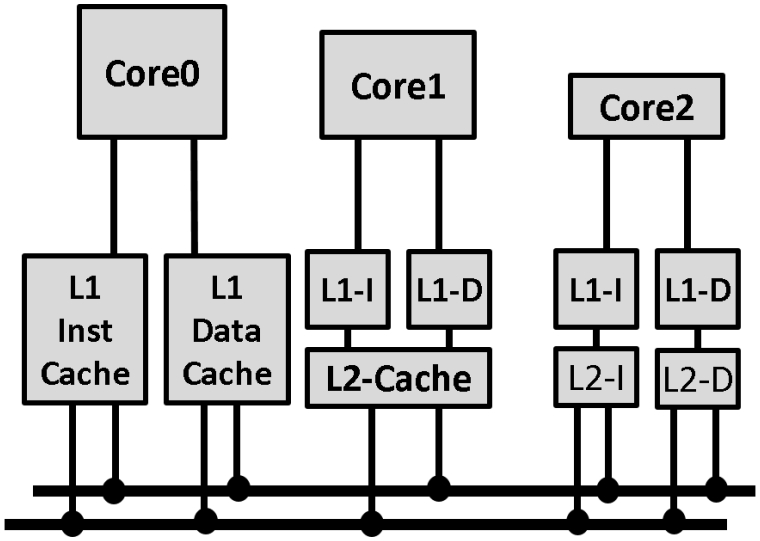

図 2.5: マルチコアプロセッサのキャッシュ構成

2.5 マルチスレッドプログラム

マルチスレッドとは，マルチコアプロセッサの並列処理性能を活用するため，プログラムを並列化し，処理性能を向上させるための手法である．これは，単一アプリケーションの実行単位をスレッドと呼ばれる単位で分割し，各スレッドを個別に処理することで並列処理を可能とする．

3 FabHetero

本章では，FabHetero の概要と検証フレームワークの詳細について述べる．検証フレームワークの動作を述べた後，その問題点を明らかにする．

3.1 概要

当研究室では，ヘテロジニアスマルチコアプロセッサを自動生成するツールセットとして，FabHetero[1] を提案している．これは，ノースカロライナ州立大学で提案されている，パラメータを与えることで様々な構成のスーパスカラコアを自動生成するツールセット FabScalar[2] に，任意の構成のキャッシュを自動生成する FabCache[3]，それらを接続するバスを自動生成する FabBus[4] を加えたものである．これらによって，論理合成可能な設計データを自動生成する．設計データは，SystemVerilog で記述されている．このような各コアやキャッシュ，バスの自動生成によって，FabHetero では，ヘテロジニアスマルチコアプロセッサの設計・検証コストの削減を目指している．図 3.6 に自動生成の概念図を示す．

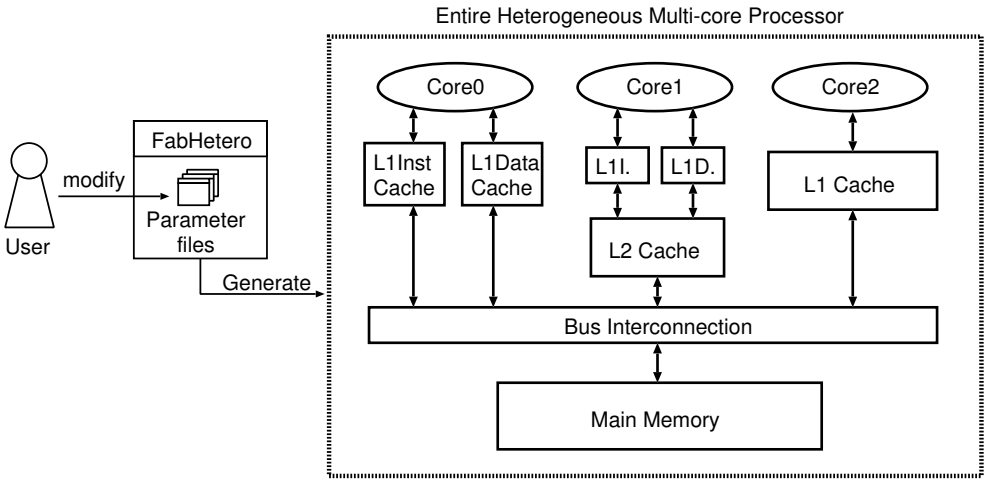

図 3.6: FabHetero によるヘテロジニアスマルチコアプロセッサの自動生成

3.2 検証フレームワーク

3.2.1 概要

FabHetero では，生成したプロセッサの動作検証を図 3.7 に示す検証フレームワークを用いて行っている．詳細は後述するが，機能シミュレータは，1 命令を 1 サイクルで実行可能な仮想的なプロセッサやレイテンシなしでアクセスできる等理想化されたメインメモリ等の動作をシミュレートする．検証フレームワークでは，検証対象となる FabHetero で生成したプロセッサと，機能シミュレータ上に構築した仮想プロセッサで，同じプログラムを同時に実行し結果を比較する．検証対象のプロセッサに問題がある場合は，実行結果に不一致が発生する．この実行結果の不一致を検出することで，生成したプロセッサの問題を容易に発見でき，検

証時間の大きな削減を行っている．

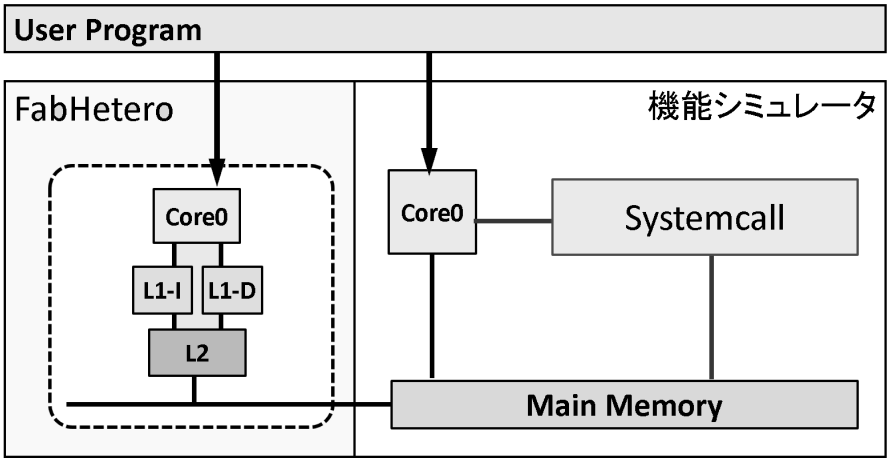

図 3.7: 検証フレームワーク

3.2.2 検証フレームワークの動作

検証フレームワークの詳細な動作について，以下で述べる．

A. 検証対象のプロセッサで命令実行

まず，検証対象となるプロセッサで命令を実行する．ただし，検証対象である FabHetero で生成するプロセッサコアは，スーパスカラ型である．第 2.3 節で述べたように，スーパスカラ型プロセッサでは複数の命令が同時に実行される．また，命令の実行順序もプログラム中の順番とは異なる場合がある．そのため，検証対象と仮想プロセッサ間で構造が異なり，単純に動作させるだけでは，結果を比

較することができない．そこで，機能シミュレータを特定のタイミングで動作させることで，構造の違いを吸収する．

B. 機能シミュレータで命令実行

機能シミュレータの仮想プロセッサは，常に同じ構成であり，1 命令を 1 サイクルで実行する．しかし，A. で述べた通り，構造の違いから，単純には結果を比較することができない．そこで，検証フレームワークでは，検証対象の命令コミット時に機能シミュレータを動作させる．命令コミットとは，命令の実行が完了し実行結果が確定する段階のことで，常にプログラム中の命令順と同じ順序で行われる．コミットされた命令の分だけ機能シミュレータで命令を実行することで，構造の違いを吸収している．

C. 動作結果比較

最後に，検証対象と機能シミュレータでの命令の実行結果を比較する．機能シミュレータは，検証対象の採用する命令セット (ISA：Instruction Set Architecture) に準拠した動作を行うため，結果が異なる場合は，検証対象に問題が発生していることが確認できる．

このように検証フレームワークを動作させることで，検証対象のプロセッサの問題を容易に発見することができ，検証コストの削減に寄与している．また，機能シミュレータ上の仮想プロセッサの構成を単一のものとしながら，様々な構成のプロセッサを検証対象とすることを可能としている．これによって，検証環境の構築コストも削減されている．

3.3 機能シミュレータ

3.3.1 仕様

検証フレームワークで用いる機能シミュレータについて述べる．機能シミュレータはC++で記述され，図3.8のようにプロセッサとメインメモリが直接接続されているシステムをシミュレーションする．仮想プロセッサは，MIPS32Release2 ISAに準拠し，1サイクルで1命令をインオーダー実行するモデルとなっている．機能シミュレータでは，静的リンクされたユーザープログラムとELF形式のLinuxカーネルが動作可能であり，ユーザープログラムの使用するOSはLinuxを想定している．

3.3.2 システムコール実行機能

システムコールとは，ファイル処理やメモリ管理といったOSの補助が必要な処理をプログラムがOSに依頼するための機構であり，対応する処

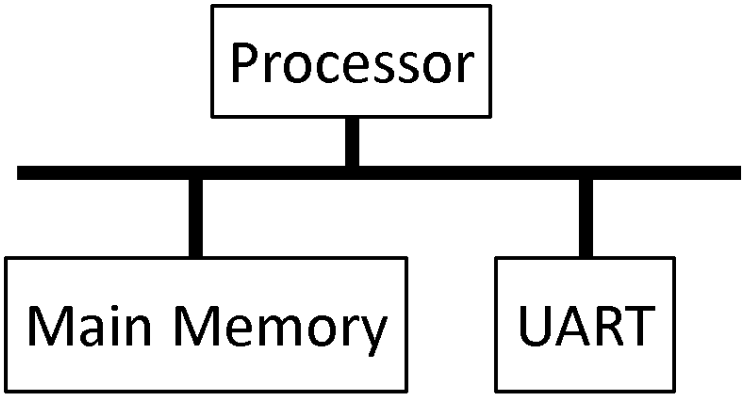

図 3.8: 機能シミュレータのシミュレートするシステム

理ルーチンはOSが保持している．したがって，システムコールを利用する一般的なプログラムの実行には，シミュレータに割込みや周辺機器等のOSの実行に必要な機構を実装した上で，OSも含めたシステム全体のシミュレーション，すなわちフルシステムシミュレーションを行う必要がある．しかし, この方式ではシミュレーション時間の増加や，OSによる処理の影響を考慮しなければならない問題がある．そこで，システムコールを内部処理する機能が機能シミュレータには実装されている．図3.9に実行機能の概要を示す．機能シミュレータでは，システムコールが要求された場合，対応した処理を内部で実行する．加えて，ファイル処理など一部のシステムコールについては，機能シミュレータ自体を動作させているOS（ホストOS）に対して，対応するシステムコールを要求する．これによって，機能シミュレータ上にOSを構築することなく，シ

ステムコールを処理することが出来る．この機能を活用し，機能シミュレータでは，プログラムを直接実行することで検証時間を削減している．

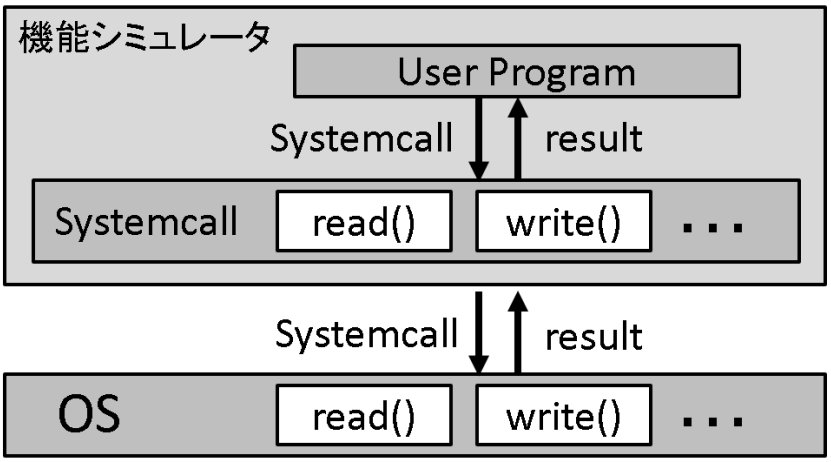

図 3.9: システムコール実行機能

3.4 問題点

マルチコアプロセッサの検証では，シングルコアプロセッサとの構造の違いから，ロード，ストア動作の不整合，キャッシュコヒーレンシの破壊およびマルチスレッドプログラムの直接実行の 3 点に問題が発生する．

3.4.1 ロード，ストア動作の不整合

検証フレームワークでは，第 3.2.2 項で示したように，機能シミュレータを検証対象のコミット時に動作させることで，検証対象との構造の違いを吸収している．マルチコアプロセッサを検証対象とする場合，各コ

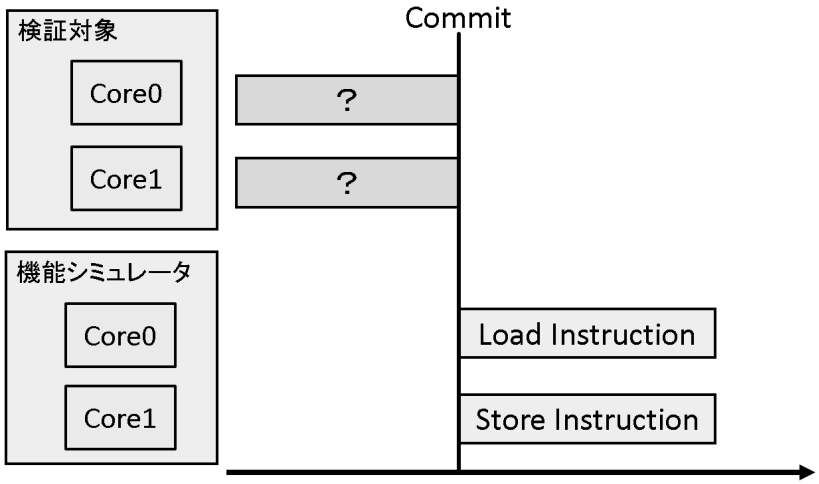

図 3.10: 2 コアプロセッサのロードストア動作

アが独立して動作するため，ロード，ストアが同時に行われる可能性がある．この場合，現在のフレームワークでは，ロードストア動作に不整合が発生し，検証が正しく行えない問題がある．2 コアのプロセッサでのロード，ストア動作を例として，図 3.10 を用いて説明する．なお，検証対象のプロセッサコアは，図 2.3 に示したようなパイプライン処理を採用しているとする．

A. 検証対象のプロセッサで命令実行

まず，検証対象となるプロセッサで命令を実行する．図 3.10 では，コア 0 でロード命令，コア 1 でストア命令が同時にコミットされている．

B. 機能シミュレータで命令実行

次に，機能シミュレータを動作させる．しかし，図 3.10 に示すように，機能シミュレータでは，検証対象が実際にメモリにアクセスしたタイミングや各コア間での優先度などを特定できない．そのため，メモリへのアクセス要求の順序が決定できず，検証基準となる仮想プロセッサの正しい動作を定義できない．

検証対象のプロセッサの動作は，実行した命令間の依存関係やパイプライン，キャッシュ構成といった様々な要素に起因して変化する．加えて，検証対象はヘテロジニアスなコア構成やキャッシュを搭載した複雑なものとなるため，正しい動作の導出は非常に困難である．

このように，マルチコアプロセッサの検証では，ロード，ストア動作に関して，仮想プロセッサの正しい動作を決定できないため，検証フレームワークが破綻する．

3.4.2 キャッシュコヒーレンシの破壊

マルチコアプロセッサでは，コア間で非共有なキャッシュを搭載する場合がある．このような構成では，第 2.4.2 項で述べたように，キャッシュコヒーレンシ機構が実装されている．

しかし，第 3.3.2 項で述べた機能シミュレータのシステムコール実行機

能では，検証対象のキャッシュコヒーレンシ機構を考慮せずにメモリに直接アクセスを行う．キャッシュコヒーレンシ機構は，ロードストア動作時にキャッシュ間で相互に通信することで実現されている．このため，メモリへのアクセスを他のキャッシュが検知できず，キャッシュコヒーレンシが破壊される．これにより，システムコールの実行機能を用いることが出来ず検証時間が増加する問題がある．

3.4.3 マルチスレッドプログラムの直接実行

マルチスレッドプログラムの実行では，スレッドの作成や終了といったスレッド管理に関するシステムコールが要求される．一般に管理機構はスケジューラと呼ばれ，レジスタなどプロセッサの内部情報を入れ替えることでOSがスレッド管理を行っている．しかし，第3.3.2項で述べたシステムコールの実行機能にはスレッドの管理機構が存在しないため，マルチスレッドプログラムの直接実行は不可能となっている．そのため，マルチスレッドプログラムの実行にはフルシステムシミュレーションを行う必要があるが，この方式では検証時間が大幅に増加する問題がある．

4　関連研究

本章では，システムコール実行機能を搭載する既存のマルチコアプロセッサシミュレータについて述べる．これらと比較し，提案手法は，マルチスレッドプログラムの実行及びマルチコアプロセッサのシミュレートを高速かつ検証用途に適用可能な手法により実現する点で優位性を持つ．

4.1　SimMips

SimMipsは，機能シミュレータと同様のシステムコール実行機能を持ち，ユーザープログラムを直接実行することができる．マルチスレッドプログラムの実行では，スレッドが作成される毎に，仮想プロセッサを新規作成し，そのプロセッサに作成したスレッドを割り当てる．つまり，スレッドと仮想プロセッサが１対１に対応するモデルを採用している．これによって，スケジューラを用いずにマルチスレッドプログラムの直接実行を実現している．しかし，実際のプロセッサではコア数は不変であるため，このような動作を行うことは不可能である．したがって，プロセッサの検証用途ではこの手法を適用することはできない．

4.2 Gem5

Gem5では2つの動作モードが実装されている．SystemEmulationModeでは，機能シミュレータと同様のシステムコールの内部実行を行うが，スケジューリングが必要なマルチスレッドプログラムは動作させることができない．そのため，マルチスレッドプログラムの実行には，Gem5上にOSを起動してのフルシステムシミュレーションを行うFullSystemModeでの動作が必要となる．フルシステムシミュレーションでは，スレッドとプロセッサが多対多に対応し，OSの持つスケジューラによって，各スレッドが管理される．つまり，実際のプロセッサと同じ動作を行うため，プロセッサの検証に利用できるが，実行時間が増加する問題がある．

5　提案手法

本章では，既存のフレームワークで対応できないマルチコアプロセッサの検証や，マルチスレッドプログラムを用いた検証を可能とする提案手法について述べる．提案手法では既存フレームワークに対し，第3.4節で述べた3つの問題点，すなわち，ロード，ストア動作の不整合，キャッシュコヒーレンシの破壊およびマルチスレッドプログラムの直接実行にそれぞれ対応する，ロード値のバイパス機構，システムコール用のキャッシュアクセス機構，内部スケジューラによるスレッド管理の3つの拡張を行う．

5.1　マルチコアプロセッサに対応する検証フレームワーク

本研究では，既存のフレームワークに対して，ロード値のバイパス機構，キャッシュへのアクセス機構，内部スケジューラによるスレッド管理の3つの拡張を行い，マルチコアプロセッサ及びマルチスレッドプログラムに対応する高速な検証フレームワークを提案，実装する．図5.11に提案する検証フレームワークを示す．以降の節でそれぞれの詳細を述べる．

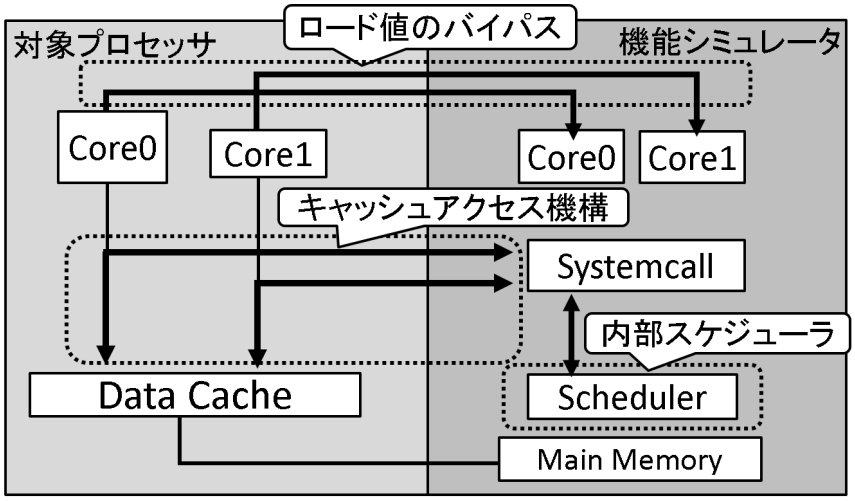

図 5.11: マルチコアプロセッサ対応検証フレームワーク

5.2 ロード値のバイパス機構

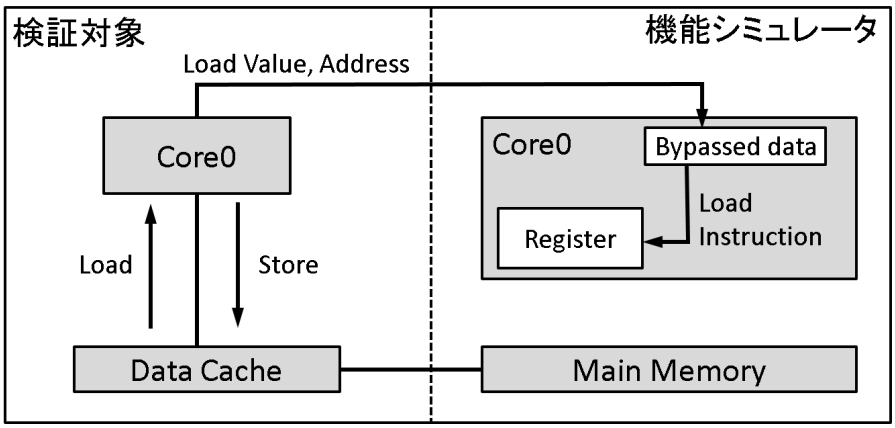

図 5.12: ロード値のバイパス機構

第 3.4.1 項で述べたように，現在のフレームワークでは，マルチコアプロセッサの検証時にロード，ストア動作に不整合が発生する．これは，プロセッサ構成が異なることが原因であるため，検証対象と同じ構成の仮

想プロセッサを作成することで解消できる．しかし，プロセッサの構成は各コアの構造や，キャッシュ構造などの組み合わせによって多岐にわたる．そのため，常に検証対象と同じ構成の仮想プロセッサを構築するには，任意の構成のコアやキャッシュへの対応が必要となり，大きなコストが必要となる．

そこで，提案フレームワークでは，検証対象がロード命令で取得した値を機能シミュレータに与えることで，ロード，ストア動作の不整合を解消する．図 5.12 を用いて，バイパス機構の動作を以下で示す．

A. ロード命令の実行

ロード命令を実行する場合について説明する．検証対象のプロセッサは，図 5.12 左のように通常と同じ動作を行う．次に，機能シミュレータの仮想プロセッサを動作させる．この際，図 5.12 上部のように，検証対象がロードした値とそのアドレスを機能シミュレータにバイパスする．仮想プロセッサでは，バイパスされたデータをレジスタに書き込むことで，メモリへのアクセスなしにロード命令を実行する．

B. ストア命令の実行

ストア命令を実行する場合について説明する．検証対象のプロセッ

サは，ロード命令と同様に通常の動作を行う．機能シミュレータの仮想プロセッサでは，メモリへの書き込みを実行しないように変更する．

このように，バイパスされた値を用いることで，検証対象と同じ値をロードすることが可能になる．また，メモリへのアクセスを行わないことで，検証対象のキャッシュコヒーレンシの破壊を防止している．これらにより，プロセッサ構成に依存するロード，ストア動作の不一致が解消し，マルチコアプロセッサの検証が可能となる．この時，仮想プロセッサの構成は単一のものとなっている．つまり，検証対象と同じ構成の仮想プロセッサを構築する必要がないため，検証に対するコストが大きく削減されたと言える．

5.3 システムコール用キャッシュアクセス機構

第 3.4.2 項で述べたように，検証対象がキャッシュコヒーレンシ機構を持つ場合，それを考慮したメモリアクセスが必要となる．しかし，システムコールの内部処理時には，仮想プロセッサでのみメモリにアクセスするため，第 5.2 節で述べた手法を適用することが出来ない．

そこで，システムコールの内部処理時に，検証対象の最上位データキャッ

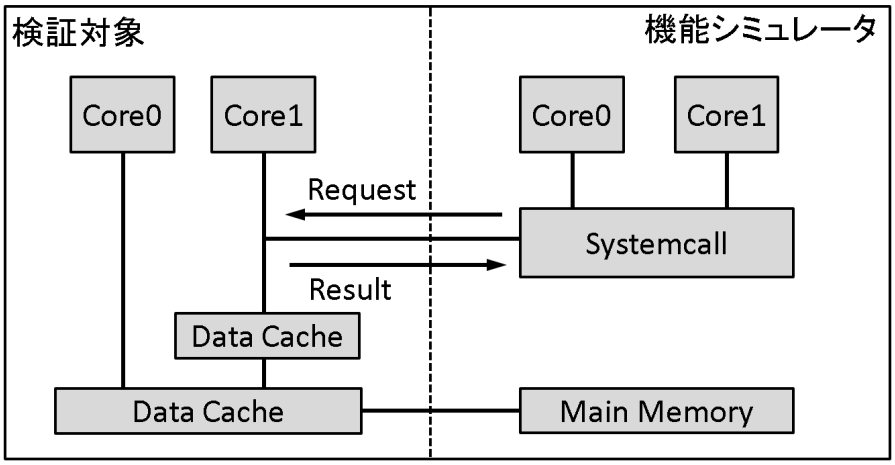

図 5.13: キャッシュアクセス機構

シュにアクセスする機構を追加する．システムコールの内部処理時にメモリアクセスが必要となる場合は，この機構を使用してアクセスする．図 5.13 を用いて，システムコール内部処理時のメモリアクセスを説明する．図 5.13 では，コア 1 がシステムコールを実行し，メモリアクセスを要求している．

A. データキャッシュへのアクセス要求

メモリへのアクセス要求が発生した場合，システムコールを実行したコアに対応する検証対象のキャッシュに対して，ロード，ストア要求を行う．図 5.13 では，コア 1 の最上位データキャッシュに対して要求を行う．この要求は，検証対象のプロセッサとキャッシュとの接続バスに対して，信号を送ることで実現する．接続バスに送る

信号は，検証対象のコアが使用する信号と同じフォーマットを用いる．また，一連のロード，ストア動作が完了するまで，検証対象の対応するコア (図 5.13 ではコア１) へのクロック供給を停止する．

B. キャッシュ動作

要求を受け取った検証対象のキャッシュは，その仕様に沿ってロード，ストア動作を行う．これは，コアがデータアクセス要求を行った場合と同じ動作である．また，実装されているキャッシュコヒーレンシ機構によって，キャッシュ間のデータ整合性が保たれる．

C. キャッシュからのデータ取得

接続バスを監視し，ロード，ストア動作の完了を検知する．ロード動作の場合は，ロード結果を機能シミュレータ側に通知し，機能シミュレータ側の動作を再開する．

このように，データキャッシュにアクセス要求を行うことで，検証対象の持つキャッシュコヒーレンシ機構を破壊せずに，メモリへのアクセスが可能となる．これにより，システムコールの内部処理によるプログラムの直接実行が可能となり，検証時間が削減できる．加えて，キャッシュコヒーレンシ機構自体は検証対象が実装しているものを使用するため，様々

なキャッシュコヒーレンシプロトコルに対応することができる．

5.4 内部スケジューラ

5.4.1 概要

第3.4.3項で述べたように，マルチスレッドプログラムの実行にはスレッド管理が必要である．第4章で述べたように，SimMipsではコア数を動的に変化させる手法によって，マルチスレッドプログラムの直接実行を可能にしているが，現実のプロセッサではコア数は不変となるため，検証用途には適用できない．したがって，プロセッサの検証用途では実際のプロセッサで可能な動作でスレッド管理を行う必要がある．また，Gem5ではフルシステムシミュレーションを行うため，シミュレーション時間が増加するという問題がある．そこで，本研究では機能シミュレータにスケジューラを実装し，内部でスレッド管理を行う．これにより，マルチスレッドプログラムをOSを実行せず直接実行することが可能となる．

5.4.2 仕様

実装したスケジューラの仕様について述べる．スケジューラは，スレッドを各コアで実行中，実行可能，待ち状態の3状態で管理し，図5.14のように，それぞれの管理領域を持つ．特に，実行可能，待ち状態のスレッ

ドを管理する領域を，ランキュー，ウェイトキューと呼び，スレッド毎に以下の情報を保持する．なお，現在の FabScalar は MIPS32Release2 ISA を実行するコアを生成対象とするため，各レジスタもそれに従ったものとなる．

- 各種レジスタ：PC，汎用レジスタ (GPR)，Hi，Lo，C0_EPC，浮動小数点レジスタ (FPR)，CP1_FCSR
- スレッド局所記憶 (Thread Local Storage,TLS) アドレス
- スレッド ID(TID)

スケジューリングアルゴリズムは，優先度付き FIFO，優先度付きラウンドロビンの 2 つが選択可能となっている．

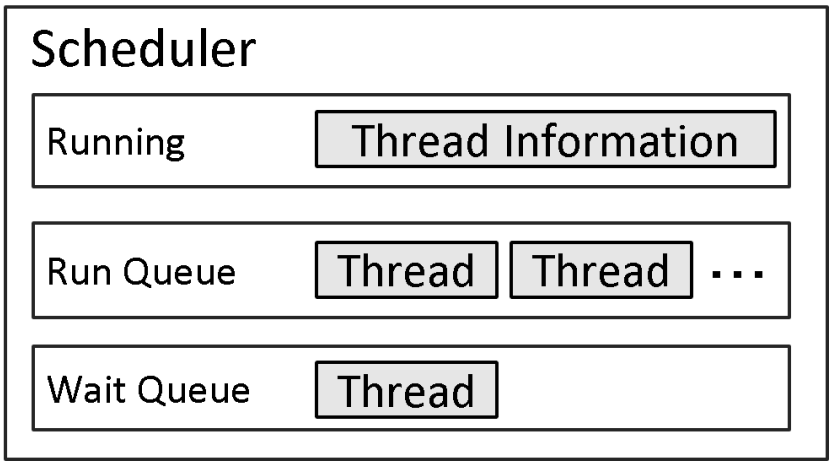

図 5.14: スケジューラの内部構造

5.4.3　スレッド管理

スケジューラのスレッド管理時の動作について述べる．以下に，スレッド管理要求の分類と対応するスケジューラの動作について示す．

A. スレッドの新規作成

対応システムコール：clone()

新規スレッドの情報をシステムコールを要求したスレッド，システムコールの引数等から適切に設定する．その後，ランキューに新規作成したスレッドを登録する．

B. 実行中スレッドの終了

対応システムコール：exit()

実行中のスレッドの情報の破棄を行う．次に，新たに実行するスレッドの情報をランキューから取得する．取得した情報を該当コアで実行中のスレッドを管理する領域に反映する．この時、ランキューから該当スレッドの情報は破棄する．

C. 実行中スレッドの休止

対応システムコール：futex()

現在実行中のスレッドの情報の更新を行う．具体的には，管理領域

の各種レジスタの値をプロセッサの保持している値に更新する．その後，ウェイトキューにスレッド情報を移動する．次に，B と同様に新たに実行するスレッドを該当コアの管理領域に割り当てる．

D. 休止中スレッドの復帰

対応システムコール：futex()

復帰要求のあったスレッドをウェイトキューから検索し，ランキューに移動させる．この時，該当スレッドはウェイトキューから破棄する．

E. 実行するスレッドの切替

対応システムコール：sched_yield()，スケジューリングによる切替

C と同様に実行中のスレッドの情報を更新した後，ランキューに情報を移動する．その後，B,C と同様に新たに実行するスレッドを割り当てる．

以上に示したように，内部スケジューラはスレッドの管理要求に応じて処理を行うことで，OS を用いずにマルチスレッドプログラムの実行を可能にする．

5.4.4　検証フレームワークにおけるスレッド管理

第5.4.3項で，スケジューラでのスレッド管理動作について述べた．A,Dの場合では，実行するスレッドそのものは変化しないため，スケジューラ内で情報を更新するのみである．しかし，B,C,Eの場合には，実行するスレッドを切り替えるため，スケジューラの更新に加え，検証対象のプロセッサ，仮想プロセッサでも情報を更新する必要がある．機能シミュレータを単体で動作させる場合は，仮想プロセッサの内部状態を直接更新可能だが，検証フレームワークの場合，検証対象のプロセッサの内部状態は直接更新できない．そのため，検証対象のプロセッサでも可能な動作によって，実行スレッドを切り替える必要がある．そこで，検証対象のプロセッサに実装されているロード，ストア命令を使用し，メモリを通して内部状態の更新を行う．図5.15を用いて説明を行う．

1. プロセッサ情報の保存

図5.15のAに示すように，対象となるプロセッサの各種レジスタの値をメモリ上の特定アドレスに対して書き込む．この書き込みはストア命令を実行することで実現する．なお，アドレスはコア毎に指定されている．

2. スケジューラによるスレッド切替

次に，図 5.15 の B に示すように，スケジューラでは保存した情報をメモリから取得し，前述の通りスレッド管理を行う．この時，メモリからの情報の取得は，第 5.3 節で実装したキャッシュアクセス機構を使用し，検証対象のキャッシュ機構を通して取得する．管理処理が完了した後は，各種レジスタの新しい値を対応するアドレスに書き込む．この書き込みも同様にキャッシュアクセス機構を用いて行う．

3. プロセッサ情報の更新

最後に，図 5.15 の C に示すように，スケジューラの更新結果をロード命令を用いて，対応アドレスから各種レジスタに反映する．

以上のように，メモリを介してスケジューリング結果を反映することで，検証フレームワークでも，OS を起動せず，マルチスレッドプログラムを直接実行可能となる．

5.4.5 ラウンドロビンスケジューリング

実装した内部スケジューラは，スケジューリングアルゴリズムとして，優先度付き FIFO と優先度付きラウンドロビンを採用している．FIFO ス

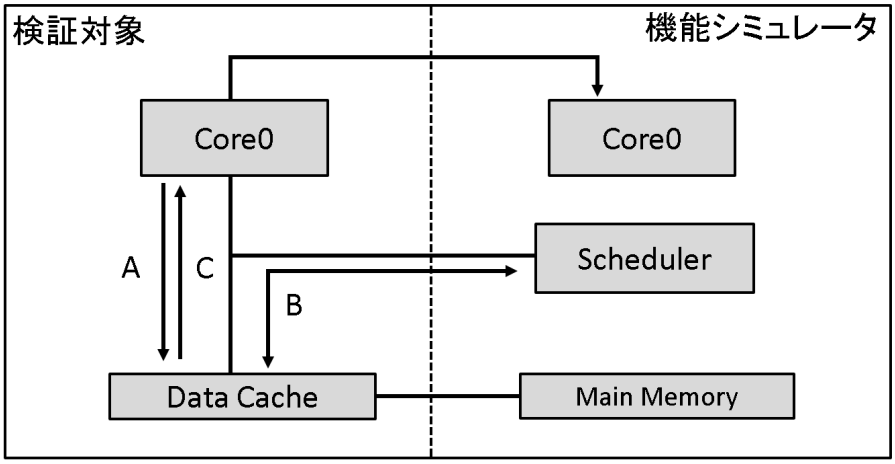

図 5.15: 提案フレームワークでのスレッド更新

ケジューリングでは，プログラムからの要求であるシステムコールを処理するのみで，スケジューリングを実現することができる．しかし，ラウンドロビンスケジューリングは，一定時間毎に実行スレッドを切り替えるアルゴリズムであり，スケジューラが主導となって，スケジューリングを行う必要がある．そこで，プロセッサの持つタイマー割り込み機構を使用して，スケジューリングを行う．タイマー割り込み機構とは，設定した命令数をプロセッサが実行した場合，特定の処理ルーチンにジャンプする機構である．以下に処理内容を示す．

A. タイマー割り込みの発生

実行中のスレッドが一定の命令を実行した場合，タイマー割り込みを発生させる．タイマー割り込みによって，スレッドの実行権を放棄

するシステムコールである sched_yield() を呼び出す．このとき，システムコールの呼び出しによって，V0，A3 レジスタの値が破壊されるため，V0，A3 レジスタを追加保存しておく．また，sched_yield() がタイマー割り込みによる呼び出しか判別するため，C0_EPC に特定の値を書き込む．C0_EPC は元のルーチンに復帰するためにも必要であるため，C0_EPC も追加保存する．

B. 実行スレッドの切替

sched_yield() の呼び出しによって，実行するスレッドを切り替える．基本的な動作は，第 5.4.3 項で述べた通りであるが，C0_EPC の値が特定の値の場合は，A. で追加保存した各レジスタ情報もスケジューラで管理を行う．

C. タイマー割り込みからの復帰

タイマー割り込みから復帰を行う場合は，スケジューラの更新結果をレジスタに反映させた後，追加保存した各種レジスタを更に書き戻すことで状態の復元を行う．

このような拡張によって，スケジューラが主導する必要のあるラウンドロビンスケジューリングを実現した．

5.4.6 アイドルタスク

マルチコア環境では，実行可能なスレッド数がコア数より少ない場合がある．その場合，スレッドが割り当てられないコアの動作を規定する必要がある．そこで，シミュレート開始時にアイドルタスクをコア数-1 個起動する．アイドルタスクは，ユーザープログラムと同じメモリ空間に存在するスレッドと定義することで，実装したスケジューラで管理できる．スケジューリング優先度は最低となっており，メモリへのアクセスは，実行する命令の取得以外には存在しない．これにより，ユーザープログラムのスレッドへの干渉を最小限に抑える．加えて，タスクの内容は無限ループとなっているため，アイドルタスクが終了することは無く，常に各コアに対して実行可能なスレッドが提供される．また，アイドルタスクは，一定の命令数毎に sched_yield() を呼び出すことで，FIFO スケジューリングを使用する場合でも，ユーザープログラムのスレッドを優先的に実行することができる．

6 評価

本章では，提案手法の有効性をする．評価環境，結果を示し，考察を行う．

6.1 評価環境

提案する内部スケジューラによるプログラムの直接実行の有効性を評価する．評価には，splash2，姫野ベンチマーク [7] を使用し，内部スケジューラを実装した機能シミュレータと Gem5 で各ベンチマークを実行し，その実行時間を計測する．表 6.1 に，シミュレータの設定及び実行環境を，表 6.2 に，各ベンチマークの設定を示す．特に記述がない部分についてはデフォルトの設定となっている．また，スレッド数は全て 16 としたため，表 6.2 からは省略する．なお，提案手法と Gem5 で ISA が異なっているが，MIPS 版 Gem5 が正常に動作しなかったため，ISA に共通点が多い Alpha を比較対象として採用したからである．

表 6.1: シミュレータの実行環境

評価対象		機能シミュレータ+内部スケジューラ (MIPS)
		Gem5 FullSystemMode(Alpha)
仮想コア数		1〜4
実行環境	CPU	Core i7-2600
	動作周波数	3.4GHz
	メモリ	16GB

表 6.2: 各ベンチマークの設定

ベンチマーク名	問題サイズ	備考
FFT	-m20	splash2 は全て Pthread
CHOLESKY	tk29.O	
LU	-n512	
LU(NON_CONTIGUOUS_BLOCKS)	-n512	
RADIX	-n10000000	
BARNES	16384 bodies	
OCEAN	-n514	
OCEAN(NON_CONTIGUOUS_PARTITIONS)	-n514	
RADIOSITY	test	-batch
FMM	input.16384	
RAYTRACE	car.env	-m64 に設定
VOLREND	head	
WATER-NSQUARED	512 mols,3step	
WATER-SPATIAL	512 mols,3step	
HIMENO(pthread)	SMALL	ループ回数は 200 固定
HIMENO(OpenMP)	XS	ループ回数は 200 固定

6.2　評価結果

図 6.16，図 6.17，図 6.18 に結果を示す．なお，図の都合から，表 6.2 とはベンチマークの並び順が異なっている．また，Gem5 では Pthread 版姫野ベンチマークが 100 時間以上経過しても終了しなかったため，正常に動作していないと判断し，図 6.18 からは除外した．結果として，内部スケジューラを用いた提案方式では，全てのベンチマークが動作し，Gem5 に対しシミュレート時間を平均 77%，最大 96%削減できた．

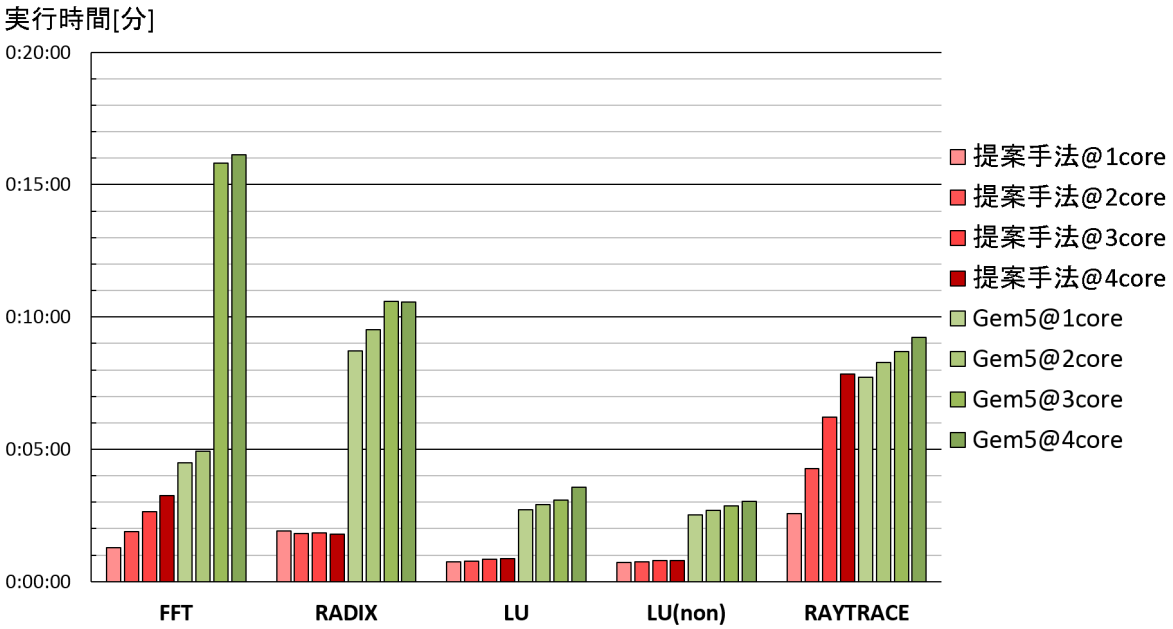

図 6.16: 評価結果 1

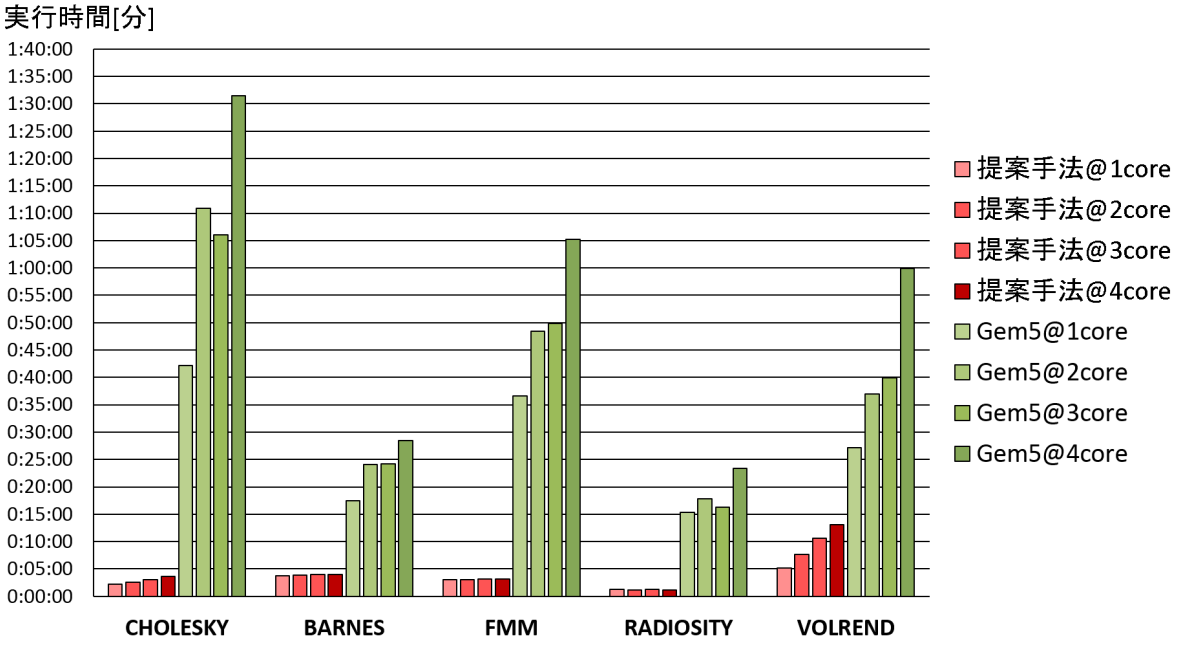

図 6.17: 評価結果 2

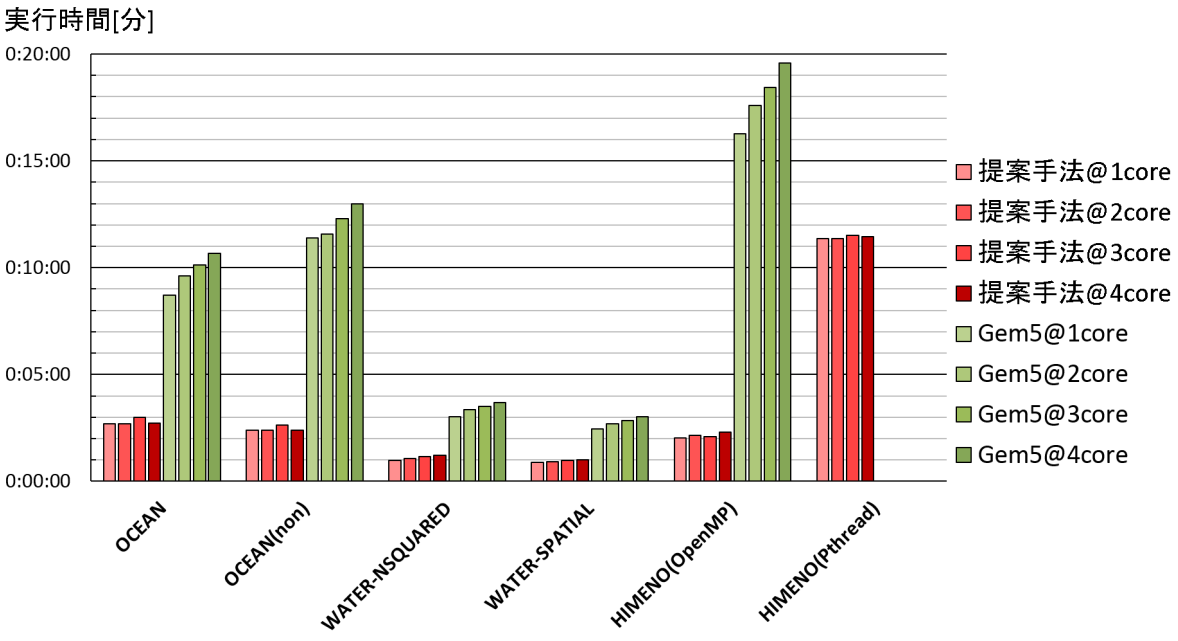

図 6.18: 評価結果 3

6.3 考察

提案手法は，全てのベンチマークセットでGem5よりも短時間でシミュレーションを行うことができた．また，全てのベンチマークが正常に実行できたことから，提案手法はPthread，OpenMPに対応している．したがって，検証用途に幅広いマルチスレッドプログラムを用いることができる．

提案手法で実行時間が大きく削減された原因としては，OSによる処理が一切発生していないことによって，プロセッサが実行する命令数が大幅に減少したことが挙げられる．また，検証対象となるプロセッサの命令実行数が減少しているため，プロセッサの検証フレームワークにおいても同様の高速化が期待できる．したがって，システムコールの実行機能と実装したスケジューラによるプログラムの直接実行は，検証時間の削減に対して有効である．

一方，コア数が増加した場合，一部のベンチマークで実行時間が増加傾向にある．これは，機能シミュレータのマルチコア動作の特性と，マルチスレッドプログラムの動作特性によるものである．機能シミュレータが単位時間あたりに処理可能な命令数は，コア数が増加しても一定である．そのため，各コアが単位時間に処理できる命令数はコア数の増加に

反比例する．つまり，機能シミュレータの動作速度は，シミュレート対象のコア数が増加するほど低下する．しかし，同時に実行されるスレッド数はコア数に比例し増加するため，実行するプログラム自体の並列度が高ければ，マルチスレッドプログラム全体での処理効率は向上する．したがって，実行効率の高いマルチスレッドプログラムでは，シミュレーション時間の増加を軽減できているが，実行効率の低いマルチスレッドプログラムでは，シミュレーション時間が増加傾向にあると考えられる．

7　おわりに

本研究では，既存のフレームワークに対し拡張を行い，マルチスレッドプログラムの直接実行及びマルチコアプロセッサの検証が可能となるフレームワークを構築した．また，Gem5 と比較して平均 77%, 最大 96%シミュレーション時間を削減できた．これにより，提案した検証フレームワークの有効性を示した．今後の展望としては，提案フレームワークを用いた FabHetero の動作検証やコア数が増加した場合のシミュレーション速度の改善が挙げられる．

謝辞

本研究を行うに当たり，多数のご指導を頂きました近藤敏夫教授，佐々木敬泰助教，並びに深澤研究員に深く感謝いたします．また，コンピュータアーキテクチャ研究室の院生，学生の皆様には多くのアドバイスを頂きました．特に院生の皆様には刺激的な議論を頂き，精神的にも支えられました．併せて感謝をいたします.

参考文献

[1] T. Nakabayashi, T. Sugiyama, T. Sasaki, E. Rotenberg, and T. Kondo. Co-simulation framework for streamlining microprocessor development on standard ASIC design flow. Proceedings of the 19th Asia and South Pacific Design Automation Conference (ASP-DAC2013), pp. 400-405, January,2014.

[2] N. K. Choudhary，et．al．FabScalar: Composing Synthesizable RTL Designs of Arbitrary Cores within a Canonical Superscalar Template. ISCA-38, pp.11-22, June 2011.

[3] T. Okamoto, T.Sasaki and T.Kondo. FabCache: Cache Design Automation for Heterogeneous Multi-core Processors. CANDAR-2013, pp.602-606, Dec 2013.

[4] Y. Seto, T. Nakabayashi, T. Sasaki, and T. Kondo. FabBus: A Bus Framework for Heterogeneous Multi-core processor. 28th International Technical Conferench on Circuits/Systems, Computers and Communications (ITC-CSCC2013), pp. 254-257, July 2013.

[5] 佐野伸太郎, 吉瀬謙二,“軽量でシンプルなマルチコアシミュレータの開発”, 第 74 回全国大会講演論文集,No.1,pp.219-220,March,2012.

[6] Gem5,http://www.gem5.org/

[7] 姫野ベンチマーク,http://accc.riken.jp/supercom/himenobmt/