修士論文

題目

ヘテロジニアスマルチコアプロセッサの効率的な検証フレームワークに関する研究

指導教員

近藤 利夫

2018年

三重大学大学院 工学研究科 情報工学専攻
コンピュータ・アーキテクチャ研究室

萱室 高樹(416M508)

内容梗概

近年，高性能と低消費電力を両立させるため，マルチコアプロセッサが広く普及している．特に，電力性能比を向上させるため，構成の異なるコアを集積するヘテロジニアスマルチコアプロセッサ (Heterogeneous Multi-core Processor:HMP) が注目されるなど，プロセッサの内部構造は複雑化，大規模化しており，設計や検証に必要な労力も増大している．これらの問題に対し，FabHetero による HMP の自動生成や C++ベースの高速なプロセッサシミュレータを活用した協調検証フレームワークが提案されている．協調検証フレームワークには，プロセッサコアの機能検証を始めとする検証支援機構が実装されている．プロセッサ検証では，ベンチマークプログラム全体の実行が望ましいが，HDL シミュレータの実行速度は低速であるため非現実的である．例として，RTL(Register Transfer Level) シミュレーションでは，フルセットのベンチマークプログラムや OS の動作を含むような大規模なテスト環境を用いた検証には数か月から数年の実行時間が必要となる．そのため，ベンチマークプログラムの核となる部分，すなわち関心領域 (Region of Interest:ROI) を重点的に検証することが一般的である．一般的な協調検証フレームワークでは，プログラムの先頭から数万あるいは数億命令を高速なプロセッサシミュレータ上で実行することで，ROI への到達時間を削減している．提案されている協調検証フレームワークでは，ファストスキップ，チェックポイントによってこれを実現し，ターンアラウンドタイムを大幅に削減している．しかし，従来の検証フレームワークでは，マルチコアプロセッサの検証時にターンアラウンドタイムの効率的な削減を行えない．これは，1. プロセッサシミュレータの速度低下と 2. ファストスキップ，チェックポイントの持つ効率性と高速性のトレードオフの 2 つの問題点に起因する．まず，従来のプロセッサシミュレータはシングルスレッドで実装されているため，対象のコア数に応じてシミュレーション速度が低下する．シミュレータの高速化手法としては，並列化が広く用いられているが，従来の並列シミュレータの多くは試行毎に振る舞いが変化する．この挙動の変化は，設計対象のバグ修正や原因追及に同じ条件で繰り返し実行する必要があるプロセッサ検証では容認出来ない．次に，効率性と高速性のトレードオフはゲートレベルシミュレーションなど細粒度なサイクル単位での検証時に問題となる．ファストスキップでは，プロセッサシミュレー

タ上でROIの直前までプログラムを実行し，レジスタ値などプロセッサの内部状態を計算する．そのため，シミュレーションを行う毎にプロセッサシミュレータの再実行が必要となる．チェックポイントでは，システム全体の情報を含むチェックポイントファイルからROI直前の状態を復元する．これにより，同じ条件でシミュレーションを繰り返す場合はファストスキップより高速である．しかし，条件毎にチェックポイントファイルの作成が必要である上，ファイルサイズは数100MB〜数GBとなるため，再粒度な検証においては効率性が損なわれる．

そこで，本論文では，従来の検証フレームワークに2つの拡張を行い，より高速で効率性の高いマルチコアプロセッサ向けの検証フレームワークを実装する．まず，高速性を実現するため，プロセッサシミュレータのマルチスレッドによる並列化を行う．このとき，メモリアクセスに注目し実行サイクルを元に各スレッドを同期することで，既存の方式では困難な再現性と高速性の両立を図る．また，チェックポイントファイルサイズの削減とファストスキップ，チェックポイントの併用を可能とすることで，ターンアラウンドタイムのさらなる削減と効率性の向上を図る．評価結果として，提案手法は従来シミュレータと比較し，最大7.94倍シミュレーション速度の向上を実現した．また，チェックポイントサイズを最大で従来の17分の1に削減し，ファストスキップとの併用も可能である．これにより，単一のチェックポイントから異なるROIに高速に復帰可能であるため，提案手法は細粒度なサイクル単位での検証においても，効率性と高速性を両立していると考えられる．

Abstract

Recently, processor architecture become more complex and difficult to improve performance and low energy consumption. For example, homogeneous multi-core processor and heterogeneous multi-core processor (HMP) are widely used to improve performance and power consumption. However, the verification of processor design requires a large effort. To solve this problem, FabHetero and co-simulation framework have been proposed. FabHetero is a tool-set of automatic HMP generation and reduces design effort of HMP. Co-simulation framework with functional simulator accelerates a processor verification flow. Generally, HDL simulation is used for development and verification of processor design, and requires huge execution time. The simulation speed is improved by co-simulation mechanism such as fast-skip and checkpoint technique. These technique run a the benchmark program before Region of Interest (ROI) using a fast function simulator, pass or restore architecture state and perform detailed simulation on HDL simulator. Nevertheless, fast-skip and checkpoint technique are not effective for multi-core processor design, because the conventional functional simulator is slow down depending on the number of target core. Although parallelized multi-core processor simulators have been proposed to improve simulation speed, most of these simulator change the simulation results every execution. Therefore conventional simulators are not suitable for processor verification phases, because the difference of results affects to the behavior of ROI. In addition, fast-skip and checkpoint have tradeoff between convenience and high speed. Therefore, conventional co-simulation framework is not enough to reduce verification effort of HMP.

In this paper, we propose effective and rapid verification framework of HMP by extend the conventional co-simulation framework. As first, we propose parallel simulation method with reproducibility of simulation results. The reproducibility is ensured by execution cycle based synchronization mechanism. Next, we propose hybrid mechanism of fast-skip and checkpoint technique, and downsizing of a checkpoint file. According to the evaluation results, proposed parallelization achieved both 7.9

times faster simulation speed at the maximum and reproducibility of results. In addition, size of checkpoint file is reduced to 1/17 at the maximum. Therefore, proposed framework achieved both convenience and high speed.

目 次

1 はじめに 1

2 プロセッサの設計・検証フロー 3

3 従来のプロセッサ検証フレームワーク 5

3.1 概要 5
3.2 機能シミュレータ 6
3.3 機能シミュレータを用いた検証支援機構 6
3.3.1 プロセッサの機能検証 6
3.3.2 システムコールエミュレーション 7
3.3.3 ファストスキップ 7
3.3.4 チェックポイント 9
3.4 従来手法の問題点 10
3.4.1 機能シミュレータの性能低下 10
3.4.2 細粒度なサイクル単位での検証 12

4 関連研究 14

5 提案手法 16

5.1 再現性のある並列シミュレーション 16
5.1.1 並列シミュレーションのフロー 16
5.1.2 実行サイクルによる同期 19
5.1.3 同期機構の最適化 23
5.2 ファストスキップ及びチェックポイントの併用 25
5.2.1 チェックポイントファイルの小サイズ化 25
5.2.2 復帰ルーチンの共通化 27

6 評価 29

6.1 並列シミュレーション手法の評価 29
6.1.1 評価環境 29
6.1.2 評価結果 30
6.2 小サイズ型チェックポイントの評価 41
6.2.1 評価環境 41
6.2.2 評価結果 41

7　まとめと今後の展望 **42**

謝辞 **42**

参考文献 **42**

図 目 次

3.1 機能シミュレータを活用したHMP設計検証フレームワーク. 5
3.2 機能シミュレータの対象とするシステム. 6
3.3 ファストスキップの概要. 8
3.4 チェックポイントの概要. 10
3.5 共有データへのアクセス. 12
5.6 提案シミュレータの概要. 17
6.7 FFT 実行時の提案手法の高速化率 34
6.8 RADIX 実行時の提案手法の高速化率 34
6.9 CHOLESKY 実行時の提案手法の高速化率 35
6.10 LU(cont.) 実行時の提案手法の高速化率 35
6.11 LU(non) 実行時の提案手法の高速化率 36
6.12 VOLREND 実行時の提案手法の高速化率 36
6.13 OCEAN 実行時の提案手法の高速化率 37
6.14 HIMENO 実行時の提案手法の高速化率 37
6.15 最適化を有効化した場合のFFTの実行時間 39
6.16 最適化を有効化した場合のFFTの実行時間 (続き) 39
6.17 最適化を有効化した場合のHIMENOの実行時間 40
6.18 最適化を有効化した場合のHIMENOの実行時間 (続き) . . 40

表 目 次

6.1 ベンチマークプログラムの設定. 30
6.2 シミュレーションの実行環境. 30
6.3 逐次シミュレーションの実行結果. 31
6.4 逐次シミュレーションの実行結果 (続き). 32
6.5 シミュレーションの実行環境. 41
6.6 チェックポイントファイルのサイズ. 41

アルゴリズムの一覧

1 Parallelized Simulation Flow 18
2 Synchronization Mechanism 20
2 Synchronization Mechanism (cont.) 21
3 Unified Bootstrap Mechanism 28

1　はじめに

近年，高性能かつ低消費電力なプロセッサを実現するため，プロセッサのマルチコア化が広く行われている．また，更なる性能向上，低消費電力化を実現するため，性能の異なるコアを集積するヘテロジニアスマルチコアプロセッサ (Heterogeneous Multi-core Processor:HMP) が注目されている．しかし，このようなプロセッサ構成の複雑化，大規模化は，性能を向上させる一方で，プロセッサの設計や検証に必要な労力も大幅に増加させる問題がある．このようなプロセッサ設計，検証の省力化を行うため，HMP 自動生成ツール FabHetero[4] やプロセッサシミュレータを活用した協調検証フレームワーク [1, 2] が提案されている．FabHetero では，プロセッサコアを自動生成する FabScalar[5] を拡張し，ヘテロジニアスなコアやキャッシュ[6]，バス [7] などの自動生成により，HMP 設計の省力化を実現している．協調検証フレームワークでは，プロセッサコアの機能検証を始めとする検証支援機構により，プロセッサ検証フローの省力化，高速化を行っている．本来，プロセッサ検証ではベンチマークプログラム全体の実行が望ましい．しかし，ハードウェア設計に用いるハードウェア記述言語 (Hardware Description Language:HDL) を元にした HDL シミュレータの実行速度は低速であるため，非現実的である．そのため，ベンチマークプログラムの核となる部分，関心領域 (Region of Interest:ROI) を重点的に検証することが一般的である．検証フレームワークでは，ROI への到達時間を削減し，ターンアラウンドタイムを削減するため，ファストスキップ，チェックポイントの 2 つの機構を提供している．しかしながら，従来の協調検証フレームワークでは，マルチコアプロセッサの検証時には十分な省力化，高速化を行えない問題がある．ファストスキップ，チェックポイントでは，プロセッサシミュレータの高速性を活用する．しかし，従来のプロセッサシミュレータは対象のコア数に応じてシミュレーション速度が低下する．加えて，この 2 つの機構には効率性と高速性のトレードオフが存在する．そのため，ゲートレベルシミュレーションのように，ROI を細分化してのシミュレーションを必要とするケースには十分対応しているとは言い難い．

そこで，本論文では，従来の検証フレームワークを拡張することで，より高速で効率性の高いマルチコアプロセッサ向けの検証フレームワークを実装する．まず，マルチスレッドを用いてプロセッサシミュレータの並列化を行い，高速性を実現する．このとき，プロセッサの検証を容易とするため，メモリアクセスに注目し各スレッドを同期させることで，従

来手法では困難な実行結果の再現性を実現する．また，ターンアラウンドタイムの更なる短縮のため，ファストスキップ，チェックポイントの併用及び，チェックポイントの小サイズ化を行う．以降，本論文は次のように構成する．まず，次章でプロセッサの一般的な設計フローについて，第3章で従来のフレームワークの詳細および問題点について述べる．第4章で他のフレームワークや高速化に寄与する並列シミュレータに関する関連研究，第5章で提案手法の詳細について説明する．最後に，第6章で評価，第7章でまとめと今度の展望について述べる．

2　プロセッサの設計・検証フロー

先行研究であるプロセッサ検証フレームワークについて述べる前に，本章では，プロセッサの設計・検証フローについて述べる．プロセッサ開発は様々な段階を経て行われ，一般的に以下のような 5 つのフェーズに分類できる．

1. ソフトウェアシミュレーション

   C などの高水準言語で記述された，シミュレータ上のモデルを用いて，設計対象の性能推定を行う．高水準言語によって実装されているため，実行速度は後述するシミュレーションと比較して非常に高速である．また，内部情報の取得やモデルの実装に必要な労力も少ない．しかし，取得可能なデータは実際のハードウェアの動作には忠実では無い．そのため，設計対象の大まかな性能の傾向を把握することが主目的である．

2. レジスタ転送レベルシミュレーション

   レジスタ転送レベル（Register Transfer Level:RTL）シミュレーションは，ハードウェアとして設計を行う初期段階である．まず，HDL を用いてレジスタ及びレジスタ間の接続として設計対象の動作を記述する．つまり，設計者の意図する動作を RTL で設計する．RTL シミュレーションでは，回路の構成要素であるレジスタの動作をサイクル単位でシミュレーションする．これにより，設計者が意図した動作をハードウェアとして正しく実装できているかの検証，すなわち設計対象の機能検証を行う．なお，シミュレーション速度は，ソフトウェアシミュレーションと比べ大幅に減少し，シングルコアプロセッサのシミュレーションでは，1 秒あたり 3000 サイクル程度が限度である．以降のフェーズでは，実チップの製作に向けて，機能検証した RTL から，より詳細な設計単位のデータを生成することで設計，検証を進める．

3. ゲートレベルシミュレーション

   ゲートレベルシミュレーションでは，AND，OR や Flip-Flop といった論理演算素子，順序回路を単位にシミュレーションを行う．RTL の論理合成を行うことで，設計単位をレジスタからこれらの素子，すなわち論理ゲートに詳細化し，ネットリストを生成する．これに

より，RTL シミュレーションでは得られない，ゲート遅延を考慮した動作や消費電力，実装面積といった面での検証を行う．しかし，RTL シミュレーションに比べ，実行速度は 10 倍から 100 倍程度悪化する．

4. トランジスタレベルシミュレーション

   ゲートレベルシミュレーションで生成したネットリストを更に詳細化し，ハードウェアの設計単位として最も詳細なトランジスタレベルでのシミュレーションを行う．シミュレーション結果は，全シミュレーション中で最もハードウェアに忠実であるが，最も実行速度が遅い．

5. チップ試作

   実際に半導体を製造し，実機での検証を行う．実行速度は最も高速であり，最も詳細なデータも取得可能だが，試作に必要な費用や時間はそれまでの段階に比べて大幅に増加する．そのため，チップ試作によるトライアルアンドエラーは，出来る限り避けるべきである．

このように，プロセッサの開発段階が進行するに従いハードウェアに忠実な結果が得られる．その反面，必要な時間や費用は大幅に増加し，特にチップ試作では端緒である．また，ソフトウェアシミュレーションでは，ハードウェアとしての動作は考慮されない．そのため，プロセッサ設計，検証においては，RTL シミュレーションをはじめとする HDL を用いたシミュレーションが重要となる．また，プロセッサのデバッグは，同条件でシミュレーションを繰り返すことでバグの原因特定や修正を行う．しかしながら，これらのシミュレーション速度は決して高速とは言えないため，この繰り返しには大きな時間が必要となる．加えて，プロセッサに実装される論理は非常に複雑なものであり，第 1 章で述べたように更なる複雑化，大規模化が進んでいる．これにより，プロセッサの検証作業は非常に労力のかかるものとなる．更に，フェーズが進行する毎にシミュレーション速度は大幅に悪化するため，論理的なエラーを含む状態でフェーズを進行させることは好ましくない．そのため，RTL シミュレーションによる機能検証の効率性は非常に重要である．

3　従来のプロセッサ検証フレームワーク

本章では，提案手法のベースとなる従来のプロセッサ検証フレームワークについて述べる．第2章で述べたように，プロセッサの開発フローは複数の段階に分かれている．特に，ハードウェアの設計，検証においては，HDLをベースとするシミュレーション検証が重要である．これらの段階の効率化を行うため，プロセッサシミュレータを活用する検証フレームワークが提案されている．

3.1　概要

従来のフレームワークの概要について述べる．図3.1は，FabHetero及び機能シミュレータを用いたHMPの設計，検証フローを示している．まず，各コアやキャッシュなどプロセッサの構成情報を記述したパラメータファイルをFabHeteroに与える．これにより，論理合成可能なRTLとして様々な構成のHMPを自動設計する．次に，設計したHMPの検証を様々な検証支援機構を活用して効率的に行う．検証支援機構は，プロセッサシミュレータである機能シミュレータを核として構築されている．以降の節で，機能シミュレータと各検証支援機構の詳細について述べる．

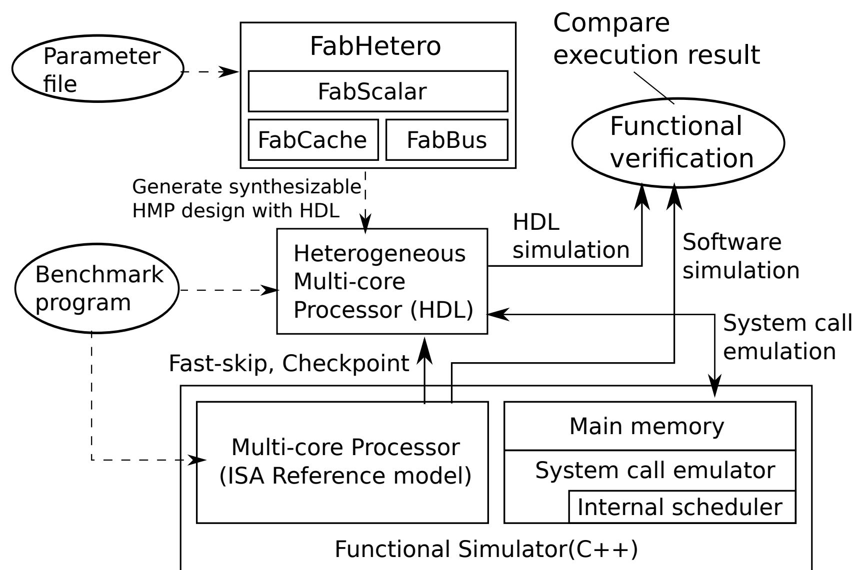

図 3.1: 機能シミュレータを活用したHMP設計検証フレームワーク.

3.2　機能シミュレータ

機能シミュレータは，検証フレームワークの提供する検証支援機構の起点となるプロセッサシミュレータである．検証支援機構の詳細は後述するが，RTL シミュレーションからチップ試作までプロセッサの設計，検証フローの様々な段階の省力化，高速化を実現する．また，ソフトウェアシミュレータとしても使用可能である．図 3.2 は，機能シミュレータのシミュレーション対象を示しており，マルチコア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．各コアは 1 命令/cycle のインオーダー実行を行うモデルであり，ISA は MIPS32 Release2 を採用している．機能シミュレータは C++で記述されており，HDL シミュレータに比べ 1000 倍以上高速である．

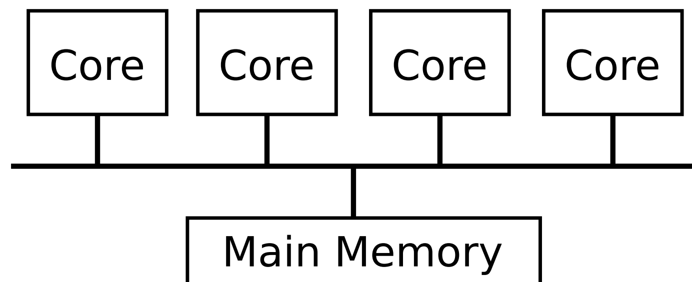

図 3.2: 機能シミュレータの対象とするシステム．

3.3　機能シミュレータを用いた検証支援機構

本節では，検証フレームワークが提供する検証支援機構の詳細について述べる．

3.3.1　プロセッサの機能検証

第 2 章で述べたように，プロセッサ設計では，HDL を用いてハードウェアとしての実装を行う必要がある．しかし，一般にプロセッサの実装は非常に複雑であり，プロセッサとしての機能の正しさを検証すること，すなわち機能検証は容易ではない．そこで，検証フレームワークでは，機能シミュレータをリファレンスモデルとすることで，任意構成のプロセッ

サの機能検証を実現している [2]．この機能では，設計したプロセッサと機能シミュレータで同じプログラムを実行し，その結果を比較する．設計したプロセッサの処理にバグが存在する場合，実行結果が不一致となる．そのため，設計データの機能検証を容易に行うことが可能である．また，検証対象がアウトオブオーダー実行を行うプロセッサである場合，機能シミュレータとは命令の実行順序が異なる．しかし，アウトオブオーダー実行においても，命令の完了順序はインオーダー順となる．この性質を利用することで，検証対象のプロセッサ構成を任意のものとしている．

3.3.2 システムコールエミュレーション

一般的なプログラムの実行には OS が必要である．これは，システムコールを通じて，プログラム側が OS の管理する資源や機能を利用するためである．しかし，OS の実行を伴うフルシステムシミュレーションには非常に多大なサイクルが必要である．加えて，ベンチマークプログラムと OS のどちらの処理を実行しているかを判別しなければならない．そのため，プロセッサの開発者に本来不要な労力が必要になる．そこで，機能シミュレータにはシステムコールエミュレーション機能 [2] が実装されている．この機能では，プログラム側がシステムコールを要求した場合に，OS の行う処理を機能シミュレータ内部で模擬する．これにより，ベンチマークプログラム単体でのシミュレーションが可能となる．この機能を用いることで，前述した OS の処理自体のシミュレーションや切り分けが不要となり，検証作業の省力化と高速化を行うことができる．

3.3.3 ファストスキップ

第 2 章で述べたように，プロセッサ設計で用いられる HDL シミュレータの実行速度は非常に低速であるため，長大なベンチマークプログラムを単純に実行することは不可能である．通常，ベンチマークプログラムは，メモリの初期化やファイル読み込みなどを経て，主となる演算部，つまり ROI に到達する．そのため，ROI での検証を重点的に行うことで，検証の効率化を行うことが一般的である．ファストスキップと呼ばれる手法では，高速なシミュレータ上で ROI の直前までベンチマークプログラムを実行し，ROI から HDL シミュレータの実行を開始することで検証時間の短縮を図る．

従来の検証フレームワークにおける実装 [1] を図 3.3 を用いて説明する．まず，ROI の直前まで機能シミュレータを用いてベンチマークプログラムを実行する．次に，ROI 直前のアーキテクチャステートすなわちレジスタ値などプロセッサの内部情報を HDL シミュレータ上のプロセッサに転送し，詳細なシミュレーションを行う．ROI 以外の部分を機能シミュレータで高速処理することで，HDL シミュレータ単体での実行に比べ，シミュレーション時間を大幅に削減できる．これは，ターンアラウンドタイムの削減に繋がるため，検証時間の短縮が可能となる．また，ROI については HDL シミュレータが実行するため，シミュレーションの精度も両立することが可能である．プロセッサの内部情報の受け渡しは全てホストマシンのメモリ上で行われるため，後述するチェックポイントとは異なり追加のリソースは必要としない．なお，機能シミュレータは前述した機能検証のため，ROI でも命令実行を継続する．

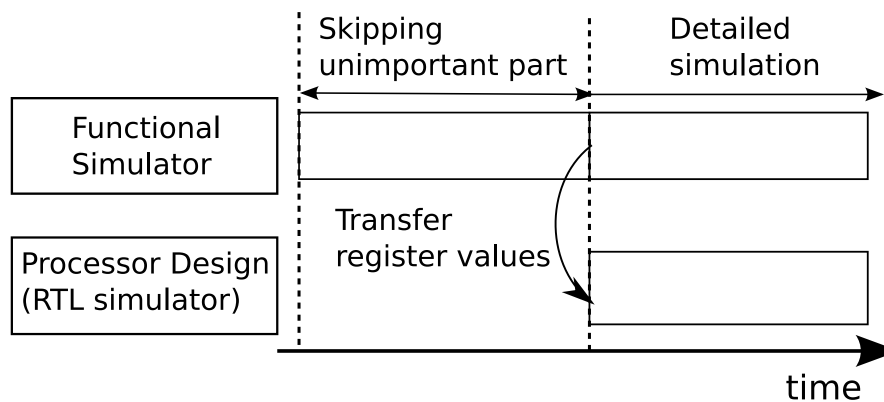

図 3.3: ファストスキップの概要.

3.3.4　チェックポイント

第3.3.3節で述べた，ファストスキップと異なる検証時間の削減手法として，チェックポイントと呼ばれる手法が挙げられる．これは，ファストスキップと同様に，ターンアラウンドタイムを削減する手法である．ファストスキップとの最も大きな違いは，ROI直前の状態を必要な情報が記録されたチェックポイントファイルから復元する点である．これにより，同条件でシミュレーションを繰り返す場合にターンアラウンドタイムをほぼ0にすることが可能である．

従来の検証フレームワークにおける実装を図3.4（出典:文献[1]）を用いて説明する．初回のシミュレーション時に，機能シミュレータを用いてROIの直前までプログラムを実行する．この時，チェックポイントファイルを作成する（図3.4.A）．チェックポイントファイルには，ROI直前のアーキテクチャステートに加え，メインメモリの差分，システムコールの実行履歴が記録されている．チェックポイントでは，アーキテクチャステートだけでなく，対象とするシステム全体の情報を正しく復元する必要がある．例として，中断前にファイルを開いていた場合には，ファイルを開いた状態で復元しなければ，その後のファイル書き込み処理を正しく行うことが出来ない．そこで，チェックポイントファイルに記録されている実行履歴を元にシステムコールを再実行する．これにより，ファイルハンドラやスレッド情報などを正しく復元する（図3.4.B）．記録されているシステムコールの再実行が全て完了した後，メインメモリの差分およびROI直前のアーキテクチャステートをシミュレータに反映する（図3.4.C）．この2つの復元処理によって，ROI直前の状態を正しく復元する．

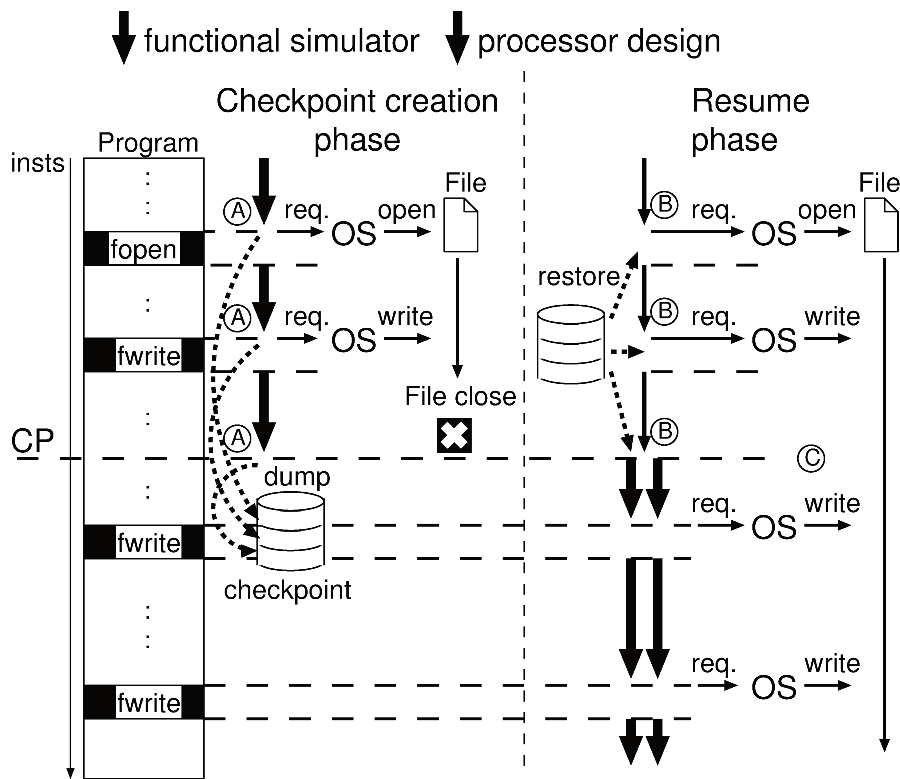

図 3.4: チェックポイントの概要.

3.4 従来手法の問題点

本節では，従来の検証フレームワークで発生する問題点について述べる．従来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十分な省力化，高速化を行うことが出来ない．これは，機能シミュレータの性能低下とファストスキップ，チェックポイント機構の効率性に起因する．

3.4.1 機能シミュレータの性能低下

従来の機能シミュレータは，シミュレーション対象のコア数に反比例し，シミュレーション速度が低下する．機能シミュレータの速度は，ファストスキップやチェックポイントの作成時間に直結する．そのため，従来の検証フレームワークでは，マルチコアプロセッサの検証時にターンアラウンドタイムが十分に削減できない．これは，機能シミュレータがシングルスレッドで実装されていることに起因する．シミュレーション速度

の改善にはシミュレータの並列化が有効だが，既存の並列シミュレータの多くは試行毎に実行サイクル数や振る舞いが変化する問題がある．シミュレータ上のモデルを用いた性能解析では，逐次シミュレーションとの結果の差が一定以内であれば，対象の性質を十分に推定することが出来るため，試行毎の振る舞いの変化はある程度許容可能である．しかし，プロセッサの検証では，結果の完全な再現性が要求される．これは，シミュレーション結果の変動がROIの振る舞いを変化させ，設計対象のデバッグ時に致命的な問題が発生するからである．第2章で述べたように，プロセッサ検証ではバグの原因特定や修正のため，ROIの詳細なシミュレーションを同じ条件で繰り返す必要がある．ファストスキップを行う場合，スキップ部分の振る舞いが変化すると後続のROIの振る舞いも変化する．そのため，実行結果が変動するシミュレータでは，バグ発生時と同じ振る舞いをする可能性が低いため，検証が非常に困難となる．チェックポイントの場合は，同一のチェックポイントファイルから復帰する場合は後続のROIの振る舞いは変化しない．しかし，シミュレーション結果が変動する場合，同一内容のチェックポイントファイルを再生成することは非常に困難となる．

並列化によるシミュレーション結果の変動は，メモリアクセス命令の処理順序が不定となることで発生する．図3.5に示す，2コアが同じアドレスに対して変更を行う場合のタイミングダイアグラムを用いて原因を述べる．逐次シミュレータでは，メモリアクセスの解決は，AからDへと決定的に行われる．しかし，並列シミュレータでは，各コアが並列かつ任意のタイミングで命令を実行するため，コア間でのアクセス順序は不定となる．並列シミュレーションでは，このような共有データへのアクセス順序の非決定性が結果の変動に繋がる．図3.5に示したアクセスパターンは，マルチスレッドプログラムで用いるロック機構の実装に現れる．例として，A，B，C，Dの順に解決された場合，コア0上のスレッドがロックを取得する．一方，C，D，A，Bの順では，コア1上のスレッドがロックを取得する．どちらの順序も，プログラムの振る舞いとしては正しい挙動だが，後続のROIの振る舞いは大きく変化する．そのため，サイクルレベルでの動作の正しさに加え，常に同じ結果が生成できる再現性が重要である．このように，検証用のプロセッサシミュレータでは高速性と再現性の両立が必要となる．

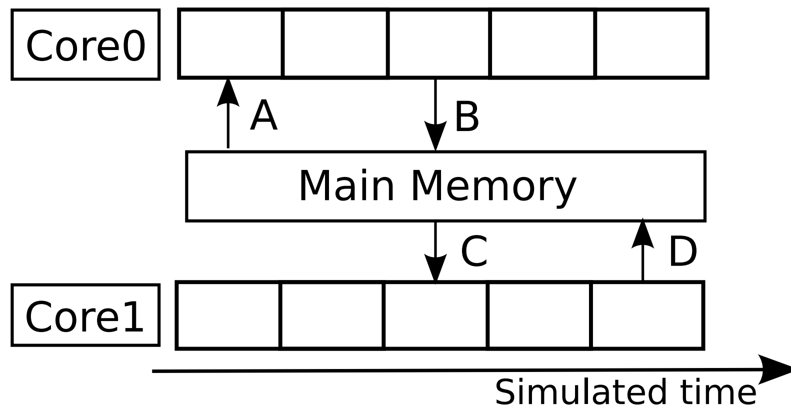

図 3.5: 共有データへのアクセス.

3.4.2 細粒度なサイクル単位での検証

プロセッサ開発では，HDL シミュレータによるシミュレーションが必要不可欠である．しかし，第 2 章でも述べたように，HDL シミュレータの速度は高速ではない．最も高速な RTL シミュレーションでも，1 秒あたり 3000 サイクル程度しか処理できず，ゲートレベルやトランジスタレベルでは，さらに 10〜100 倍以上の速度低下が発生する．従来フレームワークでは，ファストスキップ，チェックポイントによって，ターンアラウンドタイムの削減を行っている．しかしながら，従来の手法では十分な削減が出来ない場合が存在する．

文献 [1] のように，プログラムの開始地点から 1000 億サイクル後に ROI が存在する場合を例として説明する．なお，各シミュレータの速度はシングルコアプロセッサを対象とする場合とし，機能シミュレータは 1 秒あたり 1000 万サイクル程度処理可能である．まず，1000 億サイクルまでのファストスキップ，チェックポイントファイルの作成は，約 2.5 時間必要となる．ファストスキップによって検証を行う場合には，シミュレーション毎にこの 2 時間弱のオーバーヘッドが必要となる．そのため，長区間のスキップに対してはターンアラウンドタイムの削減が不十分となる．チェックポイントでは，オーバーヘッドの発生はチェックポイントファイル作成時に限定される．ただし，現実的な処理時間でゲートレベルやトランジスタレベルシミュレーションを行う場合には，ROI を更に細分化してのシミュレーションが必要である．この場合，チェックポイントファイルの作成粒度も同様に細かくする必要がある．しかし，チェックポイン

トファイルのサイズは数百 MB〜数 GB となる上，各ファイルの作成時間は個別に必要になる．そのため，細粒度な検証のためにチェックポイントファイルを多数作成することは望ましい状態ではない．ファストスキップにはチェックポイントファイルのような追加リソースは不要なため，効率性では勝るが，前述のように速度面での問題が発生する．また，マルチコアプロセッサを対象とする場合には，第 3.4.1 節で述べた機能シミュレータの速度低下に加え，HDL シミュレータ自体も低速化する [2]．そのため，これらの問題は更に悪化する．

4 関連研究

商用プロセッサにおける検証フレームワークとして，Intel 社における事例 [9] や IBM 社における事例 [10] が挙げられる．これらのフレームワークでは，専用の言語，テンプレートを用いて生成する小規模なテストプログラムや各機能ユニットに対応するモデル構築を用いて検証を行っている．第 3 章で述べた本論文で扱う検証フレームワークは，検証対象として FabHetero により生成される HMP を想定している．FabHetero の最終的な目標は，プロセッサ内部の構成を任意に変更でき，研究者が独自の機構を追加することを想定している．そのため，これらのフレームワークのように，各機能ユニットに対応するモデル構築やプロセッサの論理を網羅するといったことは困難である．しかし，独自機構の追加後も，プロセッサの基本的な機能は ISA に沿ったものとなるため，機能検証の効率化は重要である．そこで，提案するフレームワークでは，第 3.3 節で述べた ROI での機能検証の高速化と効率性の改善に焦点を置く．設計検証の省力化に向けた異なるアプローチとして，ソフトウェアで構築したモデルとハードウェアを混合させる手法 [11, 12] が挙げられる．この手法では，ハードウェアの一部をソフトウェア上のモデルに置換することで，設計，検証の省力化が可能であり，検証フレームワークと相反するものではない．しかし，シミュレーションに必要なサイクル自体を減少させるわけではないため，本論文で対象とするターンアラウンドタイムの短縮には繋がらない．

第 3 章で述べた，先行研究である協調検証フレームワークは，プロセッサシミュレータを起点として様々な支援機構を提供している．そのため，結果の再現性や高速性などシミュレータの性質はフレームワークに大きな影響を与える．最も広く使われているプロセッサシミュレータの 1 つとして，SimpleScalar[13] が挙げられる．アウトオブオーダー実行を行うパイプラインプロセッサのシミュレーションを行うことができるが，マルチコアプロセッサのシミュレーションには非対応である．また，シミュレーションの高速化手法として，ファストフォワーディングが実装されている．これは，ROI の直前まで，一部の処理を省略しインオーダー実行型のプロセッサとして振る舞うことで ROI へのターンアラウンドタイムを削減する手法である．しかし，本論文で用いる機能シミュレータは，インオーダー実行型のプロセッサとして振る舞うため，機能シミュレータ自体に同様の最適化を行う事は不可能である．SimMips[14] は，マルチコアプロセッサのシミュレーションに対応している．このような逐次シ

ミュレータは結果の再現性を有するが，コア数に応じてシミュレーション性能が低下する問題がある．これに対し，並列化されたシミュレータ [15, 16, 17] は実行環境の並列度に応じたシミュレーション性能のスケーラビリティを持つ．並列シミュレーションでは，演算ノード間の同期を行う必要があるが，これらの手法は逐次実行との誤差を一定以下に抑えるものであるため，結果の再現性を保証することは出来ない．これらと異なる並列化手法として，時間軸方向にシミュレーションを並列化する手法が提案されている [20, 21]．これは，各演算ノードで異なる区間の処理を行う方式だが，分割した区間同士の整合性の保証が困難な上，前処理に機能シミュレーションが必要であるため，本論文の趣旨とは合致しない．結果の再現性を実現する機構として，ロールバックが挙げられる [18]．シミュレーションの不正状態を検知した場合，発生前の状態に巻き戻した上で，不正状態が発生しないシミュレーション方式で再度該当区間を実行する方式である．しかしながら，逐次実行よりも低速となる場合 [19] が示されている．バイナリ変換型シミュレータ [22] は，実行速度が非常に高速である．ターゲットプログラムを実行環境の ISA に動的もしくは静的に変換することで，本論文で用いるインタプリタ型のシミュレータに比べ，大幅な性能向上を実現している．しかしながら，バイナリ変換型シミュレータはその性質上，可搬性が無く，ターゲットシステムのサイクル単位での詳細な状態の取得が難しいため，機能検証に用いるのは難しい．高速性と再現性を両立した例として，文献 [23, 24] が挙げられる．この手法ではバイナリ変換時に得られる情報を元に同期を行うため，インタプリタ型シミュレータへそのまま適用することは難しい．加えて，例外処理に対応出来ないなど対応可能なプログラムに制限が存在する．

5　提案手法

本章では，高速性と効率性を両立するための，従来のフレームワークに行う拡張の詳細について述べる．第3.4節で述べたように，従来の検証フレームワークでは，マルチコアプロセッサの検証時にターンアラウンドタイムを十分に削減できていない．加えて，従来のファストスキップ及びチェックポイントには，効率性と高速性のトレードオフが存在する．そこで，本論文では従来のフレームワークに2つの拡張を行う．はじめに，再現性のあるシミュレーションの並列化を行うことで，ターンアラウンドタイムの改善を行う．次に，ファストスキップ及びチェックポイントの併用を行うことで，効率性と高速性のトレードオフの解消を行う．これらにより，高速性と効率性を両立するフレームワークを提案，実装する．

5.1　再現性のある並列シミュレーション

5.1.1　並列シミュレーションのフロー

第3.4節で述べたように，検証用シミュレータでは高速性と再現性の両立が要求される．第3.4.1項で述べたように，並列シミュレーションの結果のゆらぎは，メモリアクセス順序が不定となるため発生する．また，第4章で挙げたように，同期方式は高速性にも大きな影響を与える．そこで，本論文では，シミュレーション対象のメモリアクセス命令に注目することで，結果の再現性と高速性を両立する並列シミュレーション手法を提案する [3]．図5.6に提案する並列シミュレータの概要，Algorithm 1にシミュレーションフローを示す．まず，マルチスレッドを用いてシミュレーションを並列化する．各スレッドにシミュレーション対象のコアを1つ以上割り当て，1サイクルずつ割り当てられたコアのシミュレーションを行う（Algorithm 1 : 5行目，RunThread）．このとき，コンテキストスイッチの頻発による性能低下を防止するため，*sched_setaffinity*() を用いて，各スレッドをホストコアに割りつける．次に，シミュレーション結果の再現性を確保するため，ロード，ストア命令の実行時に実行サイクル，メモリアドレスを用いて同期を行う．これにより，メモリアクセスの実行順序を一意にすることで，結果の再現性を保証する．また，ロードストア以外の命令は並列実行可能なため，高速性が期待できる．Sync() は，要求されたメモリアクセスが実行可能かどうかを検査し，実行が許可されない場合はfalseを返す関数である（Algorithm 1 : 24行目）．不許可の

場合，スレッドはメモリアクセスを要求したコアのシミュレーションを中断し，担当する他のコアのシミュレーションを実行する．同期に用いる実行サイクル，アクセスアドレスの更新は，アクセスするメモリアドレスの計算後に毎サイクル行う（Algorithm 1 : 20 行目，UpdateCycle）．なお，Sync，UpdateCycle の詳細については次項で述べる．

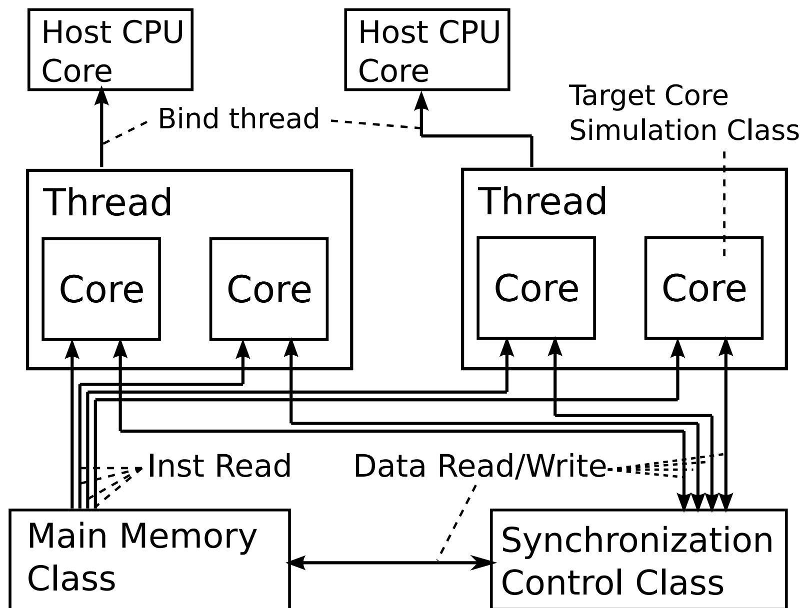

図 5.6: 提案シミュレータの概要.

Algorithm 1 Parallelized Simulation Flow

```
 1: Function Init
 2:     ClearCycles()
 3:     for all thread do
 4:         assign cores to thread
 5:         RunThread(number of assigned_cores)
 6:     end for
 7:     JoinAllThread()
 8: end Function

 9: Function RunThread(in assigned_cores)
10:     loop
11:         for i = 0 to assigned_cores do
12:             Step(assigned_core_id[i])
13:         end for
14:     end loop
15: end Function

16: Function Step(in cpu_id)
17:     if stage == front then
18:         current_cycle = current_cycle+1
19:         FrontEnd(cpu_id)          ▷ フェッチ，デコード，アドレス計算
20:         UpdateCycle(cpu_id,current_cycle,addr,inst)
21:         stage = back
22:     end if
23:     if inst == LOAD || inst == STORE then
24:         if Sync(cpu_id,addr) == false then
25:             return
26:         end if
27:     end if
28:     LoadStore(cpu_id)
29:     Writeback(cpu_id)
30:     stage = front
31: end Function
```

5.1.2　実行サイクルによる同期

本項では，実行結果の再現性を保証するための同期機構の詳細について述べる．提案する同期機構では，実行サイクル，アクセスアドレスの2つの情報を用いてメモリアクセスを時系列に沿って決定的に解決する．Algorithm 2に同期機構を示す．同期機構は，メモリアクセスの優先度に従い，メモリアクセスを要求したコアおよび他のコアの実行サイクルを以下のように比較する．ここで，メモリアクセスを要求したコアを Core_m，高優先度のコアを Core_h，低優先度のコアを Core_l とし，各コアの実行サイクル，アクセスアドレスをそれぞれ Cycle_m，Cycle_h，Cycle_l，Addr_m，Addr_h，Addr_l と表記する．また，アクセス優先度は，逐次実行でのシミュレーション順序と等価とする．

Core_h との比較

条件1：$\mathrm{Cycle}_h < \mathrm{Cycle}_m$
　Core_h が Core_m より過去のサイクルを実行している．
　Algorithm 2：4行目

条件2：$\mathrm{Cycle}_h = \mathrm{Cycle}_m$ かつ $\mathrm{Addr}_h = \mathrm{Addr}_m$
　Core_h と Core_m が同じサイクルで同アドレスにアクセス．Core_h の実行命令がロード，ストアでない場合，アクセスアドレスは不一致とみなす．Algorithm 2：7-8行目

Core_l との比較

条件3：$\mathrm{Cycle}_l < \mathrm{Cycle}_m$
　Core_l が Core_m より過去のサイクルを実行している．
　Algorithm 2：15行目

いずれかの条件が満たされる場合，未解決のメモリアクセスが Core_m のメモリアクセスに対して干渉する可能性がある．この場合，Core_m が要求したメモリアクセスの結果の変動につながるため，アクセスが不許可となる．全コアで条件が満たされない場合，Core_m が要求したメモリアクセスが最も古く高優先度となる．これにより，未解決のメモリアクセスによる干渉が無いため，アクセスを許可する．加えて，条件2により異なるアドレスへのアクセスは同時に許可される．このように，実行サイクルを基準にメモリアクセスを解決することで，シミュレーション結果を保証しつつ並列にシミュレーションを行うことが可能である．この

手法の利点として，幅広いプログラムに対応可能な点が挙げられる．再現性と高速性を両立している文献 [23, 24] の手法では，例外処理など実行時にのみ判別可能な処理に対応することが出来ない．一方，提案手法では，実行サイクルを元に動的に制御を行うため，例外処理にも対応することが出来る．

Algorithm 2 Synchronization Mechanism

```
1:  Function Sync(in cpu_id,addr)
2:      ▷ 高優先度のコアと比較
3:      for i = 0 to cpu_id do
4:          if GetCycle(i) < current_cycle then
5:              return false
6:          end if
7:          if GetCycle(i) == current_cycle
8:            && SameAddr(i,addr) == true then
9:              return false
10:         end if
11:     end for
12:
13:     ▷ 低優先度のコアと比較
14:     for i = cpu_id+1 to number of cores do
15:         if GetCycle(i) < current_cycle then
16:             return false
17:         end if
18:     end for
19:
20:     ▷ メモリアクセス許可
21:     return true
22: end Function
```

Algorithm 2 Synchronization Mechanism (cont.)

```
23: Function SameAddr(in cpu_id,addr)
24:     myaddr = addr & 0xfffffff0
25:     if IsLoadStore(cpu_id) == 1
26:        && GetAddr(cpu_id) == myaddr then
27:         return true
28:     end if
29:     return false
30: end Function

31: Function UpdateCycle(in cpu_id, current_cycle, addr, inst)
32:     upper_bit = addr & 0xfffffff0 >> 1
33:     if inst == LOAD || inst == STORE then
34:         upper_bit = upper_bit | 0x80000000
35:     end if
36:     sync_info[cpu_id] = (upper_bit << 32) | current_cycle
37: end Function

38: Function GetCycle(in cpu_id)
39:     return sync_info[cpu_id] & 0x00000000ffffffff
40: end Function

41: Function GetAddr(in cpu_id)
42:     return sync_info[cpu_id] & 0x7fffffff00000000 >> 31
43: end Function

44: Function IsLoadStore(in cpu_id)
45:     return sync_info[cpu_id] >> 63
46: end Function
```

また，同期機構の実装は，実行サイクルの単調増加性を利用することで，ロックを用いずに行う．これにより，同期処理を並列に進行させることが出来る．同期機構は，基本的には実行サイクルの値を元に，メモリアクセスを古いものから順に解決する．この時，他コアの更新が反映される前の値を使用し同期を行った場合でも，実行不可能なタイミングで実行可能と判断することはない．そのため，メモリアクセスの順序そのものは破綻しない．実行サイクルの単調増加性を保証するには，実行サイクルをアトミックに更新する必要がある．アトミック性とは，ある変数への操作に対し，中間状態が取得できないこと，つまり，操作前，操作後のどちらかの値のみを取得できる性質を指す．同期機構は複数の情報を用いるが，一部の情報のみが更新されるとアトミック性が失われる．そこで，ビットマスクを用いて 64bit 長の一変数に全情報を格納することで，アトミック性を確保する．Algorithm 2 では，最上位ビットに実行中の命令がロードストアかどうか，上位 31bit をアクセスアドレス，下位 32bit を実行サイクルとして割り当てている（Algorithm 2：UpdateCycle）．なお，Algorithm 2 では省略しているが，高優先度のコアとの比較では，sync_info をローカル変数に一度コピーし，コピーした変数から各情報を取得する．これによって，別のサイクルの情報を同時に取得することを防ぐ．また，実行サイクルのオーバーフローを防ぐため，2^{32} サイクル毎に全スレッドを同期し，カウンターをリセットする．

5.1.3 同期機構の最適化

本項では，シミュレーション速度を更に向上させるために行った，同期機構の最適化について述べる．

同期情報へのアクセス効率化

第5.1.2項で述べたように，提案手法では，実行サイクルなどの同期情報を64bitの変数に格納し，全スレッドで共有している．これはAlgorithm 2のsync_infoで示しているように，単純な配列で実現可能である．しかし，複数スレッドから同時かつ頻繁にアクセス，更新されるため，単純な配列ではホストマシンにキャッシュコヒーレンシの激しい遷移が発生する．そのため，同期情報の更新，読み込みに大きなオーバーヘッドが発生し，シミュレーション性能のボトルネックとなる．

そこで，同期情報の更新，読み込みそれぞれにホストマシンのキャッシュ構成を考慮した最適化を行う．まず，各コア毎の同期情報が全て独立したキャッシュラインに配置されるように，sync_infoをキャッシュラインサイズにパティングする．これにより，複数スレッドが同時にsync_infoを更新した場合でも，同一ラインへの書き込みが発生しない．そのため，同期情報更新時のキャッシュコヒーレンシの遷移を軽減でき，性能が向上する．次に，同期情報の読み込みによる負荷を軽減するため，スレッド毎に同期情報のキャッシングを行う．各スレッドに同期情報のコピーを保持する配列cache_infoを実装し，同期機構では各条件の判定を2段階に分割する．まず，cache_infoを元に条件判定を行う．このとき，メモリアクセス不許可と判定された場合，sync_infoを用いて再度判定を行う．再判定時にメモリアクセスが許可された場合，該当するsync_infoのエントリをコピーすることで，cache_infoを更新する．また，各スレッドが担当するコアの同期情報については，自身のcache_infoを直接更新する．スレッド間で非共有であるcache_infoには，原理上キャッシュコヒーレンシが発生しないため，その読み込み，更新に追加のオーバーヘッドは発生しない．これにより，cache_infoによる条件判定は高速に行える．また，同期に成功した場合は，sync_infoへのアクセスが不要となるため，同期情報の読み込みで発生するsync_infoのコヒーレンシの遷移を削減できる．

同期失敗時のオーバヘッド削減

提案手法では，実行サイクルを元にメモリアクセス順序を決定的に解決するが，第5.1.2項で述べたように，実行サイクルは単調増加性を有し

ている．この性質により，一度不満足と判定された条件が再評価により満足されることはあり得ない．これを利用し，冗長な条件判定をスキップする．まず，同期失敗時に最後に判定した条件を記録する．これは，コアの ID や同期処理の進行度を保存することで行う．次回の同期処理では，記録した情報を用いて前回最後に判定した条件から同期処理を再開することで，既に同期が成功しているコアとの冗長な比較を除去する．これにより，同期失敗時のオーバーヘッドを削減する．また，同期情報へのアクセス量自体を削減できるため，キャッシュコヒーレンシによる他スレッドへの悪影響の軽減も期待できる．

同期対象の削減

提案手法では，全てのメモリアクセスを監視することで結果の再現性を確保している．しかし，同期頻度の高さはシミュレーション速度に悪影響を与える [15, 23, 24]．Wu ら [23, 24] は，共有メモリアクセスにのみ同期点を挿入することで，同期頻度を削減している．しかし，マルチスレッドプログラムにおいては，グローバル変数を介したローカル変数の共有アクセスが発生するかユーザーが指定する必要がある．これを判別するには，ライブラリなども含めプログラムの実行パス全てを網羅的に確認する必要があり，検証者の労力を削減するという目的には反する．そこで，読み込み専用データのアドレスを実行するプログラムの展開時に取得し，該当アドレスでの同期を省略する．読み込み専用データはどのタイミングで読み出しても変化しないため，省略可能である．機能シミュレータは，Executable and Linkable Format(ELF) 形式 [25] のプログラムを実行対象としている．ELF 形式では，グローバル変数や実行する命令列といった使用目的に応じたセクションにデータが格納される．各セクションにはセクション内のデータの扱いに関する属性が定義されており，これを解析することで，読み込み専用データを特定する．これに該当するセクションは，実行時にメモリに展開され，かつ書き込み禁止となるセクションである．具体的には，SHF_ALLOC が定義され，SHF_WRITE が未定義のセクション内へのアクセスの同期を省略する．セクションの解析は，機械的に行うことが可能であるため，プロセッサの検証者に追加の負担は発生しない．

5.2　ファストスキップ及びチェックポイントの併用

第3.4.2項で述べたように，従来フレームワークに実装されているファストスキップ，チェックポイントでは，ターンアラウンドタイムの十分な削減と効率性の両立は実現できていない．チェックポイント方式では，チェックポイントファイルの大きさが効率性を低下させる最大の要因である．従来の仕様では，数百MBから数GB単位のファイルサイズとなるため，複数のチェックポイントファイルの作成に負担が大きい．加えて，ファイルの作成時間自体も作成数に応じて必要であるため，初回のシミュレーション時間自体の増加を招く．一方，ファストスキップ方式では，細粒度なサイクル単位での検証時に効率性は低下しない．しかし，毎回スキップを行う必要があるため，特に長区間のスキップではターンアラウンドタイムの削減が不十分となる．

そこで，ファストスキップ，チェックポイントを併用することで，効率性と高速性を両立させる．まず，効率性を向上させるために，チェックポイントファイルの小サイズ化を行う．次に，ROIへの復帰ルーチンをシミュレーションの開始ルーチンと共通化することで，ファストスキップ，チェックポイントの併用を可能にする．これにより，ファストスキップの持つ細粒度への対応性とチェックポイントの持つ高速性の両方を活用可能にする．

5.2.1　チェックポイントファイルの小サイズ化

第3.3.4節で述べたように，従来のフレームワークにおける実装では，チェックポイントファイルに記録されているシステムコールの実行履歴を元に，システムコールの再実行を行う事で状態の復元を行う．しかし，全てのシステムコールの記録を行うため，ファイルサイズの肥大化を招いている．そこで，対象とするシステムコールと記録内容を再検討し，ファイルサイズの削減を行う．

まず，再実行が必要なシステムコールを判別する．第3.2節で述べたように，検証フレームワークではシステムコールエミュレーション機能を提供している．この機能によるシステムコールの実行は，機能シミュレータ内部で実行が完結するものとホストOSのシステムコールを呼び出すものの2種類に大別できる．また，システムコールの性質として，実行結果がシステム側の状態に関連するものと関連しないものの2つに分類出

来る．これらの性質から，チェックポイントにおけるシステムコールの扱いは以下のようになる．

- ホスト OS 側を呼び出す かつ システム状態に関連する
  ファイル操作に関するシステムコールが該当する．第 3.2 節でも述べたように，ファイルハンドラやファイル自体の状態を正しく復元する必要がある．これらの情報はホスト側の OS が管理しているため，復元には再実行が必要である．そのため，チェックポイントに記録が必要である．

- 内部で完結する かつ システム状態に関連する
  スレッドスケジューリングに関するシステムコールが該当する．ファイル操作に関するものと同様に，スケジューラの状態を正しく復元しなければ，結果の変動を起こす．機能シミュレータにおいては，スレッドスケジューリングは内部スケジューラによって実現されている [2]．そのため，スケジューラの内部状態は容易に取得可能である．そこで，スケジューラの最終状態をファイルに出力する．システムコールによるスケジューラの再構築を行う代わりに，この情報を元にスケジューラの最終状態を直接復元する．

- システム状態に関連しない
  その他のシステムコール全てが該当する．これらのシステムコールは実行結果が変動しない．また，他のシステムコールに影響を与えることも無い．そのため，状態の復元に再実行は不要であり，記録の必要はない．

このように，記録対象のシステムコールをファイル操作に関するものに限定することで，チェックポイントファイルサイズを大幅に削減できる．

次に，システムコールの実行時に記録する情報について検討する．従来のフレームワークでは，システムコールの実行毎に，その時点でのメモリの差分全てを記録する．この方式では，システムコールの実行に必要な部分以外も記録することになる．加えて，同じ領域への変更が複数行われる場合，差分が重複しファイルサイズの肥大化を招く．そこで，システムコールの引数に関する情報のみを記録することで，データ量を削減する．システムコールの再実行には，引数の値そのものに加え，引数がポインタである場合には，ポインタの指す先の領域が必要である．例として，*write*() では，書き出すデータそのものが引数に加えて必要であ

る．これら 2 つの情報は，システムコールエミュレーション機構自体が要求するため，各システムコール毎に必要なデータを詳細に取得できる．これにより，記録する情報を最小限に抑えることが可能となる．

5.2.2 復帰ルーチンの共通化

第 3.4.2 項で述べたように，長い区間のスキップを複数行う場合には，ファストスキップ，チェックポイント単体では，効率性と高速性は両立しない．そこで，両方式を併用することで高速性と効率性の両立を図る．まず，長区間のスキップにはチェックポイントが適している．対して，短区間のスキップにはファストスキップが適している．これらの性質から，以下のように併用することが考えられる．はじめに，各区間の共通部分をチェックポイントでスキップする．その後，残りの短区間にファストスキップを行う．これにより，単一のチェックポイントファイルから任意の地点まで効率よく到達することが出来る．

これを実現するために，中断地点からの復帰ルーチンをシミュレーションの開始ルーチンと共通化する．Algorithm3 に共通化した起動ルーチンを示す．まず，プロセッサ本体の初期化を行う．検証フレームワークでは特定のメモリ領域を通じて，機能シミュレータと検証対象のプロセッサ間で通信を行うことが出来る [1, 2]．これを利用し，実行するスレッドのコンテキスト情報を機能シミュレータに要求する．コンテキストには，プロセッサのレジスタ値，復帰先アドレス及びアイドルタスク判別用フラグが含まれる．機能シミュレータは要求に対し，実行対象となるスレッドのコンテキストをメモリ上に出力する．最後に，メモリ上に展開されたコンテキストからレジスタ値を復元し，復帰対象へとジャンプする．シミュレーション開始時と中断地点から復帰する場合の差異は，実行済みのスレッドの有無である．コンテキスト情報は機能シミュレータ内部のスケジューラが管理しており，ファストスキップやチェックポイントでは，中断直前のコンテキスト情報を機能シミュレータ内部のスケジューラに保存する．そのため，この差異は完全に機能シミュレータ側に隠蔽することが可能である．これにより，プロセッサ側ではファストスキップ，チェックポイント，シミュレーションの開始を区別すること無く行うことができ，各方式を任意に組み合わせることが可能になる．また，メモリを用いた通信方式はチップ設計の幅広い段階に対応する．そのため，このフローも同様に様々な検証に適用することが出来る．

Algorithm 3 Unified Bootstrap Mechanism

```
 1: ▷ プロセッサ側の起動ルーチン
 2: Function Boot
 3:     ▷ TLB などプロセッサ本体の初期化
 4:     ProcessorInitialize()
 5:     ▷ 復帰させるコンテキストの取得要求
 6:     TriggerGetContext(cpu_id)
 7:
 8:     ▷ 対応する領域からレジスタ値，復帰する PC を復元
 9:     for all register do
10:         register = memory[register+offset]
11:     end for
12:     if context is idle task then
13:         entrypoint = entrypoint_of_idle_task
14:     else
15:         entrypoint = memory[offset_entrypoint]
16:     end if
17:     Jump to entrypoint
18: end Function

19: ▷ 機能シミュレータ側のコンテキスト転送処理
20: Function GetContext(in cpu_id)
21:     if exist context of core[cpu_id] then
22:         copy exist context to memory
23:     else
24:         make new context
25:         copy context to memory
26:         if cpu_id ≠ 0 then
27:             set flag of idle task
28:         end if
29:     end if
30: end Function
```

6 評価

本章では，提案フレームワークの高速性と有用性を評価する．まず，マルチコアプロセッサのシミュレーションを行い，提案手法と従来手法での実行時間を比較し，シミュレーション速度を評価する．次に，チェックポイント作成時のファイルサイズを比較することで，有用性を評価する．

6.1 並列シミュレーション手法の評価

6.1.1 評価環境

第 5.1 節で提案した，並列シミュレーションについて評価する．提案手法及び従来の逐次実行で 2，4，8，16，32，64 コアのマルチコアプロセッサのシミュレーションを行う．ベンチマークプログラムとして，SPLASH-2[26]，姫野ベンチマーク [28] を各 3 回実行する．平均実行時間，実行した命令数を取得し，提案手法の高速性と再現性を評価する．加えて，適用した最適化の効果を評価するため，最適化前および各最適化を有効化した機能シミュレータで FFT，姫野ベンチマークを実行する．提案手法は，メモリアクセス命令をトリガーとし同期を行うため，メモリアクセス頻度の低い FFT とメモリアクセス頻度の高い姫野ベンチマークを用いる．また，本研究では，ROI の直前までのシミュレーションを高速化対象としている．シングルスレッドプログラムであれば，SimPoint[8] を用いて ROI を特定することが可能である．しかし，マルチスレッドプログラムにおいては，同様の手法が存在しない．そこで，今回の評価ではベンチマークプログラムを最後まで実行するのに必要なサイクル数に 0.75 を乗したサイクルを ROI の開始点とした．表 6.1 にベンチマークプログラム設定，表 6.2 に実行環境を示す．なお，各ベンチマークプログラムは 64 スレッドで動作するように設定する．

表 6.1: ベンチマークプログラムの設定.

Benchmark	Problem size
FFT	-m20
RADIX	-n100000000
CHOLESKY	tk29.O
LU (contiguous_block)	-n512
LU (non_contiguous_block)	-n512
VOLREND	head-scaleddown2.den
OCEAN	-n514
HIMENO	XS,loop iteration:200

表 6.2: シミュレーションの実行環境.

CPU	Xeon E5-2660v4@2.00GHz(14cores)
RAM	128GB
Compiler	gcc4.8.5 -O3

6.1.2 評価結果

表 6.3，表 6.4 は，逐次シミュレーションの実行結果を示している．シミュレーション対象の増加に応じて，シミュレーション速度が大きく低下していることが確認できる．特に，64 コアのシミュレーションは，2 コアプロセッサに比べ，最大約 40 倍低下しており，プロセッサの検証効率への悪影響は明らかである．

表 6.3: 逐次シミュレーションの実行結果.

ベンチマーク	コア数	実行サイクル数	平均実行時間 [sec.]	スループット [cycle/sec.]
FFT	2	585,536,787	152.1	3,849,683.0
	4	487,635,045	261.5	1,864,761.2
	8	438,687,375	474.5	924,525.6
	16	414,304,198	903.4	458,605.5
	32	402,040,553	1764.7	227,823.7
	64	395,934,673	3513.9	112,676.7
RADIX	2	8,438,599,696	1,944.4	4,340,018.0
	4	4,219,614,071	1,889.2	2,233,563.5
	8	2,110,250,164	1,871.4	1,127,659.1
	16	1,055,606,908	1,871.8	563,936.1
	32	528,362,278	1,890.9	279,421.6
	64	264,828,758	1,920.5	137,894.8
CHOLESKY	2	1,703,633,956	384.1	4,435,529.1
	4	1,129,637,325	528.1	2,139,258.1
	8	689,930,522	692.2	996,655.1
	16	477,441,418	1,028.7	464,127.4
	32	399,704,552	1,761.2	226,955.9
	64	366,016,231	3,237.3	113,060.8
LU(cont.)	2	235,614,725	55.2	4,266,298.3
	4	135,789,005	63.9	2,124,763.3
	8	85,925,636	84.1	1,022,088.5
	16	63,008,741	129.2	487,637.4
	32	52,279,706	224.1	233,317.9
	64	48,768,948	429.4	113,576.1
LU(non)	2	554,061,602	129.1	4,291,895.5
	4	297,280,552	138.1	2,153,329.3
	8	169,924,847	160.2	1,060,819.2
	16	109,753,100	214.9	510,616.3
	32	84,105,193	349.8	240,406.5
	64	76,703,608	674.2	113,763.8

表 6.4: 逐次シミュレーションの実行結果 (続き).

ベンチマーク	コア数	実行サイクル数	平均実行時間 [sec.]	スループット [cycle/sec.]
VOLREND	2	1,594,116,159	417.9	3,814,930.0
	4	1,001,029,724	531.8	1,882,195.5
	8	564,531,174	601.2	938,978.1
	16	322,972,310	698.1	462,625.3
	32	199,512,507	878.5	227,114.6
	64	145,233,395	1,295.4	112,116.7
OCEAN	2	896,293,048	208.0	4,309,004.4
	4	463,374,611	216.2	2,143,703.6
	8	248,129,113	236.7	1,048,083.7
	16	139,273,394	278.9	499,394.6
	32	86,160,329	366.2	235,273.6
	64	57,364,029	506.7	113,206.0
HIMENO	2	636,937,193	150.7	4,226,758.2
	4	336,076,934	157.4	2,134,693.7
	8	176,835,715	169.0	1,046,174.9
	16	95,138,119	186.9	509,000.2
	32	56,109,944	230.5	243,403.3
	64	56,133,288	515.5	108,884.3

図6.7から図6.14は，各ベンチマークを提案手法による並列シミュレーションで実行した場合の高速化率を示している．高速化率Speedupは以下の式で定義する．

$$Speedup = \frac{\text{逐次実行シミュレータの平均実行時間}}{\text{提案シミュレータの平均実行時間}}$$

結果として，全てのベンチマークプログラムでシミュレーション速度の高速化が確認され，最大で7.94倍(FFT)の高速化を達成した．加えて，全ての試行で実行結果は一致しており，提案手法は再現性と高速性を両立しているといえる．また，スレッド数に応じたシミュレーション速度の高速化が確認できるため，提案手法はスケーラビリティを持つと言える．文献[23, 24]の手法は再現性と高速性を両立しているものの，このようなスケーラビリティは実現していないため，提案手法は有用である．加えて，シミュレーション対象のコア数が増加した場合でも，シミュレーション速度は低下せず，むしろ向上する傾向が確認できる．全てのベンチマークプログラムで，64Core/14Threadの実行が最も高速化率が高く，コア数の多い大規模なシミュレーションにおいても提案手法は有効と言える．ただし，16コアのシミュレーション時には，14スレッドでの実行時間が改善が見られず，場合によっては悪化している．これは，シミュレーション対象のコア数がスレッド数の整数倍ではないため，シミュレータ上のスレッドに割り振られる最大コア数が8スレッドと14スレッドで変化しない．これにより，最も多くのコアを担当するスレッドがボトルネックとなることに加え，並列度が上昇したことにより同期機構のメモリアクセスの負荷が増加したと考えられる．そのため，16スレッドを同時実行可能な環境では実行性能は向上すると考えられる．姫野ベンチマークとOCEANでは高速化率が低い．これらのベンチマークプログラムでは，各コアにほぼ同数のメモリアクセス命令が分布している．そのため，各スレッドの同期タイミングが重複し，同期処理が集中することで性能が低下した可能性が考えられる．

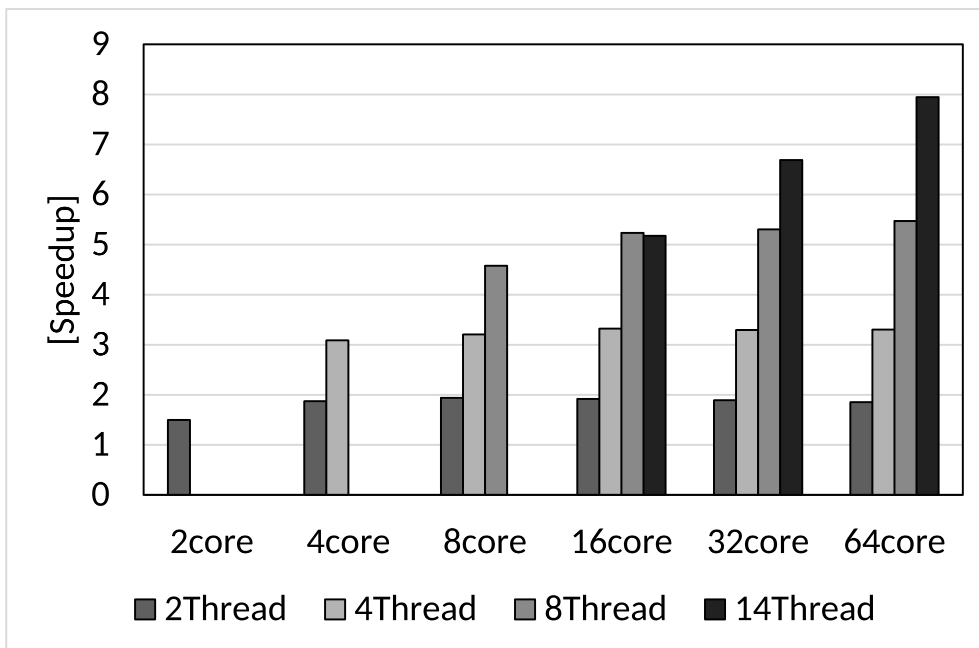

図 6.7: FFT 実行時の提案手法の高速化率

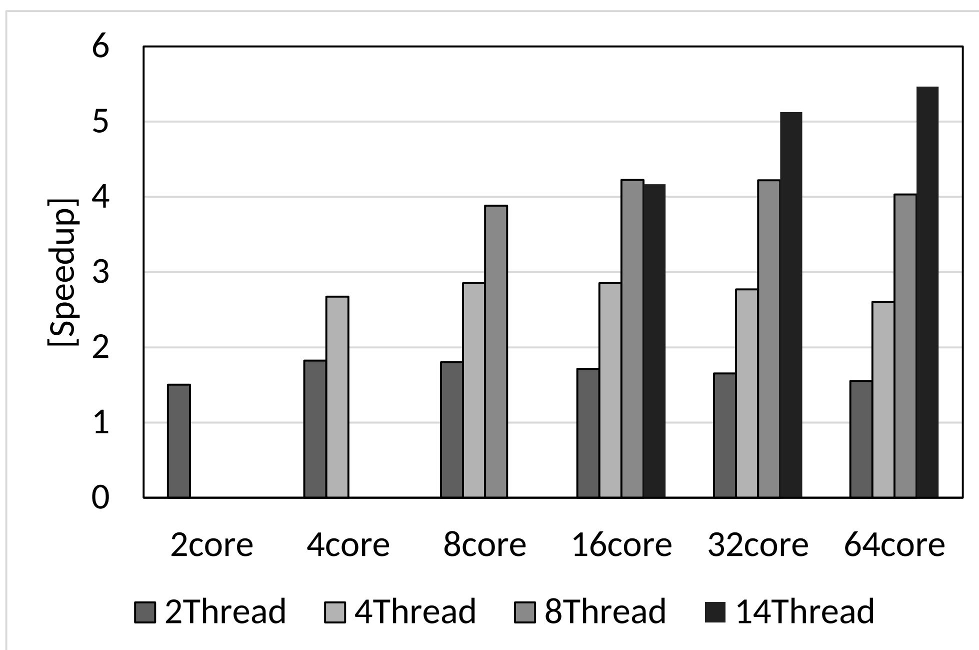

図 6.8: RADIX 実行時の提案手法の高速化率

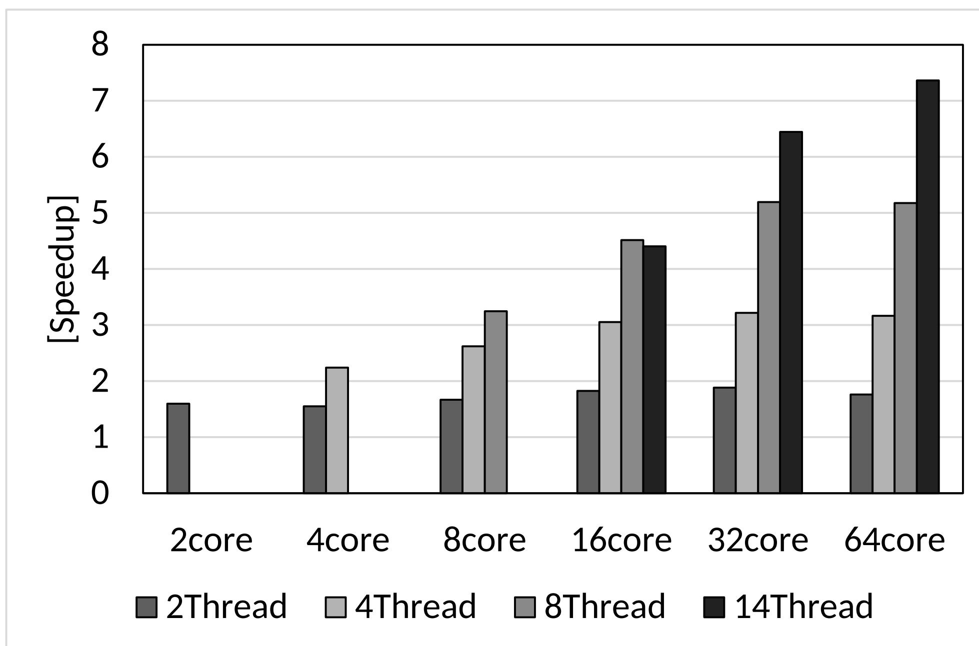

図 6.9: CHOLESKY 実行時の提案手法の高速化率

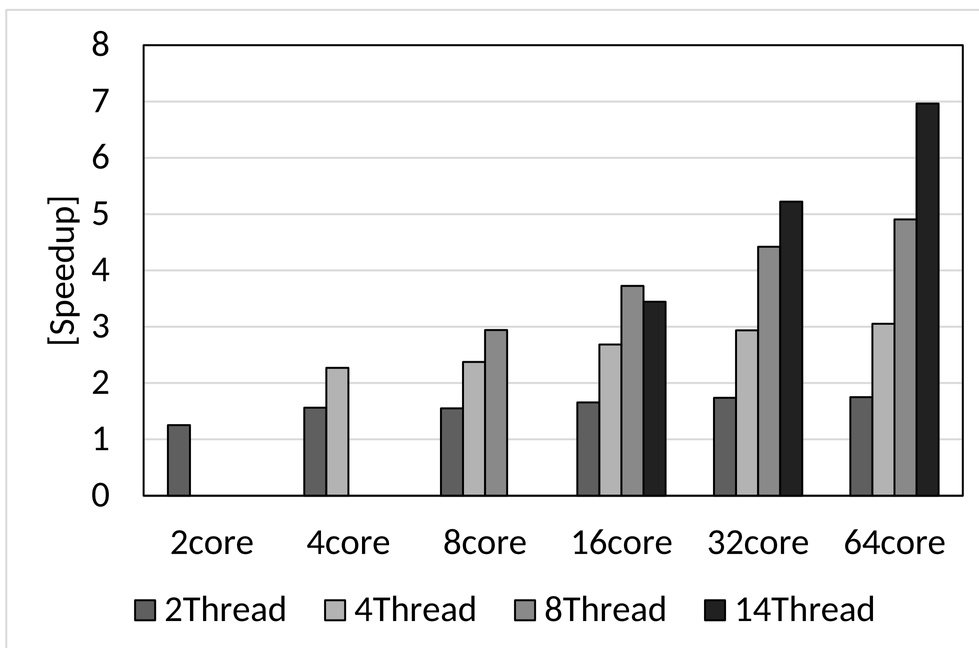

図 6.10: LU(cont.) 実行時の提案手法の高速化率

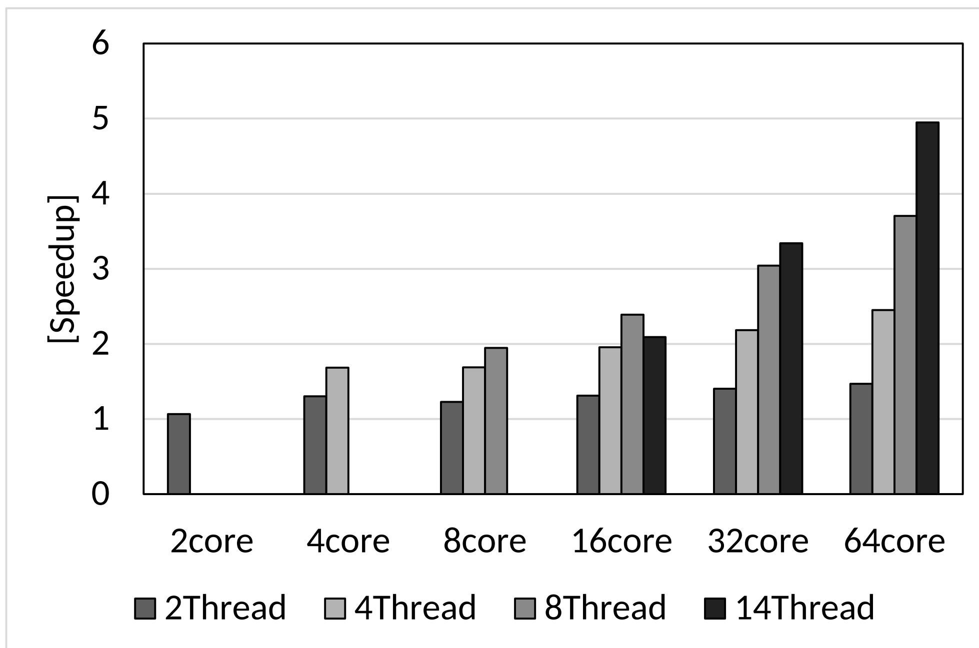

図 6.11: LU(non) 実行時の提案手法の高速化率

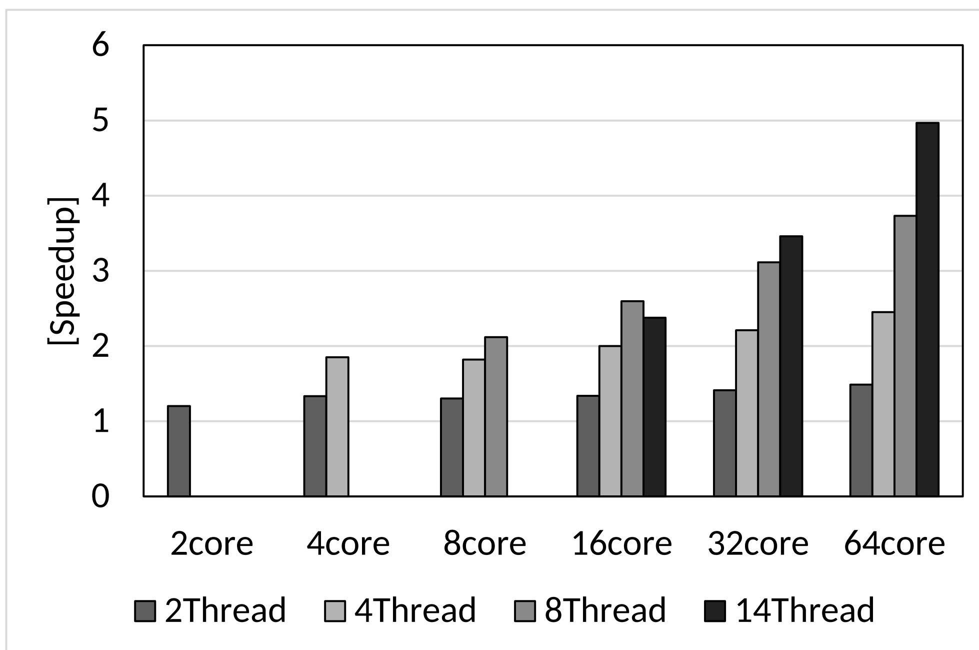

図 6.12: VOLREND 実行時の提案手法の高速化率

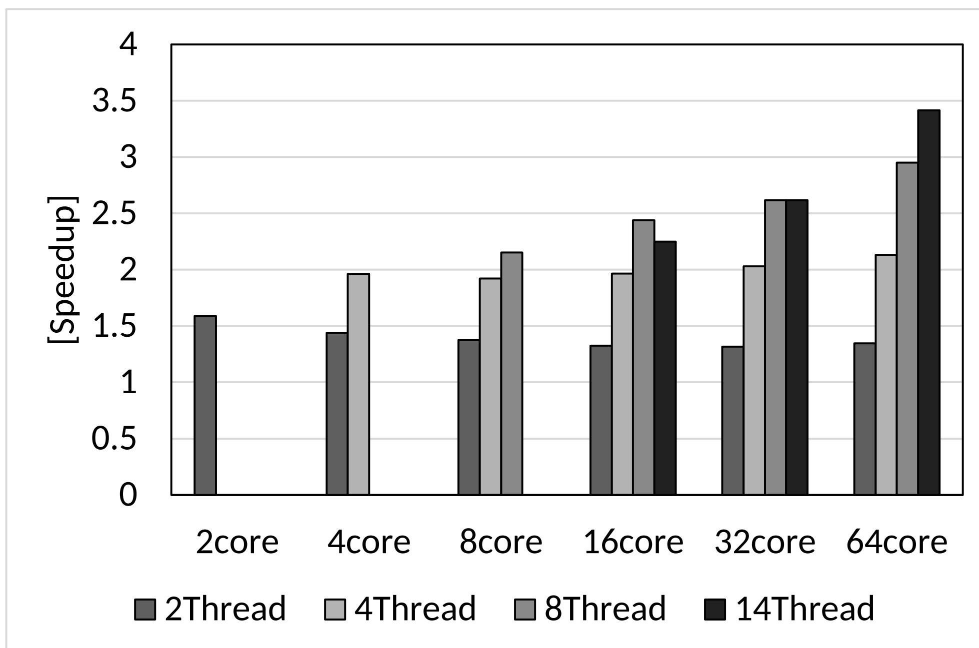

図 6.13: OCEAN 実行時の提案手法の高速化率

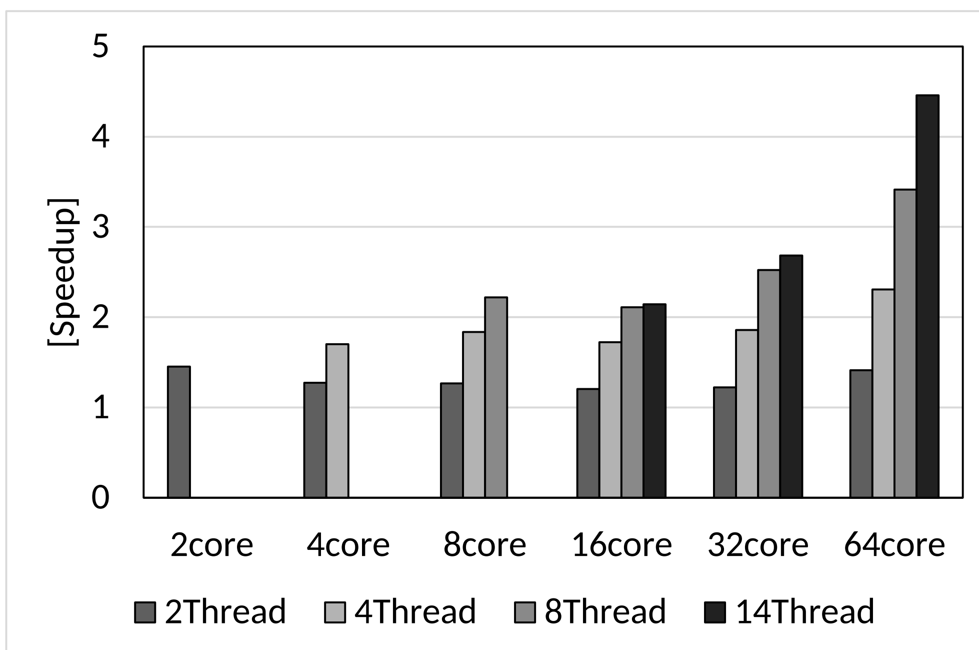

図 6.14: HIMENO 実行時の提案手法の高速化率

図 6.15，図 6.16, 図 6.17，図 6.18 は各最適化を有効化した場合の FFT 及び姫野ベンチマークの実行時間を示している．なお，図中の凡例は以下の通りである．

- Baseline：最適化なし．
- Padding_cache：同期情報のキャッシング，キャッシュラインサイズへのパディング．
- Overhead_reduction：同期失敗時のオーバーヘッド削減．
- Readonly_segment：同期対象の削減 (読込専用データの同期除外)．
- ALL_optimize：全最適化を有効化 (図 6.14 と同じ)．

最適化を行わない場合は，スレッド数の増加に対して実行性能がスケールしていないことが分かる．また，全最適化を有効化したものがほとんどの場合で最も高速であり，最適化によって実行環境の並列性を活用できていると言える．同期情報のキャッシュに対する最適化は特に効果が大きく，スケーラビリティの大幅な改善に成功している．キャッシュラインサイズへのパディング，同期情報のキャッシングは共に，キャッシュコヒーレンシ負荷を低減する一方，ヒット率を低下させる可能性を持つ．そのため，負荷の低い 2 スレッド環境では実行時間が若干悪化したと思われるが，多スレッド環境での効果を考慮すると無視できる範囲である．同期失敗時のオーバーヘッド削減に関しても一定の効果が確認できる．特に，コア数とスレッド数が多い場合に効果が確認できる．同期機構の条件判定数はコア数に依存し，同期の同時処理数はスレッド数は依存するため，これらの増加に伴い同期の失敗件数自体が増加したためと考えられる．一方，読み込み専用データの同期除外は FFT と姫野ベンチマークで効果が異なる．対象となるメモリアクセスが FFT では一定数存在するのに対し，姫野ベンチマークにはあまり含まれていないためと考えられる．また，後者においても他の最適化を阻害するほどの悪影響は見られない．前述の他のベンチマークプログラムの実行結果も考慮すると，提案する同期機構や最適化手法は実行対象のメモリアクセス傾向に大きな影響を受けることが分かる．

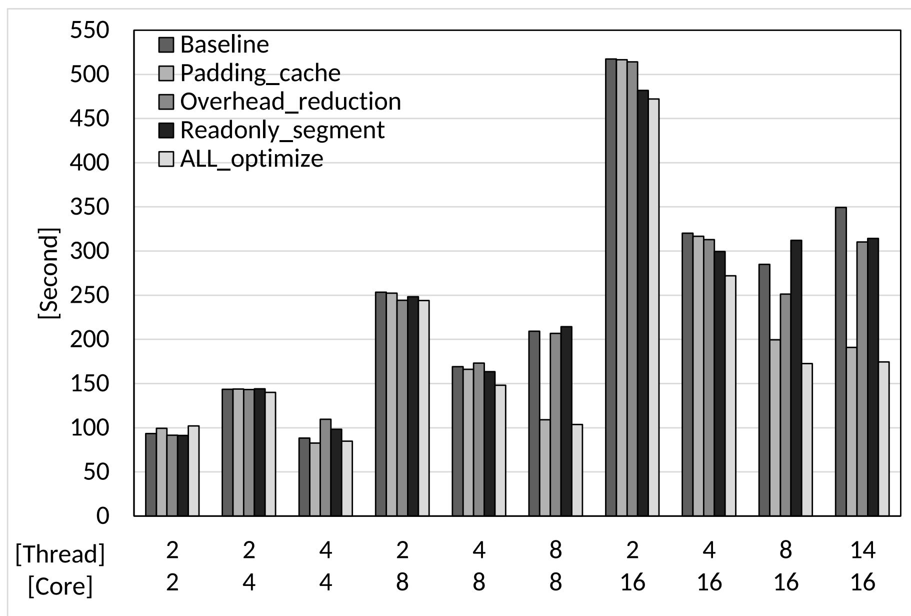

図 6.15: 最適化を有効化した場合の FFT の実行時間

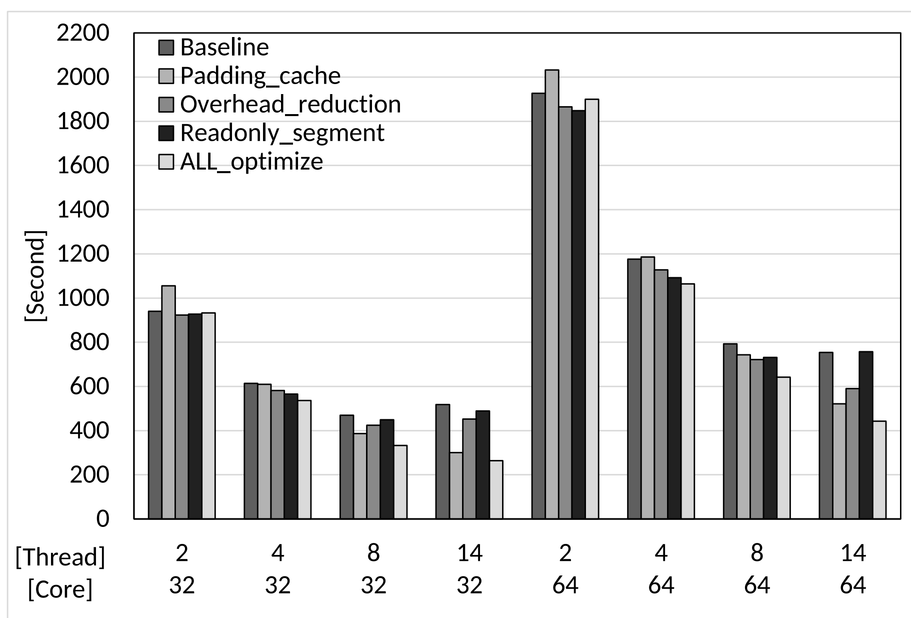

図 6.16: 最適化を有効化した場合の FFT の実行時間 (続き)

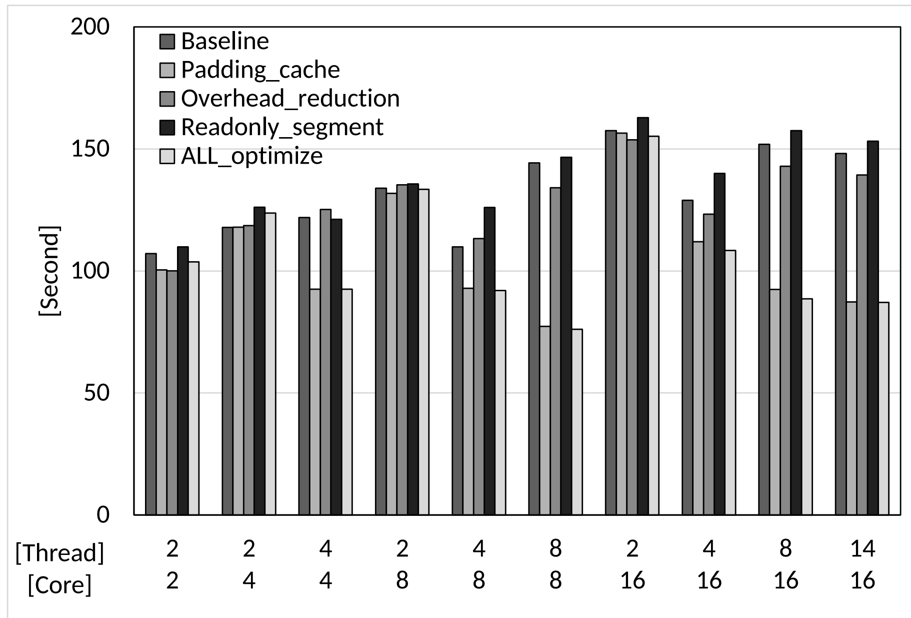

図 6.17: 最適化を有効化した場合の HIMENO の実行時間

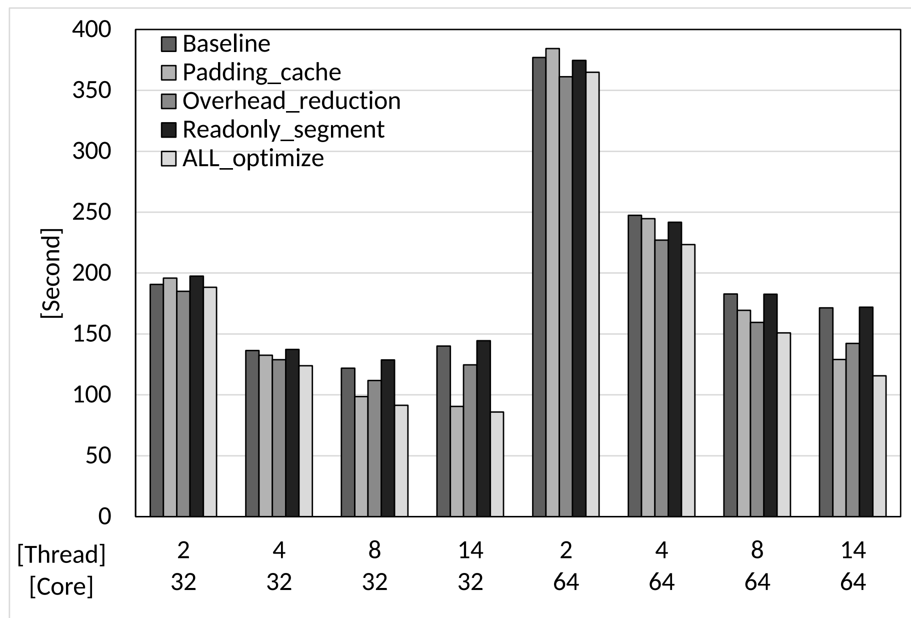

図 6.18: 最適化を有効化した場合の HIMENO の実行時間 (続き)

6.2 小サイズ型チェックポイントの評価

6.2.1 評価環境

小サイズ型チェックポイントの評価を行う．従来のチェックポイントとの比較を行うため，SPEC2000 INT[27] を文献 [1] に示されている条件で実行し，チェックポイントファイルサイズを比較する．なお，シミュレーションの実行環境のメモリ容量が異なり，本稿で述べた部分以外に機能シミュレータ自体の大きな差異が存在する．そのため，ベンチマークプログラムの進行度が大幅に異なると考えられる，mcf，parser については評価から除外する．同様に，実行時間については評価しない．表 6.5 に実行環境を示す．

表 6.5: シミュレーションの実行環境.

CPU	Core i7-2600@2.40GHz
RAM	16GB

6.2.2 評価結果

表 6.6 は，SPEC2000 のチェックポイントを作成した結果を示している．提案手法では従来手法に比べ，全てのベンチマークプログラムでファイルサイズが減少し，最大 17 分の 1 に減少した．これにより，提案手法によるチェックポイントは効率性が大幅に向上していると言える．

表 6.6: チェックポイントファイルのサイズ.

	Skipped Cycle	File Size (MB)	
	(100 million)	Conv.	Propose
gzip	1,189	832	190
bzip	977	834	192
twolf	1,066	119	7

7　まとめと今後の展望

本論文では，従来のプロセッサ検証フレームワークに拡張を行うことで，従来困難であった高速性と効率性の両立を実現した．提案手法による並列シミュレーションでは，最大 7.94 倍の高速化を達成し，全ての試行で同じ結果が得られた．これにより，従来の並列シミュレータでは実現が困難であった，再現性と高速性の両立を実現した．また，ファストスキップ，チェックポイントの併用及び，チェックポイントサイズの小サイズ化を行った．チェックポイントファイルサイズを，最大で従来の 17 分の 1 に削減したことで，大幅な効率性の向上を実現した．加えて，各手法を任意に組み合わせることで，様々な検証フローに柔軟に対応することが可能となった．今後の展望として，NUMA 対応などによる更なる高速化，Slacksim[19] の手法によるコア間通信への対応や，提案フレームワークを活用しての FabHetero の開発が挙げられる．

謝辞

本研究を行うにあたり，多数のご指導を頂きました近藤利夫教授，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学生には常に刺激的な議論を頂き，精神的にも支えられました．あわせて感謝をいたします．

参考文献

[1] T. Nakabayashi, et. al.: “Co-simulation framework for streamlining microprocessor development on standard ASIC design flow”, ASP-DAC2013, pp. 400-405, January, 2014.

[2] K. Kayamuro, et al.: “A Rapid Verification Framework for Developing Multi-core Processor”, CANDAR’16, pp. 384-394, November, 2016.

[3] 萱室高樹, 佐々木敬泰, 深澤祐樹, 近藤利夫, “マルチコアプロセッサの効率的な設計検証に向けたプロセッサシミュレータの並列化”, 電子情報通信学会技術研究報告, Vol.117, No.278, pp.53-58, November, 2017.

[4] T. Okamoto, et. al.: "Detail Design and Evaluation of FabCache", CANDAR'14, pp. 591-595, December, 2014.

[5] 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.

[6] T. Okamoto, et. al.: "FabCache: Cache Design Automation for Heterogeneous Multi-core Processors", CANDAR'13, pp. 602-606, December, 2013.

[7] Y. Seto, et. al.: "FabBus: A Bus Framework for Heterogeneous Multi-core processor", ITC-CSCC2013, pp. 254-257, July, 2013.

[8] G. Hamerly, et. al.: "Simpoint 3.0: Faster and more flexible program phase analysis.", Journal of Instruction Level Parallelism 7.4, pp.1-28, Jun, 2005.

[9] R Kaivola, et. al.: "Replacing Testing with Formal Verification in Intel®Core™i7 Processor Execution Engine Validation.", CAV, Vol. 9, pp.414-429, 2009.

[10] A Adir, et. al.: "A unified methodology for pre-silicon verification and post-silicon validation.", Design,Automation & Test in Europe Conference & Exhibition 2011,pp.1-6,2011.

[11] L. Séméria, et. al.: "Methodology for hardware/software co-verification in C/C++.", ASPDAC'00,pp.405-408,2000.

[12] A. Hoffmann, et. al.: "A Framework for Fast Hardware-Software Cosimulation", Proceedings of the conference on Design, Automation and Test in Europe, pp.760-765, March 2001.

[13] D. Burger and T. M. Austin, "The Simplescalar tool set, version 2.0" ACM SIGARCH Computer architecture News 25.3 (1997): 13-25.

[14] N. Fujieda, et. al.: "SimMips:A MIPS system simulator", Proc. WCAE, pp.32-39, December, 2009.

[15] J. E. Miller, et al. "Graphite: A distributed parallel simulator for multicores", HPCA-16 2010, pp. 1-12, 2010.

[16] Lv. Huiwei, et al.: "P-GAS: Parallelizing a cycle-accurate event-driven many-core processor simulator using parallel discrete event simulation", PADS'10, pp. 89-96, 2010.

[17] S. S. Mukherjee, et al.: "Wisconsin Wind Tunnel II: a fast, portable parallel architecture simulator", IEEE Concurrency 8.4 (2000): 12-20.

[18] H. Matsuo, et al.: " Shaman: A distributed simulator for shared memory multiprocessors", MASCOTS'02, pp.347-355, 2002.

[19] J. Chen, et. al.: "Adaptive and speculative slack simulations of CMPs on CMPs", MICRO-43, pp. 523-534, 2010.

[20] T. Kiesling, et al.: "Time-parallel simulation with approximative state matching", PADS'04, pp. 195-202, 2004.

[21] P. D. Bryan, et al.: "Accelerating multi-threaded application simulation through barrier-interval time-parallelism", MASCOTS'12, pp. 117-126, 2012.

[22] F. Bellard, "QEMU, a fast and portable dynamic translator", USENIX Annual Technical Conference, FREENIX Track, 2005.

[23] M. Wu, et al.: "An effective synchronization approach for fast and accurate multi-core instruction-set simulation", EMSOFT'09, PP. 197-204, 2009.

[24] M. Wu, et. al.: "A distributed timing synchronization technique for parallel multi-core instruction-set simulation", ACM Transactions on Embedded Computing Systems, Vol. 12, No. 1s, Article 54, 2013.

[25] TIS Committee, "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2", May, 1995.

[26] S. C. Woo, et. al.: "The SPLASH-2 programs: Characterization and methodological considerations", ACM SIGARCH Computer Architecture News., Vol. 23, No. 2, ACM, 1995.

[27] SPEC2000, http://www.spec2000.com/

[28] Himeno benchmark, http://accc.riken.jp/supercom/himenobmt/