卒業論文

題目

マルチコアプロセッサの動作検証用
シミュレータの改良と評価

指導教員

近藤 利夫 教授

2016年

三重大学 工学部 情報工学科
計算機アーキテクチャ研究室

伊藤 良 (412807)

内容梗概

近年，構成の異なるコアを複数個用いたヘテロジニアスマルチコアプロセッサが注目を集めている. ヘテロジニアスマルチコアは特徴の異なるアプリケーションに対して最適なコアで実行することにより，消費電力の低減や高速化が期待される. しかし，構成の異なるコアやそれに付随するキャッシュ，バスなど設計・検証にかかる時間がヘテロジニアスマルチコアプロセッサの研究の大きな障害となっている．そのため，スーパースカラコアを自動生成する FabScalar が提案されている．FabScalar には，自動生成されたプロセッサの動作を検証するためにソフトウェアのプロセッサシミュレータである機能シミュレータが搭載されている．

機能シミュレータは浮動小数点演算ユニットがない問題，バイエンディアン非対応の問題，チェックポイント機能の再現性の問題，マルチコアプロセッサ・マルチスレッドプログラムに非対応の問題がある．これらにより，限定的な環境の動作検証にしか利用することができない．そこで，本研究では機能シミュレータに次の改善を実装することによって動作検証機能の改良を目指す．浮動小数点演算ユニットの実装，バイエンディアンへの対応，チェックポイント機能の再現性の向上，チェックポイント機能のマルチコアプロセッサ・マルチスレッドプログラムへの対応を行う．

これらの改良によって，浮動小数点演算ユニットを持ち，バイエンディアンであるプロセッサの動作検証が可能になり，さらに，マルチコアプロセッサ・マルチスレッドプログラムの動作検証にかかる時間の削減に成功した．

Abstract

Single-ISA heterogeneous multi-core architecture which consists of diverse superscalar cores is increasing importance in the processor architecture. Using a proper superscalar core for characteristic in a program contributes to reduce energy consumption and improve performance. However, designing a heterogeneous multi-core processor requires a large design and verification effort. Therefore, FabScalar are proposed that can automatically generate a superscalar core. FabScalar has a function simulator that is software processor simulator for verification.

Function simulator has some problems. As First, Function simulator doesn't have floating-point arithmetic unit. Secondly, Function simulator doesn't adapt bi-endian processor. Thridly, Function simulator have low repeatability of checkpoint functions. Finally, checkpoint functions of Function simulator doesn't adapt to execute multi-core processor and multi-thread programs.

Function simulator can't verify flexible environments becuouse above problems. This study aims to improve a verify function by implementing the followings. The adde function is to implement floating-point arithmetic unit to function simulator, to add endian converter, to improve of repeatability of checkpoint function and to adapt to verify multi-core processor and to execute multi-thread programs.

These improvements are enable to verify a bi-endian processor with floating-point arithmetic unit and to reduce the time of verity using multi-core processor and executing multi-thread programs.

目 次

図 目 次

表 目 次

1 はじめに

1.1 背景

近年，高性能・低消費電力のアプローチとして，構成の異なるコアを複数個用いたヘテロジニアスマルチコアプロセッサが注目を集めている. ヘテロジニアスマルチコアは特徴の異なるアプリケーションに対して最適なコアで実行することにより，消費電力の低減や高速化が期待される. しかし，構成の異なるコアやそれに付随するキャッシュ，バスなどの設計・検証にかかる時間が，ヘテロジニアスマルチコアプロセッサの研究の大きな障害となっている．そのため，スーパースカラコアを自動生成する FabScalar[1] が提案されている．しかし，FabScalar の開発時及び独自開発で回路を追加時に FabScala ｒが自動生成するコアの設計データが正常が動作するか確認する必要がある．そこで，プロセッサの動作を再現し，詳しい動作を確認することが可能であり，検証環境などに柔軟に対応できる，ソフトウェアのプロセッサシミュレータが必要である．FabScalar には，それらの要件を満たした機能シミュレータが搭載されている.

1.2 研究目的

機能シミュレータは動作検証用のプロセッサシミュレータであるが，現在限定的な環境での動作検証にしか利用することができない．限定的な

環境というのは浮動小数点演算ユニットがなく，ビッグエンディアンであるプロセッサである．さらに動作検証時間の削減に効果的なチェックポイント機能が不完全であり，マルチコアプロセッサ検証マルチスレッドプログラムの実行にも対応していない．これらの限定的な環境の動作検証にしか利用できない原因として以下の４つの問題がある．

1. 画像処理などの演算用を担う浮動小数点演算ユニットの動作検証ができない問題
2. 汎用性の高い組み込み向けプロセッサにおいて重要なバイエンディアンプロセッサの動作検証ができない問題
3. チェックポイント機能 [2] の再開後の実行結果が正常では無く，動作検証に影響が出る問題
4. チェックポイント機能をマルチコアプロセッサ・マルチスレッドプログラム環境の動作検証に使用できないため動作検証に時間がかかる問題

そこで，本研究は，機能シミュレータに以下の改善を実装することによって動作検証機能の改良を目指す．

1. 浮動小数点演算ユニット実装

2. バイエンディアン対応

3. チェックポイント機能の再現性向上

4. チェックポイント機能のマルチコア・マルチスレッド対応

2 機能シミュレータ

2.1 概要

本章では，既存の機能シミュレータの問題点を指摘するのに先立ち，問題点に関連する部分の機能シミュレータの理解のため，機能シミュレータの概要及び機能について説明を行う．機能シミュレータはC++で記述され，1命令を1サイクルで実行可能な仮想的なプロセッサをシミュレートする．機能シミュレータとFabScalarで自動生成したコアで同じプログラムを実行し，出力結果を比較することで動作検証を行う．しかし，動作検証には数十億命令のベンチマークプログラムの実行を必要とするため，数十時間のシミュレーションが必要になる．このような場合に，第2.3節と第2.4節で述べるファーストスキップとチェックポイント機能を使用することで動作検証時間の削減が可能である．

2.2 システムコールの内部実行機能

機能シミュレータは，システムコールの内部実行機能がある．システムコールはOSの補助が必要な処理であるため，通常OSの起動が必要になるが，機能シミュレータがシステムコールを内部実行することで，OSを実行せずプログラムの実行ができ，検証時間を削減することが可能である．

2.3 ファーストスキップ

機能シミュレータのファーストスキップ機能でプログラムを実行した場合の概要を図2.1に示す．ファーストスキップはプログラムを任意の命令数まで機能シミュレータ単体で実行後 (図上では検証の対象外の部分)，レジスタなどの必要な情報を自動生成したコアに渡し，動作検証を開始する機能である．これにより，検証部分以外のプログラムの一部を機能シミュレータ単体で実行するため，検証時間の削減が可能である．

2.4 チェックポイント機能

プログラムをチェックポイント機能で実行した場合の概要を図2.2に示す．チェックポイント機能はプログラムを任意の命令数で一時停止させ，チェックポイントファイルを作成し，その後，チェックポイントファ

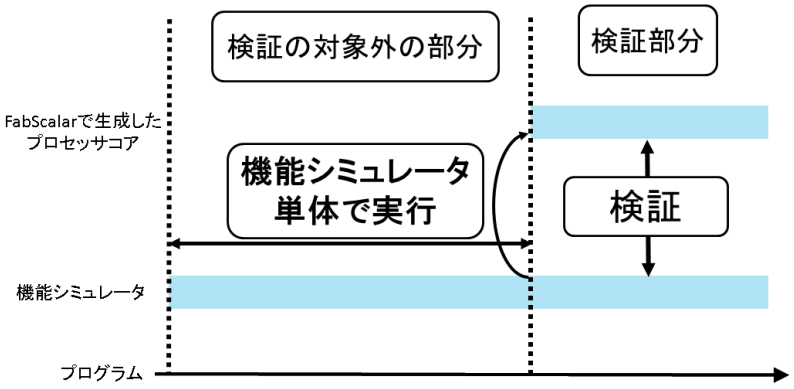

図 2.1: ファーストスキップの概要図

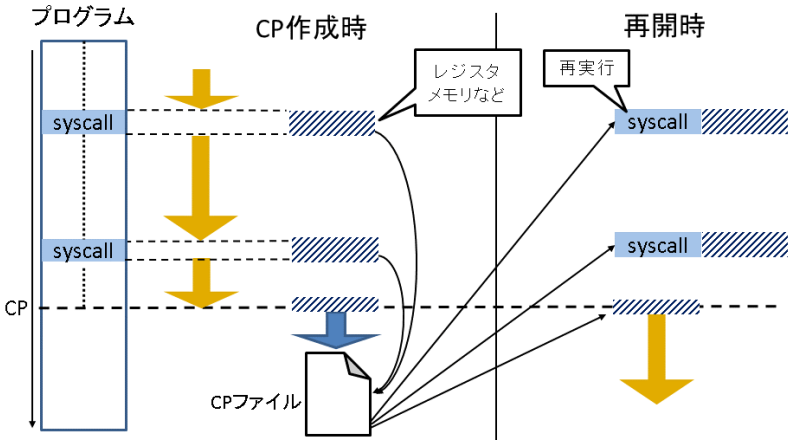

図 2.2: チェックポイント機能の概要図

イルを読み込むことにより，プログラムを途中から再開できる機能である．チェックポイントファイルは，プログラムを途中から再開するために必要な情報を保存するファイルである．チェックポイント機能により，任意の位置からシミュレーションを開始できるため，プログラム内の特定の位置での動作検証，性能評価を行いたい場合に非常に有効である．チェックポイントファイルに保存される情報としては，システムコールの実行履歴，及びレジスタやメモリの情報が挙げられる．

2.4.1 システムコールの再実行

図 2.3 はシステムコールの再実行を行わない場合である．図の場合，チェックポイントファイルからの再開時にファイルのオープンが起きていないため，ファイルへの書き込みが行われない．システムコールはファイルのオープンなど OS の補助が必要な処理であるため，プログラムの実行が正常に行われない問題が起きる．そのため，システムコールの再実行として，チェックポイントファイルの作成よりも前に実行されたシステムコールの実行履歴を復帰時に再実行することにより情報を復元し，正常な実行を行う．

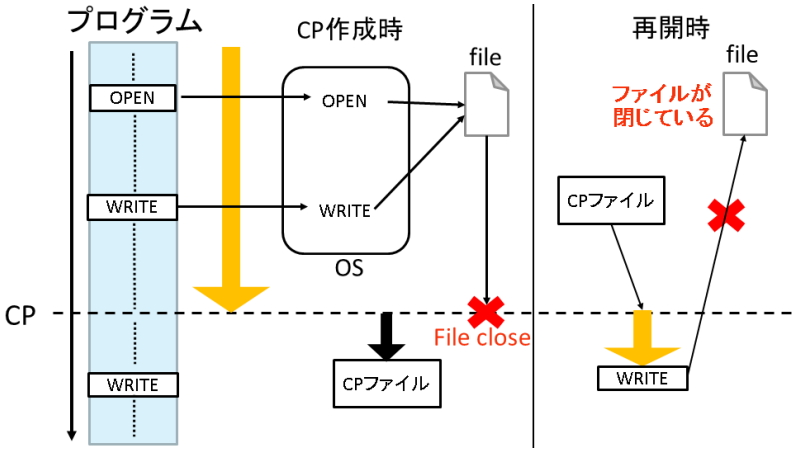

図 2.3: システムコール再実行無しの場合の図

3 関連研究

本研究への関連研究として，研究用のツールで多く使われているプロセッサシミュレータである SimpleScalar[3] と SimMips[4] がある．本章では，これらの関連研究について述べる．

3.1 SimpleScalar

SimpleScalar は，多くの研究やプロセッサの検証などで使われているプロセッサシミュレータである．高速に動作することが目的に作られており，動作が高速であるが，マルチコアプロセッサ・マルチスレッドプログラムのシミュレートにおいては機能シミュレータは可能だが，SimpleScalar は不可能である．さらに，FabScalar の生成するプロセッサとの動作検証に

対応しておらず，高速に動作することを目的に作られたSimpleScalarをFabScalarとの動作検証に対応を行うより，FabScalarとの検証用に作られた拡張性の高い，機能シミュレータを利用したほうが動作検証方法の変更などに柔軟に対応が可能である点で優れている．

3.2 SimMips

SimMipsはプロセッサのハードウェア構成を意識し，シンプルなコードで構成され，拡張性に優れたプロセッサシミュレータである．さらに，機能シミュレータと同じくマルチコアマルチスレッド実行が可能である．しかし，機能シミュレータにはチェックポイント機能があり，動作検証時間を削減することが可能である．そのため，動作検証用シミュレータとして，検証に有効なチェックポイント機能があるため機能シミュレータが優れている．

4 機能シミュレータの問題点

本章では機能シミュレータの４つある問題点

1. 浮動小数点演算ユニット非搭載

2. バイエンディアン非対応

3. チェックポイント機能のマルチコアマルチスレッド非対応

4. チェックポイント機能の再現性問題

以上の説明を順に行う.

4.1 浮動小数点演算ユニット非搭載

FabScalar が自動生成するプロセッサは組み込み機器に多く使われている MIPS32ISA に準拠している．近年，組み込み機器などではグラフィックの複雑化などで浮動小数点演算ユニットが求められている．浮動小数点演算ユニットは浮動小数点演算を行う処理装置であり，画像処理や科学技術演算など複雑な演算を効率よく動作をさせることが可能である. 以上から，浮動小数点演算ユニット含んだプロセッサを自動生成する機能が FabScalar に必要である．しかし，動作検証用の機能シミュレータに浮動小数点演算ユニットが搭載されていない．そのため，浮動小数点演算ユニットを含んだプロセッサの動作検証ができない．よって，機能シミュレータに浮動小数点演算ユニットを搭載する必要がある.

4.2 バイエンディアン非対応

エンディアンとは，メモリとプロセッサレジスタ間の多バイトのデータ格納・取得時のメモリ上の並びの順番の方式のことである．エンディア

ンの種類は一般的にビッグエンディアンとリトルエンディアン，バイエンディアンがある．ビッグエンディアンはデータの上位バイトが最上位アドレスに格納される方式である．リトルエンディアンはデータの上位バイトが最下位アドレスに格納される方式である．バイエンディアンはリトルエンディアンとビッグエンディアンのどちらとも可能な方式である．機能シミュレータが準拠している MIPS32ISA はバイエンディアンプロセッサを対象としている．しかし，機能シミュレータは現在機能シミュレータを実行するホストマシンのエンディアンがリトルエンディアンであり，FabScalar が生成するプロセッサのエンディアンがビッグエンディアンの場合のみ動作検証に対応している．よって，バイエンディアンプロセッサの動作検証が不可能である．そこで，ホストマシンや FabScalar が生成するプロセッサのエンディアンに影響されないシミュレーション環境を実装する必要がある．よって，本研究では機能シミュレータのバイエンディアン対応を行う．エンディアン変換の必要な場合をを図 4.4 に示す．ホストマシンと FabScalar が生成するプロセッサ間で，エンディアンが異なる場合 (図上では左) はエンディアン変換が必要であり，エンディアンが同じ場合 (図上では右) エンディアン変換は必要ない．さらに，現在のエンディアン変換は低速であり，オーバーヘッドになっているため．

エンディアン変換を高速化し実装する必要がある．エンディアン変換が低速な理由は第 5.2 節で詳しく述べる．

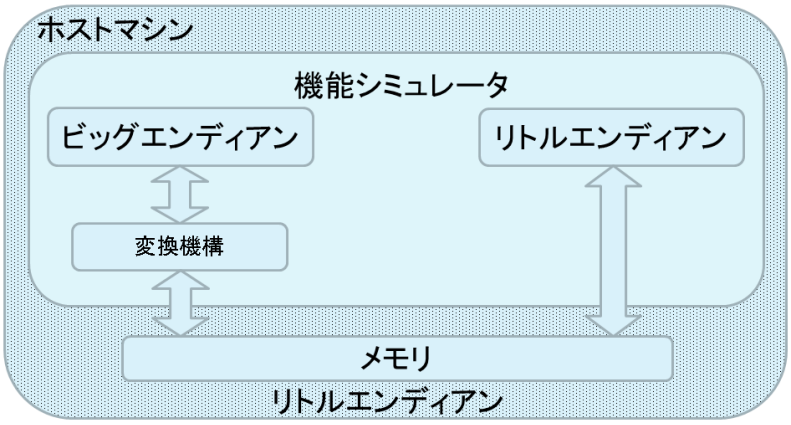

図 4.4: エンディアン変換の概要図

4.3 チェックポイント機能の再現性問題

チェックポイント機能には，再現性が必要である．再現性とは，チェックポイントファイルからの再開時に通常実行と同じ結果が得られることである．しかし，既存の機能シミュレータは以下に述べる 2 つの原因から，再現性が低い問題を抱えている．

4.3.1 再開時のファイル更新処理の問題

ファイルへの追加の書き込みを行うプログラムを，機能シミュレータのチェックポイント機能で，実行した場合に起きる問題である．これは，

チェックポイントファイルからのシミュレーションの再開時に，全てのシステムコールを再実行するため，チェックポイントファイルの作成時・システムコールの再実行時で 2 回のファイルの書き込みが発生してしまう．図 4.5 のようにファイルの中に重複分が発生するため，ファイルの内容が通常実行時とは異なり，再現性が低くなる．

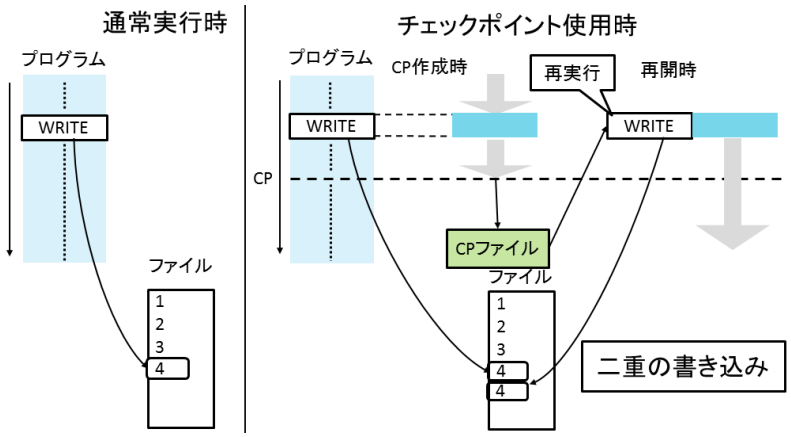

図 4.5: 再開時のファイル更新処理の問題

4.3.2 再開時の時間取得処理問題

機能シミュレータ上でシステム時間を取得するプログラムを実行する場合に起きる問題である．図 4.6 にチェックポイントファイルからの再開を複数回行った場合のプログラムの動作の概要を示す．チェックポイントファイルからの再開後にシステム時間の取得を行うと，再開ごとに取得する時間が異なる（図上では円の部分）ことが原因である．そのため，最

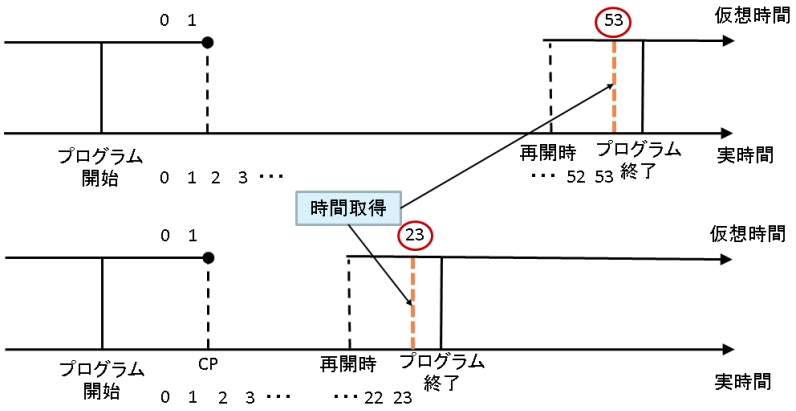

図 4.6: 時間取得処理問題の図

終的な出力結果が通常実行時と比べ異なるため，再現性が低くなる問題である．

4.4 チェックポイント機能のマルチコアマルチスレッド非対応

チェックポイント機能はマルチコアプロセッサ・マルチスレッドプログラムに非対応である．マルチコアプロセッサ対応では，コアごとに別々の情報を扱うため，各コアのレジスタなどの情報とチェックポイントファイル上の情報を関連付ける機構が必要になる．マルチスレッドプログラム対応ではスレッドの入れ替えなどのスレッドの管理が必要である．図 4.7 はスレッド管理処理の概要を示す．スレッドの管理はシステムコールを元にスケジューラが行っているため，システムコールを保存することによって対応できる．しかし，マルチスレッドプログラムでは定期的にス

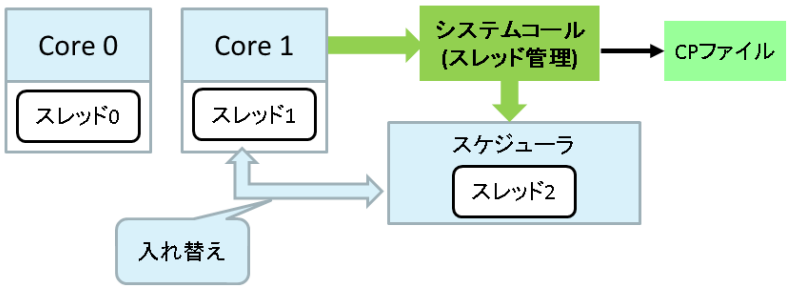

図 4.7: スレッド管理処理

レッド管理のシステムコールが実行されるため，マルチスレッドプログラムをチェックポイント機能で実行すると，多くのシステムコールがチェックポイントファイルに保存されてしまうため，チェックポイントファイルが肥大化する問題がある．

5 実装要求及び提案手法

本章では，実装要求及び上記の各問題に対する提案を述べる．

5.1 浮動小数点演算ユニット

MIPS32ISA の浮動小数点演算ユニットは 32 ビット幅の浮動小数点レジスタが 32 本と 32 ビット幅のステータスレジスタが 5 本あり，浮動小数点レジスタは浮動小数点レジスタと同じビット幅の 32 ビットの float 型と 64 ビットの double 型を扱う．そのため，浮動小数点レジスタは，64 ビット幅と 32 ビット幅に対応できるような実装を行う必要がある．よって，

同じメモリの領域を複数の型が使用することが可能な構造である共用体を使用し，float 型 (32 ビット)，double 型 (64 ビット) 共に扱うことのできる浮動小数点レジスタを実装する．

5.2 バイエンディアン対応及びエンディアン変換高速化

第 4.2 項で述べたように現在エンディアン変換がオーバーヘッドになっているため，エンディアン変換を高速化し，バイエンディアン対応を行う必要がある．

バイエンディアン対応では，ビッグエンディアン，リトルエンディアンどちらで実行するかの選択をプログラムから読み取ることで実行可能な実装を行う．実行ファイルである ELF ファイルのヘッダ情報を読み取り，対応エンディアンを判別し，判別したエンディアンプロセッサとして動作を開始する．

エンディアン変換は従来手法では，メモリから１バイトずつ読み込み，そして変換を行いレジスタに１バイトずつ書き込むことで実装している．しかし，メモリアクセスは低速であるためのオーバーヘッドになっている．そこで，メモリアクセスを減らすことが必要と考え，エンディアン変換高速化を考案した．エンディアン変換高速化はメモリから必要なビット数をすべて取り込み，変換命令を使い変換し，一回でレジスタにすべて

書き込む処理を行う．この手法によりメモリアクセスを 64 ビット幅の場合で 8 回から 1 回に削減することが可能である．32 ビット幅のエンディアン変換の従来手法と提案手法の比較を図 5.8 に示す．変換命令には x86 系

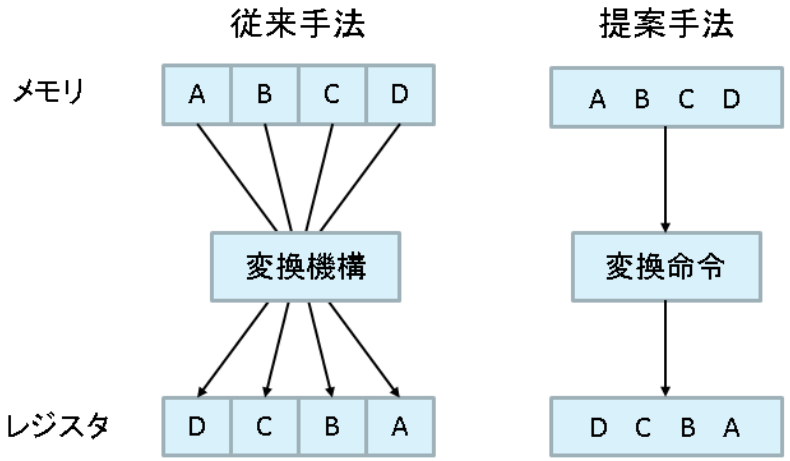

図 5.8: エンディアン変換 (32 ビット幅)

プロセッサの独自のエンディアン変換命令である bswap 命令使用し，実装を行う．x86 系プロセッサ以外の場合は以下のような，ビット演算を用いて実装を行う．

```
x=((((x) & 0xff000000) >> 24) | (((x) & 0x00ff0000) >>  8) |
(((x) & 0x0000ff00) <<  8) | (((x) & 0x000000ff) << 24))
```

5.3　チェックポイント機能の再現性問題に対する提案

第 4.3 節で述べたチェックポイント機能の再現性問題の原因である再開時のファイル更新処理の問題と再開時の時間取得処理問題に対する提案を以下の各項で述べる．

5.3.1　ファイル更新処理の問題に対する提案

第 4.3.1 項で述べた問題に対する提案手法として，復帰時のファイル更新処理スキップと対象ファイルの複製の提案を行う．

復帰時のファイル更新処理スキップは，チェックポイントファイルの容量を抑えることが可能で，動作も高速だが，再現性は完全ではない．

対象ファイルの複製は，チェックポイントファイルの容量が増え，低速となるが，再現性が確保される．本研究では，動作検証に必要な高速動作を求めるため，前者の復帰時のファイル更新処理スキップの実装・評価を行った．以下にて，それぞれについての詳細を述べる．

A. 復帰時のファイル更新処理スキップ

チェックポイントファイルから再開時のシステムコールの再実行時に 2 重の書き込みが起きてしまうことが再現性低下の原因である．そこで，復帰時のファイル更新処理スキップの提案を行う．復帰時のファイル更新

処理スキップは，必要のない書き込みを実行しないことにより２重の書き込みを防ぐ手法である．ファイルの状態を保存するファイル管理テーブルを作成し，ファイルの操作時にテーブルを参照することで，システムコール再実行の実行・不実行を判別する．図 5.9 にファイル管理テーブルの動作を示す．動作としては，チェックポイントファイルからの再開時のシステムコール再実行処理でファイルのオープンの処理が起きた時に，ファイルの状態をファイル管理テーブルに保存する．このとき保存する情報はファイルのオープンの状態 (書き込みモード，追記モードなど）とOS がファイルを管理する際に使用するファイルディスクリプタという識別子とファイルの書き込み位置などの情報をもつファイルポインタを保存する．ファイル管理テーブルは，上記の情報を格納できる配列で実装し，ファイルディスクリプタを配列番号とすることによって管理を行う．

保存後，再実行時にファイルに書込み処理があるとファイル管理テーブルをファイルディスクリプタで参照し，追記モードとして開いているファイルであった場合，処理のスキップを行う．これによって，２重の書き込みを回避することが可能である．

また，チェックポイントファイルからの再開後にファイルへの書込みが起きる場合，ファイルへの書き込み位置がずれてしまうため，ファイル

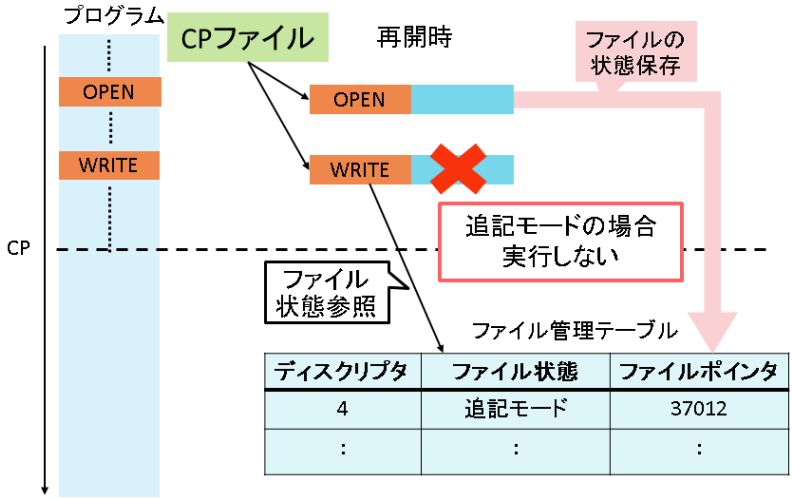

図 5.9: ファイル管理テーブルの動作

ポインタをチェックポイントファイル作成時に保存しておいた状態にファイル管理テーブルを用いて復元する事で対応する．これにより，高速かつ再現性の高いチェックポイント機能の実現が可能である．

B. 対象ファイルの複製

２重の書き込みの問題は，チェックポイント機能使用時，チェックポイントファイルからの再開時に同じファイルに変更を行っているため起きる問題である．そのため，複数回実行を行った場合，前回実行時の処理がファイルの中に残ってしまい，再現性が低くなる．つまり，チェックポイントファイルからの再開時にファイルの内容を常に同じ状態にすることにより，高い再現性が可能になる．そこで，対象ファイルの複製を提案

する．対象ファイルの複製はチェックポイントファイル作成時にファイルを複製保存し，チェックポイントファイルからの再開の度に保存した更新ファイルを復元する手法である．ファイルの複製はファイルオープン時にファイルの状態（書き込みモード，追記モードなど）を確認し，追記モードであった場合のみファイルの複製を行う．このとき複製したファイルを別の名前で保存する．その後チェックポイントファイルからの再開時に複製したファイルを複製することによりオリジナルのファイルの復元を行う．概要を図 5.10 に示す．提案手法により，常に同じファイル内容を

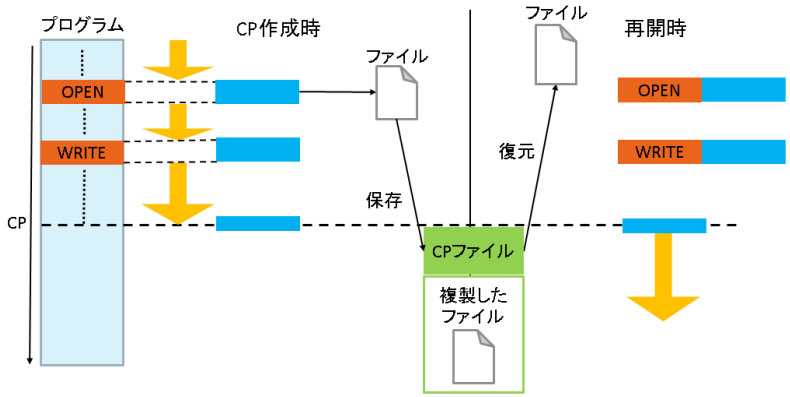

図 5.10: 対象ファイルの複製

対象に書き込み，読み込みを行うため，非常に高い再現性を保つことができる．しかし，動作が低速なため，本研究では採用しない．

5.3.2　機能シミュレータ内動作クロック固定

第 4.3.2 項で述べた再開時の時間取得処理の問題はホストマシンの実時間を元に時間取得を行っているために起きる問題である．従来手法では機能シミュレータ内の時間をホストマシンの実時間を元に取得しているため，チェックポイント機能を使用した場合に時間取得が正常では無くなる．そこで，機能シミュレータ内の時間を実行サイクル数と仮想の CPU 動作クロックを元に算出し進める機能シミュレータ内動作クロック固定を提案する．機能シミュレータ内動作クロック固定の説明は図 5.11 を元に説明を行う．プログラム開始時（図上では A）に機能シミュレータ内の

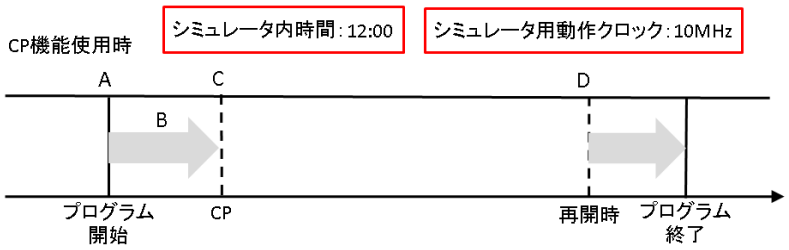

図 5.11: プログラムの流れの図

動作クロック固定とシミュレート開始時間の設定を行う．動作クロックはデフォルト値で 10MHz であり，機能シミュレータの引数でも与えることができる．シミュレート開始時間は実時間をシミュレート開始時に取得し保存を行う．シミュレートを進めていく時（図上では B）に，実行サ

イクル数の数もカウントも行う．時間取得のシステムコールが起きた場合は以下のようなの計算式でシミュレータ内時間の算出を行う．

$$\text{シミュレータ内の時間} = \text{シミュレート開始時間} + \frac{\text{実行サイクル時間}}{\text{動作クロック}} \quad (1)$$

このように，同じサイクル数の場合シミュレータ内で同じ時間を進むため，再現性を保つことが可能になる．さらに，この手法では，チェックポイント機能を使用する場合，チェックポイントファイル作成時（図上ではC）のシミュレータ内の時間と動作クロックをチェックポイントファイルに保存を行い，その後，プログラム再開時（図上ではD）にチェックポイントファイルに保存済みである時間と動作クロックを復元することによって対応が可能である．

5.4 スレッド管理処理スキップ

第4.4節で述べたとおり，マルチコア・マルチスレッド対応により起こる問題としてチェックポイントファイル肥大化問題がある．チェックポイントファイル肥大化問題はスレッド管理の処理における頻繁なシステムコールの呼び出しがファイルの肥大化の原因である．スレッド管理処理スキップの概要を図5.12に示す．スレッド管理のシステムコールは定期的に呼ばれるため，実行される回数が多い．そこで，スレッド管理のシス

テムコールにはスレッドの入れ替えが起きる場合とスレッドの入れ替えが起きない場合があることに注目し，スレッド管理処理スキップを提案する．スレッド管理処理のスキップはスレッド管理のシステムコールが実行され，スレッドの切り替えが起きなかった場合，チェックポイントファイルにシステムコールの実行履歴の保存を行わない．これにより，ファイ

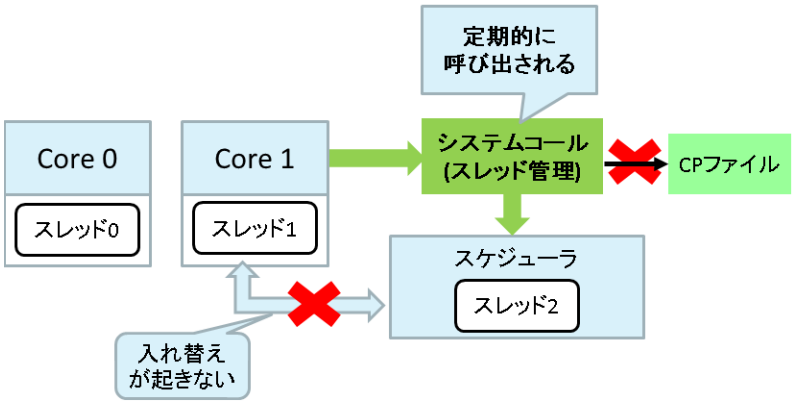

図 5.12: スレッド管理処理スキップの概要

ルに書き込むシステムコールの数を削減でき，チェックポイントファイルの肥大化を抑える．

6 性能評価

本章では，前章述べた提案手法，実装についての評価及び考察を以下の各項で述べる．

6.1 評価方法，評価結果及び考察

6.1.1 浮動小数点演算ユニット追加及びバイエンディアン対応の評価

第 5.1 節，第 5.2 節で述べた浮動小数点演算ユニット追加及びバイエンディアン対応の評価として，プログラムを動作させ，実行結果を比較し，浮動小数点演算ユニットの動作確認とバイエンディアン対応の動作確認を行う．さらに，実行時間を比較することにより，エンディアン変換の高速化の評価を行う．

評価環境は SPEC2000 CFP2000[6] から複数個ランダムに抽出し，リトルエンディアン用とビッグエンディアン用のプログラムをエンディアン変換の高速化を実装した機能シミュレータと従来の機能シミュレータで実行する．

図 6.13 に評価結果を示す．横軸は実行したベンチマークであり，縦軸は機能シミュレータをリトルエンディアンとして実行した実行時間を 1 に正規化し，提案手法と従来手法を実装した機能シミュレータをビッグエンディアンとして実行した実行時間である．

エンディアン変換の高速化を実装した機能シミュレータと従来手法の機能シミュレータの実行時間を比較し，浮動小数点演算ユニットの追加及びバイエンディアン対応が正しく実装できていることと，最大 7 %，平

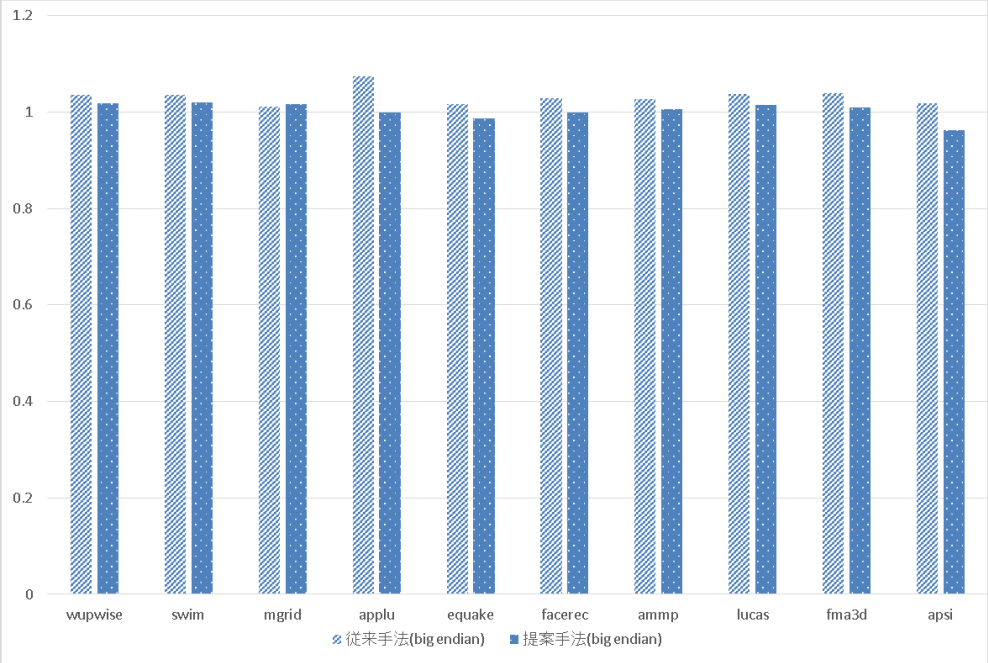

図 6.13: 実行時間の比較

均３％の削減ができていることが確認できた．エンディアン変換の高速化によりメモリのアクセスが減ったことが要因と考えられる．

6.1.2 再現性の向上の評価

第 5.3 節で述べた，再現性の問題に対しての提案手法の評価として，プログラムを動作させ，実行結果を比較し，機能シミュレータ内動作クロック固定と復帰時のファイル更新処理スキップにより再現性が保たれることを確認する．

姫野ベンチを機能シミュレータ内動作クロック固定の実装された機能

シミュレータと従来手法の機能シミュレータで実行する．上記の実行は前者をチェックポイント機能で実行し，10 億命令実行後にチェックポイントファイルを作成し一時停止させ，その後，チェックポイントファイルから再開するという動作を行う．後者は通常実行を行う．実行結果の一部を図 6.14 に示す. 実行結果を確認し，チェックポイント機能のメッセージ以外値が同じであることが確認できる．

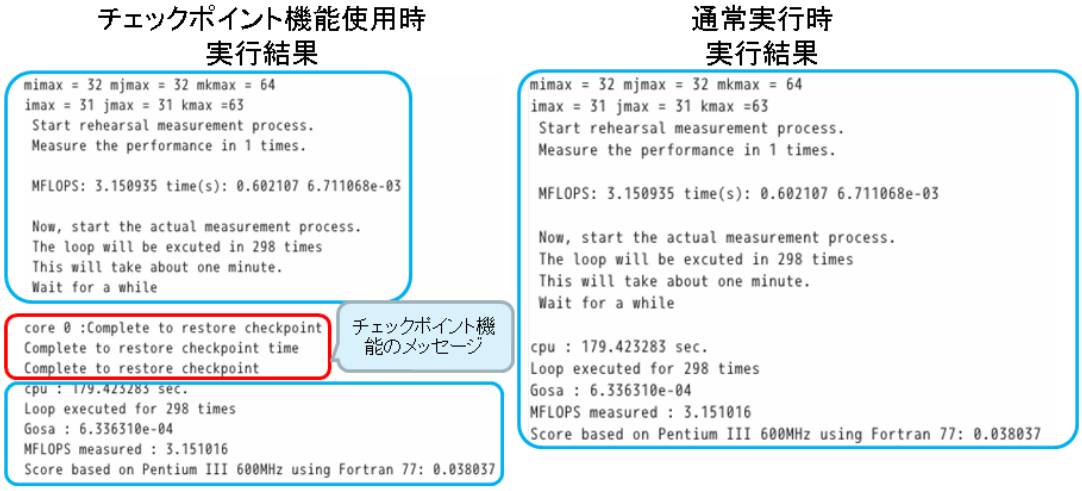

図 6.14: 姫野ベンチ実行結果 (一部抜粋)

ファイル更新処理が含まれているプログラムを復帰時のファイル更新処理スキップを実装した機能シミュレータと従来手法の機能シミュレータで実行する．上記の実行は前者をチェックポイント機能で実行し，10 億命令実行後にチェックポイントファイルを作成し一時停止させ，その後，

チェックポイントファイルから再開するという動作で実行する．後者は通常実行を行う．ファイル内容の一部を図 6.15 に示す．プログラムは変数を１から１ずつインクリメントをし、その経過をファイルに書き込むというプログラムであるがファイル内容を確認し，ファイルの内容が同じことが確認できる．

図 6.15: ファイル更新処理を含むプログラム実行後のファイル内容 (一部抜粋)

機能シミュレータ内動作クロック固定と復帰時のファイル更新処理スキップの提案手法を実装した機能シミュレータで，チェックポイント機能を使用した場合と通常実行時のファイル内容を比較し，再現性に問題ないことを確認した．

6.1.3 マルチコアマルチスレッド対応及びスレッド管理処理スキップの評価

マルチコアマルチスレッド対応の評価として，プログラムを動作させ，実行結果を比較し，チェックポイント機能のマルチコアマルチスレッド対応を確認する．さらに，チェックポイントファイルの容量を比較し，第5.4節で述べた，チェックポイントファイルの肥大化問題に対する提案手法のスレッド管理処理のスキップの評価を行う．評価環境は2コア4スレッドで姫野ベンチ [5] をスレッド管理処理スキップの実装された機能シミュレータと従来手法の機能シミュレータで実行する．上記の実行はチェックポイント機能を，10億命令で一時停止させ，チェックポイントファイルを作成した後，チェックポイントファイルを読み込むことにより一時停止した場所から再開するという動作を行う．表6.1に評価結果を示す．

表 6.1: スレッド管理処理のスキップの評価結果

	機能シミュレータ	
	提案手法	従来手法
CP ファイル容量	20.4MB	40.6MB

チェックポイント機能を使用し，マルチコア・マルチスレッドプログラムが正常に動作することから，チェックポイントのマルチコア・マルチスレッド対応の実装が正しく行われていることが確認できた．また，チェッ

クポイントファイルの肥大化問題に対するスレッド管理処理スキップを実装した機能シミュレータでは，チェックポイントファイルの容量の 49%の削減に成功した．

6.1.4 チェックポイント機能による動作検証時間削減の評価

再現性の向上問題に対する提案手法により再現性が確認できたチェックポイント機能による動作検証時間削減の評価として，実行時間を比較し，再現性が保たれているチェックポイント機能を使用時と通常実行時の検証にかかる時間の削減の評価を行う．

姫野ベンチを機能シミュレータの通常実行とチェックポイントファイル (10 億命令で一時停止させたもの) からの再開後と SimpleScalar で実行する．表 6.2 が評価結果である．再現性の向上問題に対する提案手法により，

表 6.2: 復帰時のファイル更新処理スキップの評価結果

	機能シミュレータ		SimpleScalar
	CP 機能使用時	通常実行	通常実行
実行時間 (秒)	15.54	89.62	163.91

再現性が確認できたチェックポイント機能により，動作検証時間の 83%の削減を実現した．よって，機能シミュレータのチェックポイント機能を利用することによって，大きく検証時間を削減できるため，研究分野で広く使われている高速で動作することを目的としたシミュレータであるが，

その他の検証時間削減機能がない SimpleScalar と比較しても動作検証にかかる時間が少ないことから，当研究での提案手法の実装を行った機能シミュレータの優位性が証明できた.

7 おわりに

7.1 まとめ

機能シミュレータに複雑な演算を効率よく動作可能な浮動小数点演算ユニットの追加，異なるデータ通信方式のプロセッサシミュレートの為のバイエンディアン対応を行い，浮動小数点演算ユニットを持ち，バイエンディアンのプロセッサの動作検証が可能になった．加えて，エンディアン変換高速化を提案・評価を行った．その結果，実行時間の最大７％，平均３％の削減ができた.

さらに，動作検証時間に有効なチェックポイント機能のマルチコア・マルチスレッド対応を行い，マルチコアプロセッサ・マルチスレッドプログラムの動作検証時間の削減が可能になり，また付随する問題に対して，スレッド管理処理スキップを提案・評価することによりチェックポイントファイルの容量の 49%の削減に成功した．また，チェックポイント機能の再現性の問題に対して，機能シミュレータ内動作クロック固定と復帰時のファイル更新処理スキップを提案・評価し，チェックポイント機能の再

現性を高め，動作検証時間の 83%の削減に成功した.

以上の改良によって浮動小数点演算ユニットの追加，バイエンディアン対応により検証可能なプロセッサがさらに汎用性の高い検証用シミュレータになり，チェックポイント機能のマルチコアマルチスレッド対応，チェックポイント機能の再現性の向上により動作検証時間を削減することが可能になった.

7.2 今後の課題

今後の課題として，本研究で実装を行わなかった対象ファイルの復元の実装，及び，更なる再現性向上手法の提案・実装が挙げられる．また，実際に FabScalar の自動生成するコアの検証に用いて，提案手法の評価及び更に動作検証用に適した改良なども挙げられる.

謝辞

本研究の機会を与えて頂いた近藤 利夫 教授 ，並びにご指導，助言頂いた佐々木 敬泰 助教授，深沢 祐樹 研究員の皆様にも心より感謝いたします.

参考文献

[1] 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.

[2] Tomoyuki Nakabayashi , et.al.,“Co-simulation Framework for Streamlining Microprocessor Development on Standard ASIC Design Flow.”,ASP-DAC2013,pp.400-405,2014

[3] Simplescalar,http://www.simplescalar.com/

[4] 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.

[5] 姫野ベンチマーク-理化学研究所情報基盤センター, http://accc.riken.jp/supercom/himenobmt/

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