

修士論文

STRAIGHT アーキテクチャの  
アプリケーション実行モデルとその性能解析

平成 31 年 01 月 31 日提出

指導教員

坂井 修一 教授

東京大学大学院 情報理工学系研究科  
電子情報学専攻

48-176430 福田 晃史

# 概要

STRAIGHT は独自の命令セット・アーキテクチャの導入を通じて電力性能と実行効率の両立を目指したアーキテクチャである．本研究では既存の STRAIGHT アーキテクチャの評価基盤の問題点を明らかにした上で，STRAIGHT アーキテクチャのより精確な評価に必要なシステムコールなどの要件を定義し，評価基盤を新規に開発した．また，それによって今まで不可能であった実アプリケーションによる STRAIGHT アーキテクチャの評価を可能とし，STRAIGHT アーキテクチャの理想的なハードウェア構成について検討を行った．

# 目次

第 1 章	はじめに	6
第 2 章	Out-of-Order スーパースカラプロセッサ	8
2.1	概要 . . . . .	8
2.2	フロントエンド . . . . .	8
2.3	バックエンド . . . . .	11
第 3 章	STRAIGHT アーキテクチャ	13
3.1	STRAIGHT の命令表現及びレジスタ管理 . . . . .	13
3.2	特殊レジスタ . . . . .	14
3.3	In-Order スケジューラ . . . . .	15
3.4	コンパイラ . . . . .	16
3.5	STRAIGHT の性能 . . . . .	18
第 4 章	STRAIGHT シミュレータの実現	20
4.1	既存のシミュレータの問題 . . . . .	20
4.2	システムコールの実現 . . . . .	21
4.3	シミュレーション・モデル . . . . .	23
4.4	その他の問題 . . . . .	23
第 5 章	評価	24
5.1	評価環境 . . . . .	24

5.2	基本評価 . . . . .	25
5.3	命令ウィンドウの拡張について . . . . .	31
第 6 章	関連研究	36
第 7 章	おわりに	38
参考文献		39

# 図目次

2.1	Out-of-Order スーパースカラプロセッサのパイプライン . . . . .	9
3.1	STRAIGHT の命令によるプログラムの一部 . . . . .	14
3.2	フロントエンドで並列に処理される RP . . . . .	15
3.3	In-Order スケジューラ . . . . .	16
3.4	合流後から合流前の値を参照する図 . . . . .	18
5.1	473.astar の実行性能 . . . . .	28
5.2	473.astar の実行命令数 . . . . .	29
5.3	473.astar の制御フローグラフの一部 . . . . .	30
5.4	RNN の実行性能 . . . . .	31
5.5	RNN の実行命令数 . . . . .	32
5.6	命令ウィンドウのエントリ数を変えた時の実行性能 . . . . .	34
5.7	命令ウィンドウのエントリ数に加えてレジスタファイルの容量を変化させた時（表 5.3）の RNN の実行性能 . . . . .	34

# 表目次

5.1	2-way, 4-way のパラメータ . . . . .	26
5.2	6-way, 8-way のパラメータ . . . . .	27
5.3	レジスタ・ファイルの容量の変化 . . . . .	34

# 第 1 章

## はじめに

CPU の性能向上は並列化の困難な部分をも含むあらゆるプログラムの実行時間を短縮させる。これは、命令を依存関係の許す限り並び替え、投機的実行を交えて並列に実行するという命令レベルの並列性（ILP: Instruction Level Parallelism）を抽出するというアプローチによって達成されてきた。しかしそれを行う Out-of-Order スーパースカラプロセッサは、キャッシュメモリを除くと、命令のスケジューリングなどの制御を行う回路がプロセッサの大部分を占め、その消費電力はウェイト数の 2 乗から 3 乗に比例して大きくなる。トランジスタの微細化に伴って集積度を向上させても単位面積あたりの消費電力は変わらないというデナード・スケーリング則が崩壊して以降、熱の問題からチップ上のトランジスタをすべて同時に稼働させることはできないダークシリコン問題 [1] が指摘され、厳しい電力制約を満たしながら性能向上を果たすべく様々なアプローチが模索されてきた。

Out-of-Order スーパースカラプロセッサの消費電力を削減しつつ性能を向上させるアーキテクチャとして STRAIGHT アーキテクチャ [2-5] が提案されている。STRAIGHT アーキテクチャは命令間の依存関係を陽に表現する独自の命令形式を持つ。その命令表現によって、既存の Out-of-Order スーパースカラプロセッサに置いて電力消費の主要因の一つであるレジスタ・リネーミングなしに命令の Out-of-Order 実行が可能となる。加えてリネーミング・ロジックによって制限されていたフロントエンド・パイプライン幅を容易に拡張できる。また、レジスタ管理のコストが低減されるため、リカバリ機構がシン

プルになり高速なりカバリが行える。しかし、STRAIGHT アーキテクチャについては、組み込み向けとして広く使われているベンチマークプログラムである Dhrystone [6] 及び Coremark [7] でのシミュレータによる評価、FPGA による実装と評価が行われているという段階にとどまる。

本論文では既存の STRAIGHT アーキテクチャのシミュレータの問題点を明らかにした上で、STRAIGHT アーキテクチャのより精確な評価に必要なシステムコールなどの要件を定義し、シミュレータを新規に開発した。また、それによって今まで不可能であった実アプリケーションによる STRAIGHT アーキテクチャの評価を可能とし、STRAIGHT アーキテクチャの最適な構成を明らかにした。

本論文の構成は次のとおりである。まず、第 2 章で STRAIGHT アーキテクチャに関係ある要素を中心に、Out-of-Order スーパースカラプロセッサを概観する。続く第 3 章では STRAIGHT アーキテクチャについて説明を行う。第 4 章では、既存の STRAIGHT アーキテクチャのシミュレータの問題点を洗い出し、それらの解決方法を提案する。さらに、第 5 章では、問題点を改善した新たな STRAIGHT シミュレータを用いて、従来行えなかった RNN や経路探索のプログラムによる評価を行う。第 6 章では、関連研究をまとめる。最後に第 7 章では、本研究のまとめを行う。



## 第 2 章

# Out-of-Order スーパースカラプロセッサ

### 2.1 概要

本章では Out-of-Order スーパースカラプロセッサについて、特に STRAIGHT アーキテクチャと関わりの深い部分を中心に説明を行う。なお、Out-of-Order なプロセッサとは、命令の並び替えて実行するプロセッサのことである。Out-of-Order の対義語は In-Order であり、これはプログラム・オーダに従って実行を行うことを指す。

Out-of-Order スーパースカラプロセッサは、図 2.1 に示したように大きくフロントエンドとバックエンドの 2 つに分けられる。フロントエンドは In-Order に命令を取得し、依存関係を解析・緩和する。フロントエンド・パイプラインを抜けた命令は命令ウィンドウと呼ばれるバッファに溜められる。バックエンド・パイプラインは、命令ウィンドウ中の命令を命令間の依存や使用できる演算器に応じて動的に選択し、実行を行う。なお、プロセッサの状態の更新はプログラムの意味論にあわせて In-Order に行われる必要がある。

### 2.2 フロントエンド

フロントエンド・パイプラインは主に以下のステージから構成される。

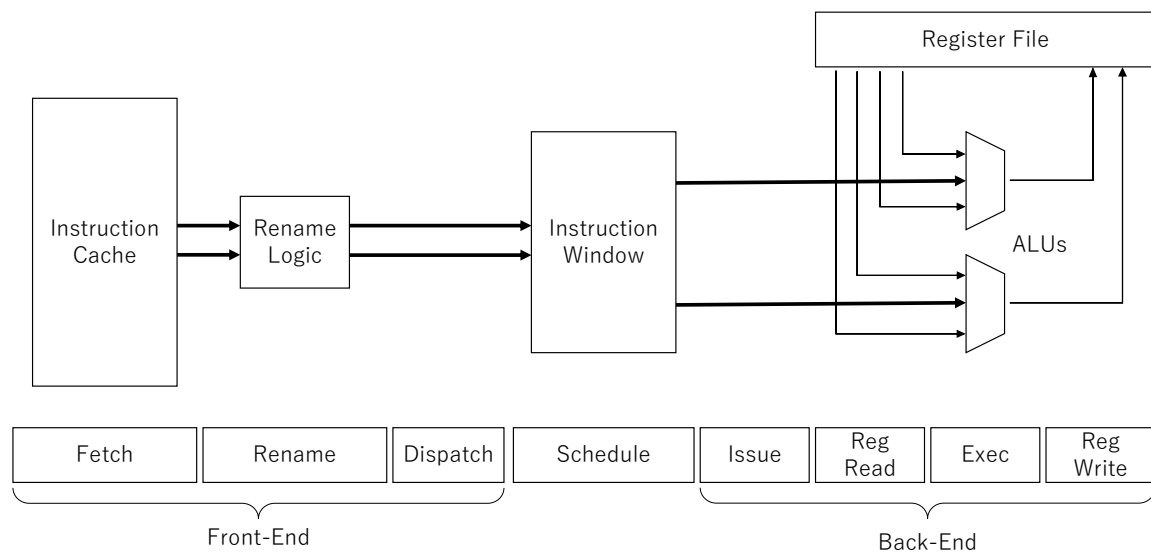


図 2.1 Out-of-Order スーパースカラプロセッサのパイプライン

1. **Instruction Fetch** : 命令キャッシュから命令を取得する
2. **Instruction Decode** : 命令のビット列を解釈し、後段で利用しやすいようにオペレーションやレジスタ番号、即値といったものに分割・整理する。場合によっては複数のマイクロ命令に分解する
3. **Rename** : 命令間の依存関係を解析し、偽の依存を含まない形に変形する
4. **Dispatch** : 命令ウィンドウにリネーム済みの命令を書き込む

リネーム・ステージは **Out-of-Order** スーパースカラプロセッサに特有のものである。

**Out-of-Order** に実行しても実行結果を正しく保つためには以下の依存関係が正しく保たなければならない。

- フロー依存 : 先行する命令の出力を後続する命令が入力とする
- 逆依存 : 先行する命令の入力を後続する命令が上書きする
- 出力依存 : 先行する命令の出力を後続する命令が上書きする

このうち、真の依存と呼ばれるフロー依存のみがプログラムの意味において重要である。

偽の依存と呼ばれる残りの2つはレジスタなどの資源が限られているためにプログラム中に出現した依存関係であり、十分な資源が存在すればこれらの依存関係を含まない等価なプログラムを考えることができる。

リネーム・ステージではこの偽の依存関係を、命令セット・アーキテクチャで定義された論理レジスタをより数の多いプロセッサ内部の物理レジスタ番号へと変換すること（レジスタ・リネーミングと呼ばれる）で解消する。このレジスタ・リネーミングは、先行する命令の入出力先となる物理レジスタと後続の命令の出力先となる物理レジスタが一致しないように行われる。リネーミング・ロジックは、主に論理レジスタ番号と物理レジスタ番号の対応を記録したレジスタ・マップ・テーブル（RMT: Register Map Table）と新たに割付け可能な物理レジスタのキューであるフリーリストから構成される。RMTはRAMあるいはCAMで構成するが、一般的な3オペランド（デスティネーション・レジスタ1, ソース・レジスタ2）の命令セット・アーキテクチャでは、 $4 \times$ （ウェイ数）のポート数が必要となる。ここでウェイ数とはパイプラインのレーンの数、つまりあるステージで同時に処理できる命令の数のことである。なぜならば、一命令の処理にソース・レジスタ番号の読み出しに2ポート、デスティネーション・レジスタ番号の読み出しと書き込みに1ポートずつ必要であるからである。読み出された古いデスティネーション物理レジスタ番号はレジスタの解放に必要となる。そして、RAMはポート数 $p$ に対してトランジスタ数は $O(p)$ 、配線の面積は $O(p^2)$ で増加し、RMTはポート数が多いため配線がポート数の自乗に比例する大きなRAMとなる。RMTをCAMで構成した場合も同様にCAMはポート数の自乗に比例する大きなものとなる。通常は複数命令同時にレジスタ・リネーミングが行われるが、同じフェッチ・グループに属する命令間でもプログラム・オーダー上で先行する命令のリネーム結果が後続の命令のリネームに影響するため、RMTから読み出される情報に加えて並行して命令間の依存関係を解析し、リネームを行う必要がある。通常は複数命令同時にレジスタ・リネーミングが行われるが、同じフェッチ・グループに属する命令間でもプログラム・オーダー上で先行する命令のリネーム結果が後続の命令のリネームに影響するため、RMTから読み出される情報に加えて並行して命令間の依存関係を解析し、リネームを行う必要がある。

なお、フロントエンドには命令フェッチを円滑に行うための命令キャッシュ及び分岐予

測器も含まれる。命令キャッシュはメインメモリへのアクセス・レイテンシを削減するための小容量で高速なメモリである。分岐予測器は、分岐命令をフェッチした直後にもパイプラインをストールさせずにフェッチを続けられるようにするためのアドレスを予測するハードウェアである。

## 2.3 バックエンド

バックエンド・パイプラインは主に以下のステージから構成される。

1. **Wake Up/Select** : 命令ウィンドウから依存関係が満たされかつ演算器などのハードウェア資源が利用可能となった命令を選択し発行する
2. **Register Read** : 物理レジスタにある演算に必要な値を読み出す
3. **Execution** : ALU などで演算を行う。メモリアクセス命令ではアドレスの計算も含む
4. **Memory Access** : キャッシュへアクセスする
5. **Commit** : プロセッサ状態の非可逆的な更新を **In-Order** に行う

バックエンドでは、命令のオペランドと資源が確保できた命令から **Out-of-Order** に命令の実行を行う。しかし、パイプラインの最後に位置するコミットは **In-Order** に行われる。これは正確な例外を保証するためである。つまり、ある命令が例外を起こした時に、プログラム・オーダー上でその命令より前の命令はすべて実行が完了しており、それよりあとの命令はまったく実行されていないという状況を再現できるようにするためである。演算が終了した命令は、アクティブ・リストというバッファに保持される。アクティブ・リストでは **In-Order** に各命令が例外を発生していないかを確認し、例外を起こしていなければその命令の実行を完了したとし、物理レジスタなどの資源の解放を行う（これがコミットである）。例外が起きていた場合はパイプラインをフラッシュし、**RMT** の状態や **PC** の値などをその命令が実行される前の状態まで巻き戻す（リカバリ）。従って、コミットが行われるまでプロセッサは状態を巻き戻せるだけの情報を保持しておく必要がある。

通常リカバリのために **RMT** にポートを増設することではなく、レジスタ・リネーミン

グに用いたポートを利用する。そのため、リカバリはウェイ数ずつ行われることになる。ウェイ数を増やすとイン・フライトな命令がそれより速い速度で増加し、リカバリにかかる時間もイン・フライトな命令数に比例して増えることから、大規模なプロセッサの性能はこのリカバリ・ペナルティの影響を受けることが指摘されている [8].

なお、本節ではバックエンドでレジスタ・ファイルを読み出す方式 [9–11] を仮定して説明したが、フロントエンドでレジスタ・ファイル [12,13] を読み出す方式も存在する。いずれにせよハードウェアの複雑度は変わらない。本稿では一貫してバックエンドで読み出す方式を前提とする。

## 第 3 章

# STRAIGHT アーキテクチャ

### 3.1 STRAIGHT の命令表現及びレジスタ管理

STRAIGHT は命令間の依存関係を陽に表現する命令形式を持つ。従来のアーキテクチャは命令のオペランド・レジスタを論理レジスタに割り振られた番号で示すが、STRAIGHT アーキテクチャではソース・オペランドをその値を生成した命令までの距離で示す。またそのようなソース・オペランドの指定のため、デスティネーション・オペランドは命令中では明示されない。図 3.1 に STRAIGHT 命令を用いたプログラムの一部を示した。図中で [1] のように示された数値がソース・オペランドを指す。I2 の [1] は一つ前の命令である I1 の結果を指し、同様に I3 の [1] は I2 の結果を指す。ソース・オペランドとして指定できる値は命令中のビット長に制限される。現在の STRAIGHT 命令セット・アーキテクチャでは距離を 7 ビットで表現するため、参照できるのは  $2^7 - 1 = 127$  命令前の結果までとなる。なお、命令の機能は標準的な RISC 同様のものが定義されている。

マイクロアーキテクチャレベルでの命令間の値の授受は、STRAIGHT でも従来の RISC アーキテクチャ同様、物理レジスタを介することで行われる。物理レジスタの割当には RP (Register Pointer) と呼ばれる特殊なレジスタを用いる。RP は一命令フェッチされる度にインクリメントされる。命令のデスティネーション・レジスタは、その命令がフェッチされたときの RP が指し示す番号の物理レジスタである。RP の最大値は

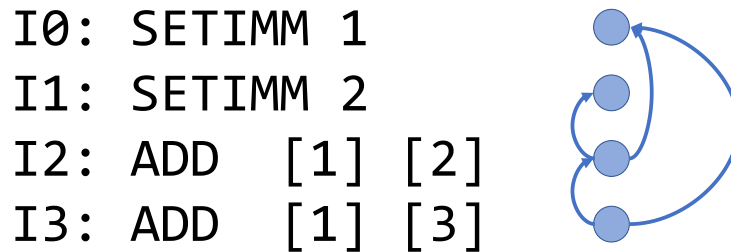


図 3.1 STRAIGHT の命令によるプログラムの一部

論理レジスタ数 + ROB のエントリ数である。この値に達した後、RP は 0 に戻る。つまり、割り当てられた物理レジスタは一定時間後に再利用される。この再利用はその値が参照されえない距離まで遠のいたあとに行われるため、再利用に伴う問題は発生しない。さらに、分岐予測ミスなどが発生してリカバリが必要となった場合も、論理的に参照できる最大の距離に加えてインフライトな命令の数だけ物理レジスタ数、RP の最大値に余裕があればリカバリに必要な情報を失うことはない。

従来の Out-of-Order スーパースカラプロセッサでのリカバリは、RMT やフリーリストの状態をフラッシュされる命令数に比例する時間をかけて順次状態を巻き戻す。STRAIGHT アーキテクチャには RMT やフリーリストは存在せず、リカバリは RP を例外を起こした命令のフェッチ時の RP に戻すだけでよい。これは単なる減算に過ぎず軽量の処理である。

## 3.2 特殊レジスタ

STRAIGHT は距離によって参照することのできる汎用レジスタの他に特殊レジスタを持つ。特殊レジスタには先述の RP と SP (Stack Pointer) がある。STRAIGHT アーキテクチャはこれらの操作・参照を行うための特殊な命令を用意している。RP は、一命令フェッチされるごとにインクリメントされる他、RPINC 命令によってもインクリメントされる。RPINC は通常の RP のインクリメント分に加えて更に即値で指定された分 RP を増加させる命令である。また、SP は SPADDi, SPLD, SPST, AUiSP 命令（まとめて SP 命令と呼称する）で操作、参照される。SPADDi 命令は SP に即値で指定された値を加算する命令である。SPLD 命令は SP に即値を足して得られるアドレスからロードを行

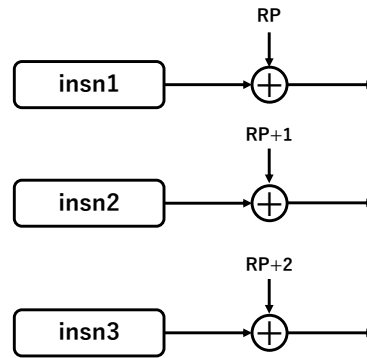


図 3.2 フロントエンドで並列に処理される RP

う。SPST 命令も SPLD 同様であり、ストアする値の入ったレジスタをオペランドとして取るもののみが異なる。

RPINC は後続のすべての命令と依存関係を持ち、SP 操作命令は SP 読出し命令と依存関係を持つ。しかし、この依存関係は **Out-of-Order** 実行に大きな影響を与えない。何故ならば、特殊レジスタ命令による特殊レジスタの読出し・書込みは、バックエンドで得られる情報、つまり汎用レジスタの値、を利用しないため、フロントエンドで処理が可能であるからである。フロントエンド・パイプラインではすべての命令が **In-Order** に処理されるため、同一フェッチグループ内での依存関係のみを考えればよく、特殊レジスタに対する操作は値の読出しと加算を行った値の書込みだけであり、追加の回路は軽量なものとなる (図 3.2)。

また、特殊レジスタの値は例外発生時に巻き戻すべきアーキテクチャの状態に含まれる。よってバックエンドにも特殊レジスタの値を送る必要がありアクティブリストを巨大化させうるが、これには分離リオーダーバッファの考え方が有効であり、ハードウェアの増加を抑制することが可能である [14]。

### 3.3 In-Order スケジューラ

STRAIGHT アーキテクチャでは、命令をディスパッチする際スケジューラのエントリ数と参照されるレジスタの距離を比較することで、その命令の依存する命令が命令ウィンドウ内に存在しうるかどうかを簡単に判断することができる。このことを用いてスケ



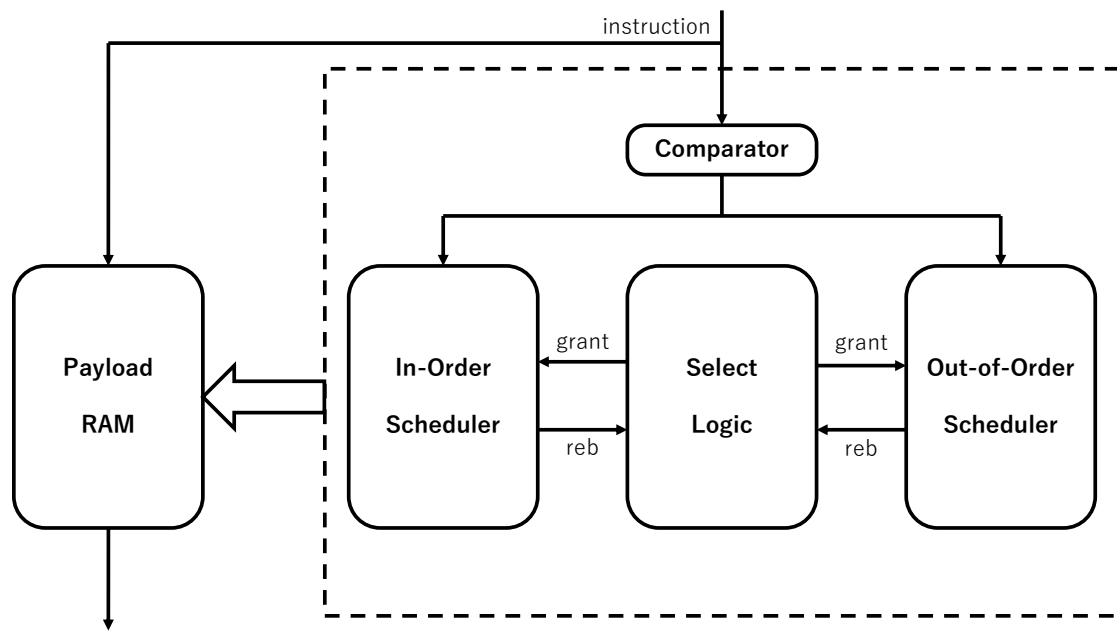


図 3.3 In-Order スケジューラ

ジューラを従来の Out-of-Order スケジューラと In-Order スケジューラに分割してハードウェアを簡素化する手法も提案されている [15]. その図を図 3.3 に示した. もしディスパッチされた命令の依存する命令がスケジューラに存在する場合は, その命令は通常のスケジューラに送られ, 一般的な場合と同様のウェイクアップ・イシューの処理が行われる. ディスパッチされた命令の依存する命令がスケジューラに存在しない場合, その命令は In-Order スケジューラに送られる. In-Order スケジューラにある命令はすでに結果が得られているのでウェイクアップは必要がなく, 単に In-Order に発行することができる. 従って, In-Order スケジューラは単なるキューで構成することができ, ハードウェアを簡素なものとすることができる.

### 3.4 コンパイラ

STRAIGHT アーキテクチャの命令表現が従来のアーキテクチャと異なるのは以下の二点である.

- 命令間に偽の依存が存在してはならない
- 距離を用いてソース・オペランドを指定する

前者に関しては、そもそも命令間に偽の依存を持たない表現は静的単一代入形式や継続渡しスタイル、A 正規型などという形でコンパイラの間表現として広く使われており、コンパイラに特段の困難をもたらすことはない。問題となるのは後者である。

例えば、図 3.4 のように制御フローの合流後の命令から合流前の値を参照するプログラムを考える。この時、一般には合流後の命令から参照したい値はパスによって異なる距離にある。従って STRAIGHT コンパイラにはこのような場合にも適切に値を参照できるような距離の調節が要請される。

最も簡単な距離調節の方法は、各ベーシック・ブロックの末尾に後続のベーシック・ブロックから参照される値を集約する方法である。これは、合流の直前のすべてのベーシック・ブロック中にあり、合流後に参照される値についてひとつずつ合流前のすべてのベーシック・ブロック末尾に固定した場所を与え、値のコピーを行う RMOV 命令や NOP 命令を用いてその場所にその値が置かれるように調節する。この手法はいかなる場合にも適用できる一方で、合流の多い制御フローを持つプログラムでは RMOV 命令の数が大幅に増加し、性能を大きく低下させる。そのようなものに対しては STRAIGHT 特有のコンパイラ最適化が必要となる。

また、関数呼出しでも従来の命令セット・アーキテクチャと異なる規約が必要となる。従来の命令セット・アーキテクチャではレジスタごとに、関数を呼ぶ前にメモリに退避しておくべき *caller-save* か呼ばれた関数が必要に応じてメモリに退避する *callee-save* かが ABI によって定められていた。しかし、STRAIGHT アーキテクチャには原理的に長寿命なレジスタが存在しない。また *callee* から復帰したあと *caller* が *callee* 以前の値がレジスタのどこに存在するかあるいはもう存在しないかを知る手段は一般にはない。従って、STRAIGHT アーキテクチャではすべてのレジスタを *caller-save* としなければならない。

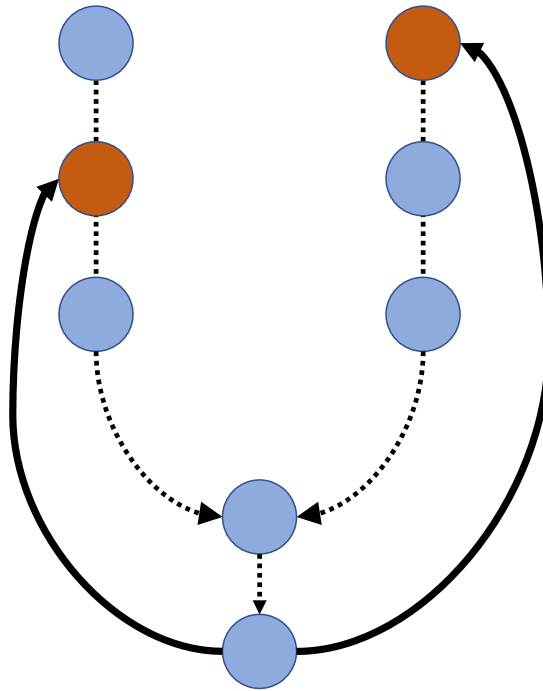


図 3.4 合流後から合流前の値を参照する図

## 3.5 STRAIGHT の性能

STRAIGHT アーキテクチャが従来のアーキテクチャに比べて実行性能において優位な点としては以下のような点があげられる。

- レジスタ・リネーミングが行われなためフロントエンド幅の拡張が容易である
- 論理レジスタ数が増加する
- リオーダーバッファで管理する状態が少ないためリオーダーバッファのエントリ数の拡張が容易である
- 予測ミス時のリカバリが高速である
- パイプライン段数が減る

一方で、次のような欠点も存在する。

- 距離調節に必要な命令が増加し、クリティカルパスを延ばす可能性がある

- callee-save なレジスタしか存在せず，既存のアーキテクチャより ABI の自由度が少ない

## 第 4 章

# STRAIGHT シミュレータの実現

### 4.1 既存のシミュレータの問題

STRAIGHT のシミュレータとしては佐保田によるもの [16] が存在し，Livermore Kernels の動作が確認されている．しかし，以下に挙げるような構造的な問題点を抱えている．

- システムコールのエミュレーション機能がない
- 特殊レジスタ命令の動作を正確にシミュレーションできない
- リカバリ・モデルが正確ではない
- ゲスト・アプリケーションが利用できるメモリ領域が十分な広さではない
- シングルコア以外の構成でのシミュレーションが困難である

以下に各問題の詳細を述べる．

システムコールのエミュレーションが行えないということは，OS の上で動く一般的なアプリケーションによる評価が行えないということであり，アーキテクチャの評価として説得力に欠ける．もちろんシステムコールのエミュレーション機能をシミュレータが提供しなくとも，OS を動作させることが可能なシミュレータならば，OS を含めてアプリケーションを動作させればよいが，当該シミュレータにはコンテキスト・スイッチなどに関する機能はなく，それも困難である．また，それに伴い，ゲスト・プログラムがファイル入

出力などの副作用を起こす手段が存在せず、シミュレータの動作の検証が困難であり、実際に十分な検証がなされていなかった。

特殊レジスタ命令は、3.2 で述べた通り、バックエンドに依存関係を持ち込まず効率的に処理することが可能な STRAIGHT プロセッサを正確にシミュレートするためにフロントエンドで処理するのが望ましい。しかし、当該シミュレータはバックエンドで特殊レジスタ命令を実行するようになっており、かつ特殊レジスタに関する依存関係の解析やレジスタ・リネーミングを行っていない。そのため、特殊レジスタ命令が Out-of-Order 実行される場合正しい動作が保証されないモデルとなっている。

リカバリについても従来のアーキテクチャと等しい時間がかかるようになっており、軽量なりカバリが可能であるという STRAIGHT アーキテクチャの特徴を反映できていない。

ゲスト・アプリケーションが利用できるメモリ領域が十分な広さではない、マルチコアのシミュレーションが困難である、というのは、全体的に柔軟性を欠いた実装であることに端を発する。佐保田がシミュレータを作る際にベースとした鬼斬式 [17] は Out-of-Order スーパースカラプロセッサの挙動を正確に再現する目的で作られており、研究の対象となるような変則的なマイクロアーキテクチャの実装が容易になるように拡張性高く実装されている。鬼斬式は十分に効率的なメモリ管理機構を持ち、マルチコア、SMT など多様な構成のプロセッサのシミュレーションを行えるものであるが、佐保田による実装は鬼斬式の拡張性を利用しないアド・ホックなものであり、それらの機能を事実上無効化してしまっている。

## 4.2 システムコールの実現

一般にシステムコールはどのレジスタにどの引数を置くかということが命令セット・アーキテクチャ、OS 毎の ABI によって定められている。このレジスタは論理レジスタ番号によって固定されている。しかし、STRAIGHT ではレジスタは命令間の依存を表すエッジとして存在し、論理レジスタ番号によって識別されない。そのため別種の規約が必要となる。本研究では新たに以下のような Linux-like な OS を想定したシステムコールの

規約を定めた。

1. ECALL 命令（システムコール呼出しを行う命令）の 1 つ前にはシステムコール番号、2 つ前からはシステムコールの引数を第一引数から順に置く
2. OS からアプリケーションへの復帰直後に有効なレジスタは戻り値の入ったレジスタのみである。つまり ECALL 命令のあとから ECALL 命令以前の命令で書き込んだレジスタを参照することはできない
3. システムコールは RISC-V(64 bit) Linux と同様のものを用意し、番号もそれに準ずるものとする

このシステムコールの規約は引数の渡し方や順番、生存変数は caller-save であるという点で関数呼び出しのそれと同様である。従ってコンパイラはシステムコール命令を関数呼び出しと同じように処理することができる。

また、STRAIGHT アーキテクチャ特有の問題として、割込み復帰時のレジスタの状態の問題がある。STRAIGHT アーキテクチャも他のアーキテクチャ同様、割込みなどから復帰するときには ERET 命令を用いてアプリケーション・プログラムに復帰する。しかし、ERET 命令が RP を進めてしまうとアプリケーションが想定しているレジスタの距離がずれてしまう。これはシステムコール呼出しの際には、システムコール呼出し直前で生存変数をアプリケーション側が保存するように規約を定めれば問題とはならないが、タイマ割り込みなどアプリケーションが予想できないものの場合に問題となる。そのため、ERET 命令は例外的に RP を進めず、デスティネーション・レジスタを割り当てられない命令とされなければならない。この特殊な扱いによるオーバーヘッドは無視できるほど小さい。何故ならば、特権レベルを変更するような命令はそもそも直列に実行せざるを得ず、またその前後には IO アクセスなどの時間のかかる処理が存在することが一般的であるため、例えば ERET 命令をフェッチしたタイミングでフェッチを止め、先行する命令がすべてリタイアするまで待つて ERET 命令を実行するような単純な実装でも性能に大きな影響はないからである。ただし、ERET についてはアプリケーション・レベルに限ったシミュレーションを行う限りはこの問題を考慮する必要はない。

### 4.3 シミュレーション・モデル

一般的な命令セット・アーキテクチャでは命令の静的な情報、つまり命令デコードによって得られる情報のみで利用するレジスタの番号が定まる。しかし、STRAIGHT アーキテクチャでは命令デコードのみで利用する論理レジスタの番号は決まらず、論理レジスタ番号の値を得るためには動的な値である RP を必要とする。従って命令セット・アーキテクチャ・レベルでのシミュレーションを行う際にも、STRAIGHT アーキテクチャでは命令の実行に先立って RP の参照・更新が発生する。サイクル・レベルでのシミュレーションでは RP によるレジスタの参照がレジスタ読出しに先行する必要がある。そのため、RP を汎用レジスタと同様の扱いとすると、従来のアーキテクチャにないレジスタ参照のパターンが増えるため、大幅なモデルの改変が必要となる。しかも、すべての命令は直前の命令の RP に依存関係が発生し Out-of-Order 実行が不可能になる。

以上の問題を解決するには、RP は通常のレジスタよりも PC に近い、その命令についてのメタ的な情報と見なし、In-Order な処理を行うフロントエンドで扱う特殊なレジスタとするのが望ましい。また、SP は意味論的には通常のレジスタに近いが、これも依存関係の解決の観点から RP 同様フロントエンドで処理されるレジスタとすべきである。

従来のアーキテクチャでは、リカバリでフラッシュされる命令に比例したサイクル数がかかるところを、STRAIGHT アーキテクチャでは 1 サイクルで終わるような軽量なりカバリとする必要がある。

### 4.4 その他の問題

ゲスト・アプリケーションが利用できるメモリ領域が十分な広さではない、シングルコア以外の構成でのシミュレーションが困難であるといった問題は鬼斬式の提供する適切な拡張機構を利用すれば解決される。



## 第 5 章

# 評価

### 5.1 評価環境

#### 5.1.1 シミュレータ

STRAIGHT シミュレータの実装は鬼斬式をベースとして新規に行った。

鬼斬式は多くの命令セット・アーキテクチャを少ない労力でサポートできるように、Linux のエミュレーション機構など命令セット・アーキテクチャに依存しないものは共通化してある。今回行った実装ではなるべくそのような共通化された機構を利用して実装を行った。それによってベンチマーク SPEC CPU [18, 19] が必要とするすべてのシステムコールを動作させることが可能となった。

#### 5.1.2 コンパイラ

コンパイラは LLVM 7 [20] のバックエンドに STRAIGHT アーキテクチャを追加したものを利用した。また、比較対象の命令セット・アーキテクチャは RISC-V [21] のうち RV64IMAFD(= RV64G) を選定した。RISC-V のコンパイラは STRAIGHT と同じバージョンの Clang/LLVM [22] に加え、GCC 7.1.1 の二種類を利用した.. LLVM の ABI は lp64 であり、GCC の ABI は lp64 または lp64d である。アーキテクチャ非依存の LLVM の中間表現である LLVM IR のコンパイルまでは STRAIGHT/RISC-V とともに同じものを利用し、その先はアーキテクチャ依存のバックエンドが処理する。C の標準ライブラリは

musl libc 1.1.16 [23] である。RISC-V と STRAIGHT で条件を揃えるために RISC-V の LLVM を基準として比較を行う。以降、特記のない限り RISC-V と称するのは LLVM を用いたものである。

### 5.1.3 ベンチマーク

ベンチマークプログラムとしては SPEC CPU 2006 に含まれる 473. astar のホットスポットである `wayobj::fill(i32 startx, i32 starty, i32 endx, i32 endy)` メンバ関数、及び RNN (Recurrent Neural Network) の亜種である LSTM (Long Short Term Memory) の LSTM ブロックの出力ゲートの計算、メモリセルのアップデート (以下、便宜的に RNN と呼称する) を対象とした。

473. astar は、`wayobj::fill` メソッドの呼出しまで実機の x86\_64 上で動かし、その時点でのメモリをダンプし、RISC-V/STRAIGHT ではそのダンプを利用して `wayobj::fill` メソッドを実行した。また、実行結果に関しては `wayobj::fill` メソッドの戻り値及び内部のループの条件変数の値を利用して検証を行い、実機の x86\_64、シミュレータの RISC-V/STRAIGHT で結果が一致することを確認した。

RNN についても同様に、実機及びシミュレータで出力結果が一致することを確認した。

### 5.1.4 パラメータ

表 5.1 に 2-way, 4-way, 表 5.2 に 6-way, 8-way のプロセッサのパラメータを示した。論理レジスタ数など特記のない 6-way, 8-way のパラメータは 4-way と同一である。以下単に  $n$ -way と称した場合はこのパラメータを指す。

## 5.2 基本評価

### 5.2.1 473. astar

図 5.1 に 473. astar の実行性能を、RISC-V、及びリカバリペナルティのない RISC-V、STRAIGHT の三種類で計測したものを示した。図 5.2 には実行命令数を示した。

表 5.1 2-way, 4-way のパラメータ

	2-way		4-way	
	RISC-V	STRAIGHT	RISC-V	STRAIGHT
Logical Register Num	64	128	64	128
Fetch Width	2		6	
Front-end latency	8	6	8	6
ROB Capacity	64		224	
Scheduler	2 way, 16 entries		4 way, 96 entries	
Register File	196		420	
LSQ	LD 16 / ST 16		LD 72 / ST 56	
Exec Unit	ALU 2, MUL 1, DIV 1, BC 1, Mem 1, FPADD 1, FPMUL 1, FPDIV 1, FPELEM 1		ALU 4, MUL 1, DIV 1, BC 2, Mem 2 FPADD 1, FPMUL 1, FPDIV 1, FPELEM 1	
Commit Width	3		4	
Branch Predictor	Gshare, Global History 10 bits, 32K entries			
L1I Cache	32 KiB, 8 way, 64 B line, 4 cycle hit latency			
L1D Cache	32 KiB, 8 way, 64 B line, 4 cycle hit latency			
L2 Cache	256 KiB, 8 way, 64 B line, 12 cycle hit latency			
L3 Cache	N/A		2 MiB, 8 way, 64 B line, 42 cycle hit latency	
Main Memory	200 cycle latency			

表 5.2 6-way, 8-way のパラメータ

	6-way		8-way	
	RISC-V	STRAIGHT	RISC-V	STRAIGHT
Fetch Width	6		8	
ROB Capacity	360		480	
Scheduler	6 way, 144 entries		8 way, 192 entries	
Register File	548		676	
LSQ	LD 96 / ST 72		LD 128 / ST 72	
Exec Unit	ALU 8, MUL 2, DIV 2, BC 4, Mem 4 FPADD 2, FPMUL 2, FPDIV 2, FPELEM 2		ALU 8, MUL 2, DIV 2, BC 4, Mem 4 FPADD 2, FPMUL 2, FPDIV 2, FPELEM 2	
Commit Width	6		8	

STRAIGHT は RISC-V に比べて 2-way では 2.5 % 性能が低下しているが、4-way では 12 %, 6-way では 15 %, 8-way では 12 % 性能が向上している。一方実行命令数は、RISC-V で 68 M 命令、STRAIGHT で 120 M 命令で、STRAIGHT は RISC-V に比べて 75 % 増加している。この増分の 52 M 命令のうち、25 M 命令を RMOV 命令が占める。

グラフ探索アルゴリズムである A\* のベンチマークプログラムである 473. astar は、その性質上単純な入れ子になった if 文が繰り返され、非常に合流の多いコントロールフローグラフとなる (図 5.3)。その上、変数の生存期間が長いという特徴がある。RMOV 命令は、その長期間必要となる変数をレジスタ上に長く保持するための中継に用いられている。

しかし、十分なウェイ数のあるプロセッサで STRAIGHT アーキテクチャが良い性能を示しているのは、距離調節用の命令はクリティカルパスには乗っておらず、余った演算器で十分に処理でき、またリカバリペナルティがないことの利得がそれ以上に大きいためである。

なお、他の主要な増加した命令は無条件ジャンプ命令 (6 M 命令)、シフト命令 (13 M) 命令、8 M 命令を NOP 命令である。が、これらは STRAIGHT コンパイラと RISC-V コンパイラでベーシックブロックの並びやアドレス計算に使う命令が異なることに起因して

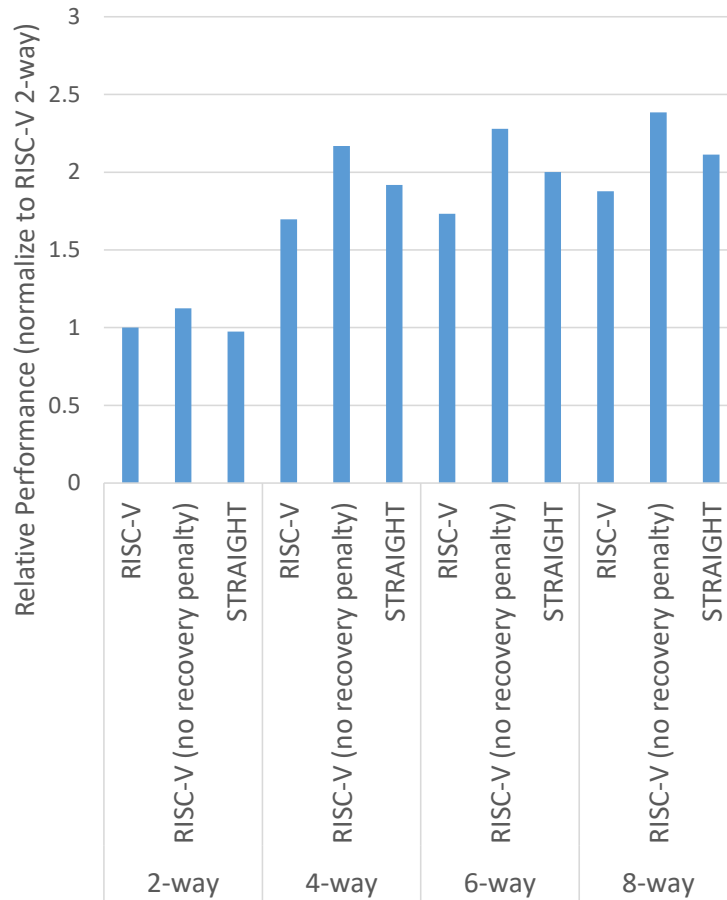


図 5.1 473.astar の実行性能

おり，STRAIGHT アーキテクチャ特有のオーバーヘッドではない。

### 5.2.2 RNN

5.1.4 で示したパラメータを用いて RNN で実行時間の逆数を計測したものを図 5.4 に示した。どのパラメータにおいても STRAIGHT は RISC-V に対して 10 % から 60 % 程度性能を向上させている。RNN の実行命令数は図 5.5 に示した。

STRAIGHT が良い性能を示している原因のほとんどは，RNN が非常に簡素な制御フローグラフを持つことに求められる。RNN のプログラムは各ブロックに対して計算を行うというのを入れ子になっていない定数回の for ループで行うだけのものであり，その中で呼び出されている指数関数や三角関数などもループを含まず，分岐と early return が多

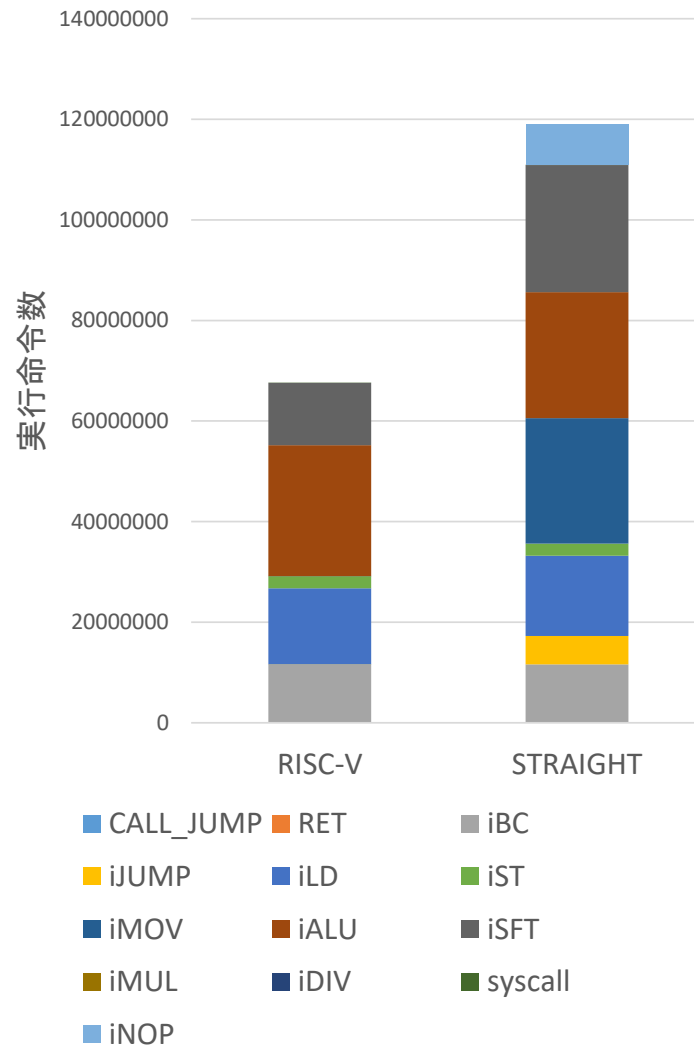


図 5.2 473.astar の実行命令数

く、ほとんど合流のない制御フローを持つ。従って、RNN には距離調節の必要のある制御フローの合流がほとんど含まれないため、RMOV 命令が 3.7 %、NOP 命令が 4.8 % とオーバーヘッドが少なくなっている。

しかし、RISC-V の ABI は STRAIGHT に比べて不利である。RISC-V の ABI である lp64 は、関数呼び出しの際に引数を浮動小数点数レジスタを用いて受け渡すことができない ABI である。STRAIGHT にはそもそも整数レジスタと浮動小数点数レジスタの区別がなく、フラットなレジスタ空間を持つのでそのような制限はない。従って、RISC-V はその分だけコピー命令が発生し不利である。RISC-V でも浮動小数点数レジスタを引数とし

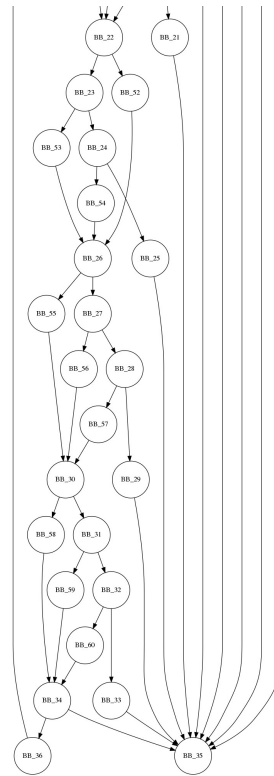


図 5.3 473.astar の制御フローグラフの一部

て利用できる `lp64d` での比較を行うことが望ましい．`lp64` と `lp64d` の影響を見積もるために，`lp64` のみにしか対応していない LLVM に代えて GCC を用いた場合の実行時間も図 5.4, 5.5 に併記した．GCC(`lp64`) から GCC(`lp64d`) へコンパイラを変更した時の性能向上幅は 4 % から 7 % であり，ウェイ数が多くなるほど差は小さくなっている．そのため，`lp64` と `lp64d` の差によって必要となる `fmv` 命令はクリティカルパス上になく，ABI が不利であることによる性能低下を差し引いても STRAIGHT を上回る性能とはなりえない．

他に、RISC-V では実数定数をレジスタ上に配置するのに浮動小数点数レジスタへのロード命令を使うか、整数演算命令を利用して整数レジスタ上に一旦定数を置き、それを浮動小数点数レジスタへコピーするかのいずれかの方法をとっている。しかし、STRAIGHT ではすべての定数を整数演算命令を利用してレジスタ上に作っており、ロード命令も浮動小数点数・整数の変換命令も利用しておらず、少ない命令数・レイテンシで実数定数を得ることができている。これは、RISC-V が整数レジスタと浮動小数点数レジスタを分けて持ち、STRAIGHT はそのような分け方をしていないことに起因する。つまり、レイテン

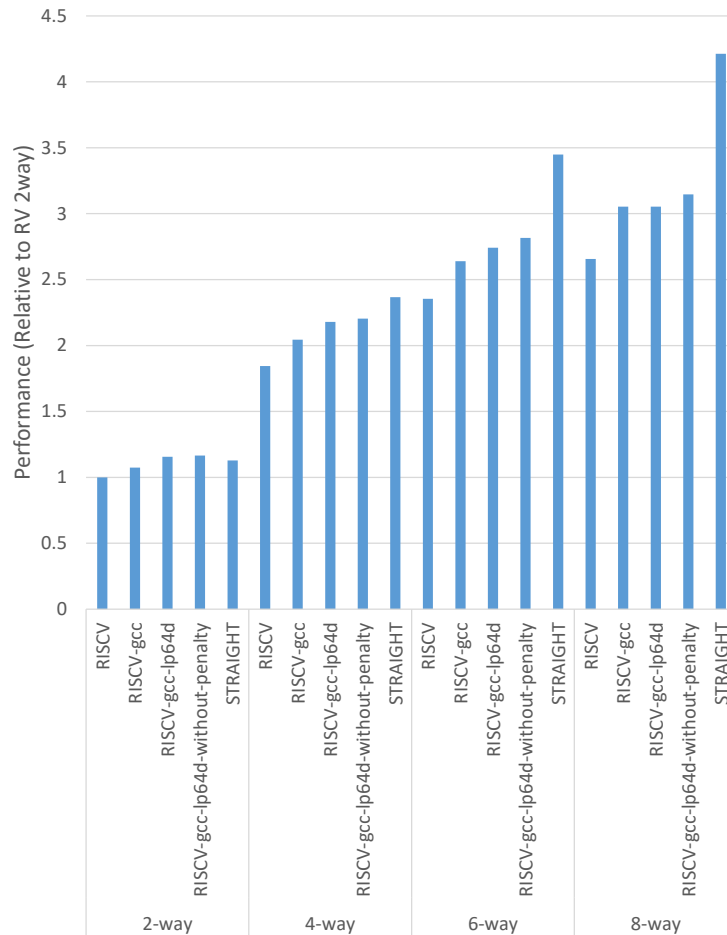


図 5.4 RNN の実行性能

シの短縮を目指すならば整数レジスタと浮動小数点数レジスタを分けずにフラットなレジスタとして保持するほうが良い。しかし、レジスタ・ファイルを構成する SRAM の面積はポート数の自乗に比例するので、仮に総容量を二倍にしてもレジスタを二分割してポート数を半分ずつに分けたほうが回路の観点からは有利である。この性能と回路のトレードオフは今まで検討されていなかったが、検討する価値がある。

### 5.3 命令ウィンドウの拡張について

前節では、473. astar と RNN が STRAIGHT アーキテクチャにとって対照的な性質を持つことを明らかにした。STRAIGHT アーキテクチャでは、命令ウィンドウで管理すべき



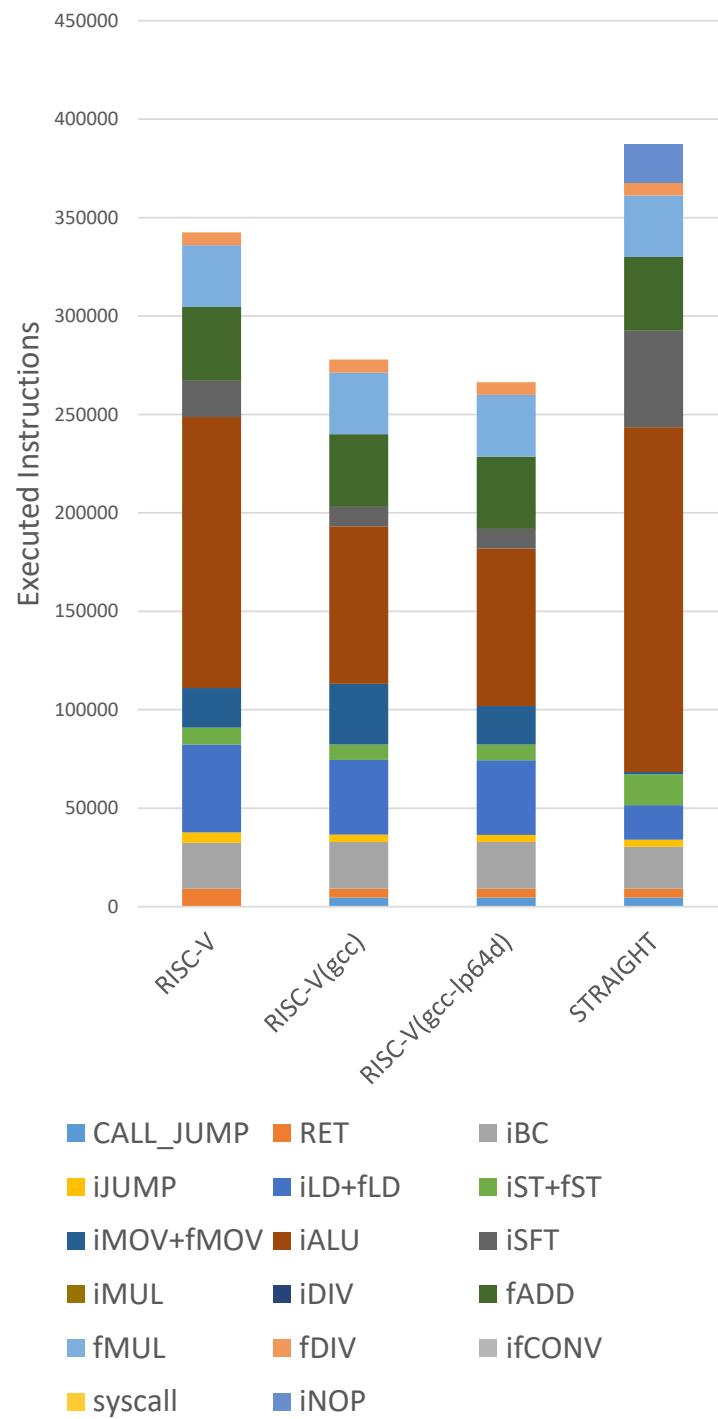


図 5.5 RNN の実行命令数

物理レジスタの情報が削減でき、分離リオーダバッファを利用することで特殊レジスタの管理も負担にならないことから、命令ウィンドウは拡張が容易な要素である。そこで本節では 4-way のプロセッサを基準に命令ウィンドウのエントリ数を変化させた時の性能評価を行う。

まず、図 5.6 に命令ウィンドウのエントリ数を変化させたときの実行性能を示した。命令ウィンドウのエントリ数は 64 (2-way と同じ)、224 (4-way)、360 (6-way)、480 (8-way)、600 と変化させた。473.astar では、ほとんどのエントリ数を広げても性能が向上していない。一方で RNN では命令ウィンドウのエントリ数 224 から 640 で 1.4 倍の性能向上を示しており、360 までも 1.3 倍の性能向上を示している。命令ウィンドウはプロセッサ全体の消費電力のうち 7 % を占める [24] とされており、命令ウィンドウの面積はその容量に比例することを考慮すると、8 % のハードウェア追加で 30 %、13 % で 40 % の性能向上が得られる。従って STRAIGHT アーキテクチャにおいて、命令ウィンドウへの資源の投入は効率の良い投資となる。

STRAIGHT で性能が向上しているものの、RISC-V ではあまり性能向上が見られない RNN に注目すると、STRAIGHT では基準の 4-way の実行サイクル数は 240k サイクルであるが、そのうち、物理レジスタが足りずにフロントエンドがストールするという事象が 55k サイクル発生していた。そこで命令ウィンドウのエントリ数を 64, 224, 360, 480, 600 と変えるとともに、レジスタ・ファイルの容量を変化させたときの RNN の実行性能を図 5.7 に示した。整数レジスタと浮動少数点数レジスタの詳細なパラメータは表 5.3 に示した。命令ウィンドウ幅を拡張しただけでは 10 % の性能向上にとどまっていたものが、整数レジスタを同時に増加させることで最大で 20 %、更に浮動小数点数レジスタを同時に増加させることで最大で 24 % の性能向上幅が得られるようになったことがわかる。これは、物理レジスタ数が少ないことがボトルネックとなっていたのみならず、整数レジスタ・浮動小数点数レジスタのどちらか一方のみが足りないだけでストールが発生して ILP の抽出が妨げられるということを示す。このような問題が発生しないことも STRAIGHT のようなフラットなレジスタを持つ利点である。

しかし、一方でフラットなレジスタは、非集中化による面積軽減と相性が悪い。命令ウィンドウを整数演算命令、浮動小数点数演算命令、メモリアクセス命令などと分類し分

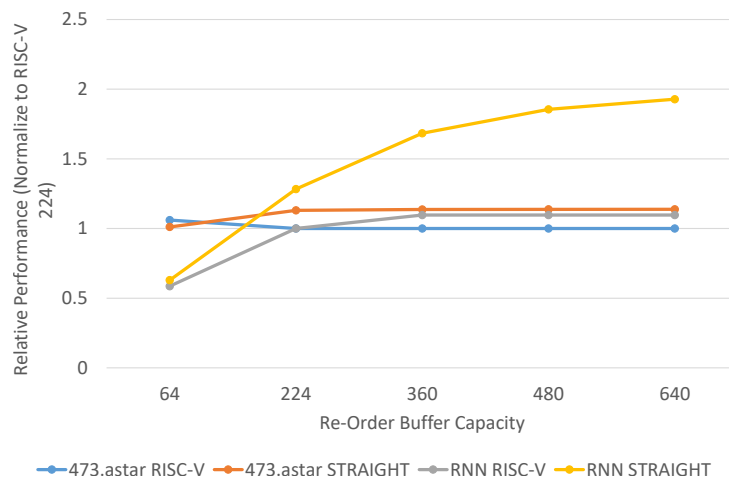


図 5.6 命令ウィンドウのエントリ数を変えた時の実行性能

表 5.3 レジスタ・ファイルの容量の変化

	命令ウィンドウの容量				
(int, fp)	64	224	360	480	640
Normal	(180,168)	(180,168)	(180,168)	(180,168)	(180,168)
RegFile	(80,80)	(180,168)	(232,220)	(276,264)	(320,320)
FPReg	(180,80)	(180,168)	(180,220)	(180,264)	(180,320)
IntReg	(80,168)	(180,168)	(232,168)	(276,168)	(320,168)

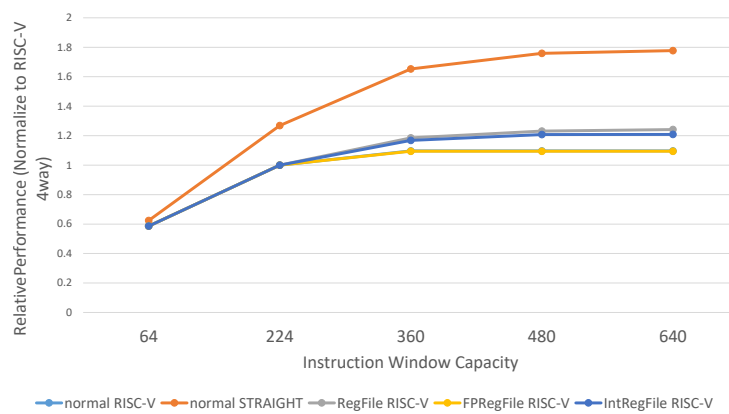


図 5.7 命令ウィンドウのエントリ数に加えてレジスタファイルの容量を変化させた時 (表 5.3) の RNN の実行性能

割を行うと、異なる命令ウィンドウにある情報を用いる命令を実行するには命令ウィンドウ間の通信によってレイテンシが延びる。また、レジスタ・ファイルもポート数の自乗に比例する面積を必要とするため、例えばポート数を半分にしたレジスタ・ファイルを 2 つ用意し、総容量を 2 倍としたとしても、総面積は元のレジスタ・ファイルより小さくなる。レジスタ・ファイルの面積削減技術としてはマルチバンクやキャッシングといったものもあるが、いずれも STRAIGHT アーキテクチャへの適用については十分な検討がなされているとは言い難い。STRAIGHT アーキテクチャは、従来のアーキテクチャでは隠蔽されていた物理レジスタをある程度まで露出させたものと捉えられるため、STRAIGHT 命令セット・アーキテクチャとレジスタ・ファイルの構成は緊密な関係を持つ。それ故、レジスタ・ファイルの構成に関しては今後慎重に検討される必要がある。

## 第 6 章

# 関連研究

命令間の依存関係を明示した形式でプログラムを記述するアーキテクチャとしてデータフローアーキテクチャが古くから研究されている [25]. データフローアーキテクチャはプログラム・オーダの概念を持たず、依存する計算結果が揃ったら次の計算が開始されるというデータ主導の計算モデルのアーキテクチャである. 近年のデータフローアーキテクチャに関する研究には Nowatzki らによるもの [26,27] があり、アクセラレータとしての可能性が模索されている.

また、プログラム・オーダを定義する制御駆動アーキテクチャでありながら、データフローアーキテクチャのように命令間のデータの受け渡しを「場所」を指定せずに行うアーキテクチャとして、五島らの Dualflow アーキテクチャがある [28]. これは命令の計算結果のコンシューマを距離で指定する命令セット・アーキテクチャであり、コンシューマ命令の指定を最大 2 つまでしかできないなどコンパイラへの制限が STRAIGHT アーキテクチャより強い. Dualflow アーキテクチャの命令配置に関する強い制限を緩和するための逆 Dualflow アーキテクチャ [29] というものも考えられている. 逆 Dualflow アーキテクチャは命令セット・アーキテクチャには手を加えず、レジスタ・リネーミングの代わりにプロデューサ命令を距離で指定する内部表現へと変換する.

単純なコアで Out-of-Order な実行を試みるものとしては Padmanabha らの Mirage Cores [30] がある. これは、Composite Cores [31], big.LITTLE [32] や FXA [33] に代表される Out-of-Order なコアと In-Order なコアを併置し、適宜使い分けることで、性能と

電力を両立させようとする試みの流れの上であり、Out-of-Order なコアのスケジューリングの結果を利用することで、省電力な In-Order なコアでも十分に良い性能を得ることができるというものである。

STRAIGHT のようなコンパイラの支援を前提としたアーキテクチャとしては、Idempotent Architecture [34] や Duong らのもの [35] がある。Idempotent Architecture は、コンパイラがプログラムを再実行による冪等性が保たれる範囲に事前に分割しておき、リカバリを状態の巻き戻しではなく、命令の再実行によって行うアーキテクチャである。これによって Out-of-Order にコミットを行うことが可能となる。また、Duong らのアーキテクチャでは、プログラムはコンパイラによって複数のブロックに分割され、プロセッサが OoO モードで動作する時は、ブロック内では InO にコミットされることが保証されるが、ブロック間では InO にコミットされることが保証されない。これによって命令が命令ウィンドウを専有する時間を短縮し、IPC を向上させることができる。

## 第 7 章

# おわりに

本研究では，既存の **STRAIGHT** シミュレータの問題点を明らかにし，大規模なベンチマークによる評価を行うのに必要とされるシステムコールなどの詳細を新たに定義し，新たなシミュレータを開発した．さらに，それを用いて，**RNN** 及び **SPEC CPU 2006** の **473. astar** による評価を行い，**STRAIGHT** アーキテクチャが実アプリケーションにおいても良い性能を示すことを明らかにした．加えて，命令ウィンドウを拡張した際の性能も評価し，**STRAIGHT** では命令ウィンドウの拡張が従来のアーキテクチャに比べて性能向上により効果的であることを確認した．また，**STRAIGHT** アーキテクチャの持つ，整数レジスタと浮動小数点数レジスタを区別しない形でのレジスタが性能に良い影響を与えることも確認した．

今後の研究の方向としては，まずより多くの **SPEC CPU** ベンチマークのようなプログラムによって **STRAIGHT** アーキテクチャを精確に評価することがあげられる．本研究で開発されたシミュレータはそのような評価に十分に堪えうるものである．更に，**STRAIGHT** アーキテクチャには **In Order** スケジューラや投機メモリフォワードリング [36] のような未検証の要素技術の評価も行うことが可能になった．また，レジスタ・ファイルの構成（とそれに関する **ISA** の改訂）の検討も必要である．

## 参考文献

- [1] H. Esmailzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. *IEEE Micro*, Vol. 32, No. 3, pp. 122–134, 2012.
- [2] Hidetsugu Irie, Toru Koizumi, Akifumi Fukuda, Seiya Akaki, Satoshi Nakae, Yutaro Bessho, Ryota Shioya, Takahiro Notsu, Katsuhiko Yoda, Teruo Ishihara, and Shuichi Sakai. STRAIGHT: Hazardless Processor Architecture Without Register Renaming. *Int. Symp. on Microarchitecture*, 2018.
- [3] Hidetsugu IRIE, Daisuke FUJIWARA, Kazuki MAJIMA, Tsutomu YOSHINAGA. STRAIGHT: Realizing a lightweight large instruction window by using eventually consistent distributed registers. In *Networking and Computing (ICNC), 2012 Third International Conference*, pp. 336–342, 2012.
- [4] 入江英嗣, 山中崇弘, 佐保田誠, 吉見真聡, 吉永努. もし ILP プロセッサのレジスタファイルが分散キーバリューストアになったら. 情報処理学会研究報告. 計算機アーキテクチャ研究会報告 2013-ARC-206(5), pp. 1–10, 2013.
- [5] Hidetsugu Irie, Daisuke Fujiwara, Kazuki Majima, and Tsutomu Yoshinaga. STRAIGHT: Realizing a Lightweight Large Instruction Window by using Eventually Consistent Distributed Registers. In *Int. Workshop on Challenges on Massively Parallel Processors*, pp. 336–342, 2012.
- [6] Reinhold P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Commun. ACM*, Vol. 27, No. 10, pp. 1013–1030, October 1984.
- [7] Cpu benchmark - mcu benchmark - coremark - eembc embedded microprocessor benchmark consortium. <https://www.eembc.org/coremark/>.



- [8] 別所祐太朗. リカバリペナルティの STRAIGHT アーキテクチャによる削減, March 2018.
- [9] Kenneth C. Yeager. The mips r10000 superscalar microprocessor. *IEEE Micro*, Vol. 16, No. 2, pp. 28–40, April 1996.
- [10] R. E. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, Vol. 19, No. 2, pp. 24–36, March 1999.
- [11] Dave Sager, Desktop Platforms Group, and Intel Corp. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, Vol. 1, p. 2001, 2001.
- [12] Gwennap L. Intel’s p6 uses decoupled superscalar design. Technical Report 2, Microprocessor Report, aug 1995.
- [13] Chetana N. Keltcher, Kevin J. McGrath, Ardsher Ahmed, and Pat Conway. The amd opteron processor for multiprocessor servers. *IEEE Micro*, Vol. 23, No. 2, pp. 66–76, March 2003.
- [14] 赤木晟也. STRAIGHT プロセッサの実装と評価, March 2018.
- [15] 赤木晟也, 入江英嗣, 坂井修一. STRAIGHT アーキテクチャの命令形式を利用したスケジューラの提案. 学生・若手研究会 IEICE-CPSY2017-68, 第 IEICE-117 巻, pp. 43–44, 2017.
- [16] 佐保田誠. プロセッサアーキテクチャ「STRAIGHT」のシミュレータ設計と評価. Master’s thesis, 電気通信大学, March 2015.
- [17] Github - onikiri/onikiri2. <https://github.com/onikiri/onikiri2>.
- [18] Spec cpu 2006. <https://www.spec.org/cpu2006/>.
- [19] <https://www.spec.org/cpu2017/>.
- [20] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO ’04, pp. 75–, Washington, DC, USA, 2004. IEEE Computer Society.
- [21] Andrew Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2016.

- [22] Risc-v llvm. <https://github.com/sifive/riscv-llvm>.
- [23] musl libc. <https://www.musl-libc.org/>.
- [24] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: speculation control for energy reduction. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, pp. 132–141, July 1998.
- [25] Arthur H. Veen. Dataflow machine architecture. *ACM Comput. Surv.*, Vol. 18, No. 4, pp. 365–396, December 1986.
- [26] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. Exploring the potential of heterogeneous von neumann/dataflow execution models. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture, ISCA '15*, pp. 298–310, New York, NY, USA, 2015. ACM.
- [27] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam. Stream-dataflow acceleration. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 416–429, June 2017.
- [28] 正裕五島, グェンハイハー, 眞一郎森, 眞治富田. Dual - flow : 制御駆動とデータ駆動を融合したプロセッサ・アーキテクチャ. Technical Report 70(1998-ARC-130), 京都大学大学院情報学研究科, 京都大学工学部情報工学科, 京都大学大学院情報学研究科, 京都大学大学院情報学研究科, aug 1998.
- [29] 一林宏憲, 塩谷亮太, 入江英嗣, 五島正裕, 坂井修一. 逆 Dualflow アーキテクチャ. 情報処理学会論文誌コンピューティングシステム (ACS) , Vol. 1, No. 2, pp. 22–33, aug 2008.
- [30] Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. Mirage cores: The illusion of many out-of-order cores using in-order hardware. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-50 '17*, pp. 745–758, New York, NY, USA, 2017. ACM.
- [31] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M. Sleiman, Ronald Dreslinski, Thomas F. Wenisch, and Scott Mahlke. Composite cores: Pushing heterogeneity into a core. In *Proceedings of the 2012 45th Annual IEEE/ACM International*

- Symposium on Microarchitecture*, MICRO-45, pp. 317–328, Washington, DC, USA, 2012. IEEE Computer Society.
- [32] ARM Ltd. Technologies — big.little – arm developer. <https://developer.arm.com/technologies/big-little> 2017/11/08 閲覧.
- [33] Ryota Shioya and Hideki Ando. Energy efficiency improvement of renamed trace cache through the reduction of dependent path length. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pp. 416–423, Oct 2014.
- [34] Marc de Kruijf and Karthikeyan Sankaralingam. Idempotent processor architecture. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pp. 140–151, New York, NY, USA, 2011. ACM.
- [35] G. B. Bell and M. H. Lipasti. Deconstructing commit. In *IEEE International Symposium on - ISPASS Performance Analysis of Systems and Software, 2004*, pp. 68–77, 2004.
- [36] 酒井一憲, 中江哲史, 入江英嗣, 坂井修一. STRAIGHT における投機メモリフォワードリングの実装の検討. 情報処理学会研究報告. 計算機アーキテクチャ研究会報告 2017-ARC-224, No. 34, pp. 1–6, 2017.

# 発表文献

## ■口頭発表（査読なし）

1. 福田晃史, 中江哲史, 入江英嗣, 坂井修一 *STRAIGHT* アーキテクチャにおけるループ内ロード命令の削減手法, 情報処理学会研究報告, 計算機アーキテクチャ研究会報告 2017-ARC-225(3), pp.1-6, 2017 年.
2. 小泉 透, 中江 哲史, 福田 晃史, 入江 英嗣, 坂井 修一: *STRAIGHT* 向けコンパイラによる冗長転送命令の削減, 情報処理学会研究報告, 2018-ARC-231, pp. 1-10, 2018 年
3. 福田 晃史, 小泉 透, 門本 淳一郎, 中江 哲史, 入江 英嗣, 坂井 修一: *STRAIGHT* アーキテクチャの評価環境の実装, 信学技報, vol. 118, no. 375, CPSY2018-52, pp. 43-47, 2018 年 12 月.

## ■国際会議会議録（査読あり）

1. Hidetsugu Irie, Toru Koizumi, Akifumi Fukuda, Seiya Akaki, Satoshi Nakae, Yutaro Bessho, Ryota Shioya, Takahiro Notsu, Katsuhiko Yoda, Teruo Ishihara, Shuichi Sakai: “*STRAIGHT: Hazardless Processor Architecture Without Register Renaming*”, Int. Symp. on Microarchitecture, pp.121–133, Oct., 2018.
2. Toru Koizumi, Satoshi Nakae, Akifumi Fukuda, Hidetsugu Irie, Shuichi Sakai: “*Reduction of instruction increase overhead by STRAIGHT compier*”, Int. Workshop on Computer Systems and Architectures, pp.92–98, Nov., 2018

■国際ポスター発表（査読あり）

1. Junichiro Kadomoto, Toru Koizumi, Akifumi Fukuda, Reoma Matsuo, Susumu Mashimo, Akifumi Fujita, Ryota Shioya, Hidetsugu Irie, Shuichi Sakai: An Area-Efficient Out-of-Order Soft-Core Processor Without Register Renaming, International Conference on Field Programmable Technology (FPT), pp. 377–380, Dec. 2018

# 謝辞

本論文の執筆の過程では多くの方に支援をいただきました。

指導教官である坂井修一教授からは、ミーティングを通じて研究の大きな流れや成果の主張の仕方などをご指導いただいた他、研究室の設備など研究生活に必要なものを整備していただきました。入江英嗣准教授には、細かい研究方針の相談、論文の添削など直接的な関わり以外にも、論文執筆を行う姿勢などから様々なものを学ばせていただきました。塩谷亮太准教授には、STRAIGHT プロジェクトと直接関わる以前からも鬼斬弐の実装の細部について親身になって助言をしていただきました。また、秘書の八木原晴水さんと赤羽彩子さんには、事務処理や出張の際には非常にお世話になりました。STRAIGHT プロジェクトに関わった皆さま、とりわけ小泉透氏には、研究のあらゆる場面でサポートをいただきました。その他の研究室のメンバーにも日々の雑談や議論を通じて研究生活を楽しいものにしていただきました。ここに深謝します。

なお、本論文の研究は、一部、民間等共同研究（株式会社富士通研究所）「Deep Learning 向け Straight アーキテクチャの研究」によります。