Vol.2018-ARC-231 No.2
2018/6/14

STRAIGHT向けコンパイラによる冗長転送命令の削減

小泉 透[1,a)]　中江 哲史[1]　福田 晃史[1]　入江 英嗣[1]　坂井 修一[1]

概要：シングルスレッド性能を高める手法であるアウトオブオーダー実行を効率よく行うためには，偽の依存を取り除くことが効果的である．これをハードウェアで行うレジスタリネーミングはその複雑度からプロセッサのボトルネックの一つとなっている．偽の依存をコンパイラが除去するアプローチのうちの一つに，STRAIGHT アーキテクチャが挙げられる．STRAIGHT アーキテクチャは，何命令前の結果を使うといった形でオペランドを指定し，レジスタの上書きを行わないため原理的に偽の依存が発生しない．一方そのコンパイラは実行経路に依存せず一定の距離でオペランドを指定するという制約を満たしたコードを生成する必要がある．これを実現する基本アルゴリズムはすでに明らかになっているものの，制約を満たすために多くのレジスタ間転送命令を追加する必要があり，STRAIGHT の潜在性能を引き出す妨げとなっている．本論文ではこの転送命令の追加を削減するアルゴリズムを示し，LLVM を用いて提案手法を実装したコンパイラを作成した．ベンチマークとして Livermore loops を用いた評価では，従来の基本アルゴリズムに比べ転送命令の数を 50%削減し，実行性能は約 12%向上した．

1. はじめに

ムーアの法則に示される使用可能なトランジスタの指数的増大は，プロセッサ上に複数のコアを実装することを可能にした．一方，デナードスケーリングが破綻したことで，電力の制限からダイ上のトランジスタ全てを同時に駆動できない，ダークシリコンと呼ばれる問題が生じており電力効率を考慮する必要性が生じた [1]．そこで近年のプロセッサでは，データレベルの並列性（data level parallelism, DLP）・スレッドレベルの並列性（thread level parallelism, TLP）・命令レベルの並列性（instruction level parallelism, ILP）の利用に適した多種多様なコアを搭載した，ヘテロジニアス構成にすることによって総合的なバランスで性能を向上させている [2][3]．

ILP はメモリアクセスやコントロールフローが規則的ではないが動的に予測できるようなワークロードを高速化させることができる．このようなコードは DLP や TLP に比べタスク全体のボトルネックとなりがちであり，プロセッサが高性能になるほど高い ILP 性能を持つコアが有用になる [4]．

DLP や TLP の抽出はプログラマやコンパイラの支援が有効であり，ハードウェアの軽量化につながっている．一方，ILP を抽出するためにはハードウェアの複雑さが指摘されているアウトオブオーダー実行コアの利用が不可欠となっている．

アウトオブオーダー実行コアでは本来の計算以外の資源管理に多くの電力を割く必要があり，電力効率が悪い．この資源管理のうち，電力上の大きなボトルネックとなっているレジスタリネーミングは偽の依存を取り除くために行われている．したがってコンパイラが偽の依存を含まないコードを生成すればレジスタリネーミングは不要になる．この点に着目し，アウトオブオーダー実行することを前提としつつもレジスタリネーミングを排し電力効率の向上を図るアーキテクチャとして STRAIGHT アーキテクチャがある [5][6]．

STRAIGHT アーキテクチャは，何命令前の結果を使うといった形式でオペランドを指定する命令セットを持つ．この形式はレジスタの上書きを行わないため，偽の依存が発生せずレジスタリネーミングが不要になる．レジスタリネーミング回路が不要になることで，フロントエンドのハードウェア量を削減し，スケーラビリティと消費電力が改善される [7]．また、論理レジスタと物理レジスタのマッピングを保持する必要がないため，分岐予測ミス等の例外からの復帰も高速化する．一方コンパイラはプログラムに分岐・合流が存在する場合でも何命令前という距離でオペランドを指定するという制約を満たすコードを生成する必要がある．

我々の先行研究 [8] では，合流地点の直前の基本ブロックの末尾に fixed 領域と呼ぶ部分を設け，以降も参照されうる値をそこに集約するという手法により，この制約を満

[1] 東京大学 大学院情報理工学系研究科
[a)] koizumi@mtl.t.u-tokyo.ac.jp

ⓒ 2018 Information Processing Society of Japan

たすコードを生成するコンパイラアルゴリズムを構成した．しかし，この手法には単に距離を調整するためだけのレジスタ間転送命令が多く出力されるという問題点が存在する．

本研究は，このような転送命令の追加を防ぎつつ制約を満たすコードを生成するアルゴリズムを構成し，STRAIGHT アーキテクチャの潜在的な性能を引き出すことを目的とする．提案手法を LLVM[9] を用いたコンパイラに実装し，Livermore loops[10] をベンチマークとしてコンパイルし，プロセッサシミュレータ「鬼斬弐」[11] を用いて性能を評価した．その結果，従来の手法に対していずれのベンチマークコードでも転送命令を削減し，平均で 50%削減した．また，実行性能は幾何平均で 12%向上することを確認した．

本研究の貢献は以下の通りである．

- STRAIGHT アーキテクチャにおいて，制約を満たすための転送命令の追加が不可欠な場合がいかなる時に発生するのかを明らかにした．
- 転送命令の追加が不可欠な場合，具体的にどのような位置に追加すると制約を満たせるかを決定するアルゴリズムを示した．
- 命令の順序を可能な範囲で入れ替えることにより，距離で指定するという制約を満たすコードを，冗長な転送命令の追加を防ぎつつ生成する方法を示した．
- 以上により，STRAIGHT アーキテクチャの潜在性能を引き出すコンパイラアルゴリズムを構成した．

以下，第 2 章では STRAIGHT アーキテクチャの概要について説明し，第 3 章では既存のコンパイラアルゴリズムを確認する．第 4 章では本研究で提案する，冗長な命令の追加を極力行わずに STRAIGHT 機械語コードを生成するアルゴリズムを示す．第 5 章では提案手法により作成された機械語コードについて評価を行う．第 6 章では他の研究との関連性について議論を行う．第 7 章では今後の課題について記す．

2. STRAIGHT アーキテクチャ

STRAIGHT コードはオペランドとしてレジスタの名前でなく，何命令前の結果を使うといった形式でオペランドを指定する特徴を持つ．命令セットはレジスタの指定方法が特殊であることを除いては RISC 系の標準的な命令をサポートしている．例として STRAIGHT アセンブリで書かれたフィボナッチ数列を計算するプログラムを図 1 に示す．ここで，`$zero` は常に 0 が得られる零レジスタを意味する．また，`[1]` は直前の命令の結果を，`[2]` は 2 つ前の命令の結果をソースオペランドとして参照することを意味する．このようなオペランド指定形式を持つため，デスティネーションレジスタをアセンブリや命令語中に記述する必要はない．

```
ADDi $zero, 1  # 1
ADDi $zero, 1  # 1
ADD  [2], [1]  # 1+1→2
ADD  [2], [1]  # 1+2→3
ADD  [2], [1]  # 2+3→5
```

図 1 STRAIGHT アセンブリで書かれたフィボナッチ数列を計算するコード

レジスタ上書きによる偽の依存を発生させないため，各命令の結果を書き込むレジスタはユニークである必要がある．これを実現するため，STRAIGHT プロセッサの内部には一命令ごとに 1 ずつ増加するレジスタポインタ（RP）が存在し，その値が命令の書き込むべき物理レジスタ番号となる．この時，命令語中で何命令前，と指定されているソースオペランドが格納されている物理レジスタ番号は単純な引き算により求めることができる．`NOP` 命令の場合バックエンドでは何もしないが，RP は通常通り 1 増加することとする．これは後述の命令間距離の調整に用いることができる．

現実的な実装を考えると，物理レジスタの数には限りがあり，いつまでもユニークなレジスタを供給できるわけではない．ここで，命令語におけるソースオペランド指定部分が n bit だとすると，2^n 命令より前の結果は参照することができない．つまり何命令前という形式でオペランドを指定する形式を採用することで，レジスタの寿命を静的に確定させることが可能であり，フリーリストでの管理なしに寿命の切れたレジスタを確保することができる．よって，インフライトな命令の最大数を m として，RP が $2^n + m$ まで増加したとき 0 にラップアラウンドして物理レジスタを使いまわしても偽の依存が生まれることはない．

STRAIGHT アーキテクチャ特有の命令として，`SPADDi` 命令と `RPINC` 命令が挙げられる．`SPADDi` 命令は，STRAIGHT アーキテクチャ唯一の上書き可能なレジスタであるスタックポインタ（SP）を操作する命令である．この命令のデスティネーションレジスタには変更後の SP の値が書き込まれるため，そのレジスタをオペランドにとることでスタックへのアクセスが可能になる．`RPINC` 命令は，正整数即値を取り，その値だけ RP を加算する命令であり，`NOP` 命令同様バックエンドでは何もしない．つまり即値の個数分だけの連続した `NOP` 命令と同等の働きをする．命令間の距離を調整するために `NOP` 命令を連続して配置しなければならない場合に，`RPINC` 命令を使うとプログラムの肥大化を防ぐことができる．

ⓒ 2018 Information Processing Society of Japan

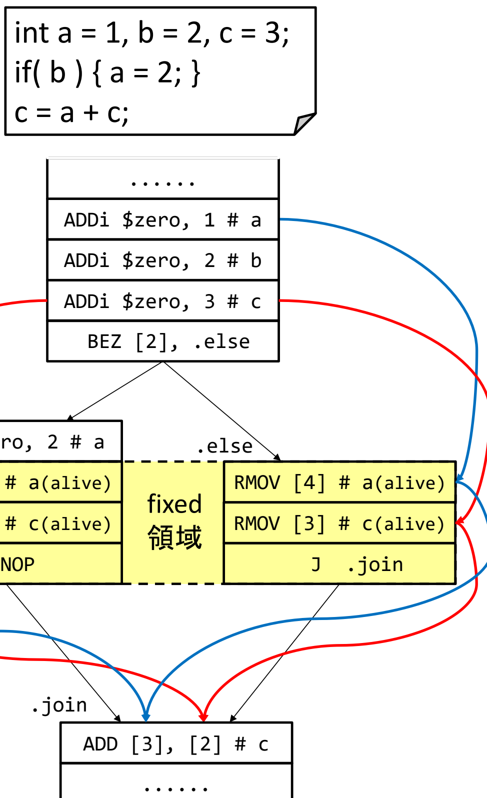

図 2 fixed 領域の作り方の例

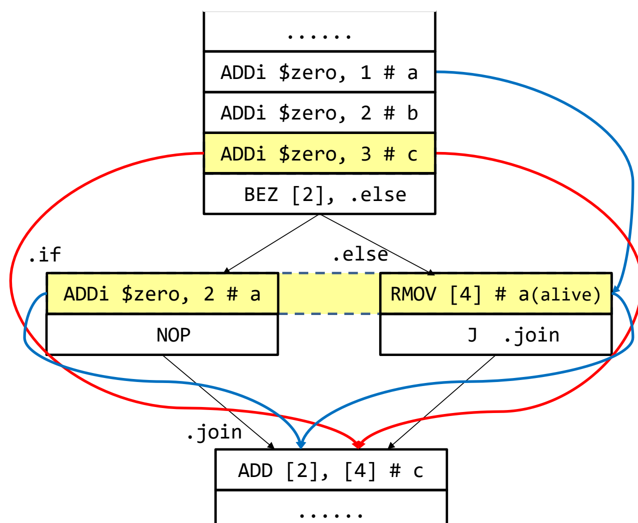

図 3 自由な位置で距離を合わせた例

3. 既存の STRAIGHT コンパイラアルゴリズム

プログラムに複数の実行経路が存在する場合，それぞれの実行経路で実行される命令数やその順序は一般に異なるため，ソースオペランドレジスタまでの距離を一意に定めるには工夫が必要である．既存の STRAIGHT アーキテクチャ向けコンパイラアルゴリズムでは，**図 2** のように複数の実行経路が合流する地点の直前に fixed 領域と呼ばれる部分を作り，その中に生存しているレジスタの値をコピーするレジスタ間転送命令（RMOV 命令）を追加することにより，いずれの実行経路でも距離が等しくならなければいけないという制約を満たすコードを生成している．その具体的なアルゴリズムは，以下のようになる．

(1) phi 関数の追加

STRAIGHT アーキテクチャのレジスタに再書き込みを行わないという特徴は，コンパイラ最適化を行う際の解析に適した，静的単一代入（static single assignment, SSA）形式と共通する．SSA 形式の場合，直前に通過した基本ブロックに応じて参照する仮想レジスタが変化する phi 関数が合流点の基本ブロックの先頭に追加される [12]．この phi 関数は実行経路の違いにより参照する仮想レジスタの可能性が複数ある場合のみ追加されるが，STRAIGHT アーキテクチャの場合，同一の仮想レジスタを参照する場合でも一般には距離の調整が必要である．そこで，後者の場合も phi 関数を追加し，fixed 領域を作るときの参照情報とする．

(2) fixed 領域の作成

合流地点の直前がフォールスルーになっている場合，NOP 命令を追加し，合流地点の直前が分岐命令か NOP 命令になるようにし，ここを fixed 領域とする．これによりどの実行経路でも fixed 領域内の命令数は同一になる．その後全ての phi 関数について実行経路ごとに，合流の直前の基本ブロックに作成した fixed 領域の上方に phi 関数のオペランドをコピーする RMOV 命令を追加する．このようにすることで，どの実行経路であっても同じ相対距離の位置に RMOV 命令が存在することになり，phi 関数の結果を参照している命令を，追加した RMOV 命令を参照するように変更できる．

このアルゴリズムを用いた場合，合流点で生存している値一つにつき，各経路に一つの RMOV 命令が常に追加される．この方法では無駄な RMOV 命令が追加されることも多いが，RMOV 命令の追加が不要であるかの判定は自明ではない．常にコンパイルできることを保証するため，常に RMOV 命令を追加するという確実な方法をとっている．

4. 冗長な転送命令の削減

4.1 提案手法

図 3 は，図 2 に示したコードの距離合わせを自由な位置で行うことにより RMOV 命令を削減した例である．このコードでも，合流点以降も参照される値はどの実行経路を通っても同じ相対距離の位置の命令で生成されるという制約を満たしている．しかし，その位置は必ずしも fixed 領域のように合流する部分の直前の基本ブロックの末尾に固まっているわけではない．また，同じ相対距離の位置になる命令は必ずしも RMOV 命令ではない．

このように，どの実行経路でも距離を一定にしなければいけないという STRAIGHT アーキテクチャ特有の制約を満たすために使える自由度は，従来のコンパイラアルゴリ

ズムが考慮しているもののほかにも多く存在する．しかし，自由度が不足し RMOV 命令の追加が必要不可欠な場合が存在する．そこで事前に必要不可欠な RMOV 命令だけを追加し，必要十分なだけ自由度を上げてから命令の順序の入れ替えや NOP 命令の追加を行うことで，不要な RMOV 命令を追加せずに距離を一意に定めることが出来ると考えられる．

まずは必要不可欠な RMOV 命令の種類について概観し，その後 RMOV 命令の追加が必要不可欠な部分を判別するアルゴリズムを示す．そしてそれを踏まえた上で冗長な RMOV 命令の追加を防ぎつつ距離をそろえる手法を提案する．

4.2 必要不可欠な転送命令の種類

4.2.1 値のコピーが必要な場合（タイプ C）

例として図 4 のように，一つの基本ブロックの複数の phi 関数から参照される命令が存在する場合を考える．この場合 (1) と (2)，(1) と (3) の距離を同時に合わせることは，(2) と (3) が同一の基本ブロックにある異なる命令であることから不可能である．このようなことが発生するのは，左の実行経路では二つの異なるレジスタに割り付ける必要のある値を一つのレジスタで持とうとしていることが原因である．よってこの問題は図 5 のように結果を複製するための RMOV 命令を追加することで解決できる．このために追加する RMOV 命令をタイプ C（copy）の RMOV 命令と呼ぶことにする．

4.2.2 ループ内定数が存在する場合（タイプ L/S）

図 6（左上）のように，phi 関数の参照が循環している場合，その閉路上には本来参照すべき命令が存在しない．このように，ループ内定数が存在する場合，ループの何周目であっても同じ距離で参照するためには，図 6（右上）のようにこのループ内に最低一回は RMOV 命令が必要になる．このようにループ内定数を保持するために追加する RMOV 命令をタイプ L（loop）の RMOV 命令と呼ぶことにする．phi 関数の参照が循環している特別な場合として，phi 関数が自分自身を参照することで循環が発生している場合が存在する．この場合は頻出であるため，特別にタイプ S（self loop）の RMOV 命令と呼ぶこととし，区別しない場合はタイプ L/S の RMOV 命令と記述する．

4.2.3 命令間に依存関係が存在する場合（タイプ D）

図 7（左上）のように，複数の phi 関数の参照先となる各命令間に依存関係（黄矢印）があり，命令の順序を自由には入れ替えられない場合，特に各経路で異なる依存関係がある場合を考える．図に示した状況では，すべての経路で同じ距離に配置するという制約を満たす以前に，それより弱い条件である，すべての経路で同じ順序になるように配置することすらできない．この問題は図 7（右上）のように RMOV 命令を追加して phi 関数のオペランドを依存関係のない命令に変更することにより解決できる．このために追加する RMOV 命令をタイプ D（Dependency）の RMOV 命令と呼ぶことにする．

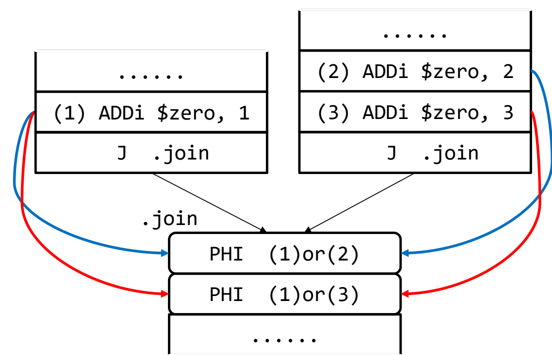

図 4 複数の phi 関数から参照される命令が存在する場合

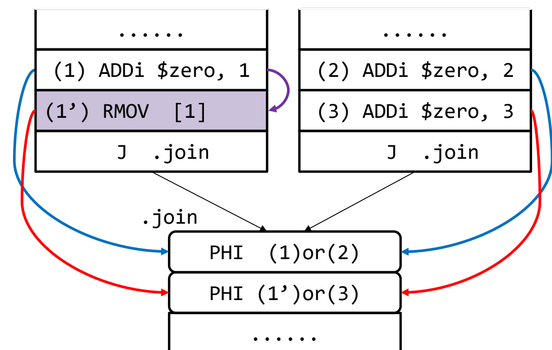

図 5 タイプ C の RMOV 命令を追加して解決した例

4.2.4 実行経路によらない値が存在する場合（タイプ B）

図 8 のように分岐があり，複数の実行経路で分岐前の同じ命令の結果を参照している phi 関数がある場合，その距離調整が他の phi 関数の距離調整と競合する場合がある．通常の phi 関数は二つの独立な命令を参照するのに対し，この場合は同じ命令を参照しており命令移動の自由度が低下していることや，二つの異なる命令を同じ位置に配置することはできないことが原因である．この問題は図 9 のように RMOV 命令を追加することにより解決できる．このために追加する RMOV 命令をタイプ B（Branch）の RMOV 命令と呼ぶことにする．

4.3 転送命令が不可欠な部分を特定し，RMOV 命令を追加するべき位置を決定するアルゴリズム

4.3.1 タイプ C の RMOV 命令

複数の phi 関数から参照される命令が存在した場合，その命令をオペランドとする RMOV 命令を（参照数-1 個）だけその直後に連続して追加し，参照している phi 関数のオペランドをそれぞれに変更すればよい．

4.3.2 タイプ L/S の RMOV 命令

まず，図 6（左下）のように phi 関数を頂点，phi 関数から phi 関数への参照を有向辺とした有向グラフを作成する．

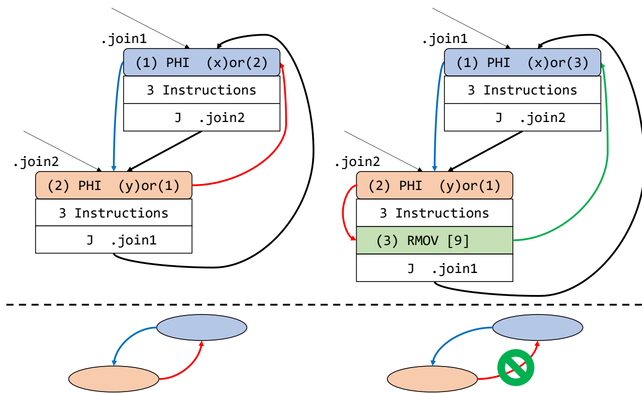

図 6 ループ内定数がある場合とタイプ L の RMOV 命令を追加して解決した例

この有向グラフに有向閉路が存在した場合，タイプ L/S の RMOV 命令の追加が必要になる．RMOV 命令を追加すべき位置を与えるには図 6（右下）のように有向閉路除去問題の解を考える．この時，除去すべき辺に対応する部分（辺の始点となる phi 関数と辺の終点となる phi 関数の間のいずれかの場所）に図 6（右上）のようにタイプ L/S の RMOV 命令を追加すればよい．

4.3.3 タイプ D の RMOV 命令

同様に図 7（左下）のように頂点を各 phi 関数とし，phi 関数が参照する命令間のデータ依存を有向辺とした有向グラフを作成する．この有向グラフがトポロジカルソートできるなら，すべての経路で同じ順序になるように命令を配置することができ，適宜間に NOP 命令を挟むことで同じ距離になるように配置することができる．しかし，有向閉路が存在した場合，トポロジカルソートを行うことができず，タイプ D の RMOV 命令の追加が必要になる．

RMOV 命令を追加すべき位置の決定方法は 4.3.5 で説明する．

4.3.4 タイプ B の RMOV 命令

タイプ B の RMOV 命令の必要性を判断するため，まず phi 関数を以下のように分類する．

phi 関数 α 二つの実行経路で同じ命令の結果を参照している phi 関数

phi 関数 β 二つの実行経路で異なる命令の結果を参照しており，そのうちの片方だけが分岐より前の命令を参照している phi 関数

phi 関数 γ 二つの実行経路で異なる命令の結果を参照しており，その両方が分岐より前の命令を参照している phi 関数

この時，phi 関数 α と phi 関数 β が同時に存在する場合タイプ B の RMOV 命令の追加が必要になる．また，phi 関数 α と phi 関数 γ が同時に存在する場合タイプ B の RMOV 命令を二つ追加する必要がある．

タイプ B の RMOV 命令の追加はタイプ D の RMOV 命令の追加で代替できる可能性がある．そこで，この制約をタイプ D の RMOV 命令が必要か判定したときに作った有向グラフ上に，有向辺を追加することで表現すると統一的に扱うことができる．具体的には，phi 関数 α と phi 関数 β の間に双方向の有向辺を，phi 関数 α と phi 関数 γ の間には双方向の二重有向辺（つまり合計四本）を，それぞれ作成する．

RMOV 命令を追加すべき位置の決定方法は 4.3.5 で説明する．

4.3.5 ハイパーグラフの利用

図 7（右）や図 9（左）のようにタイプ D/B の RMOV 命令は一つ追加するだけで複数の有向辺を除去することができることがある．そのため，同時に除去されうる有向辺をまとめた，{ src: v1, dst: { v2, v3, v4 } } といった形式の有向ハイパー辺を持つ有向ハイパーグラフを考えると一つの RMOV 命令の追加と一つの有向ハイパー辺の除去を対応付けることができる．

この時，図 7（右）や図 9 のようにこの有向ハイパーグラフにおける有向閉路除去問題の解に対応する部分（辺の始点となる命令と辺の終点となる phi 関数の間のいずれかの場所）に RMOV 命令を追加すればよい．

4.4 冗長な転送命令の追加を防ぎつつ距離を定めるアルゴリズム

Step0. 既存のアルゴリズムと同様に phi 関数を追加する．

Step1. 4.3.1 で示した手順によりタイプ C の RMOV 命令を追加する．

Step2. 4.3.2 で示した手順により有向グラフを作成し，これの有向閉路除去を考え，タイプ L/S の RMOV 命令を追加する．

Step3. 4.3.3 と 4.3.4 で示した手順により有向グラフを作成し，これをハイパーグラフとみなす．そして 4.3.5 で示した手順によりタイプ D/B の RMOV 命令を追加する．

Step4. Step2 でタイプ L/S の RMOV 命令を追加したため，Step2 で考えた有向グラフをもう一度作成すると directed acyclic graph（DAG）となっており，トポロジカルソートが可能である．以下の手順で使うためにトポロジカルソートの結果を求めておく．

Step5. Step3 でタイプ D/B の RMOV 命令を追加したため，合流点である各基本ブロックについて，Step3 で考えた有向グラフをもう一度作成すると DAG となっており，トポロジカルソートが可能である．以下の手順で使うためにトポロジカルソートの結果を求めておく．

Step6. phi 関数のオペランドとなる各命令が Step5 で判明したトポロジカル順で実行されるように命令の順序を入れ替える．この時，依存関係のトポロジカル順に並んでいることから，命令のソースオペランドがそれより後方の命令の生成した結果になってしまう事態は発生しない．

Step7. Step4, Step5 で判明したトポロジカル順と逆順に，

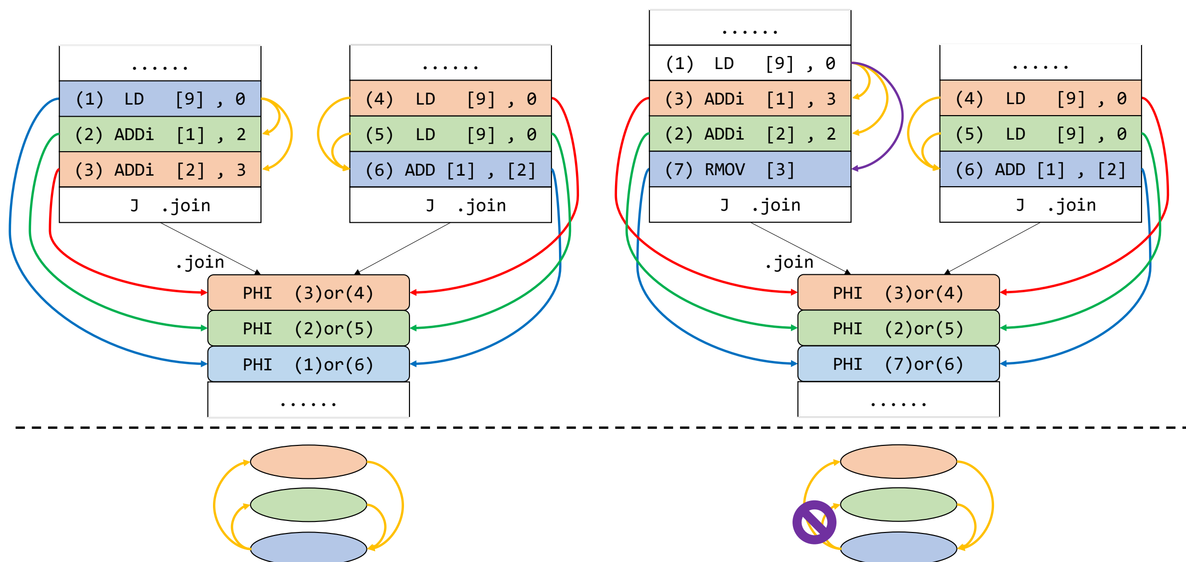

図 7 命令間にデータ依存がある場合とタイプ D の RMOV 命令を追加して解決した例

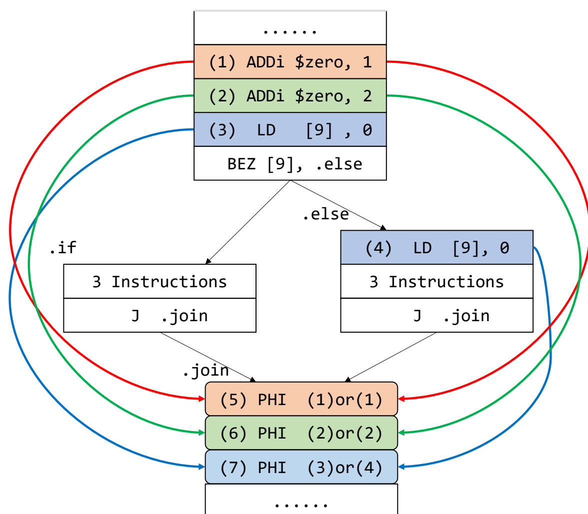

図 8 複数の実行経路で同じ命令を参照している phi 関数があるため距離調整が競合する場合

phi 関数のオペランドまでの距離調整を行う．具体的には，以下の手順を繰り返す．

Step7.1. phi 関数のオペランドとなる命令（命令 A とする）の存在する基本ブロック内に存在する，命令 A に依存している命令の数を数える．ここには間接的に依存している命令も含まれる．ただし，phi 関数にたどり着くまでの経路で実行されない命令は含めない．また，分岐命令も制御フローとして暗に依存している命令であるため数える必要がある．ここで数えた数は，命令 A を基本ブロック内で最も後ろに移動させたときのそれ以降の実行命令数に等しい．

Step7.2. Step7.1 を各実行経路で計算し，それらの最大値をとる．

Step7.3. phi 関数のオペランドとなる各命令（命令 A とする）について，Step7.2 で得られた値よりも多くの命令が後方に存在する場合，命令 A より後方に存在し命令 A に依存していない最も前方にある命令を，命令 A の直前に移動させることで phi 関数から命令 A までの距離を減少させる．

Step7.4. phi 関数のオペランドとなる各命令（命令 A とする）について，Step7.3 で得られた値よりも少ない命令しか後方に存在しない場合，命令 A の直後に NOP 命令を追加することで phi 関数から命令 A までの距離を増加させる．

Step7.5. すでに固定されているなどの理由で動かせない場合は，既存手法同様に fixed 領域を作って距離の調整を解決する．この手順で追加された RMOV 命令のことはタイプ F（fixed）の RMOV 命令と呼ぶことにする．

Step7.6. phi 関数の各オペランドまでの距離をそろえることができたため，これ以降の手順でずれが発生しないように，固定する．

Step8. 最後に，連続する NOP 命令を同等の働きをする RPINC 命令に置き換える．

5. 評価

5.1 評価方法

コンパイラ基盤である LLVM4.0[9] のバックエンドプログラム llc に STRAIGHT 用コンパイラバックエンドを追加した．ここで，距離を調整するアルゴリズムは既存手法と 4 章で述べた提案手法の二つを実装した．ベンチマークコードをこの二つの手法でコンパイルし，得られた機械語プログラムを用いてプロセッサシミュレータを用いたシミュレーションを行った．ベンチマークコードとしては Livermore loops[10] を用いた．ただし，STRAIGHT 命令セット中には無理関数を計算する命令が存在せず，またライブラリも未整備であるため，平方根関数を使用している Kernel 15 および指数関数を使用している Kernel 22 を除いた 22 個のコードを使用した．プロセッサシミュレータ

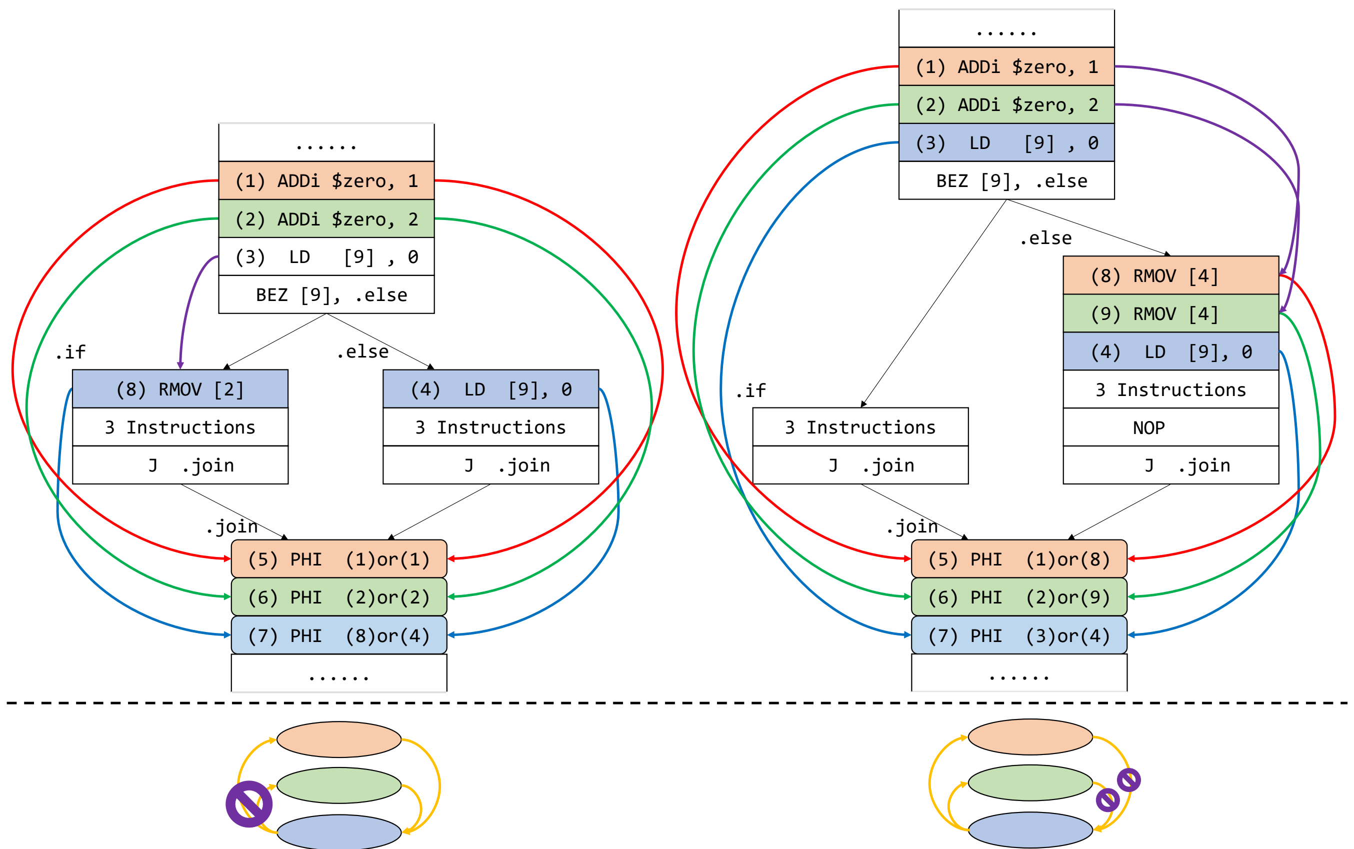

図 9 複数の実行経路で同じ命令を参照している phi 関数があるため距離調整が競合したが，タイプ B の RMOV 命令を追加することで解決した二つの例

には「鬼斬弐」[11] を使用し，STRAIGHT プロセッサのパイプラインを一クロック単位で正確にシミュレーションすることができるように改造した．プロセッサのパラメータは表 1 に示すように二種類用意した．2-way は小さなアウトオブオーダー実行コア，4-way は大きなアウトオブオーダー実行コアのモデルとなるようにパラメータを決定した．メモリへのストアアクセス系列が x86 上で動作させた時と一致していることにより，プログラムが正常に実行されていることを確かめた．

5.2 評価結果

図 10 は既存手法により追加される RMOV 命令の数を 100%としたとき，提案手法で追加される RMOV 命令の数を種類ごとにプロットしたものである．全てのベンチマークで追加される RMOV 命令の数を削減し，平均で 50%削減することができたことを示している．

また，追加される RMOV 命令のほとんどはタイプ L/S であり，タイプ D, F, B のものは多くないことがわかる．

図 11 は既存手法・提案手法でコンパイルすることで得られる機械語プログラムを実行したときの命令の種類ごとの実行割合を，既存手法でコンパイルしたものの実行命令数を 100%として表したグラフである．ただし，投機ミスにより破棄された実行パス上で実行された命令は含まれていない．

提案手法の適用により，NOP 命令や RPINC 命令以外の，本質的に必要な他の命令数を増加させることなく冗長な RMOV 命令のみ減らし，実行命令数の削減に成功していることがわかる．たとえば，ループカウンタや誘導変数は次のループでの値を計算する命令の位置と，ループの初期値を設定する命令の位置をうまく合わせることで RMOV 命令を完全に排除できているため，実行命令数の大きな削減につながっている．また，提案手法を適用しても残り，かつ実行回数の多い RMOV 命令の大半はタイプ L/S の RMOV 命令であることがわかる．

図 12 は既存手法でコンパイルしたものを実行したときの実行性能を 1 としたとき，提案手法でコンパイルしたものを実行したときの実行性能を表したグラフである．全てのベンチマークで性能が向上しており，幾何平均で見て 11.1%（4-way），13.3%（2-way）の性能向上となっている．

このように提案手法による RMOV 命令の削減はプログラム実行時間の観点から見ても有効であることがわかる．4-way での実行は 2-way での実行に比べ演算器に余裕があるため性能向上幅はやや小さくなっている．プログラムの実行時間が削減されるのは，実行すべき命令数が削減できることが主要因である．

図 13 は既存手法でコンパイルしたものを実行したとき

ⓒ 2018 Information Processing Society of Japan

表 1　プロセッサのパラメータ

		2-way	4-way
論理レジスタ数		128 ($n = 7$)	
フェッチ幅		2	6
物理レジスタ数		192	384
スケジューラサイズ		16	128
発行幅		2	4
演算器数	iALU, iBC, Mem	2	4
	fADD, fMUL	1	2
	iMUL, iDIV, fDIV	1	1
ロードストアキュー容量		LD, ST 各 48	LD 72 ST 56
リオーダバッファ容量		64	256
リタイア幅		2	6
分岐予測器		Gshare, 32K entry history12bit	
分岐先バッファ		4way, 2K entry	
メモリ依存予測器		StoreSet, 4K entry producer ID 9bit	
L1 キャッシュ		32KB+32KB 4way, 64B line 4 cycle hit latency	
L2 キャッシュ		256KB, 4way, 64B line 12 cycle hit latency	
メインメモリ		200 cycle latency	
プリフェッチャ		L2: StreamPrefetcher 16 entry distance 8, degree 2	

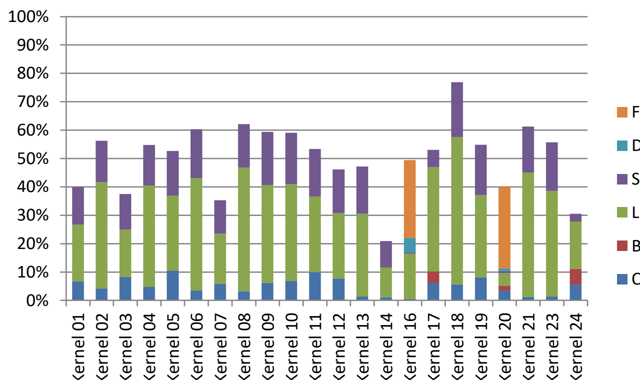

図 10　プログラム中に残存した RMOV 命令とその種類

の IPC を 1 としたとき，提案手法でコンパイルしたものを実行したときの IPC を表したグラフである．ほとんどのベンチマークでは IPC が低下しているが，これは無駄な命令を削減したからであり，性能が低下したことを意味しない．

5.3　議論

fixed 領域を作成する従来の手法では，すべての基本ブロックに RMOV 命令を追加していた．一つの基本ブロックには全く RMOV 命令を追加しなくても，他全ての基本ブロックで同じ位置に RMOV 命令を配置することで必ず距離をそ

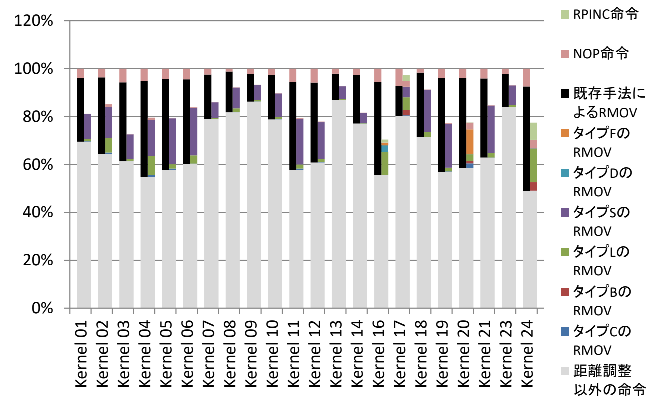

図 11　総実行命令数に占める命令種類ごとの割合

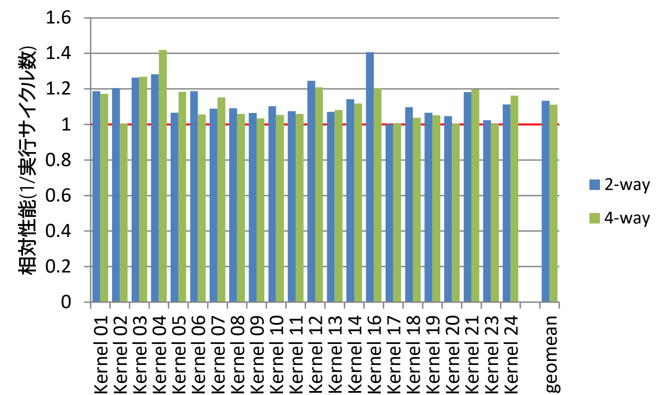

図 12　最適化前と比較した相対実行性能

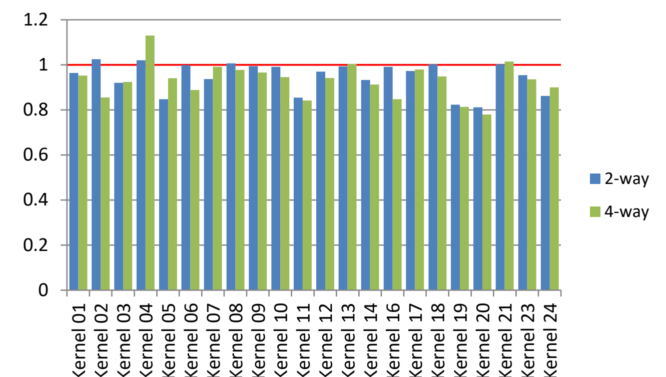

図 13　最適化前と比較した相対 IPC

ろえる事ができる．また，ほとんどの場合，合流部では二つの基本ブロックが合流する．よって，典型的な場合では RMOV 命令の追加は従来手法の半分で済む．多くのベンチマークで追加される RMOV 命令の数を半分程度に削減できるのはこれが原因である．

タイプ D の RMOV 命令は実行経路ごとに異なる順序でレジスタ変数に書き込まざるを得ない場合に追加される，STRAIGHT アーキテクチャ特有の転送命令である．評価結果から，これが必要になるプログラムはほとんどないことがわかる．タイプ F の RMOV 命令はコントロールフローが複雑な時に発生する．実際，Kernel 16 は goto 文を含む複雑なコントロールフローを持つ．Kernel 20 は複雑では

ないが，特定の一つのレジスタを書き換えるかの条件分岐が続き，上書き可能なレジスタのない STRAIGHT アーキテクチャが苦手とするコードとなっている．タイプ B の RMOV 命令はすでにレジスタ上にある値のうちどれかを実行パスごとに選択する場合に発生するが，その場合には通常の RISC アーキテクチャでも転送命令が必要になる．タイプ C の RMOV 命令はレジスタに入っている値を複製するときに発生し，通常の RISC アーキテクチャでも転送命令の追加が必要になる．タイプ L/S の RMOV 命令はループ内定数を保持するために追加される，STRAIGHT アーキテクチャ特有の転送命令である．

ループ内定数を保持するために追加されるタイプ L/S の RMOV 命令はループ内あるいは 2^n 命令内に一つあればよいため，ループアンローリングによりその数を削減できる．ループアンローリングにより冗長な RMOV 命令の数を減らすことができ，性能を向上させることができることは以前から指摘されていた [13]．ただし，提案手法はループアンローリングと異なり，コードサイズを肥大化させずに冗長な RMOV 命令を削減することができる．また，ループ構造だけではなく，あらゆるコントロールフローについて統一的に適用できる．

冗長な RMOV 命令は余っている演算器で実行可能なため，レイテンシには大きく影響を及ぼさないと考えられてきた．図 13 から，一部のベンチマークでは実行すべき命令数を削減した上に IPC の向上も得られていることがわかる．これは RMOV 命令を介した依存の連鎖が短くなりクリティカルパスが縮まることによる効果である．すなわち，RMOV 命令がクリティカルパス上に追加され，レイテンシが伸びてしまうようなプログラムの存在を示すものである．

6. 関連研究

コンパイラの支援を前提としてハードウェアの簡素化を狙う，同様のコンセプトを持ったアーキテクチャとして VLIW アーキテクチャが挙げられる [14]．VLIW アーキテクチャではコンパイル時点でどの命令が並列に実行可能かを解析するため，ハードウェアを大幅に簡素化することができる．STRAIGHT アーキテクチャの場合，並列に実行可能かはハードウェアで解析するため，ハードウェアの簡素性といった点では VLIW アーキテクチャに劣る．しかし，VLIW アーキテクチャのコンパイラはソースコードから得られる静的な情報のみをもとに並列実行可能かを解析するため，並列性の抽出には限界があり，またその潜在性能を引き出すコンパイラを構成することは大変困難である．それに対して STRAIGHT アーキテクチャの場合，その潜在性能を引き出すコンパイラを構成することが現実的に可能であることを本研究で示した．

レジスタの名前ではなく何命令前，といった形（逆デュアルフロー形式）でオペランドを指定するアーキテクチャに逆デュアルフローアーキテクチャが挙げられる [15]．逆デュアルフローアーキテクチャでは，命令セットは通常の RISC 同様のものを用い，内部で逆デュアルフロー形式の表現に変換し，トレースキャッシュに保存する．このためコントロールフローに依存する距離計算を動的に行うことが可能であり，余計な命令の実行が不要である利点がある．一方 STRAIGHT アーキテクチャでは距離の調整は完全にコンパイル時に行われ，フロントエンドで行う仕事量を大きく削減することができる．

例外発生時の巻き戻しコストを削減しようとする試みに，Idempotent アーキテクチャが挙げられる [16]．Idempotent アーキテクチャのコンパイラは，プログラムを冪等な，つまり複数回実行しても実行結果が変わらない領域に区切ることで，例外発生時の巻き戻しを単なる再実行に置き換えることを可能にする．STRAIGHT アーキテクチャも偽の依存がないことをコンパイラが保証するため例外発生時の巻き戻しが不要であるが，これはプログラムの全領域に対して成立する．また，Idempotent アーキテクチャはインオーダースーパースカラプロセッサを高速化する技術であり，アウトオブオーダー実行を前提とした STRAIGHT アーキテクチャとは異なる．

電力効率を向上させるためにコンパイラを活用する別のアプローチとして，Relax 命令セットアーキテクチャが挙げられる [17]．Relax 命令セットアーキテクチャを用いると，一定確率でのエラーを許容，あるいはやり直しによる実行時間のロスを許容することで，エネルギー効率を大きく上げることが可能になる．これは一種の投機であり，コンパイラの静的保証をもとに投機ミス時のハードウェアの仕事量を低減し電力効率を向上させるアプローチが共通する．一方，これによる恩恵を受けるためにはプログラマが明示的に指定する必要がある上，適用できる部分は純粋な関数に限られている点は STRAIGHT アーキテクチャと異なる．また，この手法は実行効率と引き換えに電力効率を上げているのに対し，STRAIGHT アーキテクチャは実行効率と電力効率を同時に上げることができる．

従来と大きく異なる ISA を採用することで高効率な実行を可能にしようとする試みに，SEED アーキテクチャが挙げられる [4]．SEED アーキテクチャでは，関数呼び出しのないループ部分をデータフローマシンで実行することにより効率化を図る．データフローマシンで実行することから，抽出できる命令レベル並列性は非常に高い．一方コンパイラはデータフローマシンを構築するだけではなく，ハードウェアの構造に合わせたマッピングを行わなければならず，レイテンシを最小にすることは難しい．

7. 今後の課題

提案手法で追加される RMOV 命令の数を最小化する場合，最小有向閉路除去と呼ばれる APX 困難 [18] な問題を解く

必要があり，プログラムの規模が大きくなると現実的な時間内に解くことができない．これは従来の RISC アーキテクチャにおいて，レジスタ割り付けの際にグラフ彩色問題を解かないといけないことに類似した問題であり，なんらかの発見的手法による近似手法が必要である．そのため，レジスタ割り付け問題の場合におけるレジスタ圧などのような，近似の指標となるものを見つけ，プログラムの規模が大きくなってもコンパイルできる方法を構成することが今後の課題となる．

謝辞　本論文の研究の一部は共同研究「Deep Learning 向け Straight アーキテクチャの研究」（株式会社富士通研究所）による．

参考文献

[1] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K. and Burger, D.: Dark Silicon and the End of Multicore Scaling, *Micro, IEEE*, Vol. 32, No. 3, pp. 122 – 134 (2012).

[2] Greenhalgh, P.: Big. little processing with arm cortex-a15 & cortex-a7, *ARM White paper*, Vol. 17 (2011).

[3] Lukefahr, A., Padmanabha, S., Das, R., Sleiman, F. M., Dreslinski, R., Wenisch, T. F. and Mahlke, S.: Composite cores: Pushing heterogeneity into a core, *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, IEEE, pp. 317–328 (2012).

[4] Nowatzki, T., Gangadhar, V. and Sankaralingam, K.: Exploring the potential of heterogeneous Von Neumann/dataflow execution models, *Int. Symp. on Computer Architecture*, pp. 298 – 310 (2015).

[5] Irie, H., Fujiwara, D., Majima, K. and Yoshinaga, T.: Straight: Realizing a lightweight large instruction window by using eventually consistent distributed registers, *Networking and Computing (ICNC), 2012 Third International Conference on*, IEEE, pp. 336 – 342 (2012).

[6] 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，Vol. 2013-ARC-206, No. 5, pp. 1 –10 (2013).

[7] 赤木晟也，入江英嗣，坂井修一：STRAIGHT アーキテクチャの HDL 実装と評価，電子情報通信学会総合大会講演論文集， Vol. 2016, No. 1, p. 69 (2016).

[8] 中江哲史，入江英嗣，坂井修一：STRAIGHT アーキテクチャのためのコンパイラ技術，電子情報通信学会総合大会講演論文集，Vol. 2016, No. 1, p. 70 (2016).

[9] LLVM: The LLVM Compiler Infrastructure. `http://llvm.org/`.

[10] McMahon, F. H.: The Livermore Fortran Kernels: A computer test of the numerical performance range, Technical report, Lawrence Livermore National Lab., CA (USA) (1986).

[11] 塩谷亮太，五島正裕，坂井修一：プロセッサ・シミュレータ「鬼斬弐」の設計と実装，先進的計算基盤システムシンポジウム SACSIS2009， Vol. 2009, No. 4, pp. 120 – 121 (2009).

[12] 中田育男：コンパイラの構成と最適化 第 2 版，朝倉書店 (2009).

[13] 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，修士論文，電気通信大学 (2014).

[14] Nicolau, A. and Fisher, J. A.: Measuring the parallelism available for very long instruction word architectures, *IEEE Transactions on Computers*, Vol. 33, No. 11, pp. 968–976 (1984).

[15] 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 (2008).

[16] De Kruijf, M. and Sankaralingam, K.: Idempotent Processor Architecture, *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, New York, NY, USA, ACM, pp. 140–151 (online), DOI: 10.1145/2155620.2155637 (2011).

[17] De Kruijf, M., Nomura, S. and Sankaralingam, K.: Relax: An architectural framework for software recovery of hardware faults, *ACM SIGARCH Computer Architecture News*, Vol. 38, No. 3, pp. 497–508 (2010).

[18] Kann, V.: On the Approximability of NP-complete Optimization Problems, PhD Thesis, Royal Institute of Technology Stockholm (1992).