Vol.2017-ARC-225 No.3
Vol.2017-SLDM-179 No.3
Vol.2017-EMB-44 No.3
2017/3/9

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

福田 晃史[1]　中江 哲史[2]　入江 英嗣[2]　坂井修一[2]

概要：現在高性能 CPU 内の各コアで主流となっているアウト・オブ・オーダ・スーパスカラ・アーキテクチャは，柔軟な高速化が可能であるものの，1 命令あたりに多くの処理を必要とし，電力効率の改善が望まれている．我々はこれまでの研究で，リネーム処理のいらない省電力アウト・オブ・オーダ実行を特徴とする “STRAIGHT” アーキテクチャを構想し，そのパイプライン構成とコンパイラの実現法を明らかにしてきた．STRAIGHT アーキテクチャは広大な論理レジスタ空間を持ち，この空間を利用して上書きのないコードを静的に生成することで，従来のレジスタ生存管理やマッピング管理を一切不要とする．さらに，広大な論理レジスタ空間とその id 付けは，より多くのメモリデータをレジスタへ同時に展開することを可能とすることが期待されている．本論文では，この特徴を利用してループ内のロード命令を削減する最適化を提案する．先行研究で開発した STRAIGHT コンパイラに実装した結果，Livermore Loops ベンチマークのうち 2 つのプログラムに対して，ロード命令をそれぞれ 31 %，45 %削減でき，また実行サイクル数も 13%，14 %減少させられた．

Akifumi Fukuda[1]　Satoshi Nakae[2]　Hidetsugu Irie[2]　Shuichi Sakai[2]

1. はじめに

マイクロプロセッサのシングルスレッド実行性能向上はあらゆるプログラムの実行時間の短縮に繋がるが，2000 年頃からは配線遅延などの問題によって限界を迎え，コア数を増やすことによる性能向上が図られるようになった．しかし，マルチコア化による性能向上はアムダールの法則 [3] が示す通り，逐次実行部分を処理するコア単体の性能に制限される．また，トランジスタの微細化を進め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・スケーリング則が破綻したために，チップ上のトランジスタすべてを同時に稼働させられないというダーク・シリコン問題が生じており，この観点からもマルチコア化による性能向上の限界が指摘されている [6]．そのため，再構成可能なハードウェアを一部に利用したプロセッサ [4] や複数のアーキテクチャのユニットを 1 つのコア内に持つプロセッサ [11] が考案されるなど，新たなパラダイムの模索が続いている．

1　東京大学工学部
Faculty of Engineering, The University of Tokyo
2　東京大学大学院　情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

上記の問題を解決するために，我々はシングルスレッド実行能力の向上と消費電力の削減を目指した **STRAIGHT アーキテクチャ**を構想している [8], [16]．STRAIGHT アーキテクチャは，広大な論理レジスタ空間を持ち，偽の依存はコンパイラによって解決される．そのため，従来のアウト・オブ・オーダ・プロセッサにおいて電力消費の大きかったリネーミング・ロジックやフリーレジスタ管理を省略できる．また，これによって命令ウィンドウサイズやフロントエンド幅の拡張が容易となり，シングルスレッド性能が向上する．初期評価では，12 % のエネルギー消費削減と 30 % の IPC（Instruction Per Clock）向上を見積もっている．

STRAIGHT アーキテクチャは，各ソース・オペランドを命令間の距離で指定するという特徴を持つため，分岐やループなど実行命令数が動的に変化するフローを辿っても適切な値を参照できるように命令間の距離の調節を行う独自のコンパイラアルゴリズムを必要とする．先行研究において我々は，辿ったフローによって異なる値をとる phi 命令を含む基本ブロックの先行ブロックの末尾に fixed 領域と呼ぶ領域を設け，phi 命令で参照される値を集約することによって基本コンパイラを実現し，同じ処理を行う Alpha

Vol.2017-ARC-225 No.3
Vol.2017-SLDM-179 No.3
Vol.2017-EMB-44 No.3
2017/3/9

アセンブリに対して実行命令数が 37 % 減ることを確認した [14]．しかし，NOP 命令を削減する以外の最適化を行っていない [15]．

本研究では STRAIGHT アーキテクチャ固有の特徴を利用してメモリアクセス命令を減らす最適化アルゴリズムを考案・実装し，プロセッサ・シミュレータ「鬼斬弐」[1], [13] を用いて評価した．その結果，最大でロード命令を 45 %を削減でき，また実行サイクル数も最大で 14 %減少させられた．

2. STRAIGHT アーキテクチャ

STRAIGHT アーキテクチャでは，各命令に対して自動的に 1 つずつデスティネーション・レジスタが割り当てられる．その割り当てられる論理レジスタは 2^n 個あり，命令間の距離が 2^n 以下の任意の 2 命令のデスティネーション・レジスタは必ず異なる．従って，2^n 以内の距離に位置する 2 命令の間には偽の依存が存在しえない．

また，レジスタの参照は名前ではなく，そのレジスタに値を書き込んだ命令までの距離で指定する．この参照可能な距離は論理レジスタ数によって制限される．プログラムがこの制約を守っていることはコンパイラによって保証されるため，コンパイルされたプログラム中には偽の依存は存在しない．

レジスタの管理は，命令のフェッチに合わせてインクリメントされる RP（Register Pointer）という特殊なレジスタで行う．RP が指す番号を持つレジスタがそのまま命令のデスティネーション・レジスタとなり，RP に命令中で指定された距離を引いたものによってソース・レジスタは決定される．一旦 2^n 以上の距離に遠のいたレジスタは以後参照されることはないため，上書きすることが可能となる．つまり，従来の OoO 実行アーキテクチャで必要であった物理レジスタの再利用のためのフリーリストの管理が STRAIGHT アーキテクチャでは RP の操作だけで行える．RP は $2^n + m$ まで数えると 0 に戻る．ただし，m はイン・フライトな命令の最大数とする．

なお，STRAIGHT アーキテクチャはデスティネーションとして利用されない特殊レジスタとして他にスタック・ポインタ，フレーム・ポインタ，グローバル・ポインタを備える．

命令セットは通常の RISC と同様に算術・論理演算命令，分岐命令，メモリアクセス命令などから構成される．

以下に例として，STRAIGHT アセンブリで書かれたフィボナッチ数を計算するプログラムを上げる．

```
ADDi $0, 1     # r[0] <- 1
ADDi $0, 1     # r[1] <- 1
ADD  [1], [2] # r[2] <- r[1] + r[0]
ADD  [1], [2] # r[3] <- r[2] + r[1]
...
```

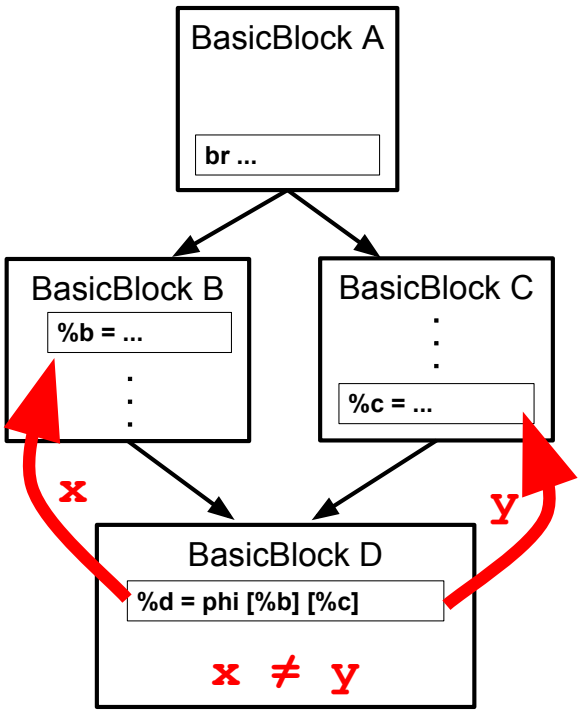

図 1　分岐を含むプログラムの制御フローグラフ
Fig. 1　An if-then-else control flow graph

3. STRAIGHT コンパイラ

STRAIGHT コンパイラは中間表現として，コンパイラ基盤 LLVM[2] が用いる LLVM IR を採用している．LLVM IR は，関数などの構造を保っている，無数のレジスタを持つ RISC アセンブリのような表現である．各命令は RISC アーキテクチャの命令とほとんど 1 対 1 に対応する．また，静的単一代入形式（Static Single Assignment form, SSA）になっているのでデスティネーション・レジスタは各々一度のみ代入先として指定される．つまり，LLVM IR で表現されたプログラム中には偽の依存が存在しない．

分岐・合流を含まないプログラムは，LLVM IR から STRAIGHT アセンブリへと次の手順でコンパイルすることができる．

- デスティネーション・オペランドを削除する
- ソース・オペランドの識別子が最後にデスティネーションとして現れた命令との距離を計算して STRAIGHT アセンブリのソース・オペランドとして出力する
- LLVM IR の命令をそれに対応する STRAIGHT アーキテクチャの命令と置換する

一方，分岐・合流を含むプログラムは異なる実行経路を持ち，**図 1** に示すようにそれぞれで実行される命令数は一般に異なる．そのため上記の単純なアルゴリズムでは実行経路によって参照すべきレジスタまでの距離を一意に定められない．適切に命令間の距離を調節し，いずれの実行経路でも参照距離が等しくなるようにする必要がある．

まず分岐をまたいだ参照では調節をする必要がない．図 1 において A→B という実行経路を辿った時は A，B が連続した命令列であるかのように処理することができ，A→C という実行経路を辿った時も A，C が連続した命令列であるかのように処理することができる．

問題が起きるのは合流後の D から合流前の値を参照する場合，つまり D から B，C あるいは A 内を参照する場合である．この時，LLVM IR では D の先頭に phi 命令が置かれ，辿ったフローに応じた値が取り出される．B, C は命令数が異なり，参照すべきレジスタがブロックの末尾から等距離にあるとは限らない．

そこで，基本ブロック B と基本ブロック C の末尾に，何もしない NOP 命令，ソース・オペランドの値をそのままデスティネーションに出力する RMOV 命令，ゼロレジスタとあわせて使うことで定数を生成できる ADDi 命令を挿入することで，基本ブロック B，C の参照される値が基本ブロックの先頭から等距離の場所に来るように調整を行う．なお，基本ブロック末尾に分岐命令などがある場合は，その後に挿入しても実行経路上の命令間距離に寄与しないのでその直前に挿入する．

より具体的には以下の手順を踏む．

(1) 全ブロックの末尾に，距離調整のための命令を置くための fixed 領域を設ける．ブロック末尾にジャンプ命令がある場合はそれを fixed 領域に入れる．なければ空とする．
(2) phi 命令を持つすべての基本ブロックについて次の処理を行う，
  (a) すべての先行ブロックの末尾の fixed 領域内の命令数がすべて等しくなるように，fixed 領域の先頭に NOP 命令を追加する．
  (b) 先行ブロックの fixed 領域の先頭に RMOV 命令，ADDi 命令を挿入して，phi 命令のあるブロックから参照される値がすべての先行ブロックで同じ順に並ぶようにする．

このようにして複数の先行ブロックを持つ基本ブロックからの参照は，その先行ブロック末尾の fixed 領域までに限られる．追加する命令すべてを fixed 領域に入れるのは後の処理で新たに命令を挿入することになった際に以前追加された命令へ影響を与えないようにするためである．

また，物理レジスタ数以上のデータを受け渡す必要があったり，以上の手順で参照が解決できなかったりする場合は，スタック領域を介してデータをやり取りする．この時，スタック領域にアクセスする命令が上で各ブロックの末尾に固定した命令に影響を与えないように追加する．

4. ループ最適化

4.1 概要

STRAIGHT アーキテクチャは一度しか書き込みが許されない多くのレジスタを持っているため，他のアーキテクチャでは上書きされてしまうような値でもレジスタから得ることができる．本論文で最適化の対象とするのは，前の周で計算された値を利用するようなループであり，例えば次に示す二重階乗の計算を行うプログラムが含まれる．

```
a[0] = a[1] = 1;
for(int i = 2; i < 10; i++){
    a[i] = i * a[i-2];
}
```

このループ内の `a[i] = i * a[i-2];` という文は，

(1) `a[i-2]` をメモリからロード
(2) 左辺値 `i * a[i-2]` を計算
(3) 計算結果を `a[i]` にストア

という 3 つの命令列にコンパイルされる．

ロードされる `a[i-2]` の値は 2 周前のループで計算され，少なくともその瞬間にはレジスタに保持されている．しかし，レジスタを名前で指定するアーキテクチャでは，1 周前のループでそのレジスタの値が上書きされてしまうので `a[i-2]` の値をメモリからロードする必要がある．STRAIGHT アーキテクチャではループ一周分の命令数が十分に少なければ 1 周前と 2 周前では同じ命令であっても値が書き込まれるレジスタが異なるため，`a[i-2]` の値をメモリを介することなく参照することができる．

本研究で行う最適化はこのように可能な限りレジスタ上の値を利用することでメモリアクセスを削減するものである．

ちなみに上の例で，ループ内の文が `a[i] = i * a[i-1];` であった場合は，上書きを行わないようにレジスタを割り当てることで，STRAIGHT アーキテクチャ以外のレジスタを名前で指定するアーキテクチャでも同様の最適化が可能である．また，ループ・アンローリングを行うことでレジスタを有効に利用しやすくなり同種の効果が得られる．しかし，STRAIGHT アーキテクチャ　に比べ他のアーキテクチャはレジスタが少なくスピルアウトによる性能低下が起こりやすいため，ループ・アンローリングによって得られる性能向上も限定的になる．STRAIGHT アーキテクチャではループの変形を行わずにレジスタを利用してメモリアクセス命令を削減できる．

5. 提案アルゴリズム

本研究で最適化するのは次のような形のループである．

```
// A, B, C は整数定数
for (int i = A; i < B; i += C){
    ...
    a[i] = ...
    ...
    ... = a[i-k] ...
```

```
    ...
    ... = a[i-k'] ...
    ...
}
```

より具体的には次の条件を満たす．

- ループの繰り返し回数が一定である
- ループ中に分岐及びポインタを介したメモリアクセスを含まない
- ループ中に以下の配列アクセスがある
  - a[i] へのストアが一度行われる
  - a[i-k]（ただし，$k \equiv 0 \mod C$）の形のロードが一度以上行われる
    - 複数のロードで k が異なっていて良い
    - このロードが削減の対象である
    - つまり，上記の例では a[i-k]，a[i-k'] がともに対象となる

ループ中に分岐を含んだ場合を除外したのは，分岐命令を含むとループ 1 周分の命令数が一定にならないため，後述のアルゴリズムでレジスタの参照距離を確定できないからである．また，ポインタを介したメモリアクセスを認めないのは，一般にポインタの参照先を解析することが困難であり，ポインタによってメモリ依存が生じる可能性があるからである．

このループに対して次に述べるアルゴリズムを適用する．これによって，a[i-k] のロード命令が，k / C 周前の a[i] = ... のストア命令のデスティネーション・レジスタの値をコピーする命令へと置き換えられる．

(1) a[i-k] のロードのみが依存しているアドレスを計算する命令を取り除く．この処理を行ったあとのループ一周の命令数を L とする．

(2) LOAD a[i-k] を RMOV (k / C * L -（STORE a[i] までの距離）) に置き換える．
- つまり，メモリから値を持ってくる代わりに (k / C) 周前のループのストア命令のデスティネーション・レジスタから値を持ってくる．
- ただし，2^n 命令以上前のデスティネーション・レジスタは参照できないので，(k / C * L -（STORE a[i] までの距離）) $\geq 2^n$ ならばこの最適化は行わない．このロード命令を最適化対象から外した上で，最適化を最初からやり直す．

(3) ループ直前の基本ブロックの fixed 領域の大きさが (k / C * L) 以上になるまで fixed 領域の先頭に NOP 命令を追加する
- ループの 1 周目から (k / C) 周目までで参照する a[i-k] の値はループ中で計算されない．そのため事前にレジスタの適切な位置にメモリから値を読み込んでおく必要がある．これはその領域を確保するための手順である

(4) 3. で調節した fixed 領域の NOP 命令のうち，a[i-k] のロードから L, 2L, ..., C×L の距離にあるものを，ループ内で計算されない a[A-k/C]，... ，a[A-C] のロードに置き換える．

(5) a[i-k] の形のロードが複数ある場合は，各 a[i-k] について 1-4 を繰り返す

(6) ループ直前の基本ブロックの fixed 領域の先頭から 1 つ以上 NOP 命令が連続しているならそれらを取り除き，また先頭以外で連続している NOP 命令があるならばそれらを RPINC 命令で置き換える

本アルゴリズムの欠点としてループの直前に追加される NOP 命令によってプログラムサイズが増大しうることがある．しかし，実行命令数においてはループの実行回数が十分に大きければループ内から削減されたアドレス計算の命令が追加された NOP 命令の影響を相殺する．また，コンパイル時に挿入される NOP 命令はほとんど IPC に影響を与えていない [15] ことがわかっている．

なお，現在の仕様では論理レジスタの数は 2^{10}，つまり $n = 10$ と定められている．

6. 評価

6.1 評価環境

コンパイラ基盤である LLVM は，最適化及びバイナリ生成のみならず，プログラムの静的な情報の解析を行う手段を提供している．本研究では LLVM を用いて抽出した，最適化可能なループの情報を用いて，STRAIGHT コンパイラが 4 章で述べたアルゴリズムによる最適化を行うという形で実装を行い，その性能をプロセッサ・シミュレータ「鬼斬弐」によって評価した．鬼斬弐はサイクル・アキュレートなシミュレータであり，IPC やキャッシュヒット率，分岐予測の精度などの評価が可能である．この鬼斬弐をベースに STRAIGHT の挙動を再現する拡張を施したものでシミュレーションを行った．

STRAIGHT シミュレータの基準とするパラメータは表 1 に示した．

ベンチマークプログラムとしては Livermore loops を用い，評価対象の最適化以外の最適化は行っていない．

6.2 評価結果

Livermore loops のうち Kernel 5（tri-diagonal elimination, below diagonal），11（first sum）に最適化アルゴリズムを適用することができた．他の Kernel は適用対象のループを含まなかった．

Kernel 11 の主要部を図 **2** に示した．ループ本体である基本ブロック for.body9 の先頭のアドレス計算及びロード命令が RMOV 命令一つに置き換えられている．また，ループ直前の for.body4 は最適化によって命令数が増加してい

Vol.2017-ARC-225 No.3
Vol.2017-SLDM-179 No.3
Vol.2017-EMB-44 No.3
2017/3/9

表 1　アーキテクチャパラメータ
Table 1　Architecture Parameters

フロントエンド幅	16
リタイア幅	12
スケジューラサイズ	int 128 + fp 64 + mem 64
レジスタファイル	int 512 + fp 512
フロントエンドレイテンシ	5 cycle
発行幅	int 2, fp 2, mem 2
L1D キャッシュ	64 KB, 8 way, 64 Bline, 3 cycle hit latency
L1I キャッシュ	64 KB, 8 way, 64 Bline, 3 cycle hit latency
L2 キャッシュ	4MB, 16 way, 64 Bline, 12 cycle hit latency, with stream + stride prefetcher
メインメモリ	200 cycle

```
  # BasicBlock : for.body4
21  ADDi  [0]   49520
22  LD    [1]
23  ADDi  [0]   57528
24  ADDi  [0]   1
25  ST    [3]   [2]

  # BasicBlock : for.cond7
26  ADDi  [0]   20
27  SLT   [3]   [1]
28  BEZ   [1]   13

  # BasicBlock : for.body9
29  SUBi  [5]   1
30  MULi  [1]   8
31  ADDi  [1]   57528
32  LD    [1]
33  MULi  [9]   8
34  ADDi  [1]   49520
35  LD    [1]
36  FADD  [4]   [1]
37  MULi  [13]  8
38  ADDi  [1]   57528
39  ST    [3]   [1]

  # BasicBlock : for.inc13
40  ADDi  [16]  1
41  J     -15
```

```
  # BasicBlock : for.body4
22  ADDi  [0]   49520
23  LD    [1]
24  ADDi  [0]   57528
25  ST    [2]   [1]
26  ADDi  [0]   57528
27  LD    [1]
28  ADDi  [0]   1
29  UNDEF

  # BasicBlock : for.cond7
30  ADDi  [0]   20
31  SLT   [3]   [1]
32  BEZ   [1]   10

  # BasicBlock : for.body9
33  RMOV  [6]
34  MULi  [6]   8
35  ADDi  [1]   49520
36  LD    [1]
37  FADD  [4]   [1]
38  MULi  [10]  8
39  ADDi  [1]   57528
40  ST    [3]   [1]

  # BasicBlock : for.inc13
41  ADDi  [13]  1
42  J     -13
```

図 2　Kernel 11 の最内ループ主要部の最適化前（左）と最適化後（右）
Fig. 2　Kernel 11 assembly code

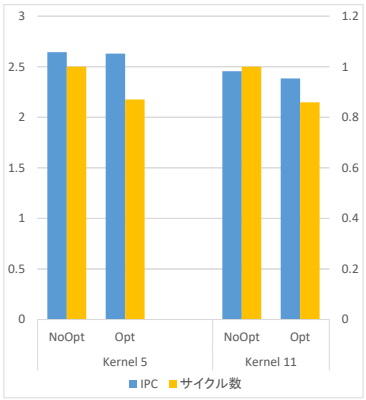

図 3　IPC 及び実行サイクル数の変化
Fig. 3　Instructions Per Cycle and Executed Cycles

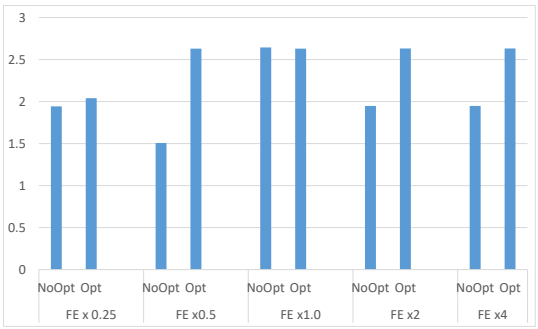

図 4　フロントエンド幅の変化に伴う Livermore Kernel 5 の IPC の変化
Fig. 4　The Relation between IPC and Front-End Width

る．最適化により，Kernel 5，11 それぞれでロード命令は 31 %，45 %減り，全実行命令数は 13 %，16 %減った．このことから，最適化によってループの直前に追加されるループ数周分の NOP 命令の影響はループ内から除去されるアドレス計算の影響に比して十分に小さいといえる．

IPC（Instruction Per Cycle）及び実行サイクル数の変化は図 3 に示した．IPC は Kernel 5 で 0.4 %，Kernel 11 で 3.2 % 低下しているが，実行サイクル数は Kernel 5 で 13 %，Kernel 11 で 14 %減少している．

従っていずれの場合もプログラム全体を実行するのに必要な時間は短縮されている．

また，フロントエンド幅（フェッチ幅及びスケジューラ幅）を表 1 を基準に 0.25 倍，0.5 倍，2 倍，4 倍と変化させた時の，最適化前後の IPC の変化を図 4 に示した．

メモリアクセスに用いられるアドレスは通常動的に生成されるため，メモリアクセス命令間の依存関係はレジスタ間のそれの解析より遥かに困難であり，並列実行を妨げ IPC を低下させる要因の一つとなる．実際に最適化前

では，フロントエンド幅が小さい場合はメモリ命令のスケジューリングをうまく行えないことによって，フロントエンド幅が大きい場合はストアアドレスの予測に失敗することによって IPC が低下している．一般にプログラムごとに最適なアーキテクチャパラメータは異なり，必ずしもプログラムを最適なパラメータのもとで実行できるとは限らない．しかし，状況を問わずメモリアクセス命令の削減によってこのように多くの恩恵が得られる．

7. 関連研究

Register Promotion [9] は可能な限りポインタ解析を行い，曖昧性のないポインタによるアクセスに関しては値の操作のたびにメモリアクセスを行うのではなく，レジスタに値をとどめ置いて性能向上を図る技法である．

具体的には，

```
for (i = 0; i < n; i++)
    *p += i;
```

このようなコードを次のように変換する．

```
tmp = *p;
for (i = 0; i < n; i++)
    tmp += i;
*p = tmp;
```

ループ内のメモリアクセス命令をループ外に移動させることでメモリアクセスの回数を減らしている．削減の対象であるメモリアクセス命令の参照先が繰り返しの間不変である点は本研究と異なる．

Register Promotion はループに焦点を当てているが，同様に関数間の値の授受にレジスタを有効に使うことでメモリアクセス命令を削減する最適化 [17] や，グローバル変数をレジスタからスピルしないようにする最適化 [7] も提案されている．

また，他にループの最適化手法としては，GPU を利用したもの [5], [10] や Scratchpad memory を用いたもの [12] などがある．いずれもマルチコアでの実行効率を向上させることを目的とした最適化である．

謝辞　本論文の研究の一部は文部科学省科学研究費補助金　No.25730028 による．

参考文献

[1] Github - onikiri/onikiri2. https://github.com/onikiri/onikiri2.

[2] The llvm compiler infrastructure project. http://llvm.org/.

[3] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pp. 483–485, New York, NY, USA, 1967. ACM.

[4] Marcelo Brandalero and Antonio Carlos S. Beck. Potential of using a reconfigurable system on a superscalar core for ilp improvements. In *Proceedings of the 2014 Brazilian Symposium on Computing Systems Engineering*, SBESC '14, pp. 43–48, Washington, DC, USA, 2014. IEEE Computer Society.

[5] Peng Di, Ding Ye, Yu Su, Yulei Sui, and Jingling Xue. Automatic parallelization of tiled loop nests with enhanced fine-grained parallelism on gpus. In *Proceedings of the 2012 41st International Conference on Parallel Processing*, ICPP '12, pp. 350–359, Washington, DC, USA, 2012. IEEE Computer Society.

[6] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pp. 365–376, New York, NY, USA, 2011. ACM.

[7] Lars Gesellensetter and Sabine Glesner. *Interprocedural Speculative Optimization ofMemory Accesses to Global Variables*, pp. 350–359. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[8] 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. 336–32, 2012.

[9] John Lu and Keith D Cooper. Register promotion in C programs. In *ACM SIGPLAN Notices*, Vol. 32, pp. 308–319. ACM, 1997.

[10] G. S. Murthy, M. Ravishankar, M. M. Baskaran, and P. Sadayappan. Optimal loop unrolling for gpgpu programs. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pp. 1–11, April 2010.

[11] 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.

[12] O. Ozturk, M. Kandemir, and S. H. K. Narayanan. A scratch-pad memory aware dynamic loop scheduling algorithm. In *9th International Symposium on Quality Electronic Design (isqed 2008)*, pp. 738–743, March 2008.

[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. Master's thesis, 電気通信大学, March 2015.

[14] 中江哲史. リネーミング・ロジックを排除する STRAIGHT アーキテクチャのためのコンパイラ技術, March 2016. 東京大学（卒業論文）.

[15] 中江哲史, 入江英嗣, 坂井修一. STRAIGHT コンパイラにおける不要コードの削減手法の検討 (コンピュータシステム). 電子情報通信学会技術研究報告 = IEICE technical report : 信学技報, Vol. 116, No. 177, pp. 25–30, aug 2016.

[16] 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.

[17] 服部直也, 峯博史, 坂井修一, 田中英彦. 関数間最適化による冗長メモリアクセスの削減. 情報処理学会研究報告計算機アーキテクチャ（ARC）, Vol. 2001, No. 76, pp. 73–78, jul 2001.