Vol.2020-ARC-241 No.5
2020/7/30

STRAIGHTコンパイラにおける ループおよび関数呼び出し最適化の評価

中江 哲史[1]　小泉 透[1]　杉田 脩[1]　入江 英嗣[1]　坂井 修一[1]

概要：STRAIGHT はソース・オペランドを命令間の距離で指定する命令セットアーキテクチャである．距離によるソース・オペランドの指定によって，STRAIGHT はリネーミング・ロジックを単純化でき，省電力や性能向上を実現している．命令間距離は STRAIGHT コンパイラのコード生成時に決定するが，ループや関数呼び出しの処理がからむ場合，命令間距離を静的解析のみでは計算出来ないコンシューマ命令が現れる．この場合はコードに転送命令やスピル命令を加える必要があり，追加方法によっては性能低下を引き起こす．本論文では STRAIGHT コードにコンパイルした CoreMark ベンチマークにループアンローリングとインライン展開を施し，これらの最適化手法が STRAIGHT の性能に与える影響を明らかにした．

キーワード：コンパイラ，最適化，ループアンローリング，インライン展開

STRAIGHT Code Optimization and Evaluation: Loop Unrolling and Inlining

Satoshi Nakae[1]　Toru Koizumi[1]　Shu Sugita[1]　Hidetsugu Irie[1]　Shuichi Sakai[1]

Abstract: STRAIGHT is an Instruction Set Architecture where source operands are identified by distance between instructions in its assembly. STRAIGHT realizes low energy consumption and high performance by simplifying the renaming logic in the source operand rule. In this rule, STRAIGHT compiler determines the distances in the code generation process. Static analysis, however, cannot calculate all the distances because of consumer instructions, especially, when there are loop and function calling codes. Forwarding and spilling instructions should be added and may cause performance degradation. In this research, we revealed how these optimization effects on STRAIGHT code, by using loop unrolling and inlining to CoreMark Benchmark program compiled to STRAIGHT codes.

Keywords: compiler, code optimization, loop unrolling, inlining

1. はじめに

プロセッサの性能向上において，1 クロックで命令レベル並列性（ILP）をどれだけ抽出できるようになるかは大きな課題である．スカラプロセッサから始まり，ハードウェアにより ILP 抽出能力を上げるパイプライン化，スーパースカラ化が実現されてきた．これらの技術は搭載可能なトランジスタの増加とともに，回路の規模を拡大することでプロセッサの性能を向上させてきた．

スーパースカラプロセッサの性能を支える技術の 1 つに，Out-of-Order（OoO）実行がある．命令の発行順を入れ替えることで，プログラムの ILP をより効率よく抽出することが可能になる．しかしながら OoO 実行に必要な回路はスケーラビリティに欠け，同時実行命令数を増やすことが難しい．

スケーリングが難しい回路のひとつに，物理レジスタの

[1] 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

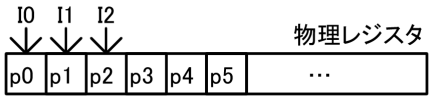

図 1　STRAIGHT アーキテクチャのレジスタ割り当て

```
I0: ADDi [zero] 1      //0 + 1
I1: ADDi [zero] 2      //0 + 2
I2: ADD  [1] [2]    //1(I0) + 2(I1)
```

図 2　STRAIGHT アセンブリの例

割り当て情報を管理するリネーミング・ロジックがある．スーパースカラプロセッサでは論理レジスタの再利用に伴い，ILP の抽出を妨げる偽の依存が生じてしまう．この偽の依存を解消するよう物理レジスタを割り当てなおす回路がリネーミング・ロジックである．

リネーミング・ロジックは多ポート RAM で構成されており，命令がフェッチされる際は例外なくアクセスされる．RAM の回路面積はポート数の 2 乗に比例して増大し，また消費電力もそれに伴って増加する．1 クロックにリネームできる命令数はポート数に比例するため，これを増やすことが難しくなる [10].

そこで我々は，偽の依存を起こさない機械語をプロセッサの入力とする STRAIGHT アーキテクチャ [5], [6] を開発している．STRAIGHT はレジスタを番号ではなく命令間の距離を指定することで，各命令の実行結果を参照する．よって STRAIGHT アーキテクチャでは偽の依存への対処が不要となり，Register Mapping Table（RMT）を省きつつも ILP を最大限に抽出可能である．先行研究 [1] では，消費電力を抑えながら性能を向上させることが分かっている．

ところが STRAIGHT アセンブリの性質上，制御が合流するようなコードではデータ転送命令の追加が必要になる場合がある．加えて関数呼び出しをまたぐようなオペランド参照では命令の距離が計算できないため，特定の変数をスピルする必要がある．

本論文ではループアンローリングと関数のインライン展開という 2 つのコード最適化手法を適用し，STRAIGHT アセンブリの転送命令，スピル命令の数がどう増減するかを確認した．さらにこれらの命令数の変化が，IPC や実行サイクル数にどう影響するか評価を行った．

2. STRAIGHT アーキテクチャ

2.1 レジスタ割り当て

STRAIGHT アーキテクチャのレジスタ割り当ては**図 1** のように動作する．STRAIGHT は OoO 実行のスーパースカラプロセッサであるため，1 クロックに複数命令がフェッチされるものとする．

レジスタ割り当て方式の特徴は以下のとおりである．

- デスティネーション・レジスタは命令が読み込まれた順、つまりフェッチ順に割り当てられる
- ソース・レジスタはレジスタの番号ではなく，命令間の距離で指定される

デスティネーション・レジスタの番号は Register Pointer（RP）によって行われる．RP は 1 命令がフェッチされる度に 1 加算される特殊レジスタである．従って命令はそれぞれ別の物理レジスタに結果を書き込むことになり，Read After Write（RAW）においては読み出されるレジスタが後続の命令に上書きされることがない．

ソース・レジスタの距離には，物理レジスタ数未満の上限を設ける．仮に物理レジスタ数以上の距離を参照したとき，結果が上書きされてしまい参照できない．よって物理レジスタ数未満の距離に制限することで，実行時には偽の依存が発生しないことを保証できる．

ソース・レジスタ番号 SrcReg の計算は，以下のように行う．

$$\mathrm{SrcReg} = (\mathrm{RP} - \mathrm{Distance} + \mathrm{MAX\_RP})\%\mathrm{MAX\_RP}(1)$$

ここで Distance は命令間距離，MAX_RP は距離の上限を意味する．

このような特徴のもと，STRAIGHT アーキテクチャはレジスタ管理に RAM を用いないリネーミング・ロジックを搭載する．

2.2 STRAIGHT アセンブリ

STRAIGHT のリネーミング・ロジックを実現するため，STRAIGHT アセンブリは従来のアセンブリと比較し，以下の特徴を備える．

- デスティネーション・レジスタの指定が不要である
- ソース・レジスタは命令間距離を意味する
- 定数やオペレーション・コードは従来と同等である

図 2 のようなコードを仮定し，STRAIGHT アーキテクチャでの演算を考える．ソース・レジスタにあたる [] で囲まれた数値は距離である．ただし，[zero] はゼロレジスタを意味する．また ADDi 命令はレジスタと即値の加算，ADD はレジスタ同士の加算を意味する．

I0 ではゼロレジスタと 1 が加算，I1 ではゼロレジスタと 2 が加算され，それぞれのデスティネーション・レジスタには 1 と 2 が書き込まれる．I2 では [1] が 1 フェッチ前の命令を意味するため，I1 のデスティネーション・レジスタがソース・レジスタとして読み出される．同様に [2] は 2 フェッチ前より I0 の結果が読み出された後，I2 のデスティネーション・レジスタには 1+2 の結果である 3 が書き込まれて計算が完了する．

2.3 命令間距離の計算と RMOV 命令

STRAIGHT コンパイラ [7] は Static Single Assign-

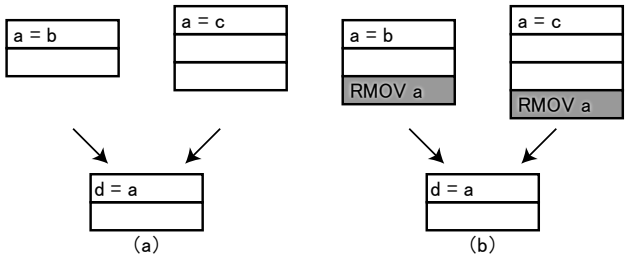

図 3 基本ブロック合流時の距離調整

ment[13]（SSA）形式の LLVM IR[11], [12] を中間表現として採用している．LLVM IR では，プログラムを基本ブロックの集合として扱う．基本ブロックは命令列であり，特に制御が入ってくる場合は必ず先頭の命令を通り，停止や分岐は必ず末尾の命令を通るような命令列である．

この LLVM IR から STRAIGHT アセンブリを出力するため，STRAIGHT コンパイラの処理は，

- ソース・レジスタの生成アルゴリズム
- 呼び出し規約

の 2 点が従来のコンパイラと異なる．

ソース・レジスタの生成においては，命令間距離の計算と，距離が上限を超えて出力されないような制限が行われる．命令間距離の計算の基本方針を以下に述べる．はじめに LLVM IR の先頭からフェッチ順に命令を調べ，デスティネーション・レジスタ番号をフェッチ順と共に記録する．次に同じ命令のソース・レジスタを調べる．記録したデスティネーション・レジスタと名前が一致した際は，現在のフェッチ順からデスティネーション・レジスタのフェッチ順を減算することで命令間距離が求まる．

ところが**図 3**（a）に見られる変数 a のように，制御が合流する際はどの制御パスを通ったかにより，算出された距離が一致しなくなる．この場合は図 3（b）に示す通り，変数 a が基本ブロック末尾から同じ距離になるよう RMOV 命令を使って値をコピーする．当然 RMOV は STRAIGHT 特有の追加命令になる．RMOV を最適な数まで削減する手法も存在する [3] が，STRAIGHT アーキテクチャの仕様上，ループなどの構文では距離の調整が必須となる [2]．

2.4 呼び出し規約

STRAIGHT アーキテクチャにおける関数呼び出しは，**図 4**（a）に示すとおりである．引数と返り値は命令間距離に基づいた次の仕様が定められており，コール・スタックではなくレジスタでの受け渡しが基本になる．

- callee への引数は，順番に JAL 命令の直前に並べる
- caller への返り値も，順番に JR の直前に並べる

従って callee の先頭にある命令は，JAL による caller のアドレスを距離 1 のレジスタで参照できる．引数の参照には距離 2 以前を指定すればよい．引数の数が距離の最大

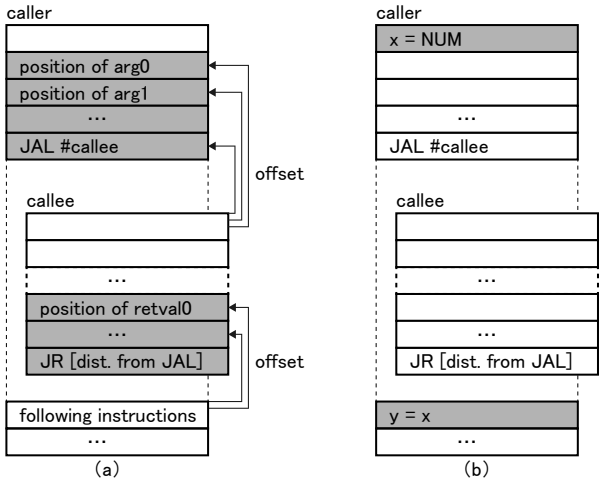

図 4 STRAIGHT アーキテクチャの呼び出し規約

値を超える場合には，従来のアーキテクチャ同様にスタックを利用することで引数の受け渡しを保証できる．同様に callee から caller へ復帰した直後の命令は，距離 2 以前を指定することで返り値を特定できる．

ところが図 4（b）における変数 x のような参照が起きた場合，STRAIGHT 固有のスピルが発生する．具体的に考えるため，命令 y=x に注目する．y=x の実行は callee の呼び出し後であるが，x=NUM は callee の呼び出し前である．従って callee 関数内で実行される命令の数がコンパイル中に確定できない限り，式 y=x が x を参照するための距離を定めることが出来ない．

そこで STRAIGHT では，関数呼び出しをまたいで変数の参照を行う際，caller の実行中に生存している変数は全てスピル命令での退避を行う．これも 2.3 節の RMOV 命令と同様に，STARAIGHT の呼び出し規約を定めたが故に追加を迫られる命令である．

そこで本論文では，距離ではない従来のレジスタ参照方式でも効果が認められるコード最適化手法を用い，STRAIGHT 固有の追加命令にはどの程度効果があるかを明らかにする．制御の合流による RMOV 命令に対してはループアンローリングを，関数呼び出しによるスピルに対しては関数のインライン展開を用いることで，固有のオーバーヘッドの削減を狙う．

3. コード最適化による STRAIGHT コードの変化

3.1 ループアンローリングを用いた RMOV の削減

STRAIGHT ではソース・オペランドが距離で表されることにより，基本ブロックの合流点で RMOV を追加しなければならない．よって合流点自体をコード最適化により減らすことが出来れば，オーバーヘッドの軽減が可能である．

基本ブロックの合流が起こる処理の代表例に，ループ文

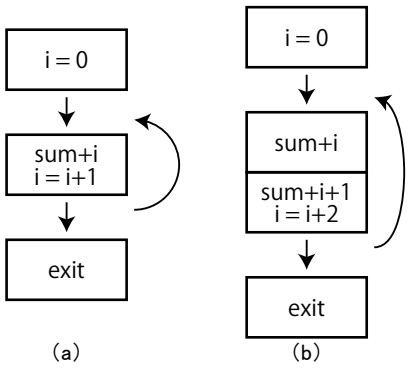

図 5 ループ文の基本ブロック構成

がある．N を偶数として以下の疑似コードを考える．

```
for (i = 0; i < N; i++) {
  sum = sum + i;
}
```

このループ文の実行を基本ブロックで考えると，**図 5**（a）のように制御が毎ループ合流点を通過する．合流点を通過する場合は，2.3 節に従って追加された RMOV 命令が実行される．

ここで，コードにループアンローリングを適用する．ループアンローリングはプログラム中の ILP を高め，静的にも動的にもスケジューリングの効率を上げる最適化手法である．2 段のループアンローリングを施したループ文は，以下のとおりである．

```
for (i = 0; i < N; i=i+2) {
  sum = sum + i;
  sum = sum + i + 1;
}
```

基本ブロックの実行は図 5（b）のように行われ，合流点の通過回数は約 1/2 に削減される．従って STRAIGHT コードに対するループアンローリングは，RMOV の追加量を削減し，実行速度を向上させることが期待できる．

3.2 関数のインライン展開

STRAIGHT コードにおいては 2.4 節で述べた通り，ある変数が関数呼び出しをまたいで参照される場合は変数の退避が必須となる．これは，caller 側が callee で実行される命令数をコンパイル時に確定できないためである．

ここで関数のインライン展開を利用する．インライン展開とは，callee 関数の呼び出しを，callee 関数本体の記述で置き換えるコード最適化手法である．結果関数本体を基本ブロック単位で処理することができ，パスによっては命令数を計算できる場合も出てくる．よって STRAIGHT コードにインライン展開を施せば，スピルする変数の数を同じかそれ未満に抑えたコードを生成できる．

例えば以下のような関数呼び出しの疑似コードを考える

```
int GetNumber() {
  return 2;
}

void main() {
  int x = NUM;
  int number = GetNumber();
  int y = x;
}
```

ここで GetNumber 関数は main 関数によって呼び出される．つまり main 関数が caller にあたり，GetNumber 関数が callee にあたる．

インライン展開を行わない場合，main 関数での変数 x は，GetNumber 関数の呼び出しをまたぐ．よって STRAIGHT コードでは，x がスピルされてしまう．

このコードに関数のインライン展開を施すと，以下のコードが得られる．

```
void main() {
  int x = NUM;
  int number = 2;
  int y = x;
}
```

GetNumber 関数の呼び出しは number=2 の初期化文に書き換わる．再び変数 x に注目すると，スピルせずにソース・オペランドで距離 2 を指定すれば正しく読み込めることが分かる．

よって STRAIGHT コードにインライン展開を施すことで，スピル命令，すなわちロード命令やストア命令数の削減が出来る．

4. 評価

4.1 評価環境

STRAIGHT コードには制御の合流や関数の呼び出し規約により，固有の命令を追加しなくてはならない．これらの命令をループアンローリングや関数のインライン展開で削減できるか確認するため，最適化を施していない STRAIGHT コードと施した STRAIGHT コードを命令セットシミュレータで動作させ，それらの実行命令数と実行にかかったサイクル数を測定した．

STRAIGHT コードとして入力するプログラムには CoreMark ベンチマークを用いた．CoreMark を STRAIGHT コンパイラのフロントエンドである clang で，C 言語ソースコードから LLVM IR へ変換したのち，STRAIGHT コンパイラのバックエンドで STRAIGHT コードへ変換した．

ループアンローリングを適用する段数は，1 段（ループアンローリングなし），2 段，4 段，最大限の 4 通りのいずれかとした．この段数の指定は，C 言語ソースコード中に pragma を記述することで行った．インライン展開の適

表 1　プロセッサパラメータ

Logical register	128
Fetch width	2
Scheduler capacity	16
Issue width	2
Load / Store queue	Load 16 / Store 16
Reorder buffer	64
Commit width	3
L1 Cache	32KB+32KB, 8way, 64B line, 4 cycle hit latency
L2 Cache	256KB, 8way, 64B line, 12 cycle hit latancy
Prefetcher	stream prefetcher, 16 entry, distance 8, degree 2

用は，フロントエンドである clang にコマンドライン引数を与えることで実施した．インライン展開の強度は指定せず，インライン展開ありとインライン展開なしの 2 つのうちどちらかとした．

STRAIGHT アーキテクチャのシミュレーションには，鬼斬弐 [8] 上に実装された STRAIGHT シミュレータを用いた．プロセッサのパラメータは**表 1** の通りである．

4.2　ループアンローリングのコードへの影響

図 6 よりループアンローリングの段数と，RMOV 命令の実行回数を確認する．unroll-max は最大限にループアンローリングを行った場合であり，clang で O2 最適化を設定したものである．unroll-4，unroll-2 は clang の O2 最適化を元に，#pragma unroll を用いてループアンローリングを抑制することで設定した．unroll-1 は clang の O2 最適化に-fno-unroll-loops オプションを加えた．

unroll-max での RMOV 実行命令数は，unroll-1 での RMOV 実行命令数の 99%である．unroll-4 の RMOV は unroll-2 の RMOV よりも若干数が多く，RMOV のみに注目すると逆効果の結果である．しかし**図 7** でループアンローリングの実行サイクル数に注目すると，ループアンローリングの段数が増加するにつれ減少していることが分かる．

この特徴は CoreMark ベンチマークにおいて，ループアンローリングが施される CRC アルゴリズムの関数呼び出しが原因である．前提として clang の O2 オプションを用いてループアンローリングの段数を調整しているため，入力 STRAIGHT コードには関数のインライン展開がかかっている．32bit の CRC を計算する関数は，16bit の CRC を計算する関数を呼び出しており，またそれは 8bit の CRC を計算する関数を呼び出すといった具合に，入れ子の関数呼び出しにより実装されている．unroll-2 の場合，ループアンローリングされた 8bit の CRC 関数は，16bit の CRC 関数と 32bit の CRC 関数のどちらにも展開されていた．対して unroll-4 でループアンローリングされた 8bit の CRC 関数は 16bit の CRC 関数にインライン展開されているものの，32bit の CRC 関数にはインライン展開されておらず，32bit の CRC 関数を呼び出すコードとなっていた．

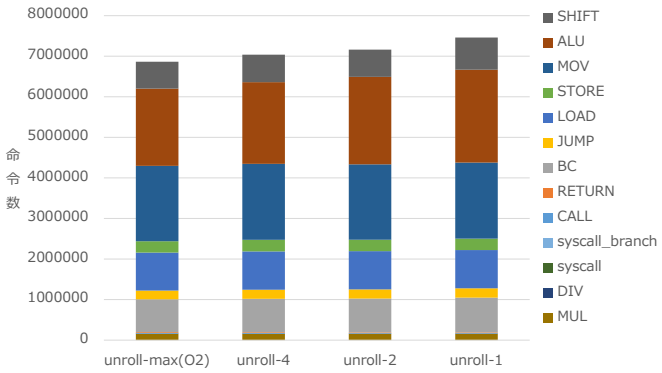

図 6　ループアンローリングによる命令数の変化

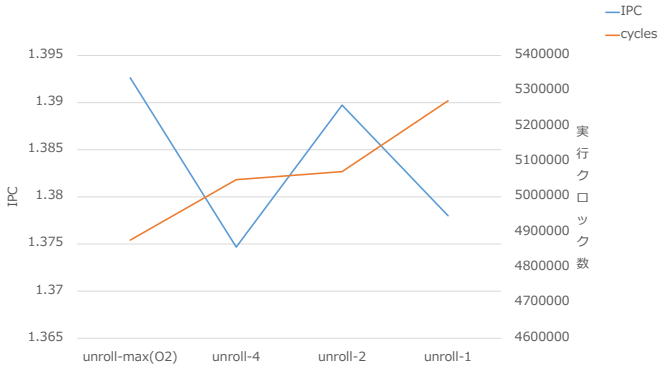

図 7　ループアンローリングによる IPC と実行命令数の変化

つまり，ループアンローリングを適用することによりコードが増大し，コンパイラの自動インライン展開を阻害したことが，unroll-4 の RMOV 実行命令数が unroll-2 の RMOV 実行命令数より多い原因である．

4.3　関数のインライン展開によるコードへの影響

関数のインライン展開による各命令数の変化は**図 8** に示すとおりだった．インライン展開のオンオフは，clang の O2 最適化に-fno-inline を加えることで設定した．ループアンローリングは unroll-max と unroll-1 の設定を用い，インライン展開のオンオフと組み合わせた．

ループアンローリングのオンオフに関係なく，ストア命

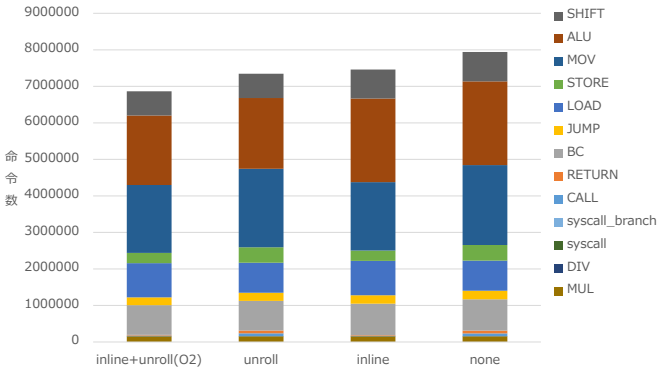

図 8 関数のインライン展開と，ループアンローリングの組み合わせによる命令数の変化

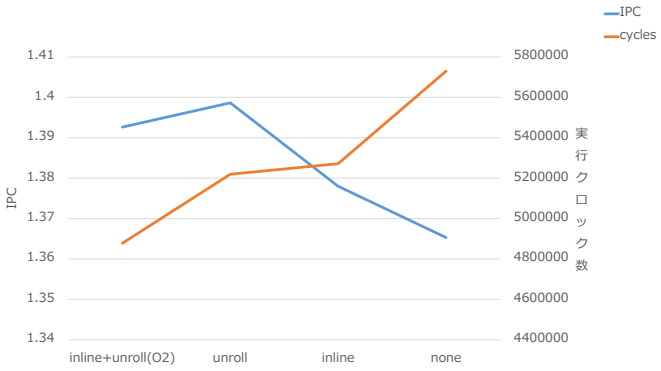

図 9 関数のインライン展開と，ループアンローリングの組み合わせによる IPC と実行命令数の変化

令は 34%削減された一方で，ロード命令は 14%増加した．

IPC と実行サイクル数は**図 9** のように変化した．どちらの最適化もかければ少なくとも実行サイクル数が短縮されることが分かる．

以上よりループアンローリングも関数のインライン展開も，出来る限り施すことで実行性能が向上することが分かった．ただしループアンローリングは関数のコードサイズを拡大させてしまうため，インライン展開の妨げとなる場合があることを確認した．これらの最適化の適切なバランスは，従来のレジスタ参照型アーキテクチャと異なり，新たにチューニングが必要なことが示唆された．

5. おわりに

本論文では STRAIGHT コンパイラで追加される転送命令やスピル命令を，コード最適化により削減できるかどうかを明らかにした．ループアンローリングでは必ずしも最適化の強さと RMOV の減少が対応はしなかったが，実行サイクル数で考えると改善が見られた．関数のインライン展開ではストア命令の大幅な削減が認められた．

ループアンローリングは，場合によっては関数のインライン展開を妨げる作用をしてしまうことも判明した．本論文ではこのような作用が実行性能の低下を招くことはなかったが，仮に性能の低下が認められる場合はバランスするポイントを見極めていく必要がある．

謝辞 本研究の一部は科研基盤研究 (B)，課題番号 19H04077 の助成による．

参考文献

[1] H. Irie, T. Koizumi, A. Fukuda, S. Akaki, S. Nakae, Y. Bessho, R. Shioya, T. Notsu, K. Yoda, T. Ishihara and S. Sakai: *STRAIGHT: Hazardless Processor Architecture Without Register Renaming In Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pp. 121-133, Piscataway, NJ, USA, 2018. IEEE Press.

[2] T. Koizumi, S. Nakae, A. Fukuda, H. Irie, and S. Sakai, *Reduction of instruction in-crease overhead by straight compiler, in 2018 Sixth International Symposium on Computingand Networking Workshops (CANDARW)*. IEEE, pp. 92-98, 2018

[3] 小泉 透, 中江 哲史, 福田 晃史, 入江 英嗣, 坂井 修一, *Straight* 向け冗長転送命令の削減, 研究報告システム・アーキテクチャ *(ARC)*, vol.2018, no. 2, pp. 1–10, 2018.

[4] 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.

[5] 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

[6] Hidetsugu IRIE，Daisuke FUJIWARA，Kazuki MAJIMA and Tsutomu YOSHINAGA：*STRAIGHT : Realizing a Lightweight Large Instruction Window by Using Eventually Consistent Distributed Registers, Networking and Computing (ICNC), 2012 Third International Conference on*, pp.336-342

[7] 中江 哲史, 入江 英嗣, 坂井 修一：*D-6-16 STRAIGHT* アーキテクチャのためのコンパイラ技術 *(D-6.* コンピュータシステム, 一般セッション*)*, 電子情報通信学会総合大会講演論文集 *2016*, p.70

[8] 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

[9] 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，電気通信大学 *(*修士論文*)*，March，2015，pp.1-69

[10] S. Petit, R. Ubal, J. Sahuquillo and P. Lopez, *Efficient Register Renaming and Recovery for High-Performance Processors, in IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 7, pp. 1506-1514, July 2014, doi: 10.1109/TVLSI.2013.2270001.

[11] C. Lattner and V. Adve, *LLVM: a compilation framework for lifelong program analysis & transformation, International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, San Jose, CA, USA, 2004, pp. 75-86, doi: 10.1109/CGO.2004.1281665.

[12] The LLVM compiler infrastructure. http://llvm.org.

[13] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. *Global value numbers and redundant computations. In Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (POPL ’88). Association for Computing Machinery, New York, NY, USA, 12-27. DOI:https://doi.org/10.1145/73560.73562