Vol.2016-ARC-221 No.5
2016/8/8

STRAIGHTコンパイラにおける不要コードの削減手法の検討

中江 哲史[1]　入江 英嗣[1]　坂井 修一[1]

概要：プロセッサの性能を向上させるため，プログラムの実行時に命令を入れ替える OoO（Out-of-Order）実行を始めとしたさまざまな機構が実現されてきた．我々はこのような技術で性能を向上させつつも電力消費を削減するような OoO 実行のアーキテクチャとして，STRAIGHT アーキテクチャを開発している．OoO 実行においてはその効果を上げるためにレジスタリネーミングという技術を用いるが，STRAIGHT アーキテクチャでは電力消費を抑えるために，レジスタリネーミングをコンパイラの処理によって省略しているという特徴がある．ところが，STRAIGHT アーキテクチャのコンパイラでは正しい動作を保証するために，何もしない NOP 命令を追加しなくてはならない．本論文ではこのような NOP 命令をデータの依存関係を保存できる範囲で他の命令に置き換える手法を提案し，サイズが削減された等価なコードの生成を実現する．またこれをもとに，STRAIGHT アセンブリによるアーキテクチャの性能への影響について考察する．

1. はじめに

プロセッサの性能向上を目指して，複数の命令を並列に処理するスーパースカラをはじめとした技術が活用されてきた．トランジスタの微細化に伴い，スーパースカラプロセッサはより多くの回路を搭載することで処理効率をさらに向上させている．スーパースカラを支える技術の 1 つに，プログラム中の命令の順番を入れ替えて実行する技術である OoO（Out-of-Order）実行がある．命令の順番を入れ替えることで，プログラム中の並列に実行できる命令，つまり ILP（Instruction Level Parallelism）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，同時に処理する命令を増やすことを狙うような規模の拡大は難しい．例えば OoO 実行を支える技術のひとつに，データを保持するレジスタを割り当てるリネーミング・ロジックがある．プロセッサでレジスタを使いまわすことを考えた場合，OoO 実行においては ILP の抽出を妨げる偽の依存が生じてしまう．したがって偽の依存を解消するようレジスタを割り当てなおすことで，より OoO 実行の効率を上げる技術がリネーミング・ロジックである．ところがリネーミング・ロジックにおいてレジスタを管理する表である RMT（Register Mapping Table）は，多くのポートを必要とする RAM（Random Access Memory）で構成されており，アクセス頻度も高い．特に RAM の回路面積はポート数の 2 乗に比例することから消費電力の増大をまねき，プロセッサの回路規模を拡大する際のボトルネックになっている．

そこで我々は，偽の依存を起こさないアセンブリを仮定することでリネーミング・ロジックを簡略化する，STRAIGHT アーキテクチャ [1], [2] を開発している．STRAIGHT アセンブリはレジスタをレジスタ番号ではなく，いくつ前に読み込まれた命令かを指定することで各命令の結果を参照することができる．そのため，STRAIGHT アーキテクチャは RMT を省きつつも ILP を最大限まで抽出することが可能であり，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．

ところがこのようなアセンブリを出力するコンパイラは，そのアルゴリズムの性質上何も処理を行わない NOP 命令を加える必要がある．これはリネーミング・ロジックの簡略化によって可能となった性能向上を妨げるオーバーヘッドとなってしまう可能性がある．本論文ではこのような NOP 命令がどの程度の負荷となっているかを調査し，これをなるべく削減する最適化手法を検討，STRAIGHT コンパイラによる性能低下をできる限り防ぐことを考える．

2. STRAIGHT アーキテクチャ

2.1 ハードウェアの動作

STRAIGHT アーキテクチャは OoO 実行のアーキテクチャである．このアーキテクチャでの物理レジスタの割り

[1] 東京大学大学院情報理工学系研究科

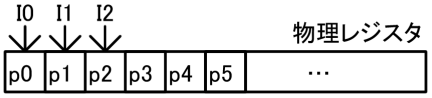

図 1　レジスタの割り当て方

```
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

当て方には，以下のような規則性を持たせている．説明にあるデスティネーション・レジスタとは命令の結果を保持するレジスタであり，ソース・レジスタは命令で処理するデータを保持するレジスタのことである．割り当ての動作は**図 1** のようになる．

- デスティネーション・レジスタを命令が読み込まれた順、つまりフェッチ順に割り当てる
- ソース・レジスタはレジスタの番号ではなくさかのぼる命令の数，つまり距離で指定する

デスティネーション・レジスタの指定は命令のフェッチの度に加算される RP（Register Pointer）によって行われる．ソース・レジスタを参照する際，物理レジスタの数を超えた距離の命令は結果が上書きされてしまい参照できないので，参照する距離の上限はコンパイラによって保証する．

このような特徴により，STRAIGHT アーキテクチャでは偽の依存を起こさないことが保障される．またレジスタ割り当ての規則により，レジスタの番号を RMT に保存せずに RP からいくつ前の命令かを減算することで結果の参照ができるため，RMT を省いたリネーミング・ロジックが実現可能となる．

2.2　アセンブリの仕様

上記の動作を実現する STRAIGHT アセンブリは従来のアセンブリと違い以下のような特徴がある．

- デスティネーション・レジスタを指定しない
- ソース・レジスタはさかのぼる命令の数で表現する
- 定数やオペレーション・コードはそのまま

具体的な動作を考えるため，**図 2** のようなコードを例にあげる．ここで [] で囲まれた数値はソース・レジスタであることを示しており，特に [zero] は常に 0 を保持している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足し算，ADD はレジスタ同士の足し算を意味する．

I0 ではゼロレジスタと 1 が，I1 ではゼロレジスタと 2 が足し算され，それぞれデスティネーション・レジスタには 1 と 2 が保存される．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・レジスタをソース・レジスタとして指定する．同様に [2] は 2 フェッチ前より I0 を指定しているため，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 が保存され，正常に計算ができたことになる．

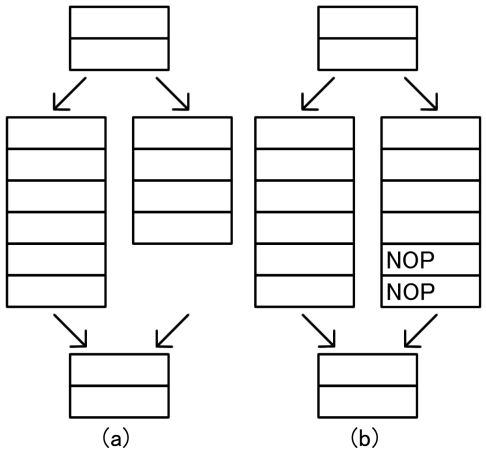

図 3　if-else 文における NOP の利用例

2.3　STRAIGHT コンパイラ [3]

コンパイラによって STRAIGHT アセンブリを出力する際，ソース・レジスタの生成に向けて従来のコンパイラとは異なるいくつかの処理を行う．STRAIGHT コンパイラの中間表現では SSA（Static Single Asignment）形式が採用されており，プログラムは基本ブロックの集合によって表現される．基本ブロックとは，初めと終わりにしか分岐命令が存在しないよう区切られた，数命令のまとまりのことである．

ソース・レジスタを生成する基本方針として，初めにデスティネーション・レジスタの行番号を記録する．次に，記録したデスティネーション・レジスタと同じ名前を持つソース・レジスタを発見した際，行番号の引き算を行うことによってソース・レジスタを生成する．ただし，**図 3** (a) のように if-else 文などの分岐が合流する，特に合流後の命令が合流前の結果を参照する際に距離が合わない場合はさらなる処理が必要である．

現在これは何も処理を行わない命令である NOP 命令を初めに末尾から挿入し，図 3 (b) のようにそれぞれの距離を合わせることで解決されている．このときに調整した距離は後の処理で変わってしまうことを避けるため，各基本ブロックが保持する fixed という変数で管理されている．

これは従来では RMT で解決できていた分岐時のソース・レジスタ参照を，STRAIGHT では距離で解決しているために行うことになった処理である．よって NOP 命令はレジスタ・リネーミングを簡略化した際の STRAIGHT アセンブリ特有のオーバーヘッドといえるので，NOP 命令の数をなるべく少なくすることが必要になる．

そこで我々は，以下のような最適化手法を用いて NOP 命令を削減することを考える．

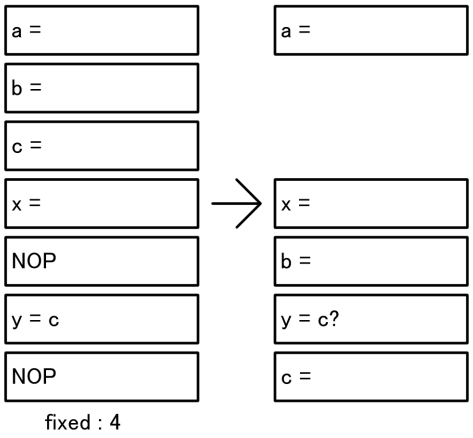

図 4　NOP 命令の置き換えの例

- NOP 命令を他の必要な命令に置き換える
- STRAIGHT アセンブリ特有の命令である RPINC を利用する

本論文ではこのような操作をした STRAIGHT アセンブリを用いてオーバーヘッドがどのように減少するかを観察し，各手法の妥当性を検討する．

3. NOP の消去

3.1 NOP 命令の置き換え

初めに，NOP 命令を同じ基本ブロック中の命令で置き換えて数を減らすことを考える．**図 4** のような操作をするときは fixed が 4 なので，末尾から 4 命令以内の x と y をデスティネーション・レジスタとする命令の距離は変わっていない．このように NOP 命令は命令の距離を合わせるためだけに追加されたものなので，他の命令に置き換える操作は距離という意味では動作に支障をきたさない．ただし，命令を置き換えるということはその命令のフェッチ順が変わってしまうため，フェッチ順の変化が起きても影響のない命令に限って操作が行える．もし命令を入れ替えたときにソース・レジスタが同名のデスティネーション・レジスタよりも前に来てしまった場合，つまり図 4 の右側のような場合はデータの依存関係が守られないため，この命令を NOP と置き換えることはできない．

NOP 命令は 3.1 節で述べた通り基本ブロックの末尾から追加されているため，NOP 命令より前の命令と入れ替える場合のみを考えればよい．実際の処理を考えると，まずは基本ブロックをフェッチされる順番に読み込み，NOP が含まれていたものに対して逐次処理を行う．NOP 命令を 1 つずつ取り出して置き換える命令を探していくことになるが，フェッチ順の変化をなるべく起こさないためにも NOP を処理する順番と置き換える命令を探す順番には工夫が必要である．

先に置き換える命令を探す順番について考える．仮にフェッチ順の早い命令から置き換え対象にすることを考えると，**図 5**（a）のように，NOP を処理する順番がどうであれ基本ブロックの先頭から 1 つめの NOP 命令の前までの命令を抜かすことになる．一方でフェッチ順の遅い命令から置き換えることにすると，図 5（b）のように抜かす命令は末尾から fixed 個の命令のうちのいくつかとなる．これは置き換える命令を先頭から探した場合も同様にまたぐ必要がある命令であるために，どちらの順番を取ろうと抜かす命令であることには変わりない．したがって，置き換える命令を探すのはフェッチの遅い順にした方が効率がよいことが分かる．

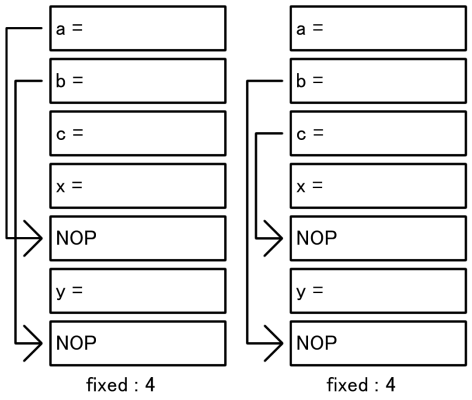

図 5　置き換える命令を探す順番の例

次に NOP を処理する順番であるが，**図 6**（a）のように仮にフェッチの早い順に処理を行ったとする．ここで 1 つめの NOP を置き換えたこととし，2 つめの NOP 命令を処理することを考える．すると置き換えに使う命令は全て NOP よりフェッチ順が速く，また置き換える命令はフェッチ順の遅いものから選んでいるため，2 つめの NOP 命令と置き換えられる命令は必ず 1 つめで置き換えられた命令を抜かさなくてはならない．逆に図 6（b）のように末尾の NOP から置き換えていけば抜かすのは固定された NOP でない命令のみとなるが，これも置き換える命令と同様どちらの順番を取ろうと結局抜かす命令である．したがって，NOP も末尾から取り出して処理を行う方が効率の良いことが分かる．

このように入れ変える命令を決定するが，実際に命令のフェッチ順が変化した場合，レジスタの指定にはどのような影響が起こるのかを考える．ソース・レジスタに関してはデスティネーション・レジスタを生成する命令は必ずその命令よりも前に存在するので，特に影響はない．一方，デスティネーション・レジスタが後続の命令のソース・レジスタを抜かしてしまった場合はデータの依存関係が守られていないことになる．よって基本ブロックの命令を末尾からチェックし，ソース・レジスタとなっているレジスタ名をすべて記録していく．入れ替えることができる命令かどうかを判定する際，デスティネーション・レジスタの名前を記録と照合する．名前が存在しない場合はソース・レ

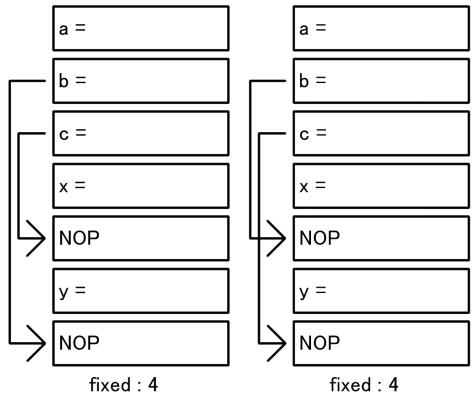

図 6　NOP を処理する順番の例

ジスタとデスティネーション・レジスタの順番が変わることは起こり得ないため，NOP と置き換えることが可能となる．

以上のような操作によって，基本ブロック内の NOP 以外の命令数は変わらず，NOP 命令は入れ替えた数だけ減らせることになる．

3.2　RPINC 命令の利用

RPINC 命令とは，STRAIGHT アーキテクチャにおいてデスティネーション・レジスタを指定する働きを持つ RP を加算する命令である．RP の加算は基本的に命令のフェッチ時に 1 だけ行われるが，この命令を利用することにより 1 命令で RP を 2 以上動かすことが可能になる．

実際の処理を考えると，1 つの NOP 命令を RPINC 命令で置き換える場合は，NOP 命令が 1 つ減って RPINC 命令が 1 つ増えるだけなので無意味である．したがってこの手法は NOP 命令が 2 つ以上並んでいる場合に効果を発揮する．例えば NOP 命令が 3 連続で並んでいる場合，3 つの NOP を削除し 1 つの RPINC と置き換えることで NOP 命令が 2 つ削除できたことになる．

3.3　NOP の置き換えと RPINC の組み合わせ

RPINC 命令はもともと中間表現に存在しない命令という意味では NOP 命令と同等の負荷になる．したがって NOP を削除するという目的のもとでは，なるべく前にある命令を置き換えるという作業が優先されるべきである．ただし，NOP を置き換える手法にも限界が存在し，フェッチ順の制約により Basic Block 内で置き換えに使える命令が NOP よりも少ない場合が生じる．よってこれらを組み合わせる場合には NOP を置き換えた後，NOP が残ってしまった場合に PRINC を利用することを考える．

単純には 3.1 節で述べたように NOP は下から順に処理されるため，例えば図 **7**(a) では NOP が 2 つ消去され，PRINC は使用されない．ここで連続している数が少ない NOP から処理していくことを考えると，図 7(b) では

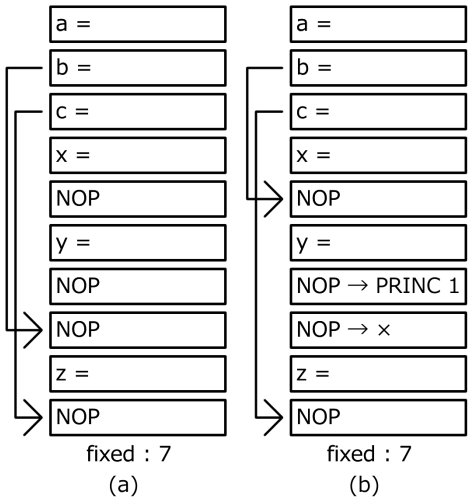

図 7　NOP 置き換えの工夫による効率化

PRINC と組み合わせて NOP が 3 つ消去できたことになる，よって NOP を入れ替える際には連続している数が少ない，かつ末尾にあるものから処理していくことでより効率のよいコードが得られる．

4.　評価

4.1　最適化による NOP 命令の削減効果

ここまでで議論した最適化を STRAIGHT コンパイラに実装し，ベンチマークである Livermore Loops をコンパイルして STRAIGHT アセンブリを得た．Livermore Loops とは，Kernel と呼ばれるループを用いたプログラムの集まりである．またアセンブリの正しさの確認には鬼斬弐 [4] と呼ばれるプロセッサ・シミュレータを基に実装された STRAIGHT シミュレータ [5] を使用し，1000 ループを正常に実行できた Kernel の値のみを利用してデータを算出した．結果を示したグラフの横軸は全て Kernel の番号と平均を表している．

まず，最適化を施す前の STRAIGHT アセンブリに含まれる NOP 命令は，図 **8** のような割合になった．縦軸は %であり，全命令数に対する NOP 命令の数を示している．これによりコード領域において平均で約 3.9 %を NOP 命令が占めていることが分かった．

続いて，NOP 命令を他の命令と置き換える最適化を施した結果が図 **9** のようになった．縦軸は割合であり，最適化前の NOP 命令数に対する最適化後の NOP 命令数を示している．この時各 Kernel の NOP 命令が平均で 67 %，最大で 33 %まで減っていることが明らかになった．

正常に実行が完了した Kernel において NOP 命令が 2 以上連続になっているものは存在しなかったため，効果は計測できなかった．

4.2　NOP の削減による IPC の変化

IPC（Instcutions Per Cycle）とは，実行命令数を実行に

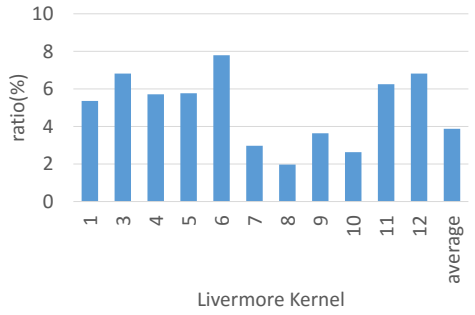

図 8　各 Livermore Kernel における NOP 命令の割合

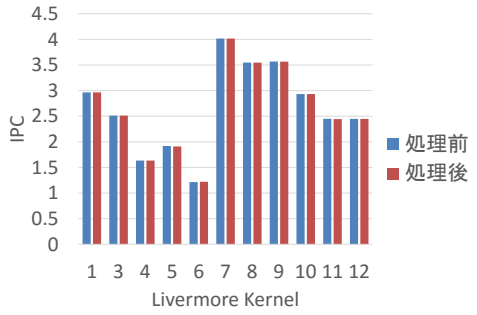

図 10　最適化による IPC の変化

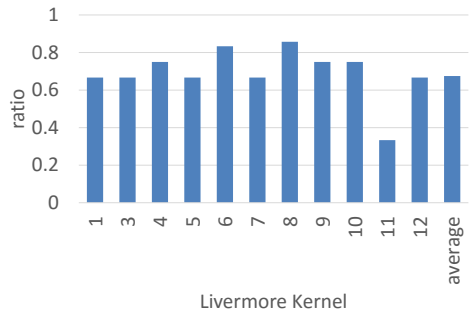

図 9　他の命令との置き換えによる NOP の減少割合

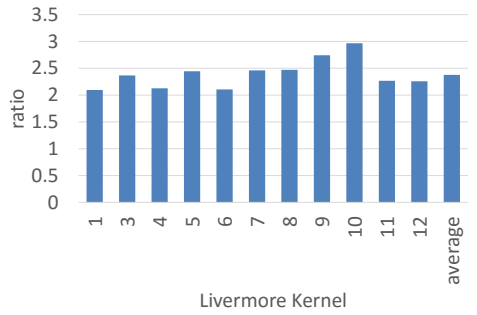

図 11　Alpha アーキテクチャに対する STRAIGHT の相対性能

かかったサイクル数で割ったものである．STRAIGHT アセンブリにおいて，NOP の削減を行う最適化の前後で IPC は図 **10** のようになり，ほとんど変化しないことが分かった．これは NOP が他の命令に置き換わることは単純に命令数の減少，それに伴って実行に必要なクロック数の減少も引き起こすために影響がなかったと考えられる．よって連続の NOP が検出されなかったことも含めて考えると，STRAIGHT コンパイラにより生成されるアセンブリは最適化前の段階でも十分に NOP 命令が少なく，従来のアーキテクチャと比べて著しく不利になるような負荷とはならないことが分かった．

4.3　相対性能評価

ここまでで STRAIGHT コンパイラ特有のオーバーヘッドとその性能に対する影響が分かったので，他のアーキテクチャとの性能比較をより正確に行うことを考える．ここでは IPC を実行命令数で割った値，つまり Kernel 実行にかかったサイクル数の逆数を用いてデータの算出を行った．比較対象には鬼斬弐に初期で実装されている Alpha アーキテクチャを利用しており，コンパイル時には最適化をかけていない．Alpha アーキテクチャを 1 とした相対性能を縦軸にとると，計算した結果は図 **11** のようになった．

平均では約 2.4 倍の性能向上を示しており，Kernel10 においては最大の約 3.0 倍もの性能向上が見られた．

5.　おわりに

本論文では STRAIGHT コンパイラによって生成される NOP 命令の影響を調査し，また NOP 命令の数をなるべく減らす方法を検討，その効果を検証した．実際に最適化を行うとアセンブリ中の NOP 命令の数が平均で 67 % にまで抑えられ，狙い通りに動作することが示された．また NOP 命令の性能への影響を IPC に基づいて考えると，少なくともベンチマークの Livermore Loops のコンパイルではほとんど負荷となっておらず，STRAIGHT コンパイラにより生成されるアセンブリは最適化前でもオーバーヘッドが十分に少ないことが判明した．

加えて Alpha アーキテクチャとの比較では約 2.4 倍もの性能向上が得られることが分かった．特に STRAIGHT アーキテクチャでは従来のアーキテクチャよりも多くのレジスタを搭載することが想定されているが，そのレジスタ数を利用することでメモリアクセス数を減らすような最適化を設計すれば，OoO 実行のプロセッサの性能をさらに向上させることが期待できる．

謝辞　本研究の一部は科研費若手 (B)25730028 の助成による．

参考文献

[1]　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
[2] 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
[3] 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
[4] 塩谷 亮太：プロセッサ・シミュレータ「鬼斬弐」，*http://www.mtl.t.u-tokyo.ac.jp/˜onikiri2/wiki/*
[5] 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