ソフトウェア工学 126−13
(2000. 3. 10)

プロセッサの仕様に適応可能な
コンパイラのための汎用最適化器

長谷川 勇[†] 鈴木正人[†] 今泉貴史[††]

コンパイラのオプティマイザはプロセッサ毎に作成しなければならず、開発効率に問題があった。これは、バックエンドで行われる最適化がプロセッサの命令に依存する一方で、各プロセッサのライフサイクルが短いためである。

本稿では、パイプライン、スーパースカラなどのプロセッサの並列性を利用した最適化を行なう汎用最適化器の構成法を提案する。目的はプロセッサのアーキテクチャに関する形式的な記述を用いて最適化を行う汎用最適化器を提案、実装することである。汎用最適化器は、仕様を入れ換えることで各プロセッサに適用できるため、開発効率に優れる。

Generic optimizer for compilers which can adapt specification of processor architecture

ISAMU HASEGAWA ,[†] MASATO SUZUKI [†] and TAKASHI IMAIZUMI[††]

Development of optimizing compiler is hard and inefficient work because of its dependency to the architecture or the instruction set of the target processor whose lifecycle is somtimes relatively short.

In this paper we propose the generic optimizer which generates code with advantage of pipeline and superscaler. Our goal is building a flamework for generic optimizer adaptable to any processor with formal specification of its architecture.

1. はじめに

1.1 背景

現在、最適化コンパイラはプロセッサ毎に新たに作成されている。これはバックエンドで行う最適化が、プロセッサ固有の特徴を利用するためである。その一方で、新しいプロセッサや既存のプロセッサを改良したプロセッサの開発が次々に行われている。このため以前に比べ各プロセッサのライフサイクルは格段に短くなった。従って、ライフサイクルの短いプロセッサごとに最適化器を新たに作らなければならないとすると開発効率の点で問題がある。そこで、プロセッサのアーキテクチャの記述を用いた汎用的な最適化を行えることが望ましい。

こうした汎用的な最適化に関する研究は以前から行われているが、これらの研究の多くは生成する命令列の効率を評価する基準として、命令列中の命令数や、各命令にかかるクロック数の和を用いている。しかし、最近のプロセッサはパイプラインやスーパースカラなどの技術により並列処理を行うため、命令数やクロック数を減らしても実際の実行速度が速くなるとは限らず、逆に遅くなることもある。そこで、最適化時にプロセッサの並列性を考慮することが必要である。

1.2 目的

本研究は汎用最適化器の構成手法の提案と実装を目的とする。汎用最適化器とは、外部からプロセッサに関する形式的な記述を与えることで、そのプロセッサに適した最適化を行う最適化器を言う。この汎用最適化器はプロセッサに非依存であり、プロセッサに関する記述を入れ替えるだけで新しいプロセッサに対応でき、開発効率の点で従来の最適化器よりも優れている。なお、最近のプロセッサをターゲットとするために、最適化はプロセッサの並列性を考慮して行う。

以上から、本研究で提案する最適化器は以下の 2 つの特長を持つことを目標とする。

- プロセッサの並列性を有効に利用する

† 東京工業大学情報理工学研究科
Graduate School of Information Science and Engineering, Tokyo Institute of Technology
†† 東京工業大学理工学研究科
Graduate School of Science and Engineering, Tokyo Institute of Technology

- ターゲットプロセッサの変更を容易に行える

2. 最 適 化

2.1 アプローチ

本研究で行う最適化は、プロセッサの並列性を有効に利用するために、パイプラインで発生するストールの回避と、スーパースカラによりプロセッサが複数持つ計算資源の利用効率を向上させることを目指す。その手段としては以下の 2 つを用いる。

- 命令列の置換
  与えられた命令列に対して、その順序を入れ替えることによる最適化である。命令列の置換により、パイプラインでストールを起こすタイミングで発行される命令を遅らせ、ストールを回避することが可能である。また、同じ計算資源を利用する 2 命令の間隔を離し、異なる計算資源を利用する 2 命令を隣接させることで計算資源の利用効率の向上も可能である。
- 命令列の変換
  与えられた命令列を意味が等価な命令列に入れ替えることによる最適化である。命令列の変換により、パイプラインでストールを起こす命令を別の命令に変換することでストールを回避することが可能である。また、同じ計算資源を利用する 2 命令が隣接するときに、片方を異なる計算資源を利用する命令に変換することで計算資源の利用効率の向上も可能である。

具体的には、まず入力であるソースを基本ブロックを単位とする命令列に分解する。ここで基本ブロックとは先頭から実行が始まり、途中で停止、分岐することなく最後の命令まで順次実行される命令列をいう。そしてそれぞれの命令列に対して全探索を行い、最適な命令列を得る。このとき探索空間は、最適化の対象である命令列に対して置換、変換を適用することで得られる命令列のうちもとの命令列と意味が等価なものすべてである。全探索による最適化では実行に膨大な時間がかかるように思われるが、この場合探索空間は高々数命令の命令列に対する置換と変換であり、またいくつかの枝狩りを行うため、実用上十分に短い時間で探索は終了する。

2.2 効　　果

ここでは本研究が行う最適化について、その効果を実例を示し説明する。

本研究では命令列の置換、変換により最適化を行うと述べた。それでは命令列の置換、変換により、どのような効果が得られるだろうか。特に命令列の置換によりコードの実行速度を向上できることは直感的には理解しにくい。しかし、本研究では命令列の置換、変換により以下の 2 つの効果が得られると考える。

- パイプラインで発生するストールの回避
- スーパースカラを採用するプロセッサの計算資源の利用効率の向上

以下では、これらの効果について述べる。

2.3 ストールの回避

2.3.1 Pentium Pro/II のデコーダ

Pentium Pro/II プロセッサのコアは RISC プロセッサであり、このコアは μop と呼ばれる RISC 命令を実行する。このため Pentium Pro/II はパイプラインの前半部分に、デコードのステージを持ち、x86 互換の CISC 命令を μop に変換する。

このデコードステージには、3 つのデコーダが用意されており、これらが 1 clock 毎に可能なら 1 つずつマクロ命令をμop にデコードする。これらのデコーダはそれぞれ能力が異なっており、デコーダ 1 は 4μop 以下のマクロ命令、デコーダ 2,3 はそれぞれ 1μop のマクロ命令をデコードできる。これらのデコーダには、in order にマクロ命令が投入されるため、デコーダ 2 でデコードができない場合ストールが発生し、次の clock で改めてデコーダ 1 に投入される。

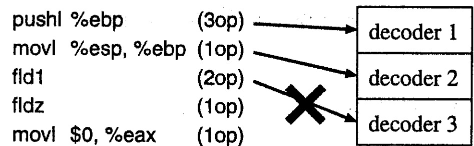

図 1　ストールの生じる例

2.3.2 デコーダでのストールの例

図 1 のプログラムを考える。このプログラムは fld1 をデコードするとき、デコーダ 2 で fld1 をデコードできないためストールが発生する。このプログラムに対して命令列の置換を行い、図 2 に変更する。すると実行結果は等価で、かつデコーダでのストールが発生せず、実行速度が向上したプログラムが得られる。

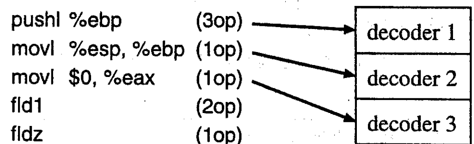

図 2　ストールの回避

例として、恣意的にストールを多く発生するコードを記述した。このコードに、本研究で試作した最適化器を使用し、最適化前後の実行速度を比較したところ、12.5% 高速化された。

以上の例から、命令列の置換によりプロセッサのパイプラインで発生するストールを回避し、最適化が可能であることが分かる。同様に、ストールを発生する命令を含む命令列を意味が等価である命令列に変換することで、ストールを回避することが可能である。

2.4 計算資源の利用効率の向上

alpha21164 プロセッサ[1] は 4 命令分のプリフェッチバッファを持つ。命令は 4 命令ずつプリフェッチバッファに入り、命令はここから各パイプラインへと発行される。

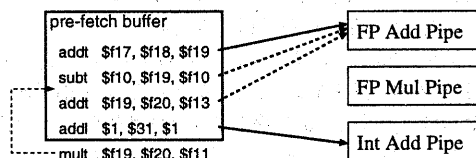

図 3 パイプラインの競合

ここで、図 3 のプログラムを考える。この例では、まず最初の 4 命令がプリフェッチされ、発行できる命令から順次パイプラインへと発行される。このときプリフェッチバッファの最初の 3 命令はすべて 浮動小数の加算、減算命令であるため同一のパイプラインを使用する。従って、プリフェッチバッファの命令をすべて発行するのに 3 clock 必要であり、例のプログラムの 5 命令目以降をプリフェッチするのはその後である。

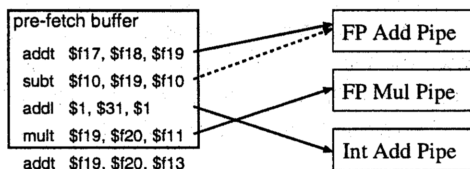

図 4 命令列の置換による競合の解消

ここで、命令列の置換を考える。図 4 に示すプログラムは図 3 のプログラムの 5 つ目の命令を 2 つ目の命令の後ろへと持ってきた例である。このプログラムにおいて、最初のプリフェッチされる 4 命令のうち、同一のパイプラインを使用するのは 2 命令だけなので、この 4 つの命令は 2 clock で発行可能である。こうして、命令列の置換を行うことで、計算資源を効率よく利用し、実行速度を向上できる。同様に命令列の変換を行い、パイプラインの競合を回避することで、計算資源の利用効率の向上が可能である。

2.5 依存解析

本節では、依存解析について、その目的、従来の方法と本研究で行っている方法について説明する。

2.5.1 依存解析の目的

本研究では最適化の手段の 1 つとして命令列の置換を行う。このとき置換が最適化となるためには、新しい命令列の意味が置換前の命令列の意味と等価である必要がある。そのため置換前の命令列から、意味が変わらないための依存関係を抜きだし、それらの依存関係が保存される置換を選択しなければならない。

それでは、どのような依存関係を解析すればよいだろうか。まず例として、図 5 を考える。

S_1 :　`addl %ebx,%eax`　; eax := ebx + eax
S_2 :　`movl %eax,%ecx`　; ecx := eax

図 5 依存関係

この例では、命令 S_1 がレジスタ eax に対して書き込みを行い、S_1 より後ろにある命令 S_2 が、S_1の書き込んだレジスタ eax を読み込んでいる。このとき、S_2 は S_1 の結果を使用しているので、S_2 は S_1 よりも後ろになければならず、もし入れ換えたら命令列の意味が変わってしまう。以上より、この S_1 と S_2 の間の依存関係は意味を等価に保つために、置換後も保存されている必要がある。

この例と同様に、置換により意味を等価に保つための依存関係を抜きだすことが依存解析の目的である。しかし、依存関係は置換を行う上での制約条件でもあるので、依存する必要のない依存関係を選ぶと逆に探索の範囲をせばめることになってしまう。

以降では、まず従来の依存解析の例を示す。そして、その依存解析が生成する条件では制約が強すぎることを述べ、本研究で提案する依存解析について述べる。

なお、本研究の依存解析ではデータフローのみを考える。なぜなら、本研究が置換の対象とするのは基本ブロックの命令列であり、コントロールフローについて考える必要がないためである。

2.5.2 従来の依存解析

データフローを考慮した依存解析には、以下の 3 つの依存関係を考える方法がある。

- Read after Write
  命令 S_jが書き込む結果を、S_jよりも後ろの S_iが読み込み使用する場合。
- Write after Read

```
S1 :  movl -4(%ebp),%eax    ; eax := [ebp − 4]
S2 :  addl %eax,%ecx        ; ecx := eax + ecx
S3 :  movl $0,%eax          ; eax := 0
S4 :  movl %eax, 8(%ebp)    ; [ebp+8] := eax
S5 :  movl $1,%eax          ; eax := 1
```

図 6 依存解析の例

```
T1  (= S1) :  movl -4(%ebp),%eax
T2  (= S3) :  movl $0,%eax
T3  (= S2) :  addl %eax,%ecx
T4  (= S4) :  movl %eax, 8(%ebp)
T5  (= S5) :  movl $1,%eax
```

図 7 Write after Read が保存されず、意味が等価でない例

命令 S_jが読み込み使用するリソースに対し、S_jよりも後ろの命令 S_iが書き込む場合。

- Write after Write
  命令 S_jが書き込むリソースに対し、S_jよりも後ろの命令 S_iも書き込む場合。

これらの 3 つを依存関係と考える。すると、任意の置換により得られる命令列は、これらの依存関係が保存されている場合もとの命令列と等価である。しかし、この依存関係の定義は十分条件ではあるが、必要条件ではない。

例として図 6 に示す命令列を考える。S_2 の add 命令は、eax レジスタを読み、S_3 の mov 命令は、eax レジスタに書いている。そのため S_2, S_3は Write after Read の関係にある。従って S_3 は S_2 に依存するので、S_3 は S_2 よりも後ろでなければならない。例えば、図 7 は図 6 の命令列にある置換を適用した命令列を表すが、これらの命令列の意味は等価ではない。このとき $T_3(=S_2)$ は $T_2(=S_3)$ よりも後ろであり、S_2, S_3 間の依存関係が保存されていないことが分かる。

しかし、意味が等価であるために、必ずしも S_3 が S_2 より後ろである必要はない。例えば、図 8 に示す命令列を考える。これは、図 6 の命令列に対し、置換を適用した命令列である。このとき、図 8 の命令列では、S_3, S_2間の Write after Read の依存関係が保存されていないにもかかわらず、実行結果は同じである。

以上の例から、Write after Read は置換により意味が等価であるために保存が必ず必要な依存関係ではないことが分かる。さらに図 6, 8 において、S_2 がない場合を考える。すると、図 6 で S_3, S_1は Write after Write の関係にあるが、図 8 ではこの依存関係が保存されていないが意味は等価である。従って、Write after Write も置換により意味が等価であるために保存が必要な依存関係でないことが分かる。

```
U1  (= S3) :  movl $0,%eax
U2  (= S4) :  movl %eax, 8(%ebp)
U3  (= S1) :  movl -4(%ebp),%eax
U4  (= S2) :  addl %eax,%ecx
U5  (= S5) :  movl $1,%eax
```

図 8 Write after Read が保存されていないが意味が等価な例

2.5.3 本研究の依存解析

前述した通り Write after Read, Write after Write は必ずしも依存関係とする必要はない。しかし、これらをまったく考慮しないのでは、意味が等価でない置換を制限できない (図 7)。

そのため、必要な依存関係としては図 6, 8 を許し、図 7 を許さない条件が記述できることが望ましい。そこで本研究では、**弱い依存関係**を導入することで、これらの条件を記述する。弱い依存関係 weak-dp(S_3, S_2, S_1) は 3 項間の関係であり、**S_1 が S_2, S_3 より前にあるならば S_3 は S_2 より後ろでなければならない** と定義する。つまり、3 命令の 6 通り並べ方の内、$S_1S_3S_2$ のみ許さない関係である。

例えば、図 6 に示したリストにおいて、Read after Write の関係にある S_1, S_2、Write after Read の関係にある S_2, S_3 の 3 命令を弱い依存関係と考え、weak-dp(S_3, S_2, S_1) とする。このとき、この弱い依存関係が保存されているためには、$S_1S_3S_2$ 以外の順で並んでいればよい。すると図 6、図 8 の命令列ではこの弱い依存関係を保存している。一方、図 7 の命令列ではこの弱い依存関係を保存していない。以上の例から、この弱い依存関係の保存を条件とすることで、図 6, 8 を許し、図 7 を許さない条件が記述できた。

本研究で行う依存解析は、Read after Write を依存関係、Read after Write の関係にある S_1, S_2、Write after Read の関係にある S_2, S_3 の 3 つの命令間の関係を弱い依存関係と考え、これらを求めることで行う。

3. 実　　装

本研究で提案する最適化器の実装を行った。実装は Java で行ない、大きさは合計で 4500 行程度である。

本章では、まずプロセッサ定義として最適化器に与える各種の設定について説明し、次に実装について説明する。

3.1 プロセッサ定義

本研究ではプロセッサ定義としてトークン表, 命令表, レジスタ定義, 変換表, パイプライン定義を与える。本節ではこれらについて順に説明する。

3.1.1 トークン表

トークン表には、ソースリスト中に現れるトークンの定義を正規表現を用いて記述する。これらのトークンはターゲットのプロセッサやアセンブラに依存したものである。表 1 に x86 系プロセッサの GNU Assembler 用に記述したトークン表の一部を示す。

```
IMM            \$(\-?[0-9]+) ...
REG_GROUP_1    %[a-d](l|h)
REG_GROUP_2    (%[a-d]x|%(d|s)i|%(b|s)p)
     :              :
```

表 1 トークン表 (x86-gas)

3.1.2 命　令　表

命令表には、ソースリスト中に現れる命令を記述する。命令表はタブ区切りの表であり、行方向、列方向両方に可変長である。命令表の 1 行目には各列の名前を記述し、2 行目以降に各命令の定義として、それらが読み書きするリソースと、最適化時に必要となるプロセッサに依存するパラメータを記述する。

表 2 に Pentium Pro プロセッサ、GNU Assembler 用に記述した命令表の一部を示す。

3.1.3 レジスタ定義

いくつかのプロセッサでは、別のリソースと領域を共有するリソースを持つことがある。例えば、x86 系の CPU では **ax** レジスタの上位 8 bit を **ah** レジスタ、下位 8 bit を`al` レジスタと言う[2)]。

複数のレジスタが同じ領域を共有するのは、特にめずらしい例ではない。しかし本研究ではリソースへの読み込みと書き込みに基づいた依存解析を行うため、ここで示したリソースの共有の概念を正しく扱う必要がある。本研究では、プロセッサ定義の一部としてレジスタ定義を記述することで、この概念を表す。表 3 に x86 系プロセッサ GNU Assembler 用のレジスタ定義を一部示す。

```
%eax   %al, %ah, %ax
%ax    %al, %ah, %eax
%ah    %ax, %eax
%al    %ax, %eax
  :      :
```

表 3 レジスタ定義 (x86, gas)

3.1.4 変　換　表

変換表は命令列の変換を生成する規則を定義し、最適化器はこの情報をもとに命令列の変換を行う。変換表は 変換前の命令列、変換後の命令列、変換前の命令列が満たすべきオペランドの条件、変換後の命令のオペランドの定義式のリストである。

```
[[[xorl REG_3, REG_3]], [[movl IMM, REG_3]],
[[0.0 == 0.1]], [[0.0 == "$0"], [0.1 := 0.0 ]]],
...
```

表 4 変換表

3.1.5 パイプライン定義

パイプライン定義はパイプラインの動作に関する定義であり、Prolog のプログラムとして与える。

パイプライン定義をプログラミング言語で記述するのは記述が困難になるというデメリットがあるが、本研究では以下の理由により Prolog による記述が必要であると判断した。

例えば、一般的なパイプラインの動作を記述するには、パイプラインの数と、それぞれのパイプラインが処理できる命令の種別とステージ数など、いくつかのパラメータを与えることで十分かも知れない。しかし、研究の目的は汎用最適化器の構成であるため、特殊な機構 (x86 系の最近のプロセッサの持つ命令のデコーダや Out of Order 実行など) にも対応できる必要がある。さらに将来どのような機構が追加されたとしても、それらに対応する必要があるため、これらの機構を ad-hoc に追加するのは好ましくない。そのため、本研究ではパイプラインの規則を一階述語論理で記述し、それを Prolog のプログラムへと変換ことで パイプライン定義としている。なお、いくつかのパラメータから パイプライン定義を自動生成し、それをプロセッサの動作の近似として用いることも可能であり、Prolog による記述が困難であるというデメリットを緩和できる。

パイプライン定義には eval_str(N, T) という名前の述語を記述する。この述語は、与えられた命令列のうち N 番目の命令までを時間 T で実行できる場合は true 、できない場合は false である。また、 パイプライン定義から lookup_int(N, C, V), lookup_str(N, C, V) という述語により命令表を参照する。これらの述語は、与えられた命令列の N 番目の命令を命令表から探し、その C 列目の値を V に返す。

3.2 構　　成

本研究で作成した最適化器は字句解析器、構文解析器、依存解析器、スケジューラから構成される (図 9)。

- 字句解析器, 構文解析器

  字句解析器と構文解析器は、トークン表と命令表

name	operands	read	write	impl_read	impl_write	m_op_size
movl	REG_3,REG_3	0	1			1
	IMM,REG_3		1			1
	IMM,MEMORY	1	1			1
⋮	⋮	⋮	⋮	⋮	⋮	⋮
pushl	REG_3	0		%esp	%esp,(%esp)	3
⋮	⋮	⋮	⋮	⋮	⋮	⋮

表 2　命令表 (Pentium Pro - gas)

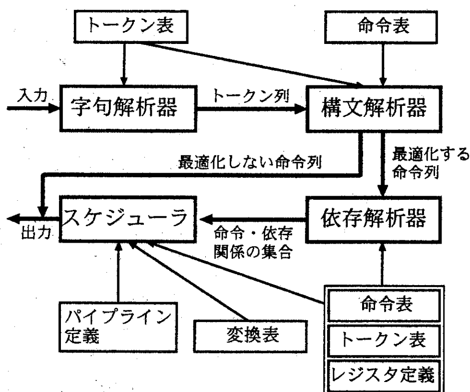

図 9　最適化器の構成

を参照して動作する。これらの定義は外部から与えるので、字句解析器と構文解析器はプロセッサ、アセンブラに非依存である。
また構文解析器は、命令表に定義が記述された命令のみ最適化する命令列として、それ以外は最適化しない命令列として扱う。このため、命令表には最適化に不要なプロセッサの命令や、アセンブラのマクロ命令に関する定義を与える必要がない。

- スケジューラ
  スケジューラは入力の拡張命令列と依存関係の集合から最適化された命令列を生成し、出力する。最適化は与えられた命令列に置換、変換を行い得られる命令列の中から最も効率がよいと判断した命令列を出力することで行う。命令列の評価は、Prolog インタプリタで パイプライン定義を使用し、述語 eval_str により命令列のコストを計算することで行なう。

4. 考　　察

4.1 実　　験

実験は C 言語で記述した十数行程度のプログラムに対して行った。これらのプログラムをコンパイルして得たアセンブリ言語のソースに対し、本研究で試作した最適化器を用いて最適化を行い、最適化前後の実行時間を調べた。なお実験環境は、Pentium II 333MHz BSD/OS 4.0.1 である。コンパイルは gcc 2.7.2.1 で行い、O3 オプションを付けて最大限に最適化を行ったコードに対して本研究の最適化器を適用した。その結果を表 5 に示す。

	最適化前 [s]	最適化後 [s]
素因数分解	7.73	7.68
分数の分解	4.60	4.55

表 5　実行例

この結果から、すでに最適化が行われているコードに対しても、デコーダにおけるストールの回避により実行速度の改善が可能であることがわかる。

4.2 評　　価

本研究で提案した汎用最適化器は以下の特徴を持つ。

- プロセッサの並列性を考慮した最適化
  生成するコードの効率の評価時に、パイプラインのストールやスーパースカラの利用効率を考慮した評価を行うため、命令数のみの基準や、各命令のレイテンシを合計しただけの単純な評価基準に比べ、実際の実行時間に近い評価が可能である。
- プロセッサ、アセンブラに非依存
  外部から与えるプロセッサの定義に従って最適化を行うため、最適化器自身はプロセッサ、アセンブラに非依存であり、プロセッサの定義を入れ換えることで様々なプロセッサに適応できる。この特徴により、最適化器をスクラッチから開発するのに比べ開発効率に優れ、プロセッサ専用の最適化器を容易に構成できる。
- すべての命令の定義を与える必要がない
  命令表には最適化に重要な命令の定義のみを記述すればよく、最適化に関係ない命令やアセンブラのマクロ命令などに関する記述は必要ない。例えば、実験で使用した 命令表の記述量は高々70 行

程度である。また、命令表の記述はオペランドへの読み込み、書き込みなどに関する記述のみで、複雑なセマンティクスの記述を行う必要がない。以上から命令表の記述量は少なく、本研究で提案する汎用最適化器は開発効率に優れる。

- スケーラブルな記述
  プロセッサの定義には最適化の程度に合わせたスケーラブルな記述が可能である。唯一レジスタ定義のみは依存解析のために正確な記述が必要だが、それ以外の記述は正確にすべての記述が必要ではない。例えば、命令表にはすべての命令を記述する必要はなく、パイプライン定義もパイプラインの動作を完全に記述する必要はない。このため、最低限名記述を与えてから少しずつ記述を増やしていく、漸進的な開発も可能である。
- 柔軟な定義が可能
  本研究ではパイプライン定義として Prolog のプログラムを用いる。そのためパイプラインの定義には、柔軟な記述が可能である。例えば、命令のデコードや Out of Order 実行など、それぞれのプロセッサに特有な性質を自由に記述できる。
  一方、パイプライン定義を Prolog のプログラムにしたことで、記述が困難になる。しかし パイプライン定義には完全に正確な記述が必要ではないため、テンプレートを元に自動生成した簡易的な記述でも最適化が可能である。
- 依存解析
  本研究では、Read after Write, Write after Read, Write after Write に基づく依存解析では不可能な置換も可能である。例えば Out of Order 実行の可能なプロセッサがターゲットの場合、命令の順序の入れ替えは意味がないと思われがちである。しかし本研究では行う命令列の置換による命令の順序の入れ替えは、これらのプロセッサでは不可能な置換が可能である。このため探索空間がより広がり、より効率のよいコードを生成できる。
- アセンブリ言語のソースが入力
  本研究で提案する最適化器は、アセンブリ言語のソースを入力とする。このため、以下に示す応用が可能である。
  - バイナリ互換性のあるオブジェクトファイルからの最適化
    例えば、Pentium プロセッサ用のオブジェクトファイルを逆アセンブルし、そこから K6 プロセッサ用の最適化をかけることで、K6 プロセッサに最適化されたコードを得ることができる。
  - 既存の最適化コンパイラと組み合わせた使用
    本研究で提案する最適化器はすでに最適化されたコードに対しても効果がある。このため、既存の最適化コンパイラと組み合わせ、最終段の最適化器としも使用できる。

4.3 問 題 点

本研究で提案した汎用最適化器には、現在以下の問題がある。

- 本研究で提案する最適化器は、プログラム中で並列に実行可能な部分を選択または生成することで最適化を行う。そのため、プログラム中に並列に実行できる部分が存在しない場合は最適化の効果が得られない。例えば、プログラムが単一のアキュムレータへの操作の連続であった場合は最適化の効果が得られない。
- 本研究ではメモリのアドレスが同一かどうかの判定を行わない。そのため、正しい依存関係を解析するために、メモリを 1 つのリソースとして扱う。このため、必要のない依存関係も生成されてしまい、本来可能である置換を制限する場合がある。

4.4 関 連 研 究

- PO[3], HOP[4]
  PO はプロセッサの記述とアセンブリ言語のソースを入力とし、最適化したソースを出力する。最適化はソース内の隣接した複数の命令をシミュレートし、それらを等価な単一の命令で置き換えることで行う。HOP は PO をコンパイラ生成時に用いて、あらかじめ命令列から命令への置き換えをパターンとして生成する。コンパイル時にはこのパターンにしたがって最適化を行う。
  本研究と比較すると、PO や HOP では生成する命令列の評価基準として命令数を使用しているが、最近のプロセッサではこれは評価基準としては問題がある。本研究は命令列の評価基準として、より正確な基準を用いている。
- TOAST[5]
  TOAST はプロセッサに関する記述に従って動作する汎用な最適化コンパイラである。TOAST では、コンパイラの最終段における最適化だけでなく、コンパイラのその他のフェーズでもプロセッサの関する記述を利用する。本研究がコンパイラの最終段での最適化を対象とするのに対し、TOAST ではコード生成から行うため、本研究では不可能な最適化が可能である。
  しかしコード生成から行っているため、プロセッ

サの定義には厳密な定義が必要である。一方、本研究では最適化に必要な記述のみでよいため、本研究と比べると他のターゲットへの移植に必要な記述量が多い。

- GNU Superoptimizer[6)]
  Superoptimizer[7)]はアセンブリ言語のソースを入力とし、全探索を行うことで意味が等価で命令数の少ないソースを出力する。探索空間はターゲットマシンのアセンブリ命令の部分集合の列である。GNU Superoptimizer は、命令列を直接実行する代わりにシミュレーションを行うことで移植性を高め、値が未定義のレジスタを入力とする命令を生成しないなどの方法で探索を改良している。
  Superoptimizer は本研究と同様、全探索による最適な命令列の探索を行う。しかし、その探索空間が大きく異なる。本研究では命令列の変換を別とすれば、命令列内での意味の等価な並べ替えなのでその探索空間の大きさは、命令列の長さを n として $n!$以下である。一方、Superoptimizer はプロセッサの持つ命令を m として m^nである。実用に耐えうるのは n が m よりも遥かに小さい場合なので、Superoptimizer は本研究と同程度の命令列を扱うのは実行時間の点で不可能である。
  また Superoptimizer は生成した命令列が等価かどうかの判定を、いくつかの入力に対するシミュレートで行っているため、各命令のセマンティクスも記述する必要がある。そのため、ターゲットのプロセッサの命令に関して、本研究に比べより多くの記述が必要となる。
- SuperFLD[8)]
  SuperFLD は命令列の評価基準として命令列の実際の実行時間を用いることで Superoptimizer を拡張している。ただし SuperFLD は本研究と異なり、Pentium プロセッサの浮動小数演算に特化している。さらに最適化時に x86 系の FPU に特有の機能を用いているため、これを汎用最適化器に拡張することは難しい。

4.5 課　　題

ここでは本研究の今後の課題を述べる。

- 枝刈り
  本研究では、全探索に基づいた命令列の探索を行うため枝狩りは重要である。現在の実装でもいくつか枝狩りを行ってるが、最適化の実行時間を短縮するために、より多くの枝狩りが必要である。
- アルゴリズムに基づいた探索
  最適化の実行時間を短縮するために、全探索ではなく、なんらかのアルゴリズムに従った探索を行なう方法もある。
  ただし最適解が得られる保証はなくなる。また、探索の途中で複数の候補から最適な命令を選択するためには、ある命令列に対してある命令が続く場合どれくらいのコストがかかるかの評価が必要となりパイプライン定義の記述がより困難になる。
- コード生成
  コード生成の段階から最適化を行う方法がある。この場合、本研究の利点のいくつかが失われるが、より効率のよい最適化が可能となる。
  例えば、本研究で行う命令列の変換や置換はレジスタアロケーションと組み合わせることでより効率のよいコードが生成できる。また、コード生成時の情報をもっている場合、現在の実装の問題であるメモリアドレスの同一性が判定でき、より正確な依存解析が可能である。

参　考　文　献

1) COMPAQ. *Alpha 21164 Microprocessor Hardware Reference Manual*, 1998.
2) Intel Corporation. *Intel Architecture Software Developers Manual, Volume 1 : Basic Architecture*, 1997.
3) J.W.Davidson and C.W.Fraser. The design and application of a retargetable peephole optimizer. *ACM TOPLAS*, Vol. 2, No. 2, pp. 191–202, April 1980.
4) J.W.Davidson and C.W.Fraser. Automatic generation of peephole optimizations. *ACM SIGPLAN Notices*, Vol. 19, No. 6, pp. 111–116, June 1984.
5) R.Hoover and K.Zadeck. Generating machine specific optimizing compilers. *ACM POPL*, pp. 219–229, 1996.
6) T Granlund and R Kenner. Eliminating branches using a superoptimizerand the gnu c compiler. *ACM SIGPLAN Notices*, Vol. 27, No. 6, pp. 341–352, 1992.
7) H Masslin. A look at the smallest program. *ACM SIGPLAN Notices*, Vol. 22, No. 10, pp. 122–126, 1987.
8) 藤波順久. マシン命令パイプラインスケジューリングの超最適化. ソフトウェア科学会第 15 回大会論文集, pp. 217–220, 1998.
9) 長谷川勇. プロセッサの仕様に適応可能なコンパイラのための汎用最適化器の研究. 修士論文, 東京工業大学, 2000.