

種々の最適化の効果のモデル化と
それに基づく最適化列の効果の予測

東京工業大学
情報理工学研究科
数理・計算科学専攻

今橋 孝典
(06M37036)

平成20年度修士論文

指導教官 佐々 政孝 教授

平成21年 1月 30日

研究概要

コンパイラのプログラム最適化に関する研究の中に、phase ordering problem と呼ばれる問題がある。これはプログラムに最適化を適用する際に、複数種類ある最適化の中から、どの最適化をどの順序で適用すればより高いパフォーマンスを得られるか、という問題である。しかし、最適化の効果は、適用先のプログラムや、前後に適用した最適化などの影響によって複雑に変化する。そのため、これまでは最適化の効果是直接予測するのではなく、様々なヒューリスティックな方法を用いて効果の高い一連の最適化（以後、最適化列と呼称する）を探索してきた。

本研究では、個々の最適化の効果を予測するモデルを構築する。そして個々の予測された最適化効果を積み重ねることで、与えられた最適化列全体の効果を予測する。これまでは、プログラムと最適化の相性や最適化同士の複雑な相互作用などのため、個々の最適化の振る舞いをモデル化するのは困難とされてきた。本研究では、最適化の効果に影響を与える幾つかの要因を「中間コードの特徴」に統合することで、最適化の振る舞いのシンプルなモデル化を実現した。具体的には、プログラムの中間コードの特徴と、そのコードへの最適化適用時の効果という「例題」によって学習された人工ニューラルネットワークという形で、予測モデルは構築される。他の類似の研究とは異なり、モデルの構築は前もって行われるため、実際に予測する段階でのコストは極めて低く抑えられている。

また、本手法による予測の効果を評価するために、COINS コンパイラインフラストラクチャ上で本手法を実装し、富士通 PRIMEPOWER 250 上で実験を行った。その結果、本手法を使って求めた最適化列は、COINS 上でよい効果を与えるとされている最適化列 (O2) に比べ、約 2 パーセントほど良い結果を実現した。

また、本手法は与えられたプログラムに対する最適化列の効果を予測するための手法であり、それ自体は「良い最適化列」を求める手法ではない。しかし、本手法を他のアルゴリズムに導入することで、極めて短時間に「良い最適化列」の探索が可能になる。そこで、本手法を用いて良い最適化列の探索を行う実験も行った。その結果、本手法を利用すれば、COINS 上でよい結果を与えるとされている最適化列以上の効果を得られることが示された。また、その予測のためのコストは実行数回程度であり、他の研究と比べても最も少ない部類である。

目次

第1章	はじめに	4
1.1	背景	4
1.1.1	phase ordering problem	4
1.1.2	これまでの研究	4
1.2	本研究の概要	6
1.3	構成	7
第2章	予備知識の説明	8
2.1	コンパイラにおける最適化	8
2.2	基本ブロック	9
第3章	最適化効果の予測モデル	11
3.1	予測モデルの概要	11
3.2	予測モデルの要件	11
3.2.1	予測モデルへの入力	11
3.2.2	予測モデルの出力	13
3.3	中間コードの性格	14
3.3.1	特徴による性格付け	14
3.3.2	その他の性格付け	16
3.3.3	静的な特徴と動的な特徴	16
3.3.4	実行時情報の収集	16
3.4	モデルの計算	19
3.4.1	人工ニューラルネットワーク ANN	20
3.4.2	予測モデルと ANN	22
3.4.3	予測モデルの学習	22
第4章	予測モデルの構築と適用	24
4.1	モデルの構築	24
4.2	モデルを使用した最適化列効果の予測	25
4.2.1	適用例	26
4.3	最適化列の探索	27
4.4	予測のコスト	28
4.4.1	特徴の取得	28

4.4.2	モデルの適用	28
4.4.3	2種類のコスト	29
第5章	実験	30
5.1	実験環境	30
5.1.1	マシン	30
5.1.2	コンパイルインフラストラクチャCOINS	30
5.1.3	最適化	31
5.1.4	プログラムとANNへの例題	33
5.2	実験	34
5.2.1	モデル構築におけるプログラム同士の相性	34
5.2.2	不適なプログラム	36
5.2.3	本手法を利用した、良い最適化列の探索	37
第6章	関連研究	39
6.1	Agakovらの研究	39
6.2	Dubachらの研究	39
6.3	Cavazosらの研究	40
第7章	まとめと今後の課題	41
7.1	精度の改善	41
7.2	今後の展望	42

第1章 はじめに

1.1 背景

コンパイラ技術が生まれてから数十年間にわたって、よりよいコードを生み出すために様々な最適化手法が提案されてきた。しかし同じ最適化でも、適用する状況によって効果が変わることが、これまでの研究により明らかになっている。具体的には、どのようなプログラムに適用するか、また、その前後の最適化が何であるかなどが、効果を変動させる要因となっている。よって、効率の良いコードを得るためには、どのような最適化をどのような順序で適用すればよいかの問題となる。これは最適化の phase ordering problem などと呼ばれる。

なお、本論文のタイトルにある、「最適化の効果のモデル化」とは、最適化の効果を予測するモデルの構築を意味する。

1.1.1 phase ordering problem

本節では、本研究の分野である phase ordering problem に関して概説する。

効果の高い最適化列を求めるのが本研究分野の目標であるが、前述の通り、個々の最適化の効果は適用されるプログラムによって異なる。同様に最適化列全体の効果も適用先のプログラムによって異なる。実際、あるプログラムに対して非常に効果の高い最適化列が、他のプログラムに対しても効果が高いことは稀である [23, 3]。よって、効果の高い最適化列は、与えられたプログラムに対して個別に求める必要がある。

しかし、与えられたプログラムに対して効果の高い最適化列を選択するためには、多くの場合は実際にプログラムを実行する必要があり、その結果時間が掛かることが問題として挙げられる。そのため、少ない時間でより効果の高い最適化列を求めることが望まれる。

1.1.2 これまでの研究

本節では、本研究に関連する既存の研究について述べる。

プログラムと最適化効果の関係、また最適化同士の相互作用は複雑であり、その大部分が未解明なままである。そのため、既存の研究ではこれらの因果関係の

予測・モデル化は限定的な範囲で行われるか、或いはスピル数などの特定の項目に焦点を絞って行われてきた。

今橋ら [13] は、x86 などのレジスタ数の少ないマシンにおいて、最適化の適用によるスピル数の増大を考慮することで、最適化によるパフォーマンスの悪化を防ぐ手法を考案した。これは、与えられた最適化列中の各最適化列に対し、最適化自体の効果とスピル数の増大による悪影響を考慮して、各最適化を実際に適用するか否かを判断するものである。しかしこの手法は、良い最適化列を一から探索するのではなく、与えられた最適化列の中から悪影響を与えるものを取り除くというものであり、探索する範囲は狭い。

また、Cavazos ら [8] はロジスティック回帰分析を用いて良い効果をもたらすと思われる最適化列を探索した。ただし、これも今橋らの研究と同様に、与えられた最適化列をベースに一部の最適化を取り除くというものである。よってやはり、良い最適化列を求めると言っても、限定的なものになっている。

一方、特定の最適化や特定の項目と言った限定的な範囲ではなく、広い最適化空間から良い最適化列を探す研究も行われてきた。これらの手法は、最適化の効果を直接予測するのではなく、ヒューリスティックな手法を用いて良い最適化列を探し求めてきた。主なアプローチとして、二つの手法が上げられる。一つめの主なアプローチは、与えられたプログラムに対して様々な最適化列の効果を何らかの形で評価し、その結果から良い最適化列の傾向を求める手法である。最適化列の評価の仕方は色々あるが、素朴には、最適化列を実際に適用し、出力されたコードの実行時間を計るといった方法などが考えられる。

二つめの主なアプローチは、あらかじめプログラムと良い最適化列の傾向を何らかの形でモデル化しておき、その後与えられたプログラムをこのモデルに適用して良い最適化列を求める手法である。本論文では前者を反復的手法、後者を予測的手法と呼称する。実際には、両者の特徴を併せ持つ研究もある。

反復的手法

前述の通り、プログラムに対して様々な最適化列の効果を評価し、その結果をもとに良い最適化列を見つける手法である。最も素朴な手法としては、ランダムに生成された多く (場合によっては数万種類以上) の最適化列を与えられたプログラムに適用し、出力コードを実際に実行することで最適化列の効果を評価する。そうして得られた結果から、最も高い効果を与えた最適化列を「良い最適化列」として採用する、という手法が考えられる。

この手法は、生成される最適化列の数によっては効果の高い最適化列を求められるだろうが、生成した最適化列の数だけ、実際に適用・実行をする必要があるため、求めるために時間が掛かりすぎるという欠点がある。このように、反復的手法では、最適化列の生成・評価のフェーズ (feedback-loop) を如何にして短くするかが問題となる。

Almagor ら [3] や Kulkarni ら [18] らは、最適化列を完全にランダムに生成するのではなく、遺伝的アルゴリズムを用いて高い効果が見込める最適化列を生成することで、効率的な探索を実現した。

また、Kisuki[16] らはループブロッキング (loop tiling, loop blocking) とループ展開 (loop unrolling) のパラメーターの調節のために反復コンパイルを利用した。最適化のパラメーターをどのようにして決定するかも、phase ordering problem と同様の問題を含んでいる。

予測的手法

一方、予測的手法は、あらかじめプログラムの性格と最適化列の効果の関係を、何らかの形で前もってモデル化しておく。その多くは、多くの最適化列とその効果を調べておき、統計的な手法を用いてモデル化するという手法を採っている。予測的手法は、反復的手法に比べて、良い最適化列の探索に掛かる時間が少なくて済む傾向がある。しかし、探索の結果得られる最適化列の効果は、反復的手法には劣ることも多い。

Cavazos らは、コードに対して特定の数種類の最適化列を適用し、その反応を調べることでコードの特性を分析し、任意の最適化列の効果予測する手法を開発している [7]。また、Dubach らは、与えられたコードに対して幾つかの最適化列を適用し、その結果を基に予測モデルを構築し、そのモデルを使用して最適化列の効果予測する手法を開発した [11]。ただし、この手法はプログラムごとにモデルの構築が必要なため、予測のためのオーバーヘッドが無視できない大きさになる。

前述の通り、反復的手法と予測的手法の両方の特徴をもつ研究もある。例えば Agakov らは、過去の最適化の適用データから機械学習を用いて、高い効果を見込める最適化列を絞り込む手法を提案した [1]。これは過去の最適化列の適用データを用い、プログラムの特性から良い最適化列の傾向を予測するモデルを構築するという点では予測的である。しかし、そうして得られた結果は、feedback-loop の効率化の為に利用される。そのため反復的であるとも言える。

1.2 本研究の概要

本研究では、与えられたプログラムと与えられた最適化列に対して、そのプログラムに最適化列を適用した際の効果を予測するための手法を提供する。本手法は、そのためにまず最適化列を構成する個々の最適化に着目する。つまり、個々の最適化の効果予測モデルを構築し、見積もられた個々の最適化効果を組み合わせることで、最適化列全体の効果を予測することを目指す。

本研究は以下の特徴を持つ。

- ・最適化の種類に関わらず、予測モデルが構築できる

個々の最適化の効果を予測するような既存の研究では、元となる最適化列が固定されていたり、対象となる最適化が限られているといった、限定的なモデル化しかされてこなかった。本研究の手法は、最適化の種類を問わず、モデル化が可能である。

・ 予測する手法である

本研究は、良い最適化列を直接求める手法は提供しない。しかし、予測が出来るという事は、良い最適化の探索を大いに助ける。例えば多くの反復的手法では、様々な最適化列の効果を何らかの形で評価する必要がある。素朴な手法だと実際に最適化を適用して目的コードを実行することになるが、これは非常に時間が掛かる。適用・実行の代わりに本手法の予測を用いれば、殆ど時間を掛けずに最適化列の評価が可能になる。

1.3 構成

本論文の構成を以下に示す。第2章では、本論文を読む上で必要になる予備知識の説明を行う。第3章では本研究の骨子でもある、最適化効果の予測モデルについて、説明を行う。第4章では、第3章で解説した予測モデルを使用して、最適化列の効果を予測する手順について述べる。第5章では、予測モデルの効果を評価するための実験と、その結果について述べる。第6章では、本研究に関連する研究について、本研究との比較を交えながら述べる。第7章では、本研究をまとめ、今後の課題と展望について述べる。

第2章 予備知識の説明

実装にあたって、また本論文の説明にあたって予備知識が必要となる。

2.1 コンパイラにおける最適化

最適化とは、コンパイルにおけるフェーズの一つであり、プログラムの意味を変えずにプログラムのパフォーマンスを上げるために用いられる。パフォーマンスとは、例えばメモリの消費であったり消費電力であったりすることもあるが、多くの場合は実行時速度の事を指す。本研究でも、最終的にこの実行速度を改善する事を主眼に置いている。

最適化には様々なものがある。そして一般的に、コンパイル時に最適化を行う場合には複数種類の最適化が適用される。その最適化フェーズを中心に、コンパイラにおける処理の流れを図示すると図 2.1 のようになる。コンパイラにはまずプ

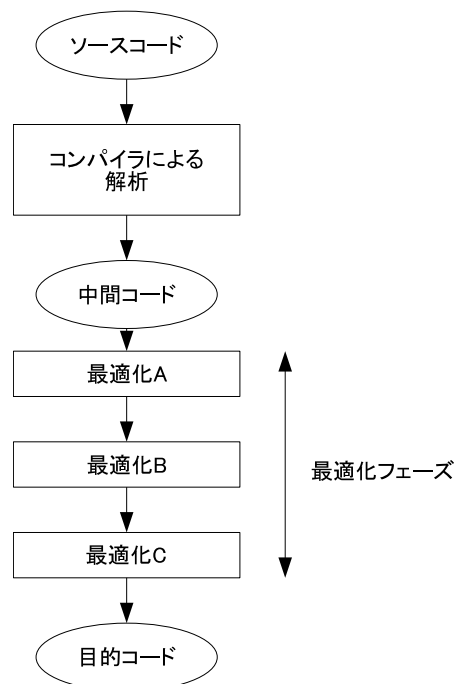


図 2.1: 最適化を中心に見た、コンパイルの流れ

プログラムのソースコードが与えられるが、このソースコードは一旦何らかの中間コードに変換され、最適化はその中間コードに対して作用するものである。図を見れば明らかであるが、最適化にも適用される順番があり、最適化を適用した結果の中間コードが次の最適化器に渡されることになる。このように、大元のソースプログラムは同じでも、最適化器に渡されるコードはそれぞれ異なり、それは最適化の効果にも影響を及ぼす。この事実こそが phase ordering problem の原因となっている。

そして前章で述べた通り、最適化フェーズを構成する一連の最適化の事を、本研究では最適化列と呼ぶ。

2.2 基本ブロック

基本ブロック (Basic Block) とはプログラムの一部分 (ブロック) であり、そのブロックの実行を開始する文が、そのブロックの先頭の文だけであり、末尾が無条件飛び越し、あるいは条件つき飛び越しであるものである。基本ブロック途中への飛び込みや、基本ブロック途中からの飛び出しはない。

基本ブロックと、そのブロック間の関係を示した制御フローグラフは、プログラムを視覚的に表現する方法として現在広く用いられている [2, 20, 4]。図 2.2 の (a) のような C プログラムを基本ブロックに分割したものが (b) である。図中の矩形は基本ブロックを表しており、矩形の左上の L1 等は基本ブロックにつけた便宜的なラベルである。

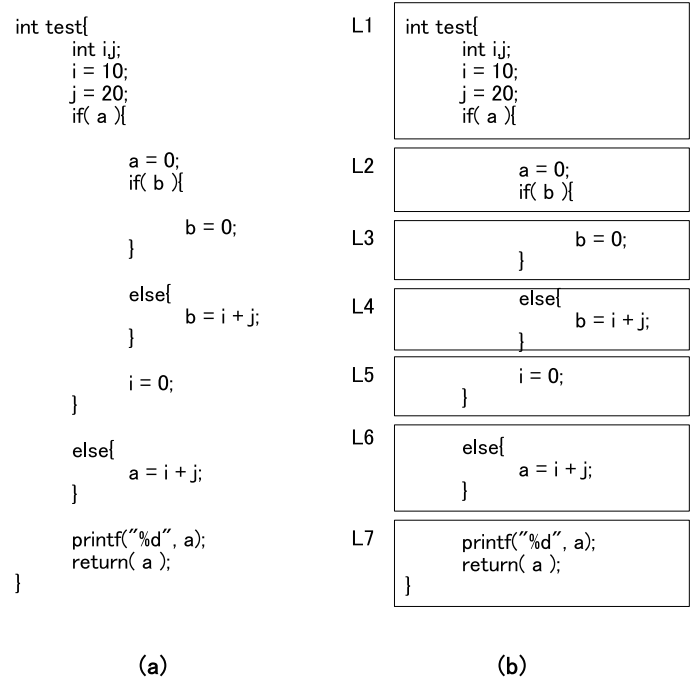


図 2.2: C プログラムと基本ブロックの例

第3章 最適化効果の予測モデル

前章で述べたとおり、本研究の大きなテーマは、最適化効果のモデル化、つまり最適化の振る舞いを予測するモデルの構築である。本章では、このモデルについて詳述する。

3.1 予測モデルの概要

本研究の手法は、最適化の種類に依存しない事を目標としている。しかし、すべての最適化の振る舞いを一つのモデルで表現するのは困難である。よって、本研究では、それぞれの最適化の種類ごとに予測するためのモデルを構築していく手法をとる。例えば最適化列が A, B, C の3つの最適化から構成される場合、最適化 A の効果、最適化 B の効果、最適化の C 効果はそれぞれ個別のモデルによって予測される。

ただし前述の通り、最適化の種類に関わらずモデルの構築が可能な汎用的なものでなくてはならない。そのため、本研究では人手で最適化の効果を解析しモデル化するのではなく、最適化適用時の振る舞いをデータとして取得し、それを機械学習の手法で学習させることでモデルを構築する。

どのようにモデルを構築するかは後で詳述するとして、まずは個々のモデルがなにをどのように予測していくかを述べる。

3.2 予測モデルの要件

最終的に、予測モデルを使って、最適化列全体の効果を予測したい。そのため、予測モデルは何を入力として受け取り何を出力すべきかを述べる。

3.2.1 予測モデルへの入力

前述の通り、既存の研究は個々の最適化の効果の予測には踏み込んでいない。もしくは、限定的にしか踏み込んでいない。これは、最適化の効果が二つの要素、つまり

1. 与えられたプログラム

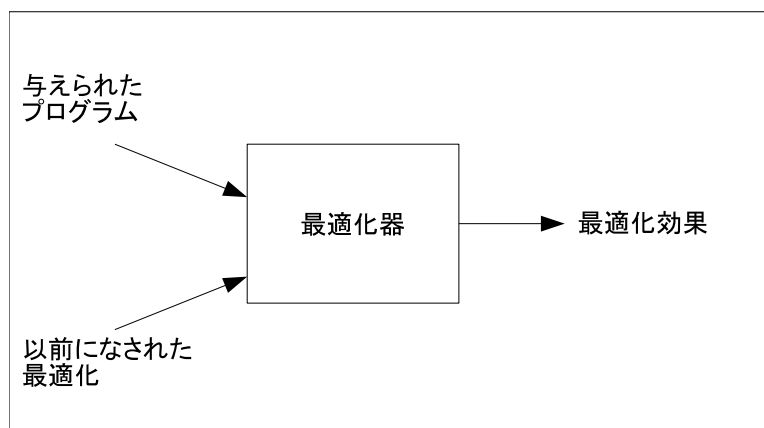


図 3.1: 最適化の振る舞いを決定する因子 (従来のとらえ方)

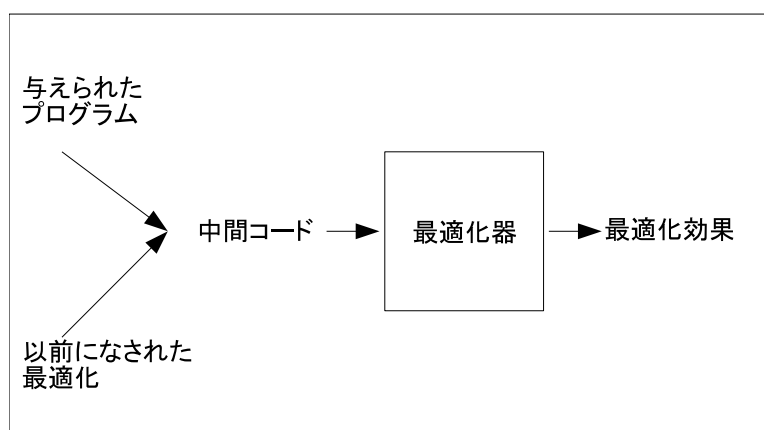


図 3.2: 最適化の振る舞いを決定する因子 (本研究でのとらえ方)

2. 以前に適用された最適化

によって決定されており、このことが最適化効果の予測の複雑さを増しているからである。しかし、実際のコンパイラの動作に着目すれば、各最適化器に与えられるのは、プログラムのコンパイル中の中間コードであることがわかる。もちろんその中間コードは、与えられたプログラムとそれまでに適用された最適化の結果である。それゆえに結局、上述の二つの要素は、「最適化に与えられる中間コード」というただ一つの要素に帰着させることができる。最適化の効果を決定する因子のとらえ方をそれぞれ図にすると、図 3.1、図 3.2 のようになる。

この事実を用いて、本研究では、「最適化に与えられる中間コードと最適化効果の関係」という形で、最適化効果を予測するモデルを構築する。つまり、予測モデルの入力は、最適化器に与えられる中間コードという事になる。但し、実際に

は処理しやすいよう、コードそのものではなくコードの特徴を抽出して入力として与えている。詳細は後述する。

3.2.2 予測モデルの出力

予測モデルの目標は、このモデルを使って最適化列全体の効果を予測することである。そのためには、モデルが何を出力すべきなのかを述べる。

ここで「最適化列全体の効果」と言ったとき、その「効果」とは、出力コードのパフォーマンス、つまり実行時間の改善の事を指す。もし、個々の最適化による実行時間の改善を知ることが出来れば、その総和として、最適化列全体の実行時間の改善も求める事が出来る。よって、個々の最適化の振る舞いを予測するモデルは、実行時間の改善を予測すればよい。

ただし、それだけでは十分ではない。予測モデルの入力は前述の通り、最適化器に与えられる中間コードである。しかし、この中間コードは最適化を適用する度に变化する（だからこそ、最適化の相互作用という概念が生まれた）。

最適化列を構成する個々の最適化の効果を予測するためには、各最適化器に与えられた中間コードの情報がそれぞれ個別に必要である。そこで、それらの個々の段階の中間コードの情報を取得するために、本モデルでは、「最適化適用後のコードの情報」もまた予測し出力する。ある最適化予測モデルで予測された「最適化適用後のコードの情報」は、次に適用される最適化の予測モデルへの入力となる。

本モデルでは最適化の効果として、

1. 最適化の適用による中間コードの変形
2. 最適化の適用による実行時間の変化

の二つを予測することにする。

モデルが各最適化の効果を予測していく様子を図示したのが図 3.3 である。まとめると、最適化の予測モデルには、対象となる最適化器に与えられる中間コードの情報が入力として与えられ、対象となる最適化によるパフォーマンス（目的コードの実行時間）の変化と、変形された中間コードの情報が出力される。そして、出力された中間コードの情報は、次の最適化の予測モデルへの入力となる。このようにして、最適化列を構成する個々の最適化の予測モデルが、対象となる最適化の効果をそれぞれ予測していく。

なお、以後は中間コードの変形と実行時間の変化を、それぞれ別のモデルとして個別に予測することにする。中間コードの変形を予測するモデルを 中間コードモデル、実行時間の変化を予測するモデルを 実行時間モデル と呼ぶ。

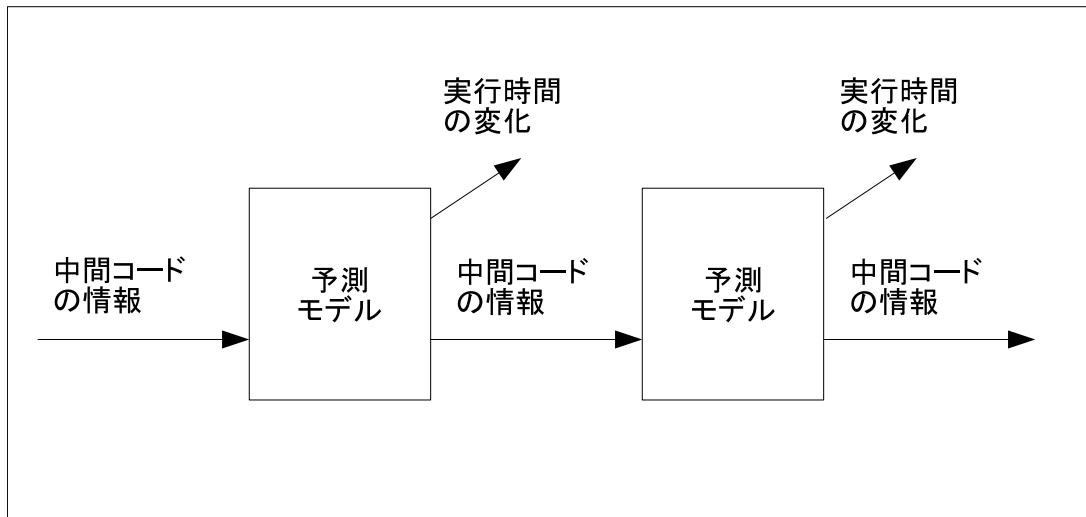


図 3.3: 予測モデルの入力と出力

3.3 中間コードの性格

最適化の効果を決定づけるのは中間コードであるが、コードをそのままモデルへの入力として与えると、処理が複雑になる。そこで本研究では、処理しやすいように、中間コードの中から最適化の効果に関する情報を抽出する手法をとる。もし、中間コードから、特定の最適化との相性といった情報を抜き出すことができれば、それらの情報は、その中間コードに対する最適化の効果を予測するのに役立つであろう。そのような情報を抽出することを、ここではコードの性格付けと呼ぶ。

本研究では、コードを性格付けるための手法として、特徴による性格付け (feature-based characterization) を使用する。

3.3.1 特徴による性格付け

特徴による性格付けとは、コードから何らかの特徴（例えば、特定の命令の数など）を抽出し、これをコードの性格を表すパラメタとする手法である。多くの場合特徴とそれに対応するパラメタは複数種類用いられる。例えば Cavazos らの研究 [8] では、関数のコードサイズや各命令の割合などの、26 個の特徴を採用している。

本研究の場合、最終的に求めたいのは最適化の効果である。よって、種々の最適化効果と密接な関係を持つと思われる特徴を抽出したい。無用命令除去の最適化を例にとると、無用命令除去の効果にもっとも直結する要素は、与えられたコード内に無用な命令がどれだけあるかであると言えよう。このように、ある最適化

1	最適化「基本ブロックの連結」の余地 1
2	最適化「基本ブロックの連結」の余地 2
3	最適化「コピー伝播」の余地 1
4	最適化「コピー伝播」の余地 2
5	最適化「共通部分式除去」の余地 1
6	最適化「共通部分式除去」の余地 2
7	最適化「条件分岐を考慮した定数畳み込みと定数伝播」の余地
8	最適化「無用命令除去」の余地 1
9	最適化「無用命令除去」の余地 2
10	最適化「式を 3 アドレス方式に変換」の余地 1
11	最適化「式を 3 アドレス方式に変換」の余地 2
12	最適化「ループ不変コードの巻上げ」の余地
13	最適化「冗長な Phi 関数の除去」の余地

表 3.1: 本手法で採用した「特徴」

にとって、その最適化の余地がコード中にどの程度残されているかが、最適化の効果と密接な関係を持つ。よって、本研究では、「コード中の各種の最適化の余地」を、特徴として採用する。

本研究で採用した特徴をまとめると下の表 3.1 のようになる。

最適化の種類と特徴の種類が一对一で対応していないのは、最適化の余地の中にも幾つかに分別できる場合があるからである。例えば無用命令除去を例にとって考えると、一口に無用命令除去の余地といっても、その無用な命令がメモリ命令か算術命令なのかでも、実行時間に対する影響は異なると考えられる。そのため、本研究では無用命令除去の余地に対応する特徴は 2 つ存在する。

実装

与えられた中間コードに対し、表 3.1 にある 13 個の特徴の値を取得したい。実は、特徴を抽出するためのプログラムの実装は容易である。今回は、「各種最適化の余地」を特徴として採用したため、最適化自体のソースコードを流用できるからだ。

例えば、無用命令除去の余地を抽出するプログラムを書きたいとする。この場合、無用命令除去のソースコード内の、無用な命令を除去する箇所に改変を加えて、実際に命令を除去する代わりに「無用命令除去の余地」をカウントするように書き直せばよい。このように改変したプログラムを実行することで、中間コード内の各種最適化の余地は取得できる。

3.3.2 その他の性格付け

尚、他の性格付けとしては、最適化列の効果による性格付けが挙げられる。これは、コードに対して特定の数種類の最適化列を適用し、その反応を調べることでコードの特性を分析する手法である。例えば、あるプログラムに対しループに関する最適化が高い効果をもたらしたとき、そのプログラムはループに関して改善の余地があることがわかる。そのようにして幾つかの最適化列を適用すれば、その効果からプログラムの性格を掴めるという発想である。もちろん、適用する最適化列は、できるだけ多くの情報を得られるよう調整されている。本研究では、前述の、特徴による性格付けの方が個々の最適化効果より密接な性格付けができると判断し、この性格付けは採用しなかった。

3.3.3 静的な特徴と動的な特徴

本研究では、それぞれの最適化の適用余地を特徴として採用した。ただし、予測したい最適化効果の中には、実行時間の変化も含まれる。実行時間への影響を予測したい場合には、コードから得られる情報のみでは不十分である。先ほどの無用命令除去の例で言えば、コード上では同じ無用な命令でも、その命令が実行時にどの程度の頻度で行われるかによって実行時間への影響は大きく変わる。よって本手法では、それぞれの特徴に対し、実際には静的な値と動的な値の二つをパラメタとして用いる。そのため、表 3.1 では特徴は 13 種類であるが、実際に収集されるパラメタの数はその 2 倍の 26 種類になる。

静的な値とは、コードのみから得られる特徴の量である。無用命令除去の例で言えば、無用な命令がコード中に何度現れるかをカウントする。コード中の命令が実際に何度計算されるかは考慮しない。一方動的な値とは、実際にプログラムを実行して得られる特徴の量である。無用命令除去の例で言えば、プログラム実行時に、無用な命令と見なされた命令が何度実行されるかをカウントする。

動的な値の取得には、実行時情報を利用する。具体的には、まず最初にプロファイラを使用して、プログラム中の各基本ブロックの実行回数を取得しておく。そして、各基本ブロック内の特徴の値（つまり、最適化の余地がどれ程あるか）と、基本ブロックの実行回数を掛け合わせることで、動的な特徴の値を取得する。

3.3.4 実行時情報の収集

本研究では、実行時情報を収集するために、伊藤らによるプロファイラを使用した。これは伊藤らの研究 [14] で、実行時にパスを何度通ったかを記録するために開発されたものであるが、本研究では、これを基本ブロックを何度通ったかを計算するために取得した。あるブロックに至るパスの実行回数を全て足し合わせると、そのブロックの実行回数を取得できる。

実行時情報を集めるために、対象のプログラムに対して、実行時情報を集めるための命令を埋め込んで予め一回実行しておく。埋め込む命令はC言語で記述し、コンパイル時に対象のプログラムを変換したものと一緒に読み込む。以下にそのソースファイルの一部を示す。

```
/* _profile.c */
struct matrix{
    long long** body;
    int size;
};
struct matrix _make_matrix(int size);
void _count(int a,int b,struct matrix m);
void _print(struct matrix m, FILE *fp);
```

構造体 *matrix* は *size * size* の行列 *body* を要素にもち、実行時にパスを通った回数を保存する。関数 *_make_matrix* は構造体 *matrix* を作成し、行列の要素を 0 に初期化する。関数 *_count* は、ブロック *a* からブロック *b* への辺を通ったときに、*m.body[a][b]* をインクリメントする。関数 *_print* は、*matrix m* をファイルに出力する。

プログラム 3.1 に命令を埋め込む前のプログラム、プログラム 3.2 に命令を埋め込んだ後のプログラムを示す。

プログラム 3.1 (命令を埋め込む前のプログラム)

```
int fact(int n){
    if(n == 1){                (ブロック 1)
        return 1;              (ブロック 2)
    }
    return n * fact(n - 1);    (ブロック 3)
}
int main(){
    scanf("%d",&n);            (ブロック 1)
    if(n > 0){                  (ブロック 1)
        printf("%d\n", fact(n)); (ブロック 2)
    }
    printf("0\n");              (ブロック 3)
    return 0;                   (ブロック 3)
}
```

プログラム 3.2 (命令を埋め込んだ後のプログラム)

```
struct matrix _counter_main;
struct matrix _counter_fact;
int fact(int n){
    int _pred;                                (ブロック 1)
    int _current = 1;                          (ブロック 1)
    if(n == 1){                                (ブロック 1)
        _pred = _current;                      (ブロック 2)
        _current = 2;                          (ブロック 2)
        _counter(_pred, _current, _counter_fact); (ブロック 2)
        return 1;                              (ブロック 2)
    }
    _pred = _current;                          (ブロック 3)
    _current = 3;                              (ブロック 3)
    _counter(_pred, _current, _counter_fact);   (ブロック 3)
    return n * fact(n - 1);                   (ブロック 3)
}
int main(){
    _counter_main = _make_counter(3);          (ブロック 1)
    _counter_fact = _make_counter(3);          (ブロック 1)
    int _pred;                                (ブロック 1)
    int _current = 1;                          (ブロック 1)
    scanf("%d", &n);                           (ブロック 1)
    if(n > 0){                                  (ブロック 1)
        _pred = _current;                      (ブロック 2)
        _current = 2;                          (ブロック 2)
        _count(_pred, _current, _counter_main); (ブロック 2)
        printf("%d\n", fact(n));              (ブロック 2)
    }
    _pred = _current;                          (ブロック 3)
    _current = 3;                              (ブロック 3)
    _count(_pred, _current, _counter_main);   (ブロック 3)
    printf("0\n");                             (ブロック 3)
    FILE* _fp = fopen("profile", "w");        (ブロック 3)
    _print(_counter_main, _fp);                (ブロック 3)
    _print(_counter_fact, _fp); 18           (ブロック 3)
    fclose(_fp);                              (ブロック 3)
    return 0;                                  (ブロック 3)
}
```

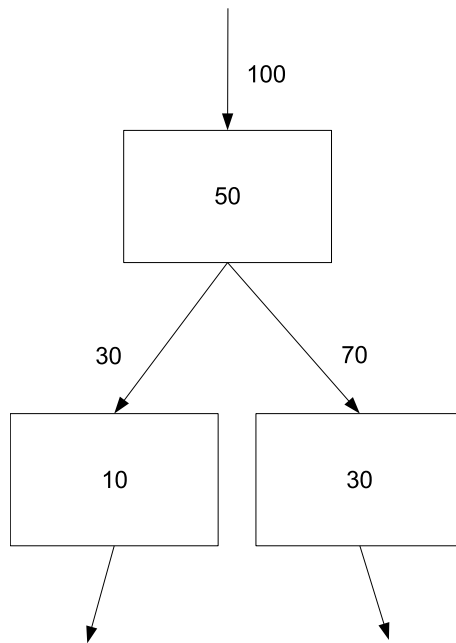


図 3.4: あるプログラムの一部のフローグラフ

この、命令を埋め込んだプログラムを実行すると、ここではファイル「profile」に実行時情報が書き込まれる。

プロファイラを使用して、静的な特徴と動的な特徴を抽出する例を示す。図 3.4 は、プログラム（の一部）のフローグラフの例である。図の四角は基本ブロックを意味し、ブロック内の数字は、そのブロック内である最適化の余地（例えば無用な命令の数）が字面上でどれだけあるかを示したものである。エッジは制御フロー、エッジのそばの数字は、そのエッジが実行時にどれだけ実行されたかを示している。この情報はプロファイリングによって得られる。

この図の場合、静的な特徴の値は、エッジの実行回数を無視して各基本ブロックにおける特徴の値の総和、つまり $50 + 10 + 30 = 90$ となる。一方動的な特徴の値は、各基本ブロックにおける特徴の値に、その基本ブロックの実行回数を掛けたものの総和、つまり $50 \times 100 + 10 \times 30 + 30 \times 70 = 7400$ となる。

3.4 モデルの計算

これまで、本研究の予測モデルとは、何を入力として受け取り何を出力するかという、言わばモデルの外枠にあたる部分を見てきた。本節では、予測モデルは具体的にどのように構築し、どのような計算を行うかを述べる。

本研究のモデルは、最適化の種類に依存しない事を目標としている。よって、どのような種類の最適化であっても予測モデルの構築が可能でなければならない。

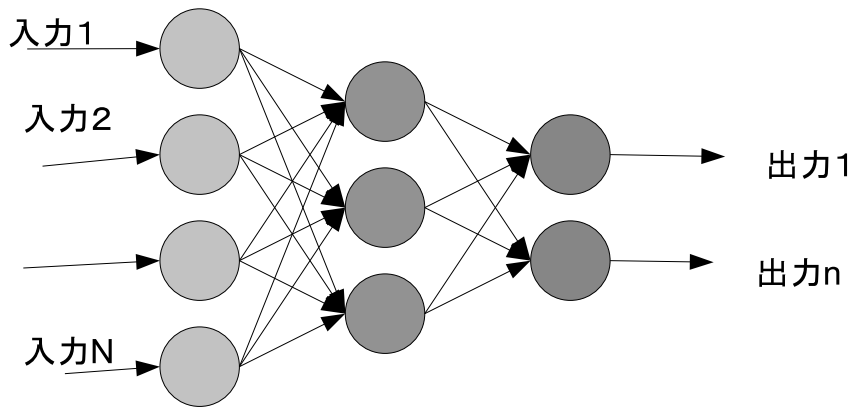


図 3.5: フィードフォワードニューラルネットの模式図

そこで、中間コードの特徴と、それに対する最適化の効果のデータを多数収集し、それを機械学習を用いて学習させるという手法で、本研究はモデルの構築を行う。機械学習の中でも人工ニューラルネットワーク (以下、ANN と略称することもある) を使用し、学習された ANN という形で予測モデルの実現を目指す。類似の手法として Mixtures of Experts[15] や回帰ツリー [6] など、あるいはロジスティック回帰分析 [5] のような機械学習の手法は他にもあるが、ANN が類似の研究 [11, 7] において実績を挙げていることなどを考慮し、本研究では ANN が最も適していると考え、採用した。

3.4.1 人工ニューラルネットワーク ANN

ANN は人の脳を模して作られた情報処理機構であり、予測的なモデルを構築するのに適している [19, 5]。本研究では、ANN の中でも、中間層を一つ持ったフィードフォワードニューラルネットを使用する。

図 3.5 はフィードフォワードニューラルネットの模式図である。色づけされたノードが 3 列にならんでいるが、これは左からそれぞれ入力層、中間層、出力層と呼ばれる。入力層に入力された値は、エッジの方向に伝えられ、最終的に出力層への出力となる。ただし、それぞれのエッジには、値をどの程度ノードに伝えるかを決定する「重み」というパラメータが存在する。一方ノードには、「閾値」というパラメータが存在する。これは、そのノードに伝えられた値の総和がそのノードの閾値以上であればノードは 1 を出力し、そうでなければ 0 を出力するというものである。これらのパラメータを適切に調節することで、ANN は様々な処理を実現することができる。

図 3.6 は、ANN の簡単な例である。この ANN は中間層を持たず、二つの値を入力として受け取り、一つの値を出力する。また、ノード内の数字はそのノードの閾値であり、エッジのそばの数字はそのエッジの重みである。なお、簡単のため、

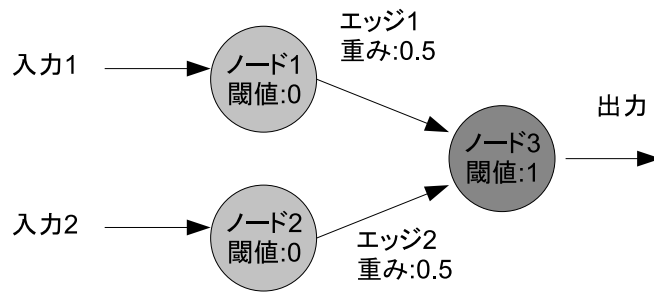


図 3.6: 簡単な ANN の例

入力 1	入力 2	出力
1	1	1
1	0	0
0	1	0
0	0	0

表 3.2: ANN への入力と出力

入力は 0 か 1 のみとする。

例えば、入力 1 に「1」を、入力 2 に「0」を出力として与えた場合、ノード 1 の閾値は 0 なので、値「1」がそのままエッジ 1 に伝わる。そしてエッジ 1 の重みは 0.5 なので、エッジ 1 に伝わった値「1」に重み 0.5 を掛けた値、つまり 0.5 がノード 3 に伝えられる。ノード 2 も閾値は 0 だが、入力 2 の値が「0」なので、エッジ 2 に伝える値もやはり「0」である。そしてエッジ 2 がノード 3 に伝える値もやはり「0」である。

まとめると、ノード 3 には、エッジ 1 から「0.5」の値が、エッジ 2 からは「0」の値が伝えられることになる。しかしこの二つの値の総和は 0.5 であり、ノード 3 の閾値「1」に満たない。よって、最終的にノード 3 から出力される値は 0 ということになる。

このような計算を、全ての入力に対して行った結果を纏めると表 3.2 のようになる。

入力と出力を見ると、これは両方の入力が 1 になった時のみ 1 を出力し、その他の場合は 0 を出力する「AND 回路」を表現していることがわかる。例えばこの ANN のノード 3 の閾値を「0.5」に変更すれば、今度はその ANN は「OR 回路」を表現することになる。このように、ANN は重みと閾値を設定することによって様々な機構を表現できる。

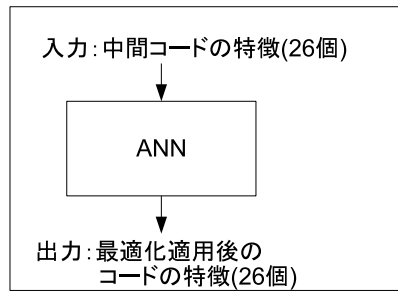


図 3.7: 中間コードモデルの入力と出力

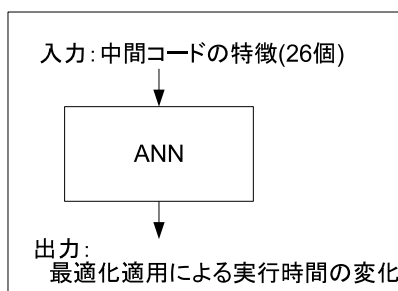


図 3.8: 実行時間モデルの入力と出力

3.4.2 予測モデルと ANN

本研究では、学習された ANN という形で予測モデルを実現させる。モデルへの入力と、モデルからの出力については 3.2.1 節と 3.2.2 節で述べた。中間コードモデルも実行時間モデルも、ANN への入力は、最適化器に与えられた中間コードの特徴である。一方、出力はモデルによって異なる。中間コードモデルの場合は、最適化適用後の予測された中間コードの特徴を出力する。実行時間モデルの方は、最適化適用後の予測された実行時間の変化を出力する。

つまり、中間コードモデルは、26 個のコードの特徴を入力とし、やはり 26 個の最適化適用後のコードの特徴を出力するような ANN として構築する。また、実行時間モデルは、26 個のコードの特徴を入力とし、最適化適用後の実行時間の変化を 1 つ出力するような ANN として構築する。

3.4.3 予測モデルの学習

上述の通り、ANN は重みや閾値を適切に調整することで様々な機構を表現することができる。逆に言えば、ANN で予測モデルを構築するには、重みや閾値を適切に設定する必要がある。この作業が学習である。実際に最適化を適用し、そのときに得られた実際の効果を例題として教師あり学習を行うことによって、ANN

が正しい予測をできるように修正してゆく。今回用いる学習は誤差逆伝播法と呼ばれる。誤差逆伝播法の要約は次の通りである。

1. ニューラルネットワークに学習のためのサンプルを与える。
2. ネットワークの出力とそのサンプルの最適解を比較する。各出力ニューロンについて誤差を計算する。
3. 個々のニューロンの期待される出力値と倍率 (scaling factor)、要求された出力と実際の出力の差を計算する。これを局所誤差と言う。
4. 各ニューロンの重みを局所誤差が小さくなるよう調整する。
5. より大きな重みで接続された前段のニューロンに対して、局所誤差の責任があると判定する。
6. そのように判定された前段のニューロンのさらに前段のニューロン群について同様の処理を行う。

以上を踏まえて、中間コードモデルの構築の手順を簡略化して示すと以下のようになる。

1. 与えられた中間コード A に最適化を適用し、中間コード B を生成する
2. A から抽出した特徴と B から抽出した特徴を、中間コードの変化予測用 ANN に入力・出力として与え、学習させる
3. 様々な中間コードに対して 1~2 を繰り返す

実行時間モデルの構築の手順を簡略化すると以下のようになる。

1. 与えられた中間コード A に最適化を適用し、中間コード B を生成する
2. A と B から生成された目的コードを実際に実行し、実行時間を計測する
3. A から抽出した特徴と実行時間の差を、実行時間の変化予測用 ANN に入力・出力として与え、学習させる
4. 様々な中間コードに対して 1~3 を繰り返す

第4章 予測モデルの構築と適用

本章では、前章で説明した予測モデルを構築し、最適化列の効果を予測するために使用する。そして最適化列の効果の予測を利用して、良い最適化列を探す手順を説明する。

予測モデルに関する手順は、大きく分けて「モデルの構築」と「モデルの使用」に分けることができる。モデルの構築は、コンパイラの著者によって一度だけ行われる。構築自体には時間が掛かるが、この手間はコンパイラの著者に掛かるものであり、コンパイラのユーザの負担にはならない。

モデルの使用は、最適化の効果を予測する度に行われる。これはコンパイラの使用時に行うため、掛かる時間はコンパイラのユーザの負担となる。しかし、モデルの構築に手間を掛けている分、モデルの使用に掛かる時間は短くて済む。

4.1 モデルの構築

まずは前節の手法を用いて最適化の予測モデルを構築する。モデルは各最適化ごとに、また中間コードモデルと実行時間モデルごとに、個別のモデルを構築する。モデルを構築する際には、ANNを学習させるために例題となる中間コードと、その中間コードに対する最適化の実際の効果のデータが必要である。

図4.1は、ある最適化Aに対し、与えられた中間コードIを用いて「中間コードモデル」の構築を行う様子を図示したものである。コードIと、コードIに最適化Aを適用したコードI'から、それぞれコードの特徴FとF'を抽出する。「F」という特徴を持つ中間コードに最適化Aを適用すると、「F'」という特徴を持つ中間コードが得られたことになる。つまり、FからF'への推移こそが、最適化Aの、中間コード上における効果という事になる。よって、FとF'の2つを、ANNへの例題として与える。

図4.2は同様に「実行時間モデル」の構築を行う様子を図示したものである。こちらは、最適化A適用前後の目的コードを実際に実行し、その差、つまり実行時間の変化を最適化の効果としている。

どちらのモデルの場合も、図4.1、4.2にあるような手順は1度きりというわけではない。様々な中間コードが与えられれば、それだけANNは様々なコード上での最適化効果を学習し、その結果汎用的なモデルが構築されることになる。よって、実際のモデルの構築の際には様々な中間コードが与えられることになり、そ

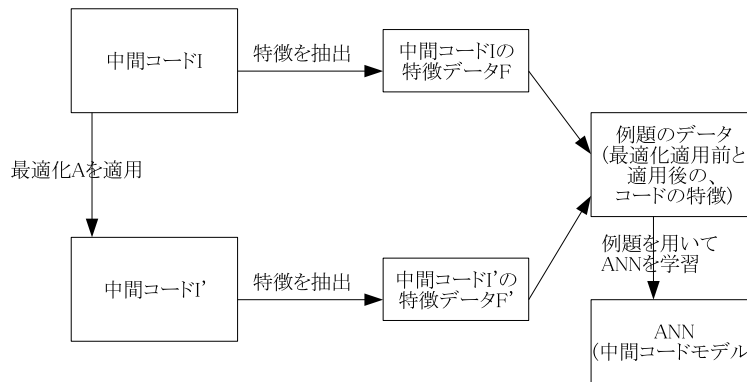


図 4.1: 中間コードモデルを構築する様子 (最適化 A の場合)

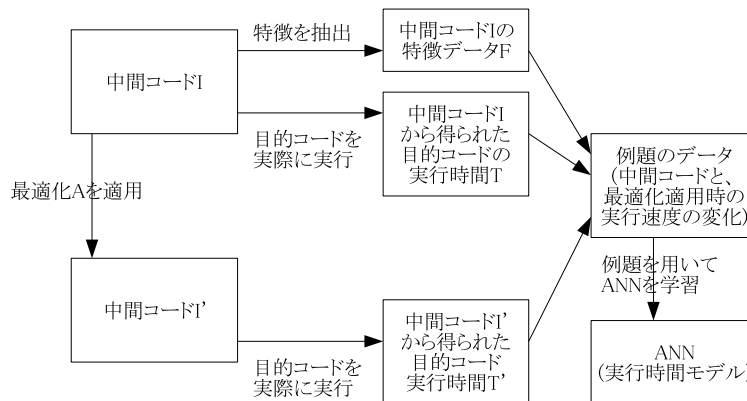


図 4.2: 実行時間モデルを構築する様子 (最適化 A の場合)

の数だけ図 4.1、4.2 の手順が繰り返されることになる。本論文の実験で使用した例題については後の節で説明する。

4.2 モデルを使用した最適化列効果の予測

本手法で構築したモデルを用いて、与えられたプログラムに対する最適化列の効果を実験的に予測する。その前に、準備として前述の二つのモデルを数式化する。中間コードモデルは、入力としてコードの特徴を受け取り、最適化適用後のコードの特徴を予測し、出力する関数 $code$ として数式化できる。一方、実行時間モデルは、同じく入力としてコードの特徴を受け取り、最適化適用後の実行時間の変化を予測し、出力する関数 $time$ と数式化できる。3章で述べた通り、本研究ではそれぞれの最適化ごとに予測モデルを構築する。そのため、それに対応した $code$ 、 $time$ 関数も最適化ごとに構築されることになる。どの最適化に対する関数なのかを区別するため、予測している最適化の名前を関数名の後に添え字としてつけること

にする。例えば最適化 A による実行時間の変化を予測する関数であれば、 $time_A$ と表記する。

最適化列全体の効果の予測は、各最適化の適用によって実行時間と中間コードが変形する様を予測 (シミュレート) することによって為される。

4.2.1 適用例

以上の関数を用いて、最適化効果を予測する手順を見ていく。ここでは、例として、最適化 A 、最適化 B 、最適化 C をプログラム P に対して順番に適用する最適化列 ABC を考える。

まず、プログラム P の中間コードに最初の最適化 A を適用する。このとき、最適化 A に与えられる中間コードの特徴を I とすると、最適化 A 適用後のコードの特徴は $code_A(I)$ 、最適化 A 適用後の実行速度の変化は $time_A(I)$ と予測される。なお、中間コードの特徴 I は、プログラムを解析・プロファイリングすることによって得られる。この様子を図示すると図 4.3 となる。

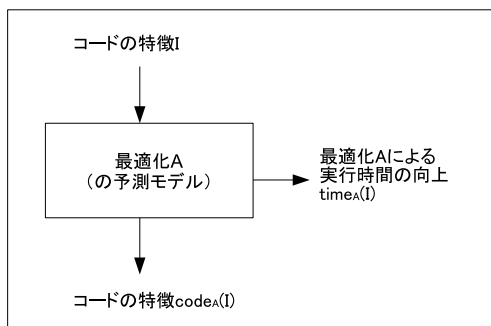


図 4.3: 最適化 A の予測モデルへの入力と出力

このように、各最適化毎に構築した中間コードモデルと実行時間モデルを用いて、与えられた中間コード (の特徴) に対する最適化の効果を予測する。そして、2つのモデルのうち、中間コードモデルの予測結果、つまり予測された「最適化適用後の中間コードの特徴」は次の最適化の予測モデルへの入力となる。最適化列を構成する各最適化に対して、順番に上記の処理を繰り返すことになる。

今回の例の場合は、最適化 B 、最適化 C に対しても同様の処理を行うことになる。その様子を述べていく。最適化 A を適用した結果のコードが次の最適化 B に渡されることになるので、最適化 B に与えられるコードの特徴は $code_A(I)$ と予測される。すると、最適化 B 適用後のコードの特徴は $code_B(code_A(I))$ となり、最適化 B の適用による実行時間の変化は $time_B(code_A(I))$ と見積もられる。同様に、最適化 B 適用後のコードが最適化 C に与えられるので、最適化 C に与えられるコードの特徴は $code_B(code_A(I))$ である。よって、最適化 C 適用後のコード

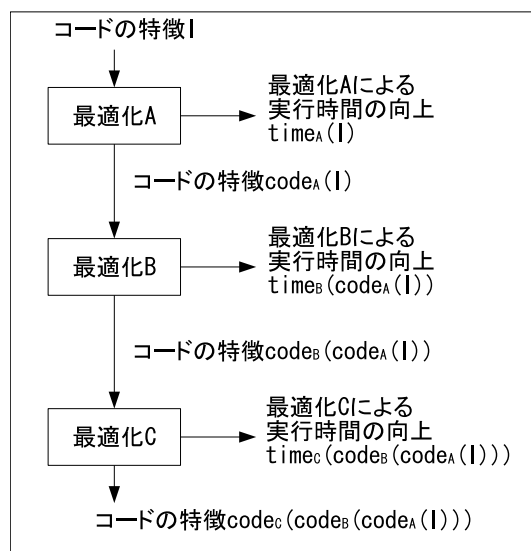


図 4.4: 各種のモデルを用いて最適化効果を予測する様子

の特徴は $code_C(code_B(code_A(I)))$ であり、最適化 C の適用による実行時間の変化は $time_C(code_B(code_A(I)))$ となる。そして最適化列全体の効果は、各最適化による実行時間の変化の和である

$$time_A(I) + time_B(code_A(I)) + time_C(code_B(code_A(I)))$$

となる。このようにして、各段階での中間コードの特徴と実行時間の変化を計算することにより、最適化列全体の実行時間への影響を予測することが可能になる。この例の手順を図で示すと図 4.4 のようになる。

4.3 最適化列の探索

本手法は、あくまでも与えられたプログラムに対する、最適化列の効果を予測する手法である。よって、本手法自体単体では、効果の高い最適化列を探索することはできない。しかし、実際のプログラムの実行をすることなく最適化の効果を見積もることができるため、既存の最適化列探索のアルゴリズムと組み合わせることで、効率的な探索が可能になる。

例えば、Almagor らの研究 [3] では、遺伝的アルゴリズムを使用して効果の高い最適化列を求めているが、最適化列の効果を評価するために何度も実行を繰り返しており、その分探索にも時間がかかってしまう。しかし、最適化列の効果を評価する際に実行を行わずに、本手法による予測を導入すれば、良い最適化列の探索の時間が大幅に改善することが見込める。

4.4 予測のコスト

本手法を用いた最適化列の予測について、どの程度のコスト、つまり時間が掛かるのかを評価する。

4.4.1 特徴の取得

まず、4.2.1 節の例における、対象プログラムの中間コードの特徴 I を取得する必要がある。3.3.3 節で述べたとおり、特徴には静的なものと動的なものがある。

静的な特徴を取得するためには、実際にプログラムを中間コードの状態まで落とし込んで、そこから解析を加える必要がある。特徴を抽出するプログラムについては3.3.1 節で述べた。この手順では目的コードを出力する必要はないが、かかる時間は実際のコンパイルと同等と見て良いだろう。

一方、動的な特徴を取得するためにはプロファイリングをする必要がある。これは、掛かる時間という観点では実際の実行と同じと見て良い。また、本プロファイラを用いてプロファイリング用実行コードを生成するためには、本プロファイラの仕様上、2度コンパイルする必要がある。

以上をまとめると、中間コードの取得のためには最低、コンパイル3回と実行1回分の時間が掛かる。

4.4.2 モデルの適用

構築されたモデルと与えられた中間コードの特徴を用いて、実際にモデルを適用する際の時間について述べる。本手法の場合、モデルの適用とは実際にはANNの計算を意味する。

ANNの構築のためには何度も学習を繰り返す必要があり非常に時間がかかるが、すでに構築されたANNの計算は極めて短時間で済む。ANNの計算の中身は単純な算術の繰り返しであり、その数は、入力層・中間層・出力層の数に依る。入力層・中間層・出力層の数をそれぞれ $input$ 、 $middle$ 、 $output$ とすると、計算の回数は $input \times middle + middle \times output$ となる。本手法の場合は、入力層・中間層は26個であり、出力層の数はモデルによって1個か26個である。よって、本手法の場合、モデルの適用のコストは、高く見積もっても算術命令約1000回程度と言える。

これはモデルを1回適用する際のコストであるが、最適化の効果を予測する際には、中間コードモデルと実行時間モデルの二つのモデルを適用する。よって、最適化1つにつき、予測のコストは $1000 \times 2 = 2000$ 程度と見積もることが出来る。

また、最適化列を構成する最適化の数が m 個だった場合、最適化列全体の効果を予測するコストは算術命令 $2000 \times m$ 回分となる。

4.4.3 2種類のコスト

特徴の取得と、実際のモデルの適用の2種類のコストを概算した。ただし、この2種類のコストはどういうときに掛かるのかが異なる。前者(特徴取得時のコスト)は入力プログラムごとに掛かるのに対し、後者(モデル適用時のコスト)は最適化列の効果を予測するたびに掛かる。そして、プログラムの数と、予測する最適化列の数は1対1であるとは限らない。例えば前節での例のように、1つのプログラムに対して何種類もの最適化列の効果を予測したいこともある。このような場合は、後者のコストは最適化列の数だけかかるが、前者のコストは最初の一度だけで済む。

前者(特徴取得時のコスト)に比べ後者(モデル適用時のコスト)の方がコストが低い事を考えると、本手法は、前節の例のように多数の最適化の効果を予測したいときに適している。

第5章 実験

本章では、本手法の効果を計るべく実験を行い、その結果をもとに本手法を評価する。本研究では、並列化コンパイラ向け共通インフラストラクチャCOINS[9]を用いて本手法の実装を行った。

5.1 実験環境

本節ではマシンやコンパイラ、使用したプログラムなどの実験環境について説明する。

5.1.1 マシン

本実験は、富士通のPRIMEPOWER 250上で行った。このマシンの主な仕様は表5.1の通りである。

5.1.2 コンパイラインフラストラクチャCOINS

背景

COINSは、コンパイラ研究の基盤となる共通のコンパイラの作成を目的に2000年度より研究が進められている[9]。つまり、組み合わせ可能なコンパイラ部品で

機種	PRIMEPOWER 250
プロセッサ種別	1.98GHz SPARC64 V
プロセッサ数	2
1次キャッシュ	256KB
2次キャッシュ	3MB
メモリ容量	10GByte
オペレーティングシステム	SunOS 5.10

表 5.1: PRIMEPOWER 250 の主な仕様

構成される共通インフラを作り、その上に各企業や研究者がそれぞれの目的に合う機能部品を加えることができるようにすることを目的としている。COINS では SSA 最適化を行っている。現在、多くの最適化コンパイラで SSA 最適化の実験が行われており、一部は実用段階に達している。現段階の COINS の完成度もかなり高まっているが、バグもまだある状況である。このような状況のため、本手法の実装や実験を行うにあたりいくつかの制約を受けることは否めない。

構成

一般にコンパイラはフロントエンド(front end) とバックエンド(back end) から構成される。フロントエンドは原始プログラム(source program) を中間コード(intermediate code) と呼ばれる内部形式に変換する。バックエンドは中間コードを計算機の機械コードに変換する。フロントエンドはさらに字句解析器(lexical analyzer)、構文解析器(syntax analyzer)、意味解析器(semantic analyzer) に分けられる。バックエンドは最適化器(optimizer) とコード生成器(code generator) に分けられる。これらの各部分はコンパイラのフェーズと呼ばれる。

COINS では、複数の入力言語、複数の対象機種に対応するため 2 つの中間コードがある (図 5.1)。入力言語の論理構造に近いレベルの中間コードを、高水準中間表現(high-level intermediate representation、HIR) とよび、機械語に近いレベルの中間コードを、低水準中間表現(low-level intermediate representation、LIR) とよぶ。COINS の概念図を図 5.1 に示す。

なお、COINS のソースはすべて Java 言語で書かれている。

5.1.3 最適化

COINS ではコンパイル時にオプションを指定することで、適用する最適化の種類・順番を任意に指定できる。これは、本手法の実装を COINS 上で行った主たる理由の 1 つである。逆に言うと、最適化の順番を指定できない gcc[12] のようなコンパイラでは本手法は実装できない。

2 章で、最適化は中間コード上で作用すると述べたが、これは COINS に関しても同様である。但し、COINS には中間表現が 2 種類あり、それに対応して最適化フェーズも 2 つ存在する。COINS でなされる最適化には、大別して高水準中間表現 HIR 上でなされる基本最適化と、低水準中間表現 LIR 上でなされる SSA 最適化の 2 種類が存在する (図 5.1)。その 2 つのうち本研究では、SSA 最適化を対象に実装・実験を行った。

本実験では最適化として COINS 上で実装されている SSA 最適化 8 種類を使用した [22]。その内容は表 5.2 にある。

これらの最適化のうち、「式を 3 アドレス方式に変換」と「冗長な Phi 関数の除去」は一般的な最適化としてはあまり見られないものなので解説を加えておく。

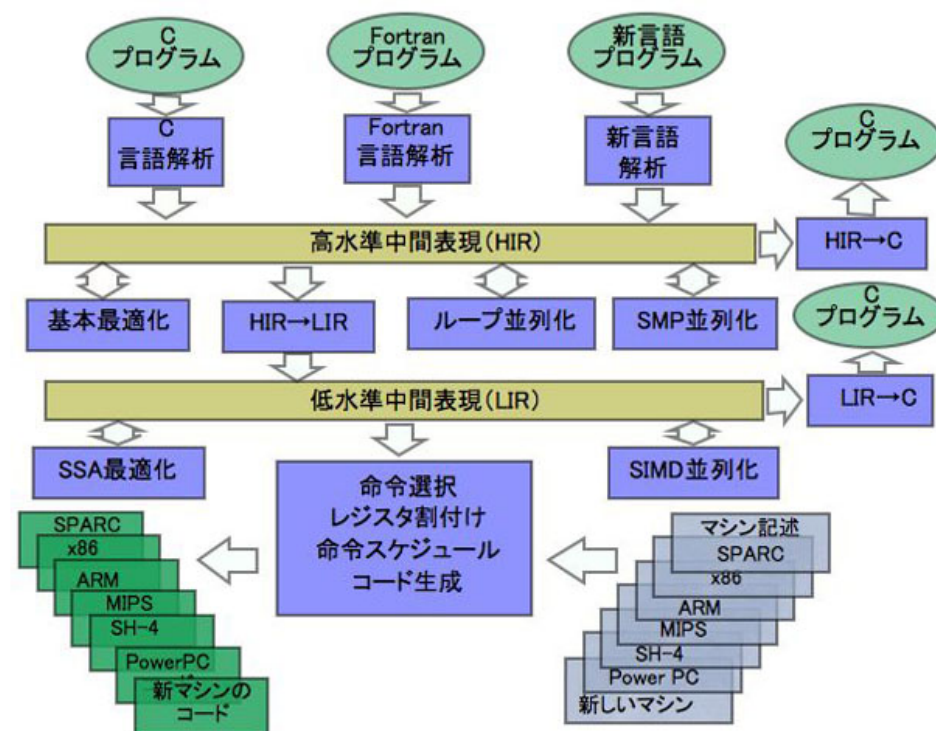


図 5.1: 並列化コンパイラ向け共通インフラストラクチャ(COINS) 概念図

- 式を 3 アドレス方式に変換

コンパイラの入力となる高級言語では、ひとつの変数に対する代入文の右辺値は必ずしもひとつの演算とは限らない場合が多い。つまり $x = a + b \times c - d$ のような式が許されている場合が多い。しかし最適化の種類によっては、式は 3 アドレス方式 (Three-Address code) のように、ひとつの代入に対してひとつの演算となっていることが都合が良い場合がある。この最適化はその 3 アドレス方式への変換を行うものである。

最適化自身がコードのパフォーマンスを改善するのではなく、他の最適化をしやすくするという点がこの最適化の特徴である。そのため、この最適化は他の最適化との適用順序が重要になる。

- 冗長な Phi 関数の除去

Phi 関数とは、SSA 形式独自の仮想的な関数の事である。Phi 関数自体は SSA 形式ではなくてはならないものであるが、コピー伝播などの最適化を行うと、不要な Phi 関数が発生してしまうことがある。そのため、この最適化ではその不要な Phi 関数の除去を行っている。

最適化名
基本ブロックの連結
コピー伝播
共通部分式の除去
条件分岐を考慮した定数畳み込みと定数伝播
無用命令除去
式を3アドレス方式に変換
ループ不変コードの巻上げ
冗長な Phi 関数の除去

表 5.2: 今回の実験で使用した最適化

この最適化もまた、他の最適化との相互作用の中で効果を発揮する最適化である。

5.1.4 プログラムと ANN への例題

3章で述べたように、本研究の予測モデルは、学習した ANN という形で実現する。よって、予測モデルを構築するには学習させるための例題が必要になる。例題とは、具体的には「中間コードの特徴」とそのコードに対する各最適化の「(実際の)効果」のデータである。このデータを取得するために、本実験では COINS のテストプログラムの中から7個、Computer Language Benchmarks Game[10]から8個、奥村によるアルゴリズムを実装したプログラム [21] から42個の、計57個のプログラムを使用した。これらのプログラムの多くは数十から数百行程度の小さなプログラムであるが、例題としてはこの程度が望ましい。何故なら、あまり大きすぎるプログラムだと、プログラム内に相反する特徴をもつコードが存在し、その結果コードの特徴や最適化の効果が打ち消される可能性が高くなるからである。

プログラムは y 個だが、プログラム1つにつき中間コード1つというわけではない。というのも、「最適化の適用による中間コードの変化」が他の最適化の効果に影響を与えるという事実が、本研究の前提となっている。よって、その中間コードの変化と他の最適化の効果の変化の関係を調べる上でも、最適化の適用によって変化した中間コードのデータを取得する必要がある。そこで本実験では1つのプログラムに対して、計73種類の異なる最適化列を適用した。こうすることで、プログラム1つにつき、73種類の中間コードのバリエーションが得られることになる。本研究では、こうして得られた $57 \times 73 = 4161$ 個の中間コードとそのコードに対する各種最適化の効果のデータを ANN への例題として使用した。

5.2 実験

本論文では、本手法の特性や効果を知るために、様々な実験を行った。本節ではそれらを順番に説明する。

5.2.1 モデル構築におけるプログラム同士の相性

本手法を用いて構築する予測モデルは、どのようなプログラムに対しても最適化の効果を予測できることが望まれる。そのため、5.1.4節で述べたように、様々な中間コードに基づく例題をANNに与え、汎用的なモデルにするように努めた。一般的には、より多くの例題を与えれば、より多くのプログラム上で正確な予測ができるようになるはずである。尚、ANNに例題として与えられるのは、あるプログラムから得られた中間コードの特徴と、そのコード上での最適化の効果であるが、ここでは簡略して「プログラムを例題として与える」や「コードを例題として与える」というような表現も行う。

しかし、全く異なる特性を持つコードを例題として与えると、ANNが一般化しきれずに精度が下がってしまうというケースも考えられる。そのため、例題として与えるプログラムは多いほど良いのか、それとも取捨選択が有効なのかを調べる必要がある。

まず、単独のプログラム(の中間コードに基づく例題)から構築したモデルをそれぞれ用意し、それらのモデルを各種プログラム上で適用して、最適化の効果をどの程度正確に予測できるかを調べた。この実験は最適化の種類ごとに行われたが、例えば最適化「冗長なPhi関数の除去」に関する結果を表にまとめると図5.2のようになった。

列は「モデルを構築したプログラム」を表し、行は「どのプログラム上でモデルを使用したか」を表している。また、表を直感的に把握できるようにするため、精度が比較的高いものには「○」を、比較的低いものには「×」をつけている。例えば1行2列目は「○」となっているが、これは「1番目のプログラム」上で、「2番目のプログラムから構築したモデル」を使用した結果、比較的少ない誤差で予測できた事を示している。

図5.2を見れば、「ほとんど○の列(または行)」と「ほとんど×の列(または行)」にくっきりと別れていることが見て取れる。例えば10行目は全ての枠が×となっている。これは、10番目のプログラムから構築したモデルは、ほとんどの他のプログラム上で正確な予測ができなかった事を示している。この事は、これらのプログラムを例題としてANNを構築すると、ほとんどプログラムに対してモデルの精度が下がるの可能性を示唆している。

そこで、あるプログラムに対して、「そのプログラム以外の全てのプログラムを用いて構築したモデル」と、「そのプログラムと、図5.2でほとんどの行が×であったようなプログラムを除いたプログラムを用いて構築したモデル」の2つを用意し

2. ランダムに生成された 20 個の最適化列 (列の長さは 8) に対し、モデル用いて最適化列の効果を予測する。
3. 2 の手順の 20 個の最適化列を実際に適用して、その効果を取得する。
4. 1~3 の手順で得られた「予測された効果」と「実際の効果」の相関係数を計算する。
5. 1~4 の手順をすべてのプログラムに対して繰り返す。

相関係数とは、2 つの確率変数がどの程度類似しているかを示す指標である。これは 1 に近いほど 2 つの変数は強く相関することを表しており、逆に 0 に近づけば 2 つの確率変数はほとんど無関係ということになる。今回は、最適化列の予測された効果と実際の効果の 2 つが確率変数になる。相関係数が 1 に近づけば、予測された効果が現実の効果に近く、精度が高いことを意味する。全てのプログラムに対して相関係数を計算し、平均を取った結果、通常モデルの相関係数は 0.27、間引きモデルの相関係数は 0.23 となった。

5.2.2 不適なプログラム

相関係数は 0.27 と 0.23 であるが、これは強い相関と呼べるものではなく、予測の精度が著しく低いことを表している。そこで相関係数が低い原因を調べると、プログラムごとの相関係数を調べると最適化列の効果自体がほとんどないプログラムは、相関係数も著しく低いという結果が得られた。これは、適用する最適化列による実行時間の変化が少ないため、相対的に予測の誤差が大きくなった為と思われる。例えば、どのような最適化列を適用しても実行時間が全く変わらないプログラムもいくつか存在した。今回は小さなプログラムを中心に実験を行ったため、そのようなプログラムが比較的多くなったと思われる。このようなプログラムの場合、どれほど正確な予測をしたとしても、相関係数は必ず 0 となる。

このようなプログラムは、予測の精度を測る対象として適切ではないと言える。そこで相関係数を調べるプログラムを、最適化列の効果が大きいときで 5% 以上認められるようなプログラム 28 個に絞り、もう 1 度実験を行った。その結果、相関係数は通常モデルが 0.62、間引きモデルが 0.57 となった。

以上から、最適化の効果が高いプログラムの場合には、本手法による予測の精度が高くなることがわかる。これは逆に言うと、最適化の効果が低いプログラムの場合には本手法の精度は不十分であることを意味する。しかし、最適化の効果自体がほとんどないプログラムの場合は、どのような最適化列でも効果が変わらないため、良い最適化列を探す必要性も低いと言える。よってこの点は、実際にはあまり問題にならない。

5.2.3 本手法を利用した、良い最適化列の探索

本節では、本手法を、良い最適化列の探索に使用し、どの程度良い最適化列が得られるかを調べる。

手順としては、まず最初に、これまでと同様に通常モデルと間引きモデルを構築する。その後、ランダムに 10000 個の最適化列を作り出し、それぞれのモデルを使ってその効果を予測する。そして 10000 個の最適化列の中で、最も効果が高いと予測された最適化列を採用し、実際に適用・実行して効果を調べというものである。

また、予測の精度は完全ではないため、選ばれた最適化列の効果が偶然あまり高くない可能性もある。そこで、最も効果が高いと予測された最適化列一つだけでなく、効果が高いものを上から複数個選ぶ手法も考えられる。選ばれた最適化列は実際に適用・実行し、その結果最も高いパフォーマンスを出した最適化を採用する。この方法は一つだけ選ぶ方法よりも良いパフォーマンスが期待できる反面、実際に適用・実行する回数が増えるため、探索に時間が掛かる。よって、選ぶ数を変えながら実験を行うことで、どの程度の数を選べば妥当であるかを見る。

パフォーマンスの比較対象として、最適化を一切適用していない場合の実行時間と比較をした。また、COINS コンパイラ上で良い効果を与えるとされている最適化列とも比較をした。これは、COINS コンパイラ上で“-O2”とオプションを指定すると適用される最適化列であり [9]、O2 と呼称する。

実験ではまず、対象となるプログラムに対して最適化列をいくつ選ぶかを変えることで、その中の最良のパフォーマンスがどう変化するかを調べた。そしてそれを全てのプログラムに対して行い、平均をとった結果が図 4 になる。「O2」と「最適化無し」のグラフは、選ぶ数に関わらず、常に同じ最適化列なので、グラフは横ばいである。「通常モデル」と「間引きモデル」のグラフは、選ぶ数が 1~3 のときは実行時間は不安定だが、選ぶ数が 4 以上のときは、ほぼ最適化列「O2」と同等の実行時間になっている。また、通常モデルの場合、選ぶ数が 7 以上の時は、O2 よりも実行時間が少なくなっていることがわかる。

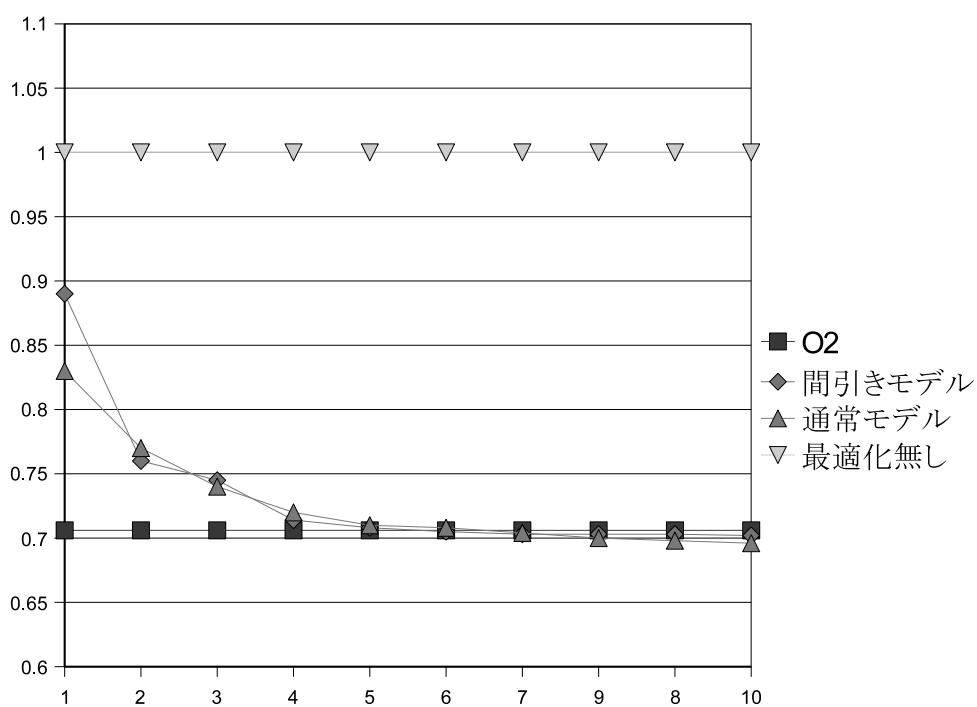


図 5.3: 選んだ最適化列の数と実行時間 (縦軸は実行時間、横軸は選んだ最適化列の数)

第6章 関連研究

本章では、本研究と同じく、統計的手法を用いて良い最適化列の探索する研究を紹介し、本研究と比較する。

6.1 Agakov らの研究

Agakov らの研究 [1] は、過去のコンパイル・実行時のデータを機械学習を用いて解析し、高い効果が見込める最適化列を絞り込む手法を提案した。これは、Almagor [3] らの研究のような、コンパイル・実行の手順を何度も繰り返すような手法と組み合わせることで効果を発揮する。

Agakov らは、絞り込みのためのモデルとして、独立同一分散モデルとマルコフモデルの2つを用い、それらを2つの探索アルゴリズムに組み込んだ。その結果、Agakov らの絞り込みを組み込むことで、それぞれの探索アルゴリズムは、極めて短時間で妥当な最適化列を探すことに成功している。

Agakov らの手法それ自身は最適化列の探索を行わないが、他の探索アルゴリズムを効率的にするという点で本手法と類似している。

ただし本手法は、与えられた最適化列の効果の評価を短時間にすることで効率的な探索を可能にしている。それに対し Agakov らの手法は、評価をする最適化列の選び方自体にバイアスを設けることで、効率的な探索を可能にしている。

この2つの研究は、それぞれ異なるフェーズで作用する。その為、本手法と Agakov らの手法は併用することが可能である。

6.2 Dubach らの研究

Dubach らの研究 [11] は、コードの特徴を抽出して解析することで、そのコードに対する最適化列の効果を予測する手法を提案した。この手法は上で述べたような点では本手法と似ているが、最適化列全体の効果を一度に予測しようとしている点、そして何よりも、モデルの構築をプログラム毎に行っている点が大きく異なる。

本手法は前もって予測モデルを構築しているが、Dubach らの手法は与えられたプログラムに対してその場でモデルの構築を行っている。その結果、与えられたプログラムに特化した精度の高いモデルを構築できる反面、プログラム毎にモデ

ル構築のコストが掛かってしまい、実用的とは言えなくなってしまっている。

本手法は逆に、予測の対象を個々の最適化に分類することで、前もってのモデルの構築を可能にした。そして前もってモデルが構築されたことで、精度では劣るものの、短時間で予測を実現した。

6.3 Cavazos らの研究

Cavazos らの研究 [8] もやはり、特徴を抽出し、機械学習を用いて良い最適化列の予測を行う研究である。良い最適化列を導くために、Cavazos らの研究ではロジスティック回帰分析を用いている。

ただし、java の JIT 上で処理を行っているという都合上、単純な特徴しか抽出できておらず、探索の範囲も広くない。つまり、良い最適化列を 1 から求めるのではなく、与えられた最適化列から無用な最適化を取り除くという形の探索に制限されている。

このように予測の制限はつくが、「良い最適化列の探索」をダイナミックコンパイラ上で実現させた事や、プラットフォームをまたぐ事を視野に入れている点など、他にはない特徴も備えている。

第7章 まとめと今後の課題

本研究では、人工ニューラルネットワークを用いて、最適化の挙動を予測するモデルを構築した。そして構築したモデルを用いて良い最適化列を探索し、その効果を評価した。

本研究の特徴として、予測に掛かるコストが低いことが挙げられる。モデルの構築には時間が掛かるが、それはコンパイルの著者に一度だけ掛かる負担であり、この手法のユーザーにはコスト掛からない。そして予測に必要なコストは実行1回とコンパイル数回程度の時間である。そのコストも、一つのプログラムにつき一度だけ払えば、あとはほとんど時間を掛けることなく最適化列の効果を予測することができる。これは、類似の研究に比べてもきわめて早いと言える。

また、本研究では最適化の個々の効果を本格的にモデル化したという点でも特徴的である。

このような特徴を持つ一方で、本手法の予測の精度はまだ十分とはいえない。5.2.3節の実験において、選ぶ最適化列の数を増やせば「O2」よりも良い結果を得ることができるが、その差は小さく、コストに見合うものかは判断が分かれる。

7.1 精度の改善

しかし本手法にはまだ改善の余地がある。まず、モデルの構築のために使用したプログラムの数を増やすという改良点が考えられる。本研究で使用したプログラムは57個だが、これを増やしていくとより一般化が見込まれる。57個のプログラムを使用した実験において通常モデルによる予測の相関係数は0.62であるが、プログラムの数を20個に減らした実験では相関係数は0.45であった。このことから、さらにプログラムを増やすことで、精度を改善することが見込まれる。

また、今回は全てのプログラムを使ってモデルを構築する通常モデルのほかに、一部のプログラムを除いて構築した間引きモデルを提案した。しかし、予測の精度は間引きモデルは通常モデルに劣っていた。これに関しては、モデルの構築に使用するプログラムと、モデルの精度の間にまだ解明すべき点があることを示唆している。それを解明すれば、より効率の良いモデルの構築が可能になると見込まれる。例えば、あらかじめ数パターンの予測モデルをよういしておき、与えられたプログラムに応じて、最も適した予測モデルを選択し使用するという手法が可能になるかもしれない。

また、今回は、ニューラルネットワークを学習させるためのデータを取得する際に、プログラム全体をひとかたまりとして扱っていた。しかし、プログラム単位ではなく、たとえば関数単位をデータを取得すれば、よりきめの細かいデータが得られるだろう。また、最適化列の探索をする際にも、関数ごとに個別の最適化列を適用するようにすれば、更なるパフォーマンスの向上が見込まれる。

ここに記した事柄は、実装上の問題で見送ったものもあれば、単に時間の都合で実装できなかったものもある。モデル構築のためのプログラムの数の増加などは、単純に時間さえあれば解決できる問題であり、実現は容易である。このように、本手法はまだ改善の余地を十分に残している。

7.2 今後の展望

いわゆる phase ordering problem と隣接する研究として、最適化をする際のパラメータをどのように設定すべきかという問題がある。本研究では類似の多くの研究と同様、パラメータの設定にまでは踏み込んでいないが、本研究のアプローチは、パラメータを考慮するモデルに拡張する事も可能である。例えば、最適化効果の予測モデルへの入力としてパラメータの値を与える手法や、同じ最適化でもパラメータが違う場合は異なる最適化として扱う手法など、様々な手法が考えられる。

また、その他に実行時間を決定する要因として、実行環境が挙げられる。これについては、まずはある環境で構築されたモデルが、他の環境でどこまで精度を維持できるかを調べる必要がある。もし環境が変化することによってモデルの予測精度が大きく下がる場合は、環境を考慮したモデルを構築するか、或いはオペレーティングシステムや環境毎にモデルを構築するといった対策が考えられる。

謝辞

本研究を進めるにあたり多大なるご指導ご鞭撻を頂いた。東京工業大学 数理・計算科学専攻教授の佐々政孝先生に深く感謝の意を表します。

また、佐々研究室の皆様、特に中川君にはさまざまな面で助力を頂きました。締め切り直前の忙しい状況を暖かく支援して下さった皆様に、ここに深くお礼申し上げます。

参考文献

- [1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pp. 295–305, Washington, DC, USA, 2006. IEEE Computer Society.
- [2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools Second Edition*. Addison-Wesley Longman Publishing Co., Inc., 2006.
- [3] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pp. 231–239, New York, NY, USA, 2004. ACM.
- [4] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java 2nd ed.* Cambridge University Press, 2002.
- [5] C. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 2005.
- [6] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Chapman and Hall, 1984.
- [7] John Cavazos, Christophe Dubach, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, Grigori Fursin, and Olivier Temam. Automatic performance model construction for the fast software exploration of new hardware designs. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pp. 24–34, New York, NY, USA, 2006. ACM.
- [8] John Cavazos and Michael F. P. O'Boyle. Method-specific dynamic compilation using logistic regression. In *OOPSLA '06: Proceedings of the 21st*

- annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pp. 229–240, New York, NY, USA, 2006. ACM.
- [9] COINS Project. COINS homepage. <http://www.coins-project.org/>.
 - [10] Computer Language Benchmarks Game. Computer Language Benchmarks Game homepage. <http://shootout.alioth.debian.org/>.
 - [11] Christophe Dubach, John Cavazos, Björn Franke, Grigori Fursin, Michael F.P. O’Boyle, and Olivier Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. In *CF ’07: Proceedings of the 4th international conference on Computing frontiers*, pp. 131–142, New York, NY, USA, 2007. ACM.
 - [12] GNU Project. GCC homepage. <http://gcc.gnu.org/>.
 - [13] 今橋孝典. 少数レジスタマシンにおけるスピルを考慮した最適化の組合せの研究. 東京工業大学 情報科学科 卒業論文, 2006.
 - [14] 伊藤陽. 実行時情報を利用した部分冗長除去. 東京工業大学 大学院情報理工学研究科 数理・計算科学専攻 修士論文, 2006.
 - [15] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton. *Adaptive Mixtures of Local Experts*. Neural Computation, 1991.
 - [16] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O’Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *PACT ’00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, pp. 237–246, Washington, DC, USA, 2000. IEEE Computer Society.
 - [17] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. pp. 1137–1143. Morgan Kaufmann, 1995.
 - [18] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. *SIGPLAN Not.*, Vol. 38, No. 7, pp. 12–23, 2003.
 - [19] 熊沢逸夫. 学習とニューラルネットワーク. 森北出版, 1998.
 - [20] 中田育男. コンパイラの構成と最適化. 朝倉書店, 1999.
 - [21] 奥村晴彦. C 言語による最新アルゴリズム事典. 技術評論社, 1991.

- [22] 佐々研究室. 静的単一代入形式最適化システム外部仕様書, 2007.
<http://www.is.titech.ac.jp/~sassa/coins-www-ssa/japanese/ssa-external-j%apanese.pdf>.
- [23] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pp. 204–215, Washington, DC, USA, 2003. IEEE Computer Society.