社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2004－EVA－9 (6)
2004／6／11

プログラム特化適用法による速度とコードサイズのトレードオフ評価

服部勇祐[†1] 山崎進[†2,†3] 中西恒夫[†3,†4]
松本充広[†2,†3] 北須賀輝明[†3] 福田晃[†3]

プログラム特化とは，プログラムの一部の変数の値がコンパイル時に既知である場合，その値に特化したプログラムを自動的に生成することで，プログラムを高速化するプログラム最適化技術の一つである．プログラム特化では，ループ時の条件判定の結果が静的である，すなわちコンパイル時に既知となる場合は，ループの展開が行われるため，特に多重ループを含むプログラムでは著しくコードが肥大することがある．

本稿では，プログラム特化器の解析において静的であると判定された変数のうち，一部を静的な変数として扱わぬよう特化器に指示することにより，特化時のループの展開を調整し，コードサイズと実行速度のトレードオフを検討する．

ベンチマークプログラムを用いた評価の結果，特化器にパラメータを与えるだけで，特化器を改変させることなく，コード肥大を抑え，実行速度の向上をはかるプログラム特化を行うことができた．

Evaluation of a Program Specialization Method Tradeoff between Execution Time and the Code Size

Yusuke Hattori,[†1] Susumu Yamazaki ,[†2,†3]
Tsuneo Nakanishi,[†1,†4] Michihiro Matsumoto,[†2,†3]
Teruaki Kitasuka[†3] and Akira Fukuda [†3]

Program specialization is a code optimization technology generating a specialized program automatically from a generic program based on predicted values of parameters at compile time. When the conditional expression of the loop is static, the loop is unrolled on program specialization. The code size of a program including loops, especially nested loops, is overgrown.

In this paper loop unrolling is controlled by treating static parameters, parameters whose values are predicted at compile time as dynamic parameters, parameters whose values are not predictable at compile time.

This treatment prevents loop unrolling. This paper demonstrates tradeoff between the code size and the execution time. The evaluation shows that we can specialize with reducing execution time and without overgrowing the code size. Moreover, our method never requires any modificaton of the program specializer.

1. はじめに

特殊な環境で使用されるソフトウェアでない限り，プログラム開発は汎用的なライブラリを用いて行われる．汎用的なライブラリは再利用が可能であるため，プログラムの保守性が向上し，バグの発生を抑えることができる．しかし，ライブラリはどこでも使われることを想定して作られているため，最終プログラムのコードサイズや実行速度の点で無駄を生じがちである．

そこで，汎用的な関数を，よく使われる場面に合わせて特化することによって，プログラムの効率化をはかるプログラム特化の研究が行われ，実行速度の改善の試みがなされてきた[3]．しかし，プログラム特化には，実行速度の向上と引き替えに，コードサイズが肥大するという問題がある．これは，ループを含む関数にプログラム特化を行うと，コードサイズが肥大するというものであり，特に多重ループを含む関数では，入れ

†1 九州大学大学院システム情報科学府
Graduate School of Information Science and Electrical Engineering, Kyushu University
†2 福岡県産業・科学技術振興財団福岡知的クラスター研究所
FLEETS, Fukuoka Industry, Science and Technology Foundation
†3 九州大学大学院システム情報科学研究院
Graduate School of Information Science and Electrical Engineering, Kyushu University
†4 九州大学システム LSI 研究センター
System LSI Research Center, Kyushu University

```
int power(int x,  int n)
{
  int r = 1;              …(1)
  while (n > 0) {         …(2)
    r = r * x;            …(3)
    n--;                  …(4)
  }
  return r;               …(5)
}
```

図 1　プログラム特化の例 (特化前)
Fig. 1　An example of program specialization(before)

```
int power3(int x)
{
  int r;
  r = 1;
  r = r * x;
  r = r * x;
  r = r * x;
  return r;
}
```

図 2　プログラム特化の例 (特化後)
Fig. 2　An example of program specialization(after)

子になったループのそれぞれの回数の積だけ内側のプログラムが展開されるため, コードサイズが非常に大きくなることがある.

本研究では, この多重ループ時のコードサイズ肥大に注目し, どのようにすればコードサイズを抑えた効率的な特化が行えるかを調査する. そして, 多重ループをプログラム内に含む際の特化の手法として, 一部ループの展開を抑止するプログラム特化を提案し, 評価結果を基に考察していく.

まず, 2 章でプログラム特化と今回評価に用いる C-Mix/II の簡単な説明を行い, 3 章で多重ループ時の効率的な特化の方法を提案する. そして, 4 章で 3 章で提案した方法を実際に適用した評価結果を述べて, 5 章でその考察を行う. 最後に 6 章でまとめを述べる.

2. プログラム特化の概要

プログラム特化とは, プログラムの中の一部の変数の値がコンパイル時に既知である場合, その変数の関与する計算をコンパイル時に行ったプログラムを生成することで, プログラムを高速化する技術である. 具体的には, 既知の値を元にプログラムの一部を評価する. まずは, 図 1 に特化の適用例を示す.

図 1 のプログラムは $y = power(x, n) = x^n$ の計算を行う関数のコードである. そしてコンパイル時に, この関数が $n = 3$ で使われることが多いということがわかったとする. その時は, 図 1 のような汎用的な関数よりも, 図 2 のような $n = 3$ のときの場合をを専門的に計算する関数 $y = power3(x) = x^3$ を利用した方が処理を高速化することができる.

このように, $n = 3$ など, パラメータがあらかじめわかっているときに, その値を元にプログラムの一部 (この場合は while 文) を評価しておいて, より高速なプログラムコードを自動的に生成することを, プログラム特化と呼ぶ.

プログラム特化には二つの作業が必要である. 一つは特化するときにそれぞれの変数が動的なものか静的なものかを判断する**解析**, もう 1 つは解析を元に特化されたプログラムのソースコードを生成する**変換**である. 関数の引数などの値が与えられていなくても, 前もって計算できるかどうかのことを **binding time** と呼び, 前もって計算できることを **static**, 実際値が与えられないと計算できないことを **dynamic** と呼ぶ. 特化で行われる解析は, 特化時にそれぞれの変数や制御構造が static なものか dynamic なものかが判断される. binding time を判定するための解析のことを, **BTA** (Binding Time Analysis : 束縛時解析) と呼ぶ.

本研究ではプログラム特化器に C-Mix/II[1] を用いた☆. この特化器は, DIKU, University of Copenhagen の Lars Ole Andersen によって開発された C 言語用のプログラム特化器である. BTA には, logic-based BTA を用い, pointwise で monovariant な特化を行う. 詳しい説明については参考文献 1) を参照.

3. 効率的な特化の提案

3.1 コード肥大の原因

特化により, プログラムが高速化されるが, その代償として一般的にはコードが肥大化される. 例えば前章の power の例では, 常に n が 3 であることが保証されない限り, n=3 として特化された power のコードのみならず, 元の power のコードも生成しなければならない.

また, power を特化するとループが展開されるが, この場合コードサイズが n に比例して大きくなる. n が小さい時はともかく, n が大きくなるとコードサイズが非常に大きくなる.

コードが肥大するとさまざまな弊害が現れる.

- 肥大したプログラムを載せるための大きなメモリが必要になる
- 命令キャッシュのミスヒットによるメモリアクセスのため, 実行時間がかかる
- 命令キャッシュのミスヒットによるメモリアクセ

☆ その他の特化器には, LABRI(Laboratoire Bordelais de Recherche en Informatique) の compose プロジェクトで開発された Tempo[2] がある.

スのため, 消費電力が大きくなる

これらの弊害が原因で, 特化前よりかえって速度が落ちることもあり得る.

ゆえに, 効率的なプログラム特化を行うためには, コードの肥大を抑えることが重要である. では特化によるコード肥大の原因を考える. 特化によって主に以下のプログラム変換が施される.

- 定数伝搬
- 複写伝搬
- ループ展開
- 不要コードの削除

定数伝搬と複写伝搬はコード肥大には繋がらない. また, 不要コードの削除は使われなくなったコードを削除する変換であり, コード肥大の原因にはなり得ない. 一方, ループ展開は, ループ時の条件判定をなくし, プログラムを高速化する変換であるが, これは大幅なコード肥大化を招く. コードサイズを抑えるためにはループ展開を抑えることが効果的であると考えられる.

3.2 ループの特化

特化時のループの扱いは, ループの条件判定とループ内部のプログラムの binding time によって, 3 つに分けられる. まず, ループの条件判定とループ内部のプログラムが static であるときは, ループ全体が計算され定数となる. 次に, ループの条件判定が static で, ループ内部のプログラムが dynamic なときは, ループが展開される. 最後に, ループの条件判定が dynamic であるときは, ループは展開されず元のコードのままとなる.

特化後コードサイズが極端に大きくなる場合, 大抵はプログラム内に多重ループを含んでおり, この多重ループが展開されることでコードサイズが膨れ上がる. そこで, 多重ループ内の全てのループの展開を抑止する, 言い換えれば, 多重ループ内の一部のループだけを展開するようにすれば, 完全に多重ループが展開されたものに比べて実行速度は若干低下するものの, コード肥大は抑えられる.

本研究では, 一部変数を dynamic 化することにより, ループ展開の抑止をはかり, コード肥大を回避することを検討する. dynamic 化とは, static な変数として扱っていたものを, dynamic な変数として扱うように特化器にパラメータを与えて指示するという作業である. static な変数の定義と dynamic な変数の定義より, static な変数を dynamic な変数として扱っても, あらかじめ計算ができるものを後で計算することになるだけであり, 特化の結果生成されるプログラムの意味の等価性は保証される.

```
int dloop(int x1){
  int i, j;
  int r = 0;
  for (i=0;i<20;i++){       …(1)
    for (j=0;j<20;j++){     …(2)
      r = r + x1;
    }
  }
  return r;
}
```

図 3 単純な多重ループの例
Fig. 3 An example of simply nested loop

展開を抑止したいループがある時, ループの条件判定に使われている変数を dynamic にすることによって, ループの条件判定自体を dynamic にし, 目的のループが展開されないようにすることができる.

3.3 多重ループの特化

多重ループを完全に展開する場合, それぞれのループの回数を掛け合わせた回数, 例えば図 3 の場合は, ループボディーの部分のコードが $20 * 20 = 400$ 回繰り返されるため, コードサイズが大幅に膨れ上がる. その時は, 前節で述べたように一部変数を dynamic 化することによって, 一部ループの展開を抑止し, コードの肥大を防ぐことができる. 図 3 の場合は, (1)(2) のどちらかのループを抑止することになる.

多重ループにおいては一部ループの展開を行う際, 内側か外側のどちらかのループを展開することになる. では, 内側のループと外側のループでは, どちらのループのみを展開したら効率のよい特化ができるか考える.

内側ループのみを展開した場合は, 内側のループが一つ減り, 外側のループが残る. 一方, 外側ループを展開すると内側ループが外側ループのループ回数だけ展開される. この二つを比較すると, コードサイズの面では制御構造ごと展開する外側ループのみの展開が, 実行速度の面でも制御構造が多くなる分, パイプラインなどの影響で外側ループのみの展開が不利であると考えられる.

以下では, 多重ループ中のループ間に, 依存関係が存在しない場合と, 存在する場合の特化について論ずる.

3.4 ループ間に依存関係が存在しない場合

3.4.1 両方のループを展開した場合

図 3 のプログラムを例に特化を行う. まず, 両方のループを展開した場合, (1) の `for (i=0;i<20;i++)` の部分と, (2) の `for (j=0;j<20;j++)` が展開され, 図 4 のように, `r = r + x1;` が, 400 回分展開される. ここで, このプログラムと特化前のプログラムを比較したところ, 実行速度は 1.1 倍速くなったものの, コードサイズは 7.3 倍にもなっていた. ここで, コードサ

```
static int
dloop(int x1)
{
  int r;
  r = 0;
  r = r + x1;
  r = r + x1;
  ...

  r = r + x1;
  r = r + x1;
  return r;
}
```

図 4 両方のループを展開した場合
Fig. 4 The case of unrolling both loop

```
static int
dloop(int x1)
{
  int i; int r;
  r = 0; i = 0;
  while (i < 20) {
    r = r + x1;
    r = r + x1;
    ...
    r = r + x1;
    r = r + x1;
    i = i + 1;
  }
  return r;
}
```

図 5 内側のループだけを展開するように特化した場合
Fig. 5 The case of unrolling inner loop

イズが 400 倍にならないのは, ループボディ以外の部分のコードサイズの影響である.

3.4.2 内側のループだけ展開した場合

次に, 図 3 の (1) の `for (i=0;i<20;i++)` の部分の展開抑止するべく, 変数 i を dynamic なものとして扱うように特化器に指示を与える. 特化した結果は, 図 5 の通りである. このプログラムと特化前のプログラムを比較したところ, コードサイズは 1.3 倍, 実行速度は 1.3 倍. 両方のループを展開した場合よりも速度が向上し, コードサイズも抑えることができた.

3.4.3 外側のループだけ展開した場合

次に, 図 3 の (2) の `for (j=0;j<20;j++)` の部分の展開抑止するべく, 変数 j を dynamic なものとして扱うように特化器に指示を与える. 特化した結果は, 図 6 の通りである. 特化前のプログラムと比較した結果, 実行速度は 1.1 倍で, コードサイズは 2.1 倍. 両方のループを展開したときよりはよかったものの, 内側のループのみを展開した場合よりも効率が落ちた.

3.4.4 ループ展開を完全に抑えた場合

最後に, 図 3 の (1) と (2) の展開を両方とも抑える特化を行った. この場合, ループ展開は全く行われず, ループ展開以外の特化作業の効果を期待することになる. こうして特化したソースコードが, 図 7 である. 図

```
static int
dloop(int x1)
{
  int j; int r;
  r = 0; j = 0;
  while (j < 20) {
    r = r + x1;
    j = j + 1;
  }
  ...
  j = 0;
  while (j < 20) {
    r = r + x1;
    j = j + 1;
  }
  return r;
}
```

図 6 外側のループだけを展開するように特化した場合
Fig. 6 The case of unrolling outer loop

```
static int
dloop(int x1)
{
  int i; int j; int r;
  r = 0; i = 0;
  while (i < 20) {
    j = 0;
    while (j < 20) {
      r = r + x1;
      j = j + 1;
    }
    i = i + 1;
  }
  return r;
}
```

図 7 ループ展開を完全に抑えて特化した場合
Fig. 7 The case of no loop unrolling

3 と比較して分かるように, for 文から while 文への変換が行われた以外は変化がない. この結果, 実行速度は 1.1 倍で, コードサイズが 1.05 倍. 普通に特化したときよりはよかったものの, 元の関数とあまり変わらない結果となった.

3.5 ループ間に依存関係が存在する場合

図 9 の (1) の部分に注目する. これを見ると, 内側のループの条件判定 `for (j=i;j<20;j++)` が外側のループ変数 i に依存していることが分かる. 従って, i の値により内側のループ回数が変わるため, 内側のループ回数が不定となる. そうすると, 今まで述べてきたような内側のループだけ展開するということが不可能になる.

このように, 多重ループのループ間に依存関係が存在する場合, 内側のループだけ, あるいは外側のループだけ展開するということができなくなることがある. 図 9 の場合は, 外側のループだけ展開することは可能なため, 一部のループ展開の評価をすることは可能だが, 内側だけ, 外側だけのループ展開がともにできない場合には, 多重ループ全体を一つのループとして見なさざるを得ない.

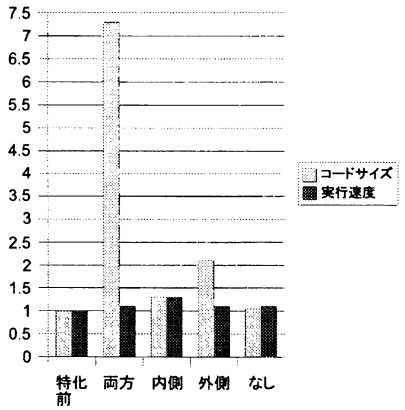

図 8 特化後のコードサイズと実行速度の比較
Fig. 8 Comparing code size and execution time

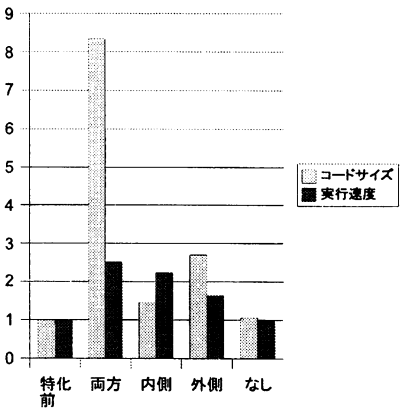

図 10 crc16 の評価結果
Fig. 10 Result of evaluation of crc16

```
int dloop2(int x1){
  int i, j;
  int r = 0;
  for (i=0;i<20;i++){
    for (j=i;j<20;j++){     …(1)
     r = r + x1;
    }
  }
  return r;
}
```

図 9 ループ間に依存関係のある例
Fig. 9 An example of including dependance between loops

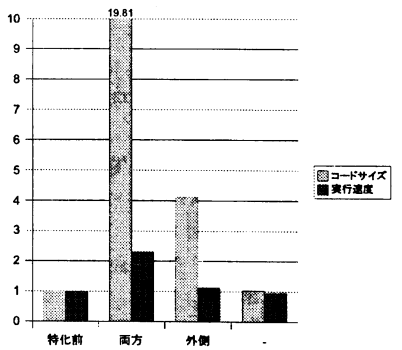

図 11 inssort の評価結果 10
Fig. 11 Result of evaluation of insert sort

4. 評　　価

4.1 評価方法

本章では，一部ループの展開によるコードサイズを抑えた特化の効果を調べるために，ループ展開の箇所（内側だけ，外側だけ，あるいは両方）の変更，最適化の有無による，プログラム速度やコードサイズの変化を調べる．

今回の評価では，一つのプログラムに対して，変数の dynamic 化の有無と，最適化の有無を変更しながら計測を行う．特化器には C-Mix/II を用いた．この評価に用いたプログラムは以下の通りである☆．

- 単純な 2 重ループの例…crc16・crc32
- 依存関係を持つループの例…バブルソート・挿入ソート・選択ソート
- 3 重ループの例（ループ間の依存関係有り）…シェルソート

実験を行った結果を次項に示す．

☆ 参考文献 4) に掲載されていたソースコードを利用している．

なお，評価結果のグラフは，コードサイズが何倍になったか，実行速度が何倍になったかを示している．よって，コードサイズのグラフは短い程良く，実行速度のグラフは長いほど良い．

4.2 評価結果

紙面の都合で，評価結果の一部を示す．ループ間の依存関係のない場合の代表例である 16bit crc の評価結果を図 10，ループ間に依存関係のある場合の評価結果を図 11，特化後に最適化をかけたものの評価結果を図 12 に示す．

5. 考　　察

5.1 ループ間に依存関係のない場合

ループ間に依存関係のない場合の結果を考察する．これらの評価結果を見ると，3 章で述べた内側のルー

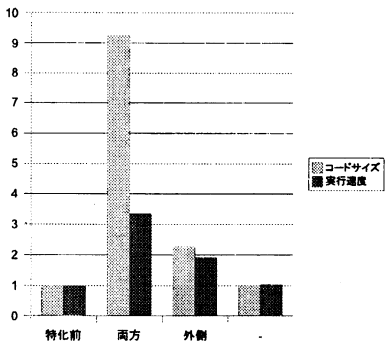

図 12 insssort+最適化の評価結果
Fig. 12 Result of evaluation of insert sort with optimization

プのみを展開した方が, 外側ループのみを展開するよりもコードサイズが小さく, 実行速度の速い効率がよい特化ができることが示されている.

コードサイズの面で比較すると, 内側のループが展開された場合, 内側のループ内のプログラムが内側のループ回数分展開されて一つのループに変換されるのに対して, 外側のループを展開した場合は, 外側のループ回数分内側のループとその周りのプログラムが並ぶことになる. すると, 外側のループのみを展開した場合は, 内側のループ内のプログラムを展開するよりも, プログラムサイズの増大が大きい. これが原因で外側のループを展開した場合に, コードサイズの面で不利になると考えられる.

また, 実行速度の面では, 内側ループを展開した場合, 展開されたプログラムをキャッシュに載せる回数は1度で済むのに対して, 外側を展開した場合は, ループの数だけキャッシュに書き込みを行う必要がある. その書き込みの回数が少ない分内側のループを展開した方が速いといえる.

さらに, パイプラインの影響も大きい. 外側ループの回数だけ while が並ぶため, 分岐の影響で命令パイプライン処理にハザードが生じ, 速度が低下する. これらが原因で実行速度の面でも外側ループのみの展開が不利になったと考えられる.

5.2 ループ間に依存関係のある場合

ループ間に依存関係のある場合の評価結果を考察する. これらは, 内側のループの条件式が外側のループに依存しているため, 内側のループの回数が不定になって, 内側のループのみを展開することができないプログラムである. この場合は外側のループのみを展開した場合と, 全て展開した場合を比較することになる.

これらの結果を見ると差はあるものの, 外側のループのみを展開する方が, 全てのループを展開した場合に比べてコードサイズは小さくなり, 実行速度は低下している. よって, 外側のみのループの展開でも実行速度の向上は少ないものの, コードサイズを抑えた特化は可能であるといえる.

また, 実行速度を向上させるため外側のみのループの展開を行ったコードに対して最適化を行うと, 実行速度が場合によってはかなり向上する. crc のプログラムの外側だけの展開の時も同じような現象が見られた. 特に, シェルソートの場合は全てのループを展開する場合よりも速度が速くなっていた. 外側のループのみを展開した場合には, while 文が並ぶプログラムになるが, そのようなプログラムは最適化の効果が大きいといえる.

6. ま と め

以上の考察で述べたことを簡潔にまとめると,

- 内側のループ内のプログラムがそれほど大きくなければ, 内側のループのみを展開した方が効率がよい
- ループ間の依存関係により, 内側のループのみを展開することができない場合, 外側のループのみの展開でもコードサイズを抑えた特化ができる.
- ただし, 外側のループのみを展開する際には最適化を併用するべきである.

という結論が得られた.

謝 辞

本研究の一部は, 文部科学省・知的クラスター創成事業, 日本学術振興会・科学研究費補助金・若手研究(B)(課題番号 16700039) の支援によるものである.

参 考 文 献

1) Henning Makholm, *Specializing C an introduction to the principles behind C-Mix/II* June 1999.
2) C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noye, S. Thibault, and E.N.Volanschi, *Tempo:Specializing Systems Applications and Beyond*, ACM Computing Surveys,Vol.30, No.3es, Article No. 19, Sep,1998.
3) Neil D. Jones, Carsten K. Gomand, and Peter Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice Hall International, 1993.
4) 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.