プログラミング 13−5
(1997. 5. 21)

アダプティブな実行時コード最適化

杉田 祐也　　原田 賢一
慶応義塾大学理工学部計算機科学科
email: yuuya@hara.cs.keio.ac.jp　　harada@hara.cs.keio.ac.jp

概要

実行時コード生成の技法を使うことによって，実行時にしか得られない情報を利用した，実行効率の高いコードを生成することができる．しかし，実行時にコードを生成するコストは大きいので，生成したコードが繰り返し実行されないと，性能低下を引き起こす場合がある．従来では，実行時コード生成を適用するプログラム中の部分を決定するためには，プログラマがプログラムの性質ばかりでなく，実行されるときの状況も考慮しなくてはならない．

本研究では，手続き型言語に対するインクリメンタルな実行時コード生成の方法を提案する．この方法では，多く実行される部分ほど実行時コード生成が進むので，無駄な実行時コード生成を自動的に避けることができる．本研究の方法を使うことによって，実行時コード生成を適用する部分の決定が比較的容易になり，より積極的に実行時コード生成を適用することが可能となる．

プロトタイプシステムを用いた測定では，一度しか実行されない部分に実行時コード生成を適用した場合のオーバヘッドが，従来の30% に減少した．

An approach for adaptive runtime code optimization

SUGITA, Yuuya　　HARADA, Ken'ichi
Dept. of Computer Science, Faculty of Science and Technology, Keio University

abstract

Run-time code generation(RTCG) is an optimization technique to generate specialized code using run-time information, which cannot be obtained by static analyses. Previous studies have shown that the technique drastically improves execution performance of certain kinds of applications.

However, to gain higher execution performance, the generated code must be executed repeatedly so as to amortize the overhead of the code gengeration. Due to this property, the target of RTCG is limited to routines that are known to be executed in many times in advance.

We propose a technique to incrementally perform run-time code generation. When a target routine is executed in a few times, the system generates a code fragments for the part of the routine. In other words, the amount of generated code is kept small when the routine is less frequently executed, while a frequently executed routine is eventually replaced with fully generated code. Our prototype system shows that the overhead of the code generation for a less frequently executed routine is reduced to 30%.

1 本研究の背景

実行中のプログラムのインストラクションストリームに動的にコードを書き加えることによって，静的な解析では得ることが不可能な情報を利用して最適化したコードを実行時に生成する技法を，実行時コード生成 (run-time code generation) という．この技法によって生成されるコードは，実行時に決定される具体的な値について特化 (specialize) できるので，高度な最適化が期待で

きる [1, 2].

近年，高級言語のソースプログラムから，実行時の情報を利用して特化したプログラムを生成する特化器 (specializer) を生成する言語処理系が提案されている [3, 4]. これらの処理系の処理の流れの概略を図 1に示す. 静的に，ソースプログラムから実行時にコードを生成する特化器を生成する. この特化器が実行中に得られる情報を利用して特化されたコードを生成し、生成したコードが実行される.

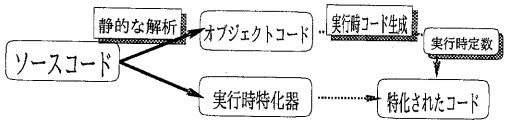

図 1: システムの全体像

これらの処理系では，特化に必要な処理のほとんどを静的に行い，実行時にはコードのコピーと，実行時に得られる情報に依存した定数，ジャンプ先の変更などの軽い処理だけを行う [1]. このように，特化器はコードを生成するコードなので，1 命令の生成に 4 命令程度しか要しない. しかし，実行時コード生成にかかる時間は大きいので，実行時に生成したコードが繰り返し使用されない場合には，実行時コード生成にかかる時間が性能の向上から得られる実行時間の短縮を上回ることになり，全体として性能の低下を引き起こすという欠点がある (図 2). さらに、実行時に生成されるコードが実行さ

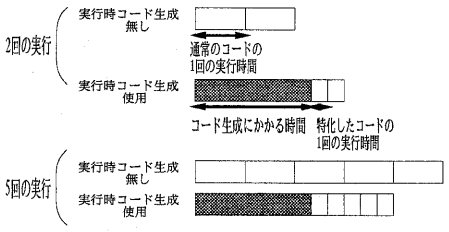

図 2: 実行時コード生成にかかる時間

れる回数を静的に解析することは一般的には不可能である. したがって，これらの処理系では，プログラマがプログラムの性質を考えて，プログラム中の実行時コード生成の対象となる部分と，その部分をどの値について特化するのかを指定しなくてはならない.

従来の方法では，実行時コード生成の対象となる部分は，実行時に生成されたコードのほとんど全てが，繰り返し利用されることが見込まれる部分でなくてはならない. 例えば，プログラムの一部をユーザの入力について特化する場合には，ユーザの入力の性質も含めて注意深く考え，ほとんどの入力に対して，実行時コード生成されたコードが，全体としての高速化を得るのに十分に多く，繰り返し実行されることが見込まれる部分を選ばなくてはならない.

したがって，プログラムを特化することによって高速化が得られることが分かっても，動的な性質がはっきりしない部分に実行時コード生成を適用することは，性能低下の危険がある. このように，実行時コード生成を適用できる範囲は限られていた.

2 本研究での提案

本研究では，実行時コード最適化のなかでも効果の大きいループの特化をインクリメンタルに行う方法を提案する. この方法では，実行時のループの特化を一括で行わずに，ループの先頭に制御が入る度に，ループの内部の適当な繰り返し数分のコードを生成する (図 3). この方法には，次のような利点がある.

- **大きな失敗をしにくい**
  少ない回数しか実行されない部分は，少ない量しかコードが生成されないので，損害が小さくてすむ. 一方，多くの回数，実行される部分はコード生成が進み，特化される.

- **特化器の停止性が問題にならない**
  特化の対象となるループが停止しない場合には，特化器も停止しないので，従来の方法では全体の実行が滞ってしまう. コード生成を分割して行うことによって，この問題を避けられる.

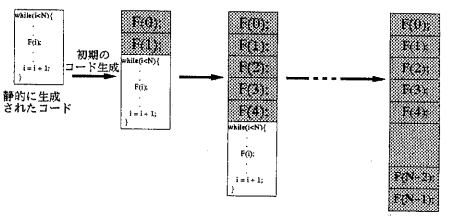

図 3: インクリメンタルなループの特化

これらの性質から，実行時の性質がはっきりしない部分を実行時コード生成の対象とした場合にも，適切に対応できる. したがって，インクリメンタルなループの特

化を使うことによって，従来の方法に比べ，積極的に実行時コード最適化を適用することが可能となる．

2.1 対象言語

対象言語はポインタを含まない手続き型言語とした．この言語には文として，代入文，条件文，繰返し文，文の並び，空文，式として，定数，変数の参照，演算子の適用がある．型は整数型だけであるので，定数は整数だけである．変数には，整数型の変数とその配列がある．なお，関数の呼出しはない．

2.2 特化の対象

通常，実行時コード生成にかかる時間は，生成したコードの1回の実行で得られる実行時間の短縮よりも大きい．したがって，実行時の特化の対象となるループは，全体が繰り返し実行される可能性のあるものに限られる．本論文では，特に2重ループの内側のループを実行時に特化することを考える．したがって，2重ループの内側のループで，ループ条件式が内側のループの入口で外側のループのループ不変値となっているものが，特化の対象となる．

2重ループの内側のループを，外側のループにおけるループ不変値 (loop invariant) について特化する．その結果，ループアンローリングによる最適化よりも大きな効果を得ることができ，ループ変数に関する計算も，すべて除去することができる．

2.3 特化の方法

特化の方法は，Consel[1] の方法と基本的に同じである．特化の対象となるプログラムは，従来の方法と同様に，部分評価 (partial evaluation) で用いるバインディングタイム解析 (binding time analysis)[5] を用いて静的に解析される．

図4のプログラムを使って，解析の例を示す．特化の対象となるループがこれを囲むループの内部にある．変数 c1，c2 は，外側のループでループ不変，変数 v は外側のループでループ不変ではないとする．すると，ループ条件式に現れる変数 i，c1 はループの入口において，外側のループのループ不変値なので，特化の対象となることができる．これらの情報をもとに，バインディングタイム解析が行われる．

このプログラム中のループは特化されて，変数 i は定

```
while(expr) { /* 外側のループ */
    .          /* 特化の対象にならない */
    .
  i = 0;
  sum = 0;
  while(i < c1) { /* 特化の対象 */
    if (even(i))
      sum += i * c2;
    else
      sum += v;
    i++;
  }
    .
    .
}
```

図 4: 特化の対象となるループ

数として扱うことができる．変数 sum は，特化の対象となるループの入口で外側のループのループ不変値 (この場合は定数) なので，これも定数として扱う．特化に

```
T1: sum += [hole];

T2: sum += v;
```

図 5: 図 4 に対応するテンプレート

```
i = 0;
sum = 0;
while (i < c1) {
  if (even(i)) {
    generate_template(T1);
    instantiate_hole([hole], i*c2);
  } else {
    generate_template(T2);
  }
  i++;
}
```

図 6: 図 4 に対応する実行時特化器

よって，実行時定数だけに依存した計算は除去することができる．特化の結果を図5と図6に示す．図5には，実行時定数以外の値に依存した除去できない計算に対応するプログラムの断片が示されている．これらの断片をテンプレート (template) と呼ぶ．これらのテンプレー

トは，図4のif文のthen節とelse節の部分に対応するものである．then節の式i * c2は，実行時定数だけに依存しているので，コードを生成するときの定数を埋め込むための穴(hole)で置き換えられている．else節の変数vは，実行時コード生成時にも値が決定できないので，変数参照が残っている．

これらのテンプレートは静的にコンパイルされる．テンプレートの各文は，同じテンプレート内の他の文との関係だけでなく，他のテンプレートとのつながりを考慮して最適化を施すことができる．

図6に示した実行時特化器(runtime specializer)が，実行時にテンプレートを組み立て，テンプレートに空いている穴に実行時定数を埋め込むことによって，ループを特化したコードを生成する．ただし，この特化器は，一括に特化を行うものである．

図4にあるif文は，条件式が実行時定数だけに依存していて実行時コード生成時に値が決定できるので，テンプレートからは除去され，実行時特化器で実行される．同様に，while文の条件式，変数iの計算も実行時特化器で1度だけ計算すればよく，テンプレートからは除去できる．ただし，ループの外に変数iの参照がある場合には，変数iにループ終了時の値を代入するためのコードが必要である．

2.4 コード生成のタイミング

インクリメンタルなループの特化では，より頻繁に実行されるループがより多く特化される．この特性は，外側のループが何回か繰り返される度に，実行時コード生成をすることで得ることができる．

このときに，既に特化されているコードを実行することによって得られる実行時間の短縮を利用して，続くコード生成を行う．特化したコードを実行した回数，即ち，外側のループが繰り返された回数によって，得られた時間短縮が推定できる．そこで，次の実行時コード生成にかかると見込まれる時間よりも大きな時間短縮が得られた時点で，実行時コード生成を行う(図7)．

最初の実行時コード生成ではコストを償却できない可能性があるコードを生成をする．初回に生成するコードの量は，実行時に生成したコードが1度しか実行されない場合の性能低下をどの程度許容するかに依存する．この許容できる性能低下の程度によって，特化の対象となるループは制限される．最初に実行時コード生成する

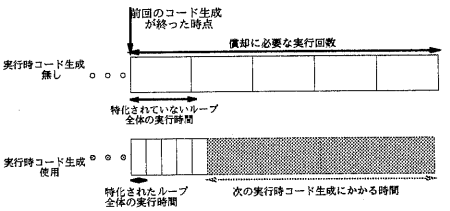

図7: 実行時に生成されるコード

コードの量が十分でない場合には，3節で述べるような実行時に生成したコードへ制御を移すためのオーバヘッドが，特化することによって得られる実行時間の短縮を上回るためである．このように，生成するコード量に対し，特化によって得られる利益の小さいループを特化の対象からはずす．

2回目の実行時コード生成は，初回に生成されたコードの実行により，初回の実行時コード生成のコストが償却され，2回目の実行時コード生成のコストの見込み分を打ち消すのに十分なだけの実行時間が得られた後に行われる．以降，同様にループの特化が続けられる．したがって，実行時コード生成のコストを保守的に見込んでいて，初回のコード生成のコストが償却できれば，実行時コード生成を利用しない場合よりも高速になる．

既に生成したコードによって得られる実行時間の短縮を利用しているので，生成したコードの量が多いほど，次に生成できるコードの量を多くするか，次の生成が可能になるまでの，既に生成したコードの実行回数を小さくすることができる．したがって，大きなループを特化することも十分に可能である．また，1回に特化する繰り返し数を大きくできるので，インクリメンタルなコード生成を行うためにかかるコストが相対的に小さくなっていく．このように，繰り返し数の多いループの特化を，従来と同様の性能を維持して行うことができる．

3 実装

2節での提案に基づいた簡単な処理系をi386系のマイクロプロセッサ上で実装した．この章では，実装上の問題について述べる．

3.1 システムの全体像

システムの全体像を図8に示す．ソースプログラムが与えられると2.3節で示したような解析を行い，テンプレートと実行時特化器をソースレベルで生成する．ソースレベルのテンプレートはコンパイルされて，バイナ

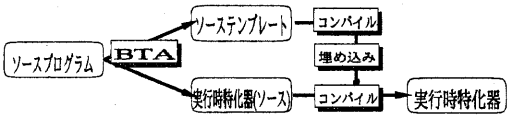

図 8: システムの全体像

リのテンプレートを生成し，実行時特化器に埋め込まれる．この実行時特化器をコンパイルして，バイナリの実行時特化器を生成する．

このように，実行時特化器はそれぞれの特化の対象について特化されており，また，コンパイラによって最適化することができるので，より高速にコードを生成できるようになる．

3.2 実行時コード生成のタイミングの決定

タイミングの決定には自由度が残されているので，最適なタイミングを決定するためには，許容できる最大のオーバヘッドを指定しなくてはならない．この実装では，実行時に生成したコードが1回しか実行されなかった場合の実行時間が，実行時コードを使用しない場合の2倍程度になるように初回に実行時コード生成するコード量を決定した．

タイミングの決定には，特化によって得られるループ1回分の実行による時間短縮，実行時特化器のループ1回分のコードの生成時間が必要である．これらの値はコードの命令数から推定値を求める．特化の対象となるのはループであり，その内側にはループがないのでよい推定値が得られる．

2回目以降のコード生成は，償却に必要な実行回数が十分小さくなるまで，1度に生成するコード量はそのままで，次のコード生成までの実行回数を半分にしていく．償却に必要な実行回数が十分に小さくなった後は，次のコード生成までの実行時間はそのままで，1度に生成するコード量を2倍にする．

3.3 実行時に生成するコード

今回の実装で用いた実行時特化器の構成と，それによって生成されるコードについて述べる．図9に示したように，静的なコード (通常にコンパイルされたコード) のループの入口に間接コールを埋め込む．このコールは，実行時特化器を呼び出す．実行時特化器は，それまでに呼ばれた回数を記録し，その回数から3.2節の方針に従い，コードの生成が可能な場合には，コード生成を

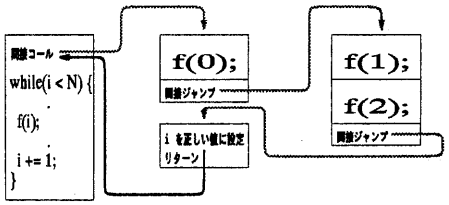

図 9: 実行時に生成されるコード

行う．ループの特化が終了した場合には，この間接コールのコール先を生成されたコードへ直接飛ぶように変更する．

図9では，初回に `f(0)` のコードを生成し，2回目に `f(1)`，`f(2)` のコードを生成した状態を示している．初回のコードだけが生成されているときには，`f(0)` のコードについている間接ジャンプは，静的なコードとの整合性をとるためのトレイラ (trailer) へのジャンプとなっている．2回目のコードを生成するときに，このジャンプ先を変更して2回目に生成したコードからトレイラへのジャンプを生成する．

このように，間接ジャンプを利用する理由は最近のマイクロプロセッサの構造にある．最近の多くのマイクロプロセッサではコード領域に対して書き込みを行っても，その変化をキャッシュに反映させるためには，フラッシュ (flush) を明示的に行う必要がある．このフラッシュにかかるコストを無くすために，間接ジャンプを使用する．

ただし，i386系のマイクロプロセッサでは上位互換のためにフラッシュを行う必要はない．この実装では，特定のアーキテクチャに依存しないために間接ジャンプを使用している．

4 評価

4.1 評価の条件

評価は，PC/AT互換機で行った．詳しい条件を表1に示す．実行時間の測定には tcsh の `time` コマンドを利用し，3回の測定の平均を結果としたものである．

表 1: 評価の条件

CPU	Pentium 90MHz
キャッシュ	asynchronous 256KB
メインメモリ	32MB
OS	FreeBSD-2.1.6-RELEASE
コンパイラ	gcc 2.6.3

4.2 評価

評価用のプログラムとして，逆ポーランド記法の式を計算するインタプリタを使った．このプログラムは，入力として逆ポーランド記法で記述された式と式に現れる変数の値を受け取り，計算結果を出力する．このプログラムを式について特化して，異なる変数の値を与えて繰返し使用する場合を考える．このような状況は，表計算のようなアプリケーションで頻発すると考えられる．

表 2: コードの使用回数に対する実行時間比

使用回数	一括 (μ秒)	分割 (μ秒)
1	20.5	10.0
2	10.9	7.4
3	7.7	6.5
5	6.3	5.8
6	4.5	5.6
10	3.2	5.3
30	1.8	4.7
50	1.6	3.8
200	1.3	2.2
∞	1.2	1.7

表2は，インタプリタで式 $x^3+y^3+z^3$ の計算 1 回あたりの実行時間の比較である．実行時コード生成を使わずに，gcc(GNU C compiler) の最適化レベル 2 でコンパイルしたコードの場合の実行時間は 5.4 μ秒である．

実行時コード生成を利用する場合には，インタプリタのエンジン部のループを，入力された式について特化する．一括にループを特化した場合の損益分岐点は 6 回の実行である．実行時コード生成したコードの実行回数が非常に多い場合には，静的なコードに比較して約 4.5 倍の速度向上が得られる．しかし、実行時に生成したコードがあまり利用されない場合の性能低下が著しい．

インクリメンタルなループの特化をした場合には，計算される式が順次，コンパイルされていく．ループ全体を一括に特化した場合に比べて，実行時に生成したコードを少数回しか使用しなかったときの実行速度の低下が小さくなっている．しかし、情報管理のオーバヘッドと，ジャンプ命令がパイプラインを乱すことの影響を受けて，損益分岐点に達するまでの実行回数が多くなってしまった．$x^3+y^3+z^3$ の式を計算するときに，インタプリタのループは 19 回と繰り返し数が小さい．また，ループ 1 回分のコードは約 3 命令と小さいので，これらのオーバヘッドの影響が大きくあらわれている．

もっと長い式についてインタプリタを特化をした場合には，生成したコードの実行回数が小さいときの実行時間の差が大きくなる．インクリメンタルなループアンローリングでは式の長さに関係なく，初回の実行時コード生成するコード量は一定なためである．また，実行回数が大きいときの実行時間の差は小さくなる．実行時に生成するコードの量が多くなると，インクリメンタルな生成をするためのコストが相対的に小さくなるためである．

5 まとめ

インクリメンタルなループの特化の技法を提案した．評価に使った例では，生成したコードが 1 回しか実行されない場合のオーバヘッドが従来の約 30% に減少した．このことから，従来に比べて，安全に実行時コード生成を適用できることが示された．また，ループ 1 回分のコードが 3 命令と小さいループに対しても有効であることが確認できた．

これらのことから，より積極的に実行時コード生成を利用できることが示された．

謝辞

本研究に貴重な示唆と助言をくださった，東京大学の増原英彦氏と慶応義塾大学の滝本宗宏氏に感謝します．

参考文献

[1] C.Consel and F.Noel: A general approach for runtime code generation and its application to C, *ACM Symposium on Priciples of Programming Languages* (1996).

[2] David Keppel, S. J. E. and Henry, R. R.: A case for runtime code generation, Technical Report 91-11-04, Department of Computer Science and Engineering, University of Washington (1991).

[3] Joel Auslander, M. P.: Fast, Effective Dynamic Compilation, *ACM Conference on Programming Language Design and Implementation* (1996).

[4] Leone, M. and Lee, P.: Optimizing ML with runtime code generation, *ACM Conference on Programming Language Design and Implementation* (1996).

[5] N. Jones, C. G. and P.Sestoft: *Partial Evaluation and Automatic Program Generation*, Prentice Hall (1993).