数理解析研究所講究録
1125巻 2000年 26-37

実行時最適化のためのコード生成インタフェース

A code generation interface for run-time optimization

藤波 順久
Nobuhisa FUJINAMI

ソニー(株) インフォメーション&ネットワーク研究所
Information & Network Technologies Laboratories, Sony Corporation

概要

プログラムの実行時に部分計算を行って、結果の機械語コードをメモリに書き込むことで、実行時最適化を行うシステムでは、結果の機械語の性能とともに、コード生成自体の性能が重要である。著者の作成した実行時最適化システム「C++ Doubler」では、最適化対象のプログラムについて特殊化された実行時コード生成機を生成するために、スタックマシンを抽象モデルとしたインタフェースを採用し、各種の最適化を実行時に適用可能にしている。本論文では、このインタフェースの実現方法について述べるとともに、tcc や tempo など、他のシステムで用いられている方法と比較する。

1 はじめに

実行時最適化は、プログラムの実行開始後に初めてわかる情報を使って、プログラムを最適化する方法である。部分計算に基づいた実行時最適化では、プログラムの実行開始時に与えられたコマンドライン引数や、実行開始後のユーザの入力、ファイルからの入力、計算の中間結果などの値を既知引数として、プログラム中の関数の部分計算を行う。実行時最適化で使われる既知引数は、実行時定数と呼ばれる。部分計算の結果をソースプログラムとして出力してしまうと、コンパイラを起動する必要があり、すみやかに実行することができないので、部分計算結果はマシンコードの形でメモリに書き込まれるのが普通である。

著者の作成した実行時最適化システム「C++ Doubler」は、部分計算に基づいた、C++言語のための実行時最適化システムである。部分計算の対象は、追加キーワード **runtime** で指定されたメンバ関数(メソッド)である。C++ Doubler は、オブジェクト指向のカプセル化機構を利用して、実行時定数の検出を行っている。すなわち、対象関数が使っている非公開データメンバ(インスタンス変数)をの使い方を解析して実行時定数の候補とする。

実行時最適化を行うシステムでは、マシンコードを新たに生成するコストに見合うだけ、生成されたコードによる性能向上が見込めなければならない。そのため、生成されるマシンコードの性能を高くすることと同様に、マシンコードを生成するコストを低くすることが重要になる。近年の実行時最適化システムでは、最適化対象の関数に対して、専用のコード生成機をあらかじめ生成しておく方法が普通である。プログラムの実行開始後は、ソースプログラムやその中間表現を扱う必要がないため、マシンコード生成は高速に行われる。

本論文では、C++ Doubler が最適化対象関数に対してコード生成機を生成する際に用いられているインタフェースである、SSIF について述べる。SSIF は、スタックマシンを抽象モデルとしたインタフェースであり、C++ Doubler のターゲットマシン依存部分を隠蔽する役割を果たす。すなわち、C++ Doubler の唯一のターゲットマシン依存部分は、SSIF の実現部である。実現部の役割は、コード生成機を生成することであるが、ターゲットマシンに依存した最適化も担当する。プログラムの実行前にわかる最適化は実現部で直接実行し、そうでないものはそれを行うプログラム片をコード生成機に埋め込む。

以下本論文では、2節で C++ Doubler の構成を概観した後、3節で、SSIF の設計方針と、その実現方法を述べる。4節では、他の実行時最適化システムで使

われているコード生成の方法を紹介し、SSIF と比較する。最後に5節でまとめを行う。

2 C++ Doubler の構成

2.1 実行時コード生成でオブジェクト指向言語を使う正当性

1章で述べたように、実行時コード生成は、実行時にしかわからない値について最適化されたマシンコードを生成することで、プログラムの効率を上げる。例えば、計算の中間結果や、ユーザの入力である。このような値を操作するプログラムがオブジェクト指向言語で書かれているなら、実行時にわかる値を表すインスタンス変数を持つオブジェクトを定義するのが自然である。例えば、ストリーム入出力関数をプログラムするなら、プログラマはファイルやソケットのデスクリプタをストリームオブジェクトのインスタンス変数に代入するだろう。ストリームオブジェクトにはストリームを読み書きするメソッドがあって、デスクリプタをその実行時定数として持つことになるだろう。別の例は、3 次元のシーンの生成とレンダリングである。プログラマは、レンダリングの最中は実行時定数であるシーンを、グラフィックオブジェクトや視点や光源などを表すインスタンス変数を持ったシーンオブジェクトとして表現するだろう。シーンオブジェクトのレンダリングのためのメソッドは、実行時コード生成によって最適化できる。

インスタンス変数に注目する利点は次の通りである:

コード生成/無効化のタイミングの自動化: オブジェクト機能のカプセル化機能のために、非公開のインスタンス変数に対する代入はすべて、ポインタ経由の間接アクセスを除いて、クラスやそのメソッドの定義から知ることができる。システムが、いつコードを生成/無効化するべきかを知っているため、プログラマはコードに埋め込まれた値と実際の値との整合性を保つためにプログラムに注釈を付けたり適切なパラメタを提供したりすることから解放される。

生成されたコードの管理の自動化: 生成されたコードはインスタンスの一部とみなせるので、その管理は、オブジェクト指向言語のインスタンス生成/破壊機構に任せることができる。同じメソッドに対する複数のマシンコードルーチンの管理は自明である。生成されたマシンコードは、元のメソッドを使う代わりに自動的に起動できる。プログラマはコードのためのメモリ管理と、コードを起動するためのプログラム書き換えから解放される。

システムのプロトタイプ [12] [13] は、不変なインスタンス変数にのみ注目している。次のような条件を満たすオブジェクトのクラスに注目している:

> 非公開インスタンス変数(C++ではprivate メンバ) のいくつかが、どのメソッドでも変更されない*。

オブジェクト指向言語のカプセル化機構により、そのような変数はオブジェクトの一生の間不変である。そのような変数を参照するメソッドはどれでも、x を不変な変数のベクトル、yをメソッドへのそれ以外の入力変数のベクトルとして、2 入力関数 $f(x,y)$ とみなすことができる。これは x に関して部分計算でき、オブジェクトのインスタンス生成時にマシンコードにコンパイルすることができる。

これは [7] [8] で擬似不変インスタンス変数—ある期間は一定の値を保つようなインスタンス変数—を扱うように拡張された。

著者のシステムはプログラマに、実行時コード生成が適用されるメソッドを指示することを要求する。適用可能性の自動検出は可能だが実用的でない。なぜなら、実行時コード生成を適用し過ぎると、コンパイル時間と実行可能ファイルのサイズを増大させるからである。

システムの実装では、入力言語として C++を使っている。新しいキーワード`runtime` をメンバ関数の宣言の前につけると、実行時コード生成を指示する。システムはそのメンバ関数で使われるが変更されないすべての「既知」データメンバが実行時定数だと仮定する。もしあるデータメンバが頻繁に変更され、プログラマがその値を生成されるコードに埋め込むことを望まないなら、プログラマはキーワード`dynamic` をそのメンバの定義の前につければよい。不適切にキーワード`runtime` と`dynamic` を挿入すると、プログラムの性能は落ちるかもしれないが、プログラムの意味は変えないことに注意してほしい。

図1はグラフィクスオブジェクトを表すクラスの例を示す。クラス`ObjectTable` は二つの`private` データメンバ`count` と`table`、コンストラクタ、そして二つの`public` メンバ関数`add` と`intersect_all` を持つ。プログラマがキーワード`runtime` を`intersect_all` の前に挿入して、実行時コード生成を指示したとしよう。`intersect_all` はレンダリングで多くの回数実行されるからである:

```
runtime const Object *
```

---

* C++ではコンストラクタを除く。

```
class ObjectTable {
 private:
  int count;                    // グラフィクスオブジェクトのカウンタ
  Object *table[MAXOBJECT]; // グラフィクスオブジェクトへのポインタ配列
 public:
  ObjectTable(): count(0) {}
  int add(Object *p);           // グラフィクスオブジェクトを追加する
  const Object *intersect_all(Ray &, myfloat &);
};                              // 交線が最初に交差するオブジェクトを返す
```

図1　クラス ObjectTable の定義

intersect_all(Ray &, myfloat &);
システムはintersect_all のためのマシンコードを、データメンバcount とtable が実行時定数であるとして、生成しようとする。メンバ関数add の起動は、データメンバを変更し、生成したコードを無効化する。

2.2　実行時コード生成機の自動生成

実行時に行われる部分計算のコストは、それが達成する節約に比べて小さくなければならない。そのため、部分計算の結果は、コンパイル時間を節約するため、ソース言語ではなく、ネイティブなマシン語でなければならない。コンパイラを起動せずに実行時コード生成を行うためには二つのアプローチがある。一つは [6] で述べられているような、中間表現からマシンコードを生成し最適化するライブラリを呼び出すものである。もう一つは [9] で述べられているような、特殊化されたコード生成機を使ってマシンコードを直接生成するものである。後者が意欲的かつ高速であるため、著者はそれを選んだ。

この節は自動的に実行時部分計算機を生成する方法について議論する。著者は実行時部分計算機を生成するのに生成拡張生成機を使った。正確に何を使ったか述べるために、生成拡張生成機の形式的記述が以下で与えられる。出力言語が入力言語と異なる部分計算機を記述するために、伝統的なものとは異なる表記法を用いる。もし f がかんすうなら、$[f]_L$ を f の言語 L での実装の一つとする。いいかえれば、プログラム $[f]_L$ の意味は f である。以下の議論を単純にするために、f は一つの既知入力 x と、一つの未知入力 y を持つとする。

もし f が言語 A で実装されていたなら、部分計算機 α^{AB} は次の式を満たす†。

$$\alpha^{AB}([f]_A, x) = [f_x]_B \quad s.t. \quad f_x(y) = f(x,y)$$

もし α^{AB} が言語 C で実装されていたら、別の部分計算機 α^{CD} を α^{AB} と f に適用できる:

$$\alpha^{CD}([\alpha^{AB}]_C, [f]_A) = [\alpha^B_f]_D \quad s.t. \quad \alpha^B_f(x) = [f_x]_B$$

結果 $[\alpha^B_f]_D$ は D で実装された f の生成拡張である。それは f の既知入力 x を引数にとり、f の特殊化されたバージョンを B で出力する。言い換えれば、それは f に特殊化された部分計算機である。関数 f はそれにハードコーディングされている。

さらに、もし α^{CD} が言語 E で実装されていれば、別の部分計算機 α^{EF} を α^{CD} と α^{AB} に適用できる:

$$\alpha^{EF}([\alpha^{CD}]_E, [\alpha^{AB}]_C) = [\alpha^D_{\alpha^{AB}}]_F$$
$$s.t. \quad \alpha^D_{\alpha^{AB}}([f]_A) = [\alpha^B_f]_D$$

結果 $[\alpha^D_{\alpha^{AB}}]_F$ は F で実装された生成拡張生成機である。それは f の A での実装を引数としてとり、f の生成拡張を D で出力する。f は生成拡張生成機の唯一の入力なので、生成拡張はコンパイル時に生成できる。A, B, C, D, E, F はすべて独立であることに注意してほしい。

生成拡張生成機を部分計算機の自己適用で素朴に生成すると、各種の問題が起きるため、著者は現代的な方法である、生成拡張生成機を直接書く (部分計算機を別の部分計算機に手動で適用する) 方法を使ってこれらの問題を避けている。

生成拡張生成機 $\alpha^D_{\alpha^{AB}}$ の A, B, D の選択には順列組み合わせがある。S をソース言語、M をマシン語とする。C-Mix [1] は $\alpha^S_{\alpha^{SS}}$ をオフライン部分計算に使っている。S は ANSI C 言語である。FABIUS [9] は $\alpha^M_{\alpha^{SM}}$ を遅延コンパイルに使っている。

著者は $\alpha^S_{\alpha^{SM}}$ を選んだ。これは実行時コード生成機の生成機である。f のソースプログラムを入力としてとり、生成拡張 α^M_f (実行時コード生成機) をソース言語で出力する。A と B の選択は自明だが、$D = S$ とした理由を述べるべきだろう。まず、生成拡張の最適化のほとんどを、ソース言語のコンパイラに任せることができる。次に、マシンコード生成機を 1 レベルでだけプログラムすればよいので生成拡張生成機を書くのが簡単になる。最後に、元のプログラムとコンパイル

† この議論は部分計算機 α^{XY} が常に止まると仮定している。

時に生成された生成拡張を混ぜ合わせるのが簡単である。生成拡張は元のプログラムのシンボルを自然にアクセスできる。

2.3 システムの構成

システムはC++コンパイラのプリプロセッサとして実装されている。システムのプロトタイプは、Sony NEWS ワークステーション (MIPS R3000/R4000) で走る AT&T C++フロントエンドと MIPS C コンパイラ用に開発された。現在の実装は、Win32API を持つ 80x86 ベースのコンピュータで走る、Borland C++コンパイラ (バージョン 4.0 以上) または Microsoft Visual C++コンパイラ（バージョン 4.0 以上）のためのものである。実行可能ファイルの名前は RPCC.EE である。C++をソース言語として選んだ理由は次の通りである:

- C++は静的型宣言があるので、実行時コード生成機で使う値の型を決めるのが容易である。
- C++はたいそう効率の良いオブジェクト指向言語なので、システムは高級言語で書かれたプログラムの可能な最良の実装を提供する潜在能力を持つ。

システムの基本設計は他のオブジェクト指向言語にも、もし静的な型付けがあり、何等化の方法でメモリに書かれた値を実行できるなら、適用可能である。さらに、静的型推論の研究結果と組み合わせれば、純粋なオブジェクト指向言語にも適用可能である。

図2はシステムの全体構成を示す。上半分がコンパイル時の動作を、下半分がプログラムの実行を表す。コンパイル時にはまず、ソースプログラム中の C++プリプロセッサ擬似命令が処理される (Borland C++では CPP32.EXE)。次に、RPCC.EXE がプログラムを分析し、必要なら実行時コード生成機を C++で生成する。コード生成機、それを起動するコード、生成したコードを起動したり無効化したりするコードが元のプログラムに挿入される。出力は通常の C++コンパイラ (Borland C++では BCC32.EXE) を使って実行可能ファイルへとコンパイルされる。ソースプログラムとその中間表現は、このコンパイル時にだけ扱われる。

実行時には、コード生成機は実行時定数を引数として起動される。それらは実行時定数に対して最適化されたメンバ関数をマシンコード形式で生成する。各コード生成機は一つのメンバ関数に特有のものである。コードは直接メモリに書かれ、ソースプログラムもその中間表現も使われないので、コード生成は効率的である。一つのコード生成機は、異なる実行時定数に対する複数のマシンコードルーチンを生成するかもしれない。生成されたルーチンは、静的にコンパイルされたコードより効率がよいと期待され、元のメンバ関数の代わりに起動される。

```
class A {
 private:
  int x;
 public:
  A(int i);
  runtime int f(int y);
};

A::A(int i): x(i) {}
int A::f(int y) { return y-x*x; }
```

図3 RPCC.EXE への入力の例

図3は RPCC.EXE の入力の例を示す。図4は出力 (読みやすいように一部省略し、コメントを追加した) である。プリプロセッサ RPCC.EXE はキーワード runtime のついたメンバ関数を処理し、実行時コード生成機を C++で生成する。生成されたマシンコードルーチンへのポインタが、データメンバとしてクラスに追加され、コード生成機がメンバ関数として追加されている。処理されたメンバ関数は、生成されたコードの有効性を調べ、必要ならコード生成機を起動し、生成されたコードを起動するようなコード片に置き換えられている。プリプロセッサはまた、デストラクタと、生成されたマシンコードに埋め込まれた値を変更するようなメンバ関数に、生成されたマシンコードを無効にするコードを挿入する。

3 実行時コード生成機を生成するインタフェース

この節では、C++ Doubler で用いられている、実行時コード生成機を生成するインタフェースである、SSIF について述べる。SSIF は、その実現部に C++の関数とマクロを使ったインタフェースである。その目的は、実行時コード生成機を C++言語で出力することである。すなわち、このインタフェースが使われるのは、プログラムのコンパイル時であり、実行時には、それの出力したコード生成機が動作する。

SSIF は現在、x86 アーキテクチャを対象に実装されている。以前の実装では MIPS アーキテクチャ用となっていた。移植の際にインタフェース部分には変更が不要であった。すなわち、SSIF はターゲットマシン依存部分を実現部の中に隠蔽できるようになって

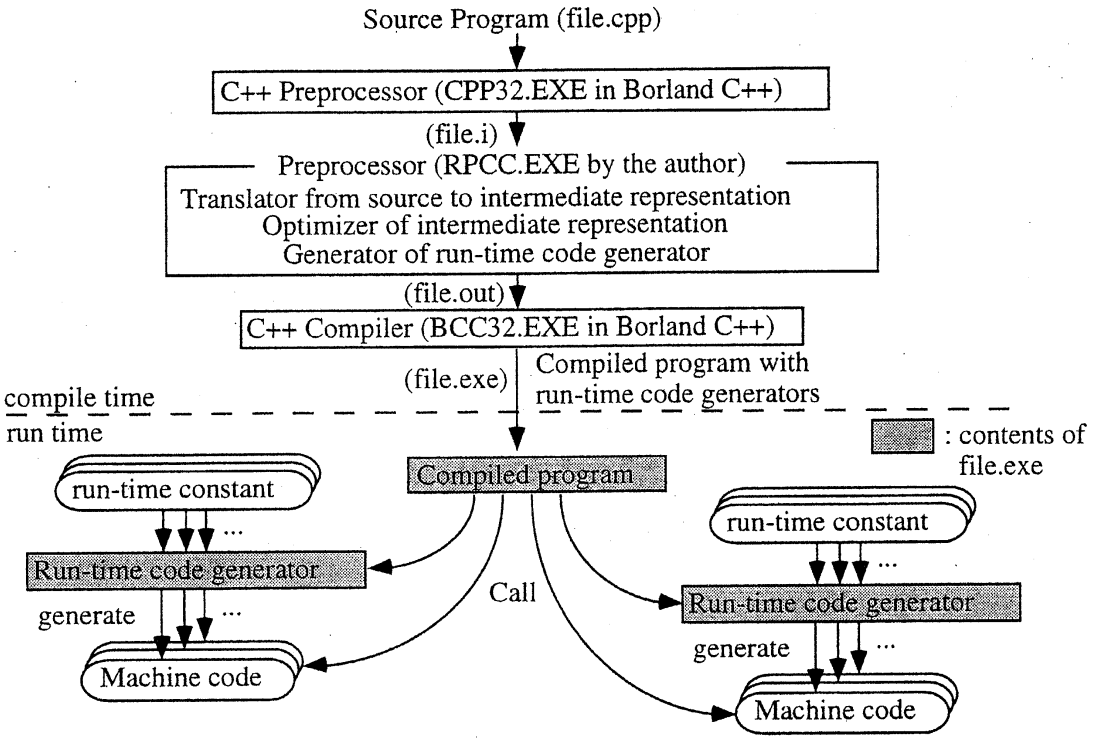

図2　実装されたシステムの構成

いる。

インタフェースを実現しているマクロまたは関数が実行されると、該当するマシンコードをメモリに書き込むような、C++プログラム片が生成される。場合によっては、このインタフェースを呼ぶ側で、そのC++プログラム片で使っているの変数に値をセットする必要がある。セットされた値は、それを使うマクロまたは関数を呼んだ後は変更してよい。

インタフェースを実装する関数は、覗き穴最適化を行うために状態を持っており、呼び出されたマクロまたは関数に対応するプログラム片が直ちには生成されないことがある。現在の実装では、覗き穴最適化は基本ブロックの中でだけ行うようになっている。状態としてバッファリングされた処理は、ssFlush 関数を実行すると吐き出されて空になる。

インタフェースの一覧は、付録Aを参照してほしい。ここでは例を用いてSSIF の働きを説明する。前節では例として、図3で示したプログラム中の関数

```
int A::f(int y) { return y-x*x; }
```

に対する実行時コード生成機を、x を実行時定数として生成したが、そのときC++ Doubler のコード生成機生成部は次のような順でSSIF を呼び出す。

- `ssAnalyze` を呼び出すことで、簡単な大域的レジスタ割り当てが行われる。
- `ssInit` を呼び出すことで、コード生成機の出力開始を指示する。
- “`qq0i=x;`” を出力する。
- “`qq1i=x;`” を出力する。
- “`qq0i=(qq0i*qq1i);`” を出力する。
- `ssPushVar` を呼び出す。引数はローカル変数`y` を表す構造体へのポインタである。これにより、`y` の値をロードするコードを生成するプログラム片が出力される。
- `ssSubG` を呼び出す。引数は0と、整数型を表すオブジェクトへのポインタである。その意味は、実行時定数として`qq0i` の値を使うことである。これにより、実行時定数を減算するコードを生成するプログラム片が出力される。
- `ssPopRet` を呼び出すことで、上で計算した値を戻り値として関数を終了するコードを生成するようなプログラム片が出力される。実際には、関数を終了するコードを生成する部分はマクロ名が出力されるだけである。
- `ssExit` を呼び出すことで、コード生成機の出力終了を指示する。
- `ssEnter()` を呼び出すことで、関数の入口のコードを生成するプログラム片と、関数を終了するコードを生成するマクロ定義が出力される。

```
#include <qqmacro.h>              // 実行時コード生成のためのマクロと関数

class A {
 private:
  int x;
 public:
  A(int i);
  int f(int y);
  ~A()                            // デストラクタ
  char *qq_0Lf;                   // 生成されたコードへのポインタ
  void qq__0Lf() const;           // コード生成機
  static char *qql_0Lf;           // f のラベル"generate"の番地
  static char *qql__0Lf();        // qql_0Lf を初期化する関数
};

A::~A() { if(qq_0Lf!=qql_0Lf) delete qq_0Lf; }

A::A(int i): x(i) ,qq_0Lf(qql_0Lf){}

int A::f(int ) {
retry:
  asm MOV ECX,this;
  asm JMP DWORD PTR [ECX].qq_0Lf; // 生成されたコードへジャンプ
generate:
  qq__0Lf();                      // コード生成機を起動
  goto retry;
}

char *A::qql_0Lf=qql__0Lf();

void A::qq__0Lf() const {
  char *qqcode;                   // コードの番地
  // 関数の入口のコード生成機（略）
  qqMOVdx(0,5,12);                // MOV EAX,[EBP+12] ; y
  qqSUB_I(0,(int)x*x);            // SUB EAX,x*x
  // 関数の出口のコード生成機（略）
  *(char **)&qq_0Lf=qqcode;       // コードの番地をセット
}
```

図4　RPCC.EXE からの出力の例
(マクロ qqXX(YY) はオペランド YY を持つ命令 XX をメモリに書き込む)

関数の入口のコードを生成するプログラム片を出力する、ssEnter() は、最初ではなく最後に呼び出される。なぜなら、関数の入口ではローカル変数などで使用するためのスタックフレームを確保したり、呼び出し先で保存するべきレジスタで使われているものを保存する必要があるが、その大きさや必要性はコード生成機を最後まで生成してみないとわからないからである[‡]。関数の出口についても、複数の出口がある場合などに同様の問題が発生するため、ssEnter() でマクロを定義して、ssPopRet でそれを使う。なお、コード生成機のコードの順序としては、ssEnter の出力す

‡ SSIF では動的なレジスタ割り当てを行っていないので、コード生成機の生成後はこれらが確定する。

るプログラム片のほうが先になるように、一時ファイルを用いて並べ換えを行っている。

出力されたコード生成機は、図4では簡略化して示したが、実際には次のようになっている。

```
// 関数の入口の処理（略）
qq0i=x;
qq1i=x;
qq0i=(qq0i*qq1i);
// ssPushVar の出力
if(qqt0!=2 || qqv0!=8131696) {
  qqMOVdx(0,5,16+0);
}
qqt0=2; qqv0=8131696;
// ssSubG の出力
qqsi=qq0i;
qqSUB_I(0,(int)qqsi);
qqt0=0;
// ssPopRet の出力
QQEXIT
// コード生成の後始末（略）
```

コード生成機の出力は、マシンコードをメモリに書き込むプログラム片に展開されるようなマクロを使っている。マクロでは簡単な覗穴最適化を行っている。例えば、整数の即値減算を行うマシンコードをメモリに書き込むマクロは、次のように定義されている。

```
#define qqSUB_I(rd,n) \
  if((n)!=0) { \
    if((n)==1) { qqDEC(rd); } \
    else if((n)==-1) { qqINC(rd); } \
    else { qqSUBi(rd,n); } \
  }
```

このマクロは、必要に応じてINC命令、DEC命令、またはSUB命令を生成するか、あるいは何も生成しない。すなわち、簡単な命令選択と覗き穴最適化が、コード生成時に行われる。参考までに、このマクロの内部で使われているマクロqqINCの定義を示す。

```
#define qqINC(rd) qqI1(0x40|rd)
#define qqI1(op) *qqpc++=(op)
```

最終的には、x86アーキテクチャの命令エンコーディングにしたがって、マシン命令がメモリに書き込まれる。

4 他のシステムとの比較

この節では、ターゲットマシンに依存しないように設計された他の実行時コード生成システムを紹介し、SSIFと比較する。

VCODE [4] は、RISCアーキテクチャ向きの実行時コード生成システムである。ロードストアアーキテクチャを仮定しているが、その範囲でターゲットマシンによらないインタフェースを定義している。その実現部は、C言語のマクロおよび、補助関数である。VCODEでは、ターゲットマシン依存の最適化として、簡単なディレイスロット処理などを行っている。

VCODEは、tcc [11] で、高速にコードを生成するために使われている。tccは、C言語を拡張して、実行時コード生成機を記述できるようにした言語'C [5] のコンパイラである。

VCODEのイタンフェースを使ったコード生成機の例（[4] より引用）を図5に示す。これが実行時に呼ばれると、次のようなコードを生成する。

```
# a0 に渡された引数に 1 を加える
addiu a0,a0,1
# 戻り番地にジャンプ
j     ra
# ディレイスロット：結果を戻り値レジスタへ
move  v0,a0
```

この例では、常に同じコードしか生成しないが、コード生成機に引数をつけることで実行時最適化が可能になる。

VCODEでは、SSIFで採用した、関数の入口のコードを生成するプログラム片を後から出力する方式を、採用していないため、生成されるコードに一部無駄がある。例えば、関数の入口では、呼び出し先で保存するべきレジスタをすべて保存できるように、スタックフレームを確保してしまうので、スタックフレームが不必要に大きくなることがある。なお、ローカル変数のために確保される領域の大きさは、バックパッチにより正しく設定される。

Tempo [2] [3] は、静的部分計算と、実行時最適化の両方に対応したシステムである。実行時最適化を行う場合には、あらかじめコンパイラを使ってテンプレートを作っておき、それをコード生成時につなぎあわせる。すなわち、SSIFやVCODEとは異なり、ターゲットマシンによらないインタフェースを定義しているわけではなく、ターゲットマシン依存性の大部分はコンパイラに任せ、残りはテンプレートをつなぎ合わせたり穴を埋めたりする部分が担当する。

例えば（[2] より引用）、次のような関数

```
int f(int x,int y) {
  int l;
  l=2*x;
  if(l==2) l=l+y;
  else l=y*x;
```

```
typedef int (*iptr)(int);
/* 実行時に呼ばれると、引数+1 を返す関数を生成する */
iptr mkplus1(struct v_code *ip) {
  v_reg arg[1];
  /* コード生成開始。型文字列"%i"は、このルーチンが整数引数一つをとる
  ことを示す。それを保持するレジスタがarg[0] にはいる。V_LEAF は、この関
  数が葉であることを示す。ip は生成したコードの格納域へのポインタである。*/
  v_lambda("%i",arg,V_LEAF,ip);

  /* 引数のレジスタに 1 を加える */
  v_addii(arg[0],arg[0],1);  /* 整数即値加算 */
  /* 結果を返す */
  v_reti(arg[0]);            /* 整数を返す */

  /* コード生成終了。v_end は後始末を行い、コードへのポインタを返す */
  return (iptr)v_end();
}
```

図5　VCODE を使ったコード生成機

```
  return 1;
}
```

に対する、x を実行時定数としたコード生成機は、コンパイラを使って用意したテンプレート（実際にはマシン命令列）

```
t1: 関数の入口
t2: l=[hole1]+y;
t3: l=y*[hole2];
t4: return l;
```

を使って、

```
void rt_spec_f(int x) {
  int l;

  dump_template(t1);
  l=2*x;
  if(l==2) {
    dump_template(t2);
    instantiate_hole(t2,l);
  } else {
    dump_template(t3);
    instantiate_hole(t3,x);
  }
  dump_template(t4);
}
```

のようになる。コード生成処理の大半はテンプレートのコピーで済むため、コード生成は比較的高速であるが、乗算の強さ軽減ができないなど、行える最適化に制限がある。静的なコンパイル結果と比べてかなり低速なコードとなることが報告されている。

この他、ターゲットマシン依存性を隠蔽する方法として、バイトコード特化という、Java のバイトコードを実行時に生成する方法がある [10]。Java の仮想マシンが Just In Time コンパイルを行う場合、その最適化機能によって静的コンパイルにかなり近い性能が得られる。

5　まとめ

本論文では、C++のための実行時最適化システムである C++ Doubler を紹介し、その中で用いられている実行時コード生成のインタフェースである SSIF について述べた。SSIF は、C++のマクロおよび関数で実装されており、ターゲットマシン依存性をその実現部に隠蔽している。他のシステムと比べた特徴は次の通りである。

- CISC プロセッサにも対応している。すなわち、可変長の命令や各種のエンコーディング形式に柔軟に対応できる。
- 固定したテンプレートを使わず、メモリに値を書き込むプログラムを出力することで、各種の最適化を可能にしている。
- 関数の入口のコードを出力するプログラム片を後から出力することで、スタックフレームを確保するコードの効率がよい。

著者は今後、現在は基本ブロック中だけに制限されている、ターゲットマシン依存の最適化処理を拡張す

るなどして、SSIF および C++ Doubler をさらに効率のよい実行時コード生成システムにしていく予定である。

参考文献

[1] Andersen, L. O.: *Program Analysis and Specialization for the C Programming Language*, PhD Thesis, DIKU, University of Copenhagen, May 1994.

[2] Consel, C. and cois Nöel, F.: A General Approach for Run-Time Specialization and its Application to C, Technical Report No. 946, INRIA/IRISA, July 1995.

[3] Consel, C., Hornof, L., Nöel, F., and Volanshi, N.: A Uniform Approach for Compile-time and Run-time Specialization, Technical Report No. 2775, INRIA, January 1996.

[4] Engler, D. R.: VCODE: A Retargatable, Extensible, Very Fast Dynamic Code Generation System, *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, May 1996, pp. 160–170.

[5] Engler, D. R., Hsieh, W. C., and Kaashoek, M. F.: 'C: A language For High-Level, Efficient, and Machine-independent Dynamic Code Generation, *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1996, pp. 258–270.

[6] Engler, D. R. and Proebsting, T. A.: DCG: An Efficient, Retargetable Dynamic Code Generation System, *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM Press, October 1994, pp. 263–272. Also appeared in SIGPLAN NOTICES, Vol.29, No.10.

[7] Fujinami, N.: Automatic Run-Time Code Generation in C++, *LNCS 1343: Scientific Computing in Object-Oriented Parallel Environments. First International Conference, ISCOPE 97 Proceedings*(Ishikawa, Y., Oldehoeft, R. R., Reynders, J. V., and Tholburn, M.(eds.)), December 1997, pp. 9–16. Also appeared as Technical Report SCSL-TR-97-006 of Sony Computer Science Laboratory Inc.

[8] Fujinami, N.: Determination of Dynamic Method Dispatches Using Run-time Code Generation, *LNCS 1472: Types in Compilation. Second International Workshop, TIC'98 Proceedings*(Leroy, X. and Ohori, A.(eds.)), March 1998, pp. 253–271. Also appeared as Technical Report SCSL-TR-98-007 of Sony Computer Science Laboratory Inc.

[9] Leone, M. and Lee, P.: Lightweight Run-Time Code Generation, *Proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, ACM Press, June 1994, pp. 97–106.

[10] Masuhara, H.: Generating Optimized Residual Code in Run-time Specialization, *Presented at Partial Evaluation and Program Transformation Day, Waseda University*, November 1999.

[11] Poletto, M., Engler, D. R., and Kaashoek, M. F.: tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation, *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, June 1997, pp. 109–121.

[12] 藤波順久: オブジェクト指向言語の実行時最適化, 日本ソフトウェア科学会第 *12* 回大会, September 1995. 高橋奨励賞受賞.

[13] 藤波順久: オブジェクト指向言語を利用した自動的かつ効率的な実行時コード生成, コンピュータソフトウェア, Vol. 15,No. 5(1998), pp. 25–37.

A SSIF のインタフェースの仕様

引数には、次のような表記を使う。

v: 値（整数または浮動小数点数）

r: 実行時定数を表す変数名（実際には番号）

e: ローカル、グローバル、またはインスタンス変数を指定するテーブルエントリ

tp: 型（型を表すオブジェクトへのポインタ）

- `r:tp` は、コード生成機の実行時定数を表す変数として、`r` 番の `tp` 型の変数を使うことを表す。この変数の値は、インタフェースを呼び出す側でセットする。変数の名前は、`qqrt` であり、*r* は変数の番号の十進表記、*t* は型によって決まる文字列（`i` は int、`Ui` は unsigned int、`l` は long、`Ul` は unsigned long、`f` は float、`d` は double、`r` は long double、`p` はポインタ）である。
- `e:tp` は、変数 `e` を `tp` 型としてアクセスすることを表す。通常は `tp` は `e` の型と等しい。構造体メンバアクセスの場合には、異なる型となり、コード生成機の変数 `qqd` に構造体の先頭からのオフセットが入る。
- `addr:tp` は、番地を `tp` 型のポインタとして解釈することを表す。
- `tmp:tp` は、値を `tp` 型の値として使うことを表す。

f: メンバアクセスであること、使用されるラベルであること、または、仮想関数呼び出しが決定されていることを表すフラグ

tf: 条件分岐の反転を示すフラグ

b,b0,b1: 基本ブロック番号（ジャンプ先）

snp: `switch` ノードへのポインタ

psysinfo: コード生成対象関数にマシン依存解析を行った結果へのポインタ

関数またはマクロ名	オペランド	説明
ssAdd	v	st+=v;
ssAddG	r,tp	st+=r:tp;
ssPopAdd		tmp=pop(); st+=tmp;
ssSub	v	st-=v;
ssSubG	r,tp	st-=r:tp;
ssSubR	v	st=v-st;
ssSubRG	r,tp	st=r:tp-st;
ssPopSub		tmp=pop(); st-=tmp;
ssAnd	v	st&=v;
ssAndG	r,tp	st&=r:tp;
ssPopAnd		tmp=pop(); st&=tmp;
ssOr	v	st|=v;
ssOrG	r,tp	st|=r:tp;
ssPopOr		tmp=pop(); st|=tmp;
ssXor	v	st^=v;
ssXorG	r,tp	st^=r:tp;
ssPopXor		tmp=pop(); st^=tmp;
ssMul	v	st*=v;
ssMulG	r,tp	st*=r:tp;
ssPopMul		tmp=pop st*=tmp;
ssDiv	v	st/=v;
ssDivG	r,tp	st/=r:tp;
ssDivR	v	st=v/st;
ssDivRG	r,tp	st=r:tp/st;
ssPopDiv		tmp=pop(); st/=tmp;
ssMod	v	st%=v;
ssModG	r,tp	st%=r:tp;
ssModR	v	st=v%st;
ssModRG	r,tp	st=r:tp%st;
ssPopMod		tmp=pop(); st%=tmp;
ssEq	tf v	st=st==v;
ssEqG	tf,r,tp	st=st==r:tp;
ssPopEq	tf	tmp=pop(); st=st==tmp;
ssNe	tf v	st=st!=v;
ssNeG	tf,r,tp	st=st!=r:tp;
ssPopNe	tf	tmp=pop(); st=st!=tmp;

表1　コード生成インタフェース (1/3)

関数またはマクロ名	オペランド	説明
ssLt	tf v	st=st<v; (signed)
ssLtG	tf,r,tp	st=st<r:tp; (signed)
ssPopLt	tf	tmp=pop(); st=st<tmp; (signed)
ssGt	tf,v	st=st>v; (signed)
ssGtG	tf,r,tp	st=st>r:tp; (signed)
ssPopGt	tf	tmp=pop(); st=st>tmp; (signed)
ssLe	tf,v	st=st<=v; (signed)
ssLeG	tf,v,tp	st=st<r:tp; (signed)
ssPopLe	tf	tmp=pop(); st=st<tmp; (signed)
ssGe	tf,v	st=st>=v; (signed)
ssGeG	tf,r,tp	st=st>=r:tp; (signed)
ssPopGe	tf	tmp=pop(); st=st<=tmp; (signed)
ssB	tf,v	st=st<v; (unsigned)
ssBG	tf,r,tp	st=st<r:tp; (unsigned)
ssPopB	tf	tmp=pop(); st=st<tmp; (unsigned)
ssA	tf,v	st=st>v; (unsigned)
ssAG	tf,r,tp	st=st>r:tp; (unsigned)
ssPopA	tf	tmp=pop(); st=st>tmp; (unsigned)
ssBe	tf,v	st=st<=v; (unsigned)
ssBeG	tf,r,tp	st=st<=r:tp; (unsigned)
ssPopBe	tf	tmp=pop(); st=st<=tmp; (unsigned)
ssAe	tf,v	st=st>=v; (unsigned)
ssAeG	tf,r,tp	st=st>=r:tp; (unsigned)
ssPopAe	tf	tmp=pop(); st=st>=tmp; (unsigned)
ssTrue	tf	push(true);
ssFalse	tf	push(false);
ssPromote		st=integral_promotion(st);
ssNeg		st=-st;
ssNot		st=~st;
ssCast	tp	st=(tp)st;
ssPshNum	v	push(v);
ssPshNumG	r,tp	push(r:tp);
ssPshVar	e,tp,f	push(e:tp);
ssPshAddr	e,f	push(&e);
ssPtr	tp,f	addr=st; st=*addr:tp;
ssFieldAddr		st=field_address(st);
ssPopAsn	e,tp,f	e:tp=pop();
ssPopPopAsnP	tp,f	addr=pop(); *addr:tp=pop();

表2　コード生成インタフェース (2/3)

関数またはマクロ名	オペランド	説明
ssJmp	b	b にジャンプ
ssJmpG	b0,b1,r,tp	もしr:tp ならb0 に、そうでなければb1 にジャンプ
ssJccG	b,r,tp	もしr:tp ならb にジャンプ
ssPopJcc	b	tmp=pop(); もしtmp ならb にジャンプ
ssLbl	b,f	基本ブロックb の開始
ssPopSwitch	snp	tmp=pop(); tmp で場合分け
ssParam	v	v は実引数
ssParamG	r,tp	r:tp は実引数
ssPopParam	tp	tmp=pop(); tmp を実引数とする
ssCallG	r	関数r を呼ぶ
ssPopCall		tmp=pop(); 関数tmp を呼ぶ
ssCallVirtual	i,f	i 番の仮想メンバ関数を呼ぶ
ssPshResult	tp	関数の返した値をスタックに積む
ssDrop		dummy=pop();
ssEnter	psysinfo	関数の入口の処理
ssRet		generate 関数から戻る
ssPopRet	tp	tmp=pop(); tmp:tp を戻り値として、関数から戻る

表3　コード生成インタフェース (3/3)

関数またはマクロ名	オペランド	説明
ssSimple	tp	tp の値がレジスタに入れられるかどうかを返す
ssInit		コード生成機を初期化する
ssExit		コード生成機を終了する
ssFlush		コード生成機の内部バッファをフラッシュする

表4　特殊インタフェース