2G-07

LLVMを活用したバイナリ変換のための ARM機械語からIRへの変換手法の検討

重信 晃太 [†] 大津 金光 [††] 大川 猛 [††] 横田 隆史 [††]
[†] 宇都宮大学工学部情報工学科 [††] 宇都宮大学大学院工学研究科情報システム科学専攻

1 はじめに

組み込み機器やモバイル端末で普及している ARM プロセッサにおいて，限られた計算リソースを最大限に利用して目標性能を達成することは非常に重要である．しかし，これらの機器にはマルチコアプロセッサや GPU，専用回路など多種多様な処理装置が備わっており，異なる処理環境を考慮して最適な処理プログラムコードを開発することは難しい．この問題を緩和する事例として，ソースコードに並列化や最適化に必要となる情報を付加することで容易に最適なプログラムを生成する手法 [1] も研究されている．しかしながら，この方法はソースコードが利用可能であることを前提としているため，プログラムの実行バイナリコードしか得られないユーザはプログラムを最適化する機会すらない．

そこで我々は，ARM プロセッサにおいてソースコードがない状況でも各機器の環境に最適化されたプログラムの実行を可能とするために，プログラムのバイナリコードを元にして最適化されたバイナリコードに変換するシステムを提案する．このシステムには，解析や最適化を行うモジュールが多数利用可能である LLVM[2] を利用する．しかし，そのためには ARM 機械語を LLVM の中間表現 LLVM IR に変換する必要がある．本稿では，この ARM 機械語から LLVM IR へのコード変換方法について検討する．

2 バイナリコード自動最適化システム

我々が提案する自動最適化システムでは，逐次実行モデルで書かれたプログラムから GPU やマルチコアプロセッサを用いた高性能な並列プログラムへの変換を主な目的とする．そのためにはプログラムの解析やそれに基いた最適化が必要である．本システムでは解析処理や並列化も含む最適化処理にオープンソースのコンパイラ基盤である LLVM を活用する．LLVM では言語やアーキテクチャから独立した独自の中間表現である LLVM IR を定義しており，LLVM IR で表現されたプログラムコードに対する解析処理，最適化処理，および各種命令セットに対するコード生成機能を提供している．また，LLVM IR に対する独自の処理を行うモジュールを開発するための各種クラスも提供されている．バイナリコード自動最適化システムの全体図を図 1 に示す．このシステムは，ARM バイナリコードを LLVM IR に変換する．変換された LLVM IR に対してプログラムの解析を行い，マルチコアプロセッサや GPU 等の実行環境に合わせて並列化された LLVM IR コードを生成する．並列化された LLVM IR コードから最適化されたバイナリコードが生成される．

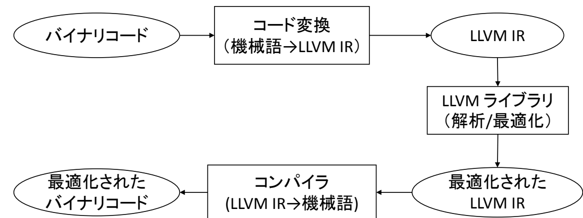

図 1: バイナリコード自動最適化システム

本システムの利点として，入力として必要なデータがバイナリコードのみであることが挙げられる．ソースコードを必要としないため，ユーザ透過的に実行バイナリコードから直接最適化された実行バイナリコードを得ることができる．また，最適化処理がプログラムの実行環境上で行われるため，元のプログラムの開発者がプログラムの最適化を考えなくてもよいという利点も得られる．

本システムでは LLVM の各種最適化モジュールを活用するため，ARM バイナリコードから LLVM IR へのコード変換が必要である．そこで，ARM 機械語から LLVM IR への変換手法について検討する．

3 ARM 機械語から LLVM IR への変換手法

ARM 機械語から LLVM IR への変換を考える前に，2 つの表現を比較する．

LLVM IR は，無限個の仮想レジスタを持つレジスタマシンをターゲットとした中間表現として定義されている．解析や最適化を行いやすくするためにすべてのレジスタ変数が SSA（Static Single Assignment）形式で表現される．SSA 形式では，各変数への代入は一度しか行われず，一度代入された変数は変更されることがない．LLVM IR は Module，Function，BasicBlock，Instruction とグローバル変数を表す Global Variable で構成される．Module は複数の Funciton と Global Variable，Function は複数の BasicBlock，BasicBlock は複数の Instruction を含む形でそれぞれ構成される．

図 2 に示す単純な加算処理を行うコードを LLVM IR に変換したものが図 3 のコードである．これは C 言語の LLVM フロントエンドである Clang を用いて

Consideration on Method of Translating ARM Binary Code into IR for LLVM-based Binary Translation
[†]Kohta Shigenobu, [††]Kanemitsu Ootsu, [††]Takeshi Ohkawa and [††]Takashi Yokota
Department of Information Science, Faculty of Engineering, Utsunomiya University (†)
Department of Information Systems Science, Graduate School of Engineering, Utsunomiya University (††)

Copyright ©2017 Information Processing Society of Japan.
All Rights Reserved.

```
1  int calc_add(){
2     int a = 1;
3     int b = 2;
4     int sum;
5     sum = a + b;
6     return sum;
7  }
```

図 2: 単純な加算処理を行う C 言語のソースコード

```
1  define i32 @calc_add() {
2   entry:
3     %a = alloca i32, align 4
4     %b = alloca i32, align 4
5     %sum = alloca i32, align 4
6     store i32 1, i32* %a, align 4
7     store i32 2, i32* %b, align 4
8     %1 = load i32* %a, align 4
9     %2 = load i32* %b, align 4
10    %3 = add nsw i32 %1, %2
11    store i32 %3, i32* %sum, align 4
12    %4 = load i32* %sum, align 4
13    ret i32 %4
14 }
```

図 3: 単純な加算処理を行う LLVM IR

生成したものである．LLVM IR では，変数は alloca 命令で領域を確保し，store 命令，load 命令を使って読み書きする．例えば，図 3 の変数 a，b は alloca 命令で int 型 32bit 分のメモリを確保し，store 命令でそれぞれ 1，2 を格納する．また，sum の値を求めるときには，load 命令で値を読み出して add 命令で使用する．ret 命令は関数の呼び出し元へ制御を移す命令で，戻り値を返す．

図 2 のソースコードから生成した ARM アセンブリコードを図 4 に示す．図中 2 行目から 4 行目，15 行目から 17 行目はフレームポインタやスタックポインタの退避および復帰や呼び出し元へ戻る命令であるため，LLVM IR に変換する際には定型的に変換することができる．5 行目以降では，str 命令でメモリに整数の 1，2 を格納し，加算で使用する前に ldr 命令で読み出している．加算した結果も同様にメモリに読み書きしている．ARM 機械語プログラムでは関数の戻り値に r0 レジスタを使う慣習となっているため，14 行目で加算結果を r0 に転送している．

図 4 のアセンブリコードを図 3 の LLVM IR と比較すると，変数をそれぞれ [fp, #-8] は%a，[fp, #-12] は %b，[fp, #-16] は%sum と対応付けることで，アセンブリ命令と LLVM IR 命令が 1 対 1 で対応付けられそうに見える．つまり，ARM 機械命令から LLVM IR への変換は単純な置き換えによって実現できると考えられる．変換の際に，ARM 機械語では変数のメモリ確保を行う命令は無いため，LLVM IR ではメモリ確保を行う alloca 命令を補う必要がある．加えて，LLVM IR は SSA 形式で記述しなければならないため，レジスタ変数の再利用をなくす必要がある．

以上より，ARM 機械語を 1 命令ごとに LLVM IR に変換することを基本方針とする．まずは ARM の機械語プログラムを読み込み，基本ブロックに分割する．以後は，基本ブロックを単位に変換処理を行う．

基本ブロックごとに 1 命令ずつ変換処理をしていく．ARM 機械語でメモリを初めて参照していた場合は対

```
1  000002f8 <calc_add>:
2   2f8: e52db004  push {fp}  ; (str fp, [sp, #−4]!)
3   2fc: e28db000  add fp, sp, #0
4   300: e24dd014  sub sp, sp, #20
5   304: e3a03001  mov r3, #1
6   308: e50b3008  str r3, [fp, #−8]
7   30c: e3a03002  mov r3, #2
8   310: e50b300c  str r3, [fp, #−12]
9   314: e51b2008  ldr r2, [fp, #−8]
10  318: e51b300c  ldr r3, [fp, #−12]
11  31c: e0823003  add r3, r2, r3
12  320: e50b3010  str r3, [fp, #−16]
13  324: e51b3010  ldr r3, [fp, #−16]
14  328: e1a00003  mov r0, r3
15  32c: e24bd000  sub sp, fp, #0
16  330: e49db004  pop {fp}  ; (ldr fp, [sp], #4)
17  334: e12fff1e  bx lr
```

図 4: 単純な加算処理を行う ARM アセンブリコード

応した LLVM IR を生成する前に alloca 命令で変数用の領域を確保する．また，LLVM IR で値を定義する命令ごとに新しいレジスタ変数を割り当てることで SSA 形式のコードを生成する．

単純な置き換えでは対処できない場合として，ARM 機械語の条件付き実行命令が挙げられる．ARM 機械語では各命令に実行条件を指定できる．LLVM IR では条件付き実行命令は使えないため，分岐命令を使って実現する必要がある．そこで，条件付き実行命令については複数の BasicBlock を組み合わせた LLVM IR コードを生成する．

初めに，条件成立時に実行する命令を格納するための BasicBlock A と，条件付き実行命令の次の命令を格納するための BasicBlock B を生成する．次に，条件成立時には A に，不成立時には B に分岐する LLVM IR 命令を生成する．最後に，A に条件成立時に実行する命令と B に分岐する命令を追加する．以降の命令は BasicBlock B に追加していく．これにより条件付き実行命令を変換できるが，LLVM IR コードを SSA 形式とするために，条件分岐により分岐した処理が合流する地点で phi 命令を挿入する必要がある．条件付き実行命令の書き込み対象レジスタの直前までの値と条件付き実行によって新たに定義された値を選択する phi 命令を BasicBlock B の先頭に挿入する．

4 おわりに

本稿では，LLVM ベースのバイナリコード自動最適化システムに必要な機能である ARM 機械語から LLVM IR への変換手法について述べた．今後は，複雑な分岐において適切な phi 命令を挿入することで SSA 形式に対応した変換手法を検討する必要がある．

謝辞

本研究は一部 JSPS 科研費 15K00068，16K00068 の助成による．

参考文献

[1] Alejandro Acosta，Sergio Afonso，Francisco Almeida: “Extending Paralldroid with object oriented annotations,” Parallel Computing, Vol.57, pp.25-36, 2016.

[2] “The LLVM Compiler Infrastructure Project,” http://llvm.org/

Copyright ©2017 Information Processing Society of Japan.
All Rights Reserved.