Goldfishをいじってみよう

2009.11.30

京都マイクロコンピュータ
小林 哲之

はじめに

- Androidのソースが公開されてから一年
- 先日Eclair(Android2.0)のソースがマスターにマージされたので急遽内容を変更しました。
- ここでの話は発表者の主観に基づくもの。無保証です。

Who am I?

- 組み込み一筋N十年。
  - リアルタイムOS iTRON
  - 組み込み向けJava実行環境
  - 組み込み向けLinux
  - gcc
- ブログ 「組み込みの人。」
  - http://d.hatena.ne.jp/embedded/
- 京都マイクロコンピュータ 2008年3月入社
  - http://www.kmckk.co.jp/

本日お話すること

- Goldfishとは
- Eclair(エクレア)のソースがマージされた
  - VFP対応、armv7対応
  - Webkit V8 JavaScriptエンジン
  - DalvikVMに実験版のJIT

Goldfishとは

- エミュレータ(コマンド名 emulator) で使われている仮想ハードウェア
- CPUは元々arm926だが「換装」可能。

ソースからビルドして動かす方法

- envsetup.shを使うと環境設定が簡単。

```
$ cd mydroid
$ source build/envsetup.sh
$ lunch generic-eng
$ time make -j4 2>&1 |tee make.log
$ emulator &
```

Eclair(エクレア)

- 2009.11.15
- デフォルトは変更されていないが、以下のものがソースに入ってきている
  - VFP対応、armv7対応
  - Webkit V8 JavaScriptエンジン
  - DalvikVMに実験版のJIT

ARMのアーキテクチャ対応の拡大

- DalvikVMのアセンブラ版のインタープリタの種類が増えた
  - armv4t
  - armv5te (default)
  - armv5te-vfp
  - armv7-a
- ビルド時の環境変数TARGET_ARCH_VARIANT

VFP対応版のビルド

```
$ export TARGET_ARCH_VARIANT=armv5te-vfp
$ time make -j4 2>&1 |tee make.log

$ emulator &
```

- インタープリタはVFP対応版が使われる。
- それ以外の部分ではVFPは使われない。(-msoft-flaot付きでビルドされるため)
- Kernel, qemuはそのままでOK

armv7a対応版のビルド

```
$ export TARGET_ARCH_VARIANT=armv7-a
$ time make -j4 2>&1 |tee make.log

$ emulator -kernel prebuilt/android-arm/
kernel/kernel-qemu-armv7 &
```

- Kernelはarmv7用にビルドされたものを使用する。
- gcc4.4.0ではコンパイルエラーになる場所があった。armv7対応はまだ枯れていないかも。もっと新しいコンパイラで後で再チャレンジ。

Webkit V8 Javascriptエンジン

- http://code.google.com/intl/ja/apis/v8/
- C++で記述されている
- X86, x86-64, ARMに対応
- ARMの場合32bitコードを生成する。(Thumbではない)

V8を有効にしてビルド

```
$ export JS_ENGINE=v8
$ time make -j4 2>&1 |tee make.log

$ emulator &
```

V8が生成したコードを見てみる

- 無理やりデバッグ用のコードを有効にする
  - webkit/v8Binding/Android.libv8.mk
    - LOCAL_CFLAGS += -DENABLE_DISASSEMBLER
  - webkit/v8Binding/src/flag_definitions.h
    - print_code を true にセット
    - print_builtin_code を trueにセット

V8が生成したコード

```
I/v8      (  201): Builtin: Illegal
I/v8      (  201): kind = BUILTIN
I/v8      (  201): name = Illegal
I/v8      (  201): Instructions (size = 40)
I/v8      (  201): 0x45a50920     0  e59fc014       ldr ip, [pc, #+20]
I/v8      (  201): 0x45a50924     4  e58c1000       str r1, [ip, #+0]
I/v8      (  201): 0x45a50928     8  e2800001       add r0, r0, #1
I/v8      (  201): 0x45a5092c    12  e59f100c       ldr r1, [pc, #+12]
I/v8      (  201): 0x45a50930    16  e59fc00c       ldr ip, [pc, #+12]
          ;; code: STUB, CEntry, minor: 0
I/v8      (  201): 0x45a50934    20  e12fff1c       bx ip
I/v8      (  201): 0x45a50938    24  03000003       constant pool begin
I/v8      (  201): 0x45a5093c    28  aa438d10       constant
I/v8      (  201): 0x45a50940    32  aa2d94e1       constant
I/v8      (  201): 0x45a50944    36  45a50120       constant
I/v8      (  201):
```

32bitのARMのコードが生成されている。
詳しいことはわかりません。。

$ adb logcat

Dalvik VmのJIT

- マスターのソースに入ったのは実験途中のスナップショットらしい。いろんなやり方を評価中。
- Cで記述されている。
- ARMのみ。
  - Armv5te (Thumbのコードを生成)
  - Armv5te-vfp (Thumb + VFP呼び出し)
  - armv7-a　(Thumb2のコードを生成)
- V8と共通のコードは無い。

JITを有効にしてビルド

```
$ export WITH_JIT=true
$ time make -j4 2>&1 |tee make.log

$ emulator &
```

JITが生成したコードを見てみる

- 無理やりデバッグ用のコードを有効にしてみる
- vm/Init.c setCommandLineDefaults()
  - gDvmJit.printMe = true;
  -

JITが生成したコード

```
D/dalvikvm(   97): -------- dalvik offset: 0x0062 @ aget-byte
D/dalvikvm(   97): 0x44a83544 (0018): ldr     r2, [r5, #124]
D/dalvikvm(   97): 0x44a83546 (001a): str     r3, [r5, #80]
D/dalvikvm(   97): 0x44a83548 (001c): mov     r3, #128
D/dalvikvm(   97): 0x44a8354a (001e): ldr     r3, [r5, r3]
D/dalvikvm(   97): 0x44a8354c (0020): cmp     r2, #0
D/dalvikvm(   97): 0x44a8354e (0022): beq     0x44a835b6
D/dalvikvm(   97): 0x44a83550 (0024): ldr     r0, [r2, #8]
D/dalvikvm(   97): 0x44a83552 (0026): add     r2, r2, #16
D/dalvikvm(   97): 0x44a83554 (0028): cmp     r3, r0
D/dalvikvm(   97): 0x44a83556 (002a): bcs     0x44a835b6
D/dalvikvm(   97): 0x44a83558 (002c): ldrsb   r0, [r2, r3]
```

$ adb logcat

JITが生成したコード

バイト配列のロードのDEXコード

```
D/dalvikvm(    97): -------- dalvik offset: 0x0062 @ aget-byte
D/dalvikvm(    97): 0x44a83544 (0018): ldr     r2, [r5, #124]
D/dalvikvm(    97): 0x44a83546 (001a): str     r3, [r5, #80]
D/dalvikvm(    97): 0x44a83548 (001c): mov     r3, #128
D/dalvikvm(    97): 0x44a8354a (001e): ldr     r3, [r5, r3]
D/dalvikvm(    97): 0x44a8354c (0020): cmp     r2, #0      nullチェック
D/dalvikvm(    97): 0x44a8354e (0022): beq     0x44a835b6
D/dalvikvm(    97): 0x44a83550 (0024): ldr     r0, [r2, #8]
D/dalvikvm(    97): 0x44a83552 (0026): add     r2, r2, #16
D/dalvikvm(    97): 0x44a83554 (0028): cmp     r3, r0 配列の範囲チェック
D/dalvikvm(    97): 0x44a83556 (002a): bcs     0x44a835b6
D/dalvikvm(    97): 0x44a83558 (002c): ldrsb   r0, [r2, r3]
```

バイトデータのロード

ほぼDEXコードに対応したコード。最適化はされていない

このJITの特徴

- インタープリタで実行頻度の高い部分をカウントし、一定回数以上実行されるとコンパイル要求がキューに入る。
- JITコンパイルは別スレッドで行われる
- コンパイルの単位はメソッド単位ではなく、もっと小さいブロック単位

体感速度はあまり変わらないが...

- マイクロベンチマークでは3倍程度の性能向上があるようだ
- 体感速度はVMの実行速度よりも描画速度の比重が大きい
-

JITコンパイラのトレードオフ

- コンパイラ自身のサイズ
- コンパイル時間
- コンパイルしたコードのサイズ
- 最適化の度合い
- ...

- よりよいバランスを目指して調整が必要
- まだまだ実験中なので期待して待とう

今後が楽しみ

- Thumb2EE
  - JITコンパイルに適したarmv7の命令セット。
    - ゼロコストnullポインタチェック
    - 配列の範囲チェック用命令
    - ハンドラ呼び出し命令
  - 生成コードのサイズが小さくなり効率化
- マルチプロセッサ(SMP)
  - バックグランドでJITコンパイル

最後に

- オープンソースの幸せ
- ぜひ自分でソースをいじってビルドして動かしてみてください
- ブログにコメント、トラックバック歓迎
-
-
- ご清聴ありがとうございました。