Portability and Optimizations of GNU Applications for ARM Embedded Linux

Vassilis Laganakos

System Design Division - ARM Ltd.

November 7, 2008

We are going to point out some common issues that someone can come across, when porting or developing an application for ARM Embedded Linux, having in mind to be cross-platform and cross-compiler compatible.

Furthermore, some features of ARM's latest architecture (v7) are going to be discussed, and what/how these features are supported by tools.

Most of the current Linux applications used for embedded Linux, were designed for desktop use. This means that when created, the special requirements of Embedded Systems world were not taken into account. Things such as: not so much available memory, constrained space, and need for low power.

This defines the **Holy Grail** for most of the cases:

"Get as much better performance, with the smallest memory footprint"

Of course memory footprint and performance are not entirely orthogonal. E.g. you might need to unroll that loop *x* more times to get *y* performance improvement, but this will lead to *z* size increase.

There might be also case-specific restrictions. E.g. We have to fit the whole lot in *x* space..

You can get the best for a case like above, by using ARM architecture v7 with it's Thumb2. This is a whole ISA, not just a mode of operation.
Is as small as Thumb and performs 95% of ARM mode

So you want to be portable...

A few things should be kept in mind in order your project to be cross-compiler/platform compatible.

Do s

- Program according to standards
- Use intrinsics

Don't s

- Use inline, embedded assembly
- Use Toolchain specific things.

So you want to be portable...

A few things should be kept in mind in order your project to be cross-compiler/platform compatible.

Do s

- Program according to standards
- Use intrinsics

Don't s

- Use inline, embedded assembly
- Use Toolchain specific things.

So you want to be portable...

A few things should be kept in mind in order your project to be cross-compiler/platform compatible.

Do s

- Program according to standards
- Use intrinsics

Don't s

- Use inline, embedded assembly
- Use Toolchain specific things.

So you want to be portable...

A few things should be kept in mind in order your project to be cross-compiler/platform compatible.

Do s

- Program according to standards
- Use intrinsics

Don't s

- Use inline, embedded assembly
- Use Toolchain specific things.

Creating a cross-compiler compatible application/library enables better portability across the hardware platforms.

Whether we are developing a project or porting an existing to another architecture there are three ways to work (well actually two):

- Cross-compilation
- Native compilation

Whether we are developing a project or porting an existing to another architecture there are three ways to work (well actually two):

- Cross-compilation
- Native compilation

Cross Compiling

Benefits

- Big variety of compilers to choose from
- It is fast
- Widespread

Drawbacks

- Have to convert existing projects
- Have to be really careful what is pulled in
- Procedure required to test the build

Cross Compiling

Benefits

- Big variety of compilers to choose from
- It is fast
- Widespread

Drawbacks

- Have to convert existing projects
- Have to be really careful what is pulled in
- Procedure required to test the build

Cross Compiling

Benefits

- Big variety of compilers to choose from
- It is fast
- Widespread

Drawbacks

- Have to convert existing projects
- Have to be really careful what is pulled in
- Procedure required to test the build

Cross Compiling

Benefits

- Big variety of compilers to choose from
- It is fast
- Widespread

Drawbacks

- Have to convert existing projects
- Have to be really careful what is pulled in
- Procedure required to test the build

Cross Compiling

Benefits

- Big variety of compilers to choose from
- It is fast
- Widespread

Drawbacks

- Have to convert existing projects
- Have to be really careful what is pulled in
- Procedure required to test the build

Cross Compiling

Benefits

- Big variety of compilers to choose from
- It is fast
- Widespread

Drawbacks

- Have to convert existing projects
- Have to be really careful what is pulled in
- Procedure required to test the build

Native

Benefits

- Don't have to worry about header, library paths
- Can test immediately

Drawbacks

- It isn't fast enough
- Most of the apps for ARM Linux are configured for Cross-Compilation
- Not many choices in compilers
- Native compiler faces different problems

Native

Benefits

- Don't have to worry about header, library paths
- Can test immediately

Drawbacks

- It isn't fast enough
- Most of the apps for ARM Linux are configured for Cross-Compilation
- Not many choices in compilers
- Native compiler faces different problems

Native

Benefits

- Don't have to worry about header, library paths
- Can test immediately

Drawbacks

- It isn't fast enough
- Most of the apps for ARM Linux are configured for Cross-Compilation
- Not many choices in compilers
- Native compiler faces different problems

Native

Benefits

- Don't have to worry about header, library paths
- Can test immediately

Drawbacks

- It isn't fast enough
- Most of the apps for ARM Linux are configured for Cross-Compilation
- Not many choices in compilers
- Native compiler faces different problems

Native

Benefits

- Don't have to worry about header, library paths
- Can test immediately

Drawbacks

- It isn't fast enough
- Most of the apps for ARM Linux are configured for Cross-Compilation
- Not many choices in compilers
- Native compiler faces different problems

Native

Benefits

- Don’t have to worry about header, library paths
- Can test immediately

Drawbacks

- It isn’t fast enough
- Most of the apps for ARM Linux are configured for Cross-Compilation
- Not many choices in compilers
- Native compiler faces different problems

Native-Like

There is another approach, whereby we cross-compile but we fool the compiler/autotools that it is compiling natively. This is what Scratchbox is all about.

Benefits

- Can compile existing GNU Applications without any modifications
- Lots of compilers to choose from - RVCT already supports Scratchbox
- It is fast to compile
- Can test immediately
- Execution on real hardware alleviates any emulation problems.

Drawbacks

- It is not native :)
- many layers of abstraction insert case-specific problems

Native-Like

There is another approach, whereby we cross-compile but we fool the compiler/autotools that it is compiling natively. This is what Scratchbox is all about.

Benefits

- Can compile existing GNU Applications without any modifications
- Lots of compilers to choose from - RVCT already supports Scratchbox
- It is fast to compile
- Can test immediately
- Execution on real hardware alleviates any emulation problems.

Drawbacks

- It is not native :)
- many layers of abstraction insert case-specific problems

Native-Like

There is another approach, whereby we cross-compile but we fool the compiler/autotools that it is compiling natively. This is what Scratchbox is all about.

Benefits

- Can compile existing GNU Applications without any modifications
- Lots of compilers to choose from - RVCT already supports Scratchbox
- It is fast to compile
- Can test immediately
- Execution on real hardware alleviates any emulation problems.

Drawbacks

- It is not native :)
- many layers of abstraction insert case-specific problems

Native-Like

There is another approach, whereby we cross-compile but we fool the compiler/autotools that it is compiling natively. This is what Scratchbox is all about.

Benefits

- Can compile existing GNU Applications without any modifications
- Lots of compilers to choose from - RVCT already supports Scratchbox
- It is fast to compile
- Can test immediately
- Execution on real hardware alleviates any emulation problems.

Drawbacks

- It is not native :)
- many layers of abstraction insert case-specific problems

Native-Like

There is another approach, whereby we cross-compile but we fool the compiler/autotools that it is compiling natively. This is what Scratchbox is all about.

Benefits

- Can compile existing GNU Applications without any modifications
- Lots of compilers to choose from - RVCT already supports Scratchbox
- It is fast to compile
- Can test immediately
- Execution on real hardware alleviates any emulation problems.

Drawbacks

- It is not native :)
- many layers of abstraction insert case-specific problems

Native-Like

There is another approach, whereby we cross-compile but we fool the compiler/autotools that it is compiling natively. This is what Scratchbox is all about.

Benefits

- Can compile existing GNU Applications without any modifications
- Lots of compilers to choose from - RVCT already supports Scratchbox
- It is fast to compile
- Can test immediately
- Execution on real hardware alleviates any emulation problems.

Drawbacks

- It is not native :)
- many layers of abstraction insert case-specific problems

Native-Like

There is another approach, whereby we cross-compile but we fool the compiler/autotools that it is compiling natively. This is what Scratchbox is all about.

Benefits

- Can compile existing GNU Applications without any modifications
- Lots of compilers to choose from - RVCT already supports Scratchbox
- It is fast to compile
- Can test immediately
- Execution on real hardware alleviates any emulation problems.

Drawbacks

- It is not native :)
- many layers of abstraction insert case-specific problems

Scratchbox is a cross-compilation (OpenSource, under GPL) toolkit designed to make embedded Linux application development easier.

From www.scratchbox.org

Scratchbox creates a virtual native ARM Linux environment. Which means:

- A standard Linux/Unix filesystem is in place with C-library compiled for ARM in default niches.
- use of a ”native” compiler

Comes with tools such as:

- Qemu - to execute the ARM binaries
- SBRSH - to execute ARM binaries on real hardware
- Other DevKits (e.g. Matrix)

Short Demo...

A simple example GCC - RVCT equivalent flags

Example

GCC Command line

```
gcc -mcpu=Cortex-A8 -mfloat-abi=softfp -O3 ...
```

This will compile code for the Cortex-A8 using software FP library to perform the calculations.

Example

RVCT Command line

```
armcc -cpu=Cortex-A8 -vfp=softvfp+vfpv3 -O3 \
      -Otime ...
```

Note that we'll have to use the legacy mode of `softvfp+vfpv3`. This mode enables the generation FP code used by the FPU on the ARM core, and the procedure call (APCS) would use soft.

From RVCT 4.0 onwards, you *can* use the automatic translation of GCC command line flags. Therefore you won't need to manually generate another set of Makefiles to compile things with RVCT.

how?

```
armcc --arm_linux_configure --arm_linux_config_file=<path/to/xmlfile> \
      --configure_gcc=<path/to/csgcc>

armcc --translate_gcc --arm_linux_config_file=<path/to/file>
```

Diagnose

You could also diagnose if everything was translated correctly as well as other diagnostics if you add `--remarks` flag on your command line:

```
armcc --translate_gcc --remarks --arm_linux_config_file=<path/to/file>
```

Compiling applications such as Mesa3D, DirectFB, XULRunner, showed that they were using GCC-specific extensions, that would actually prevent compilation with another toolchain.

Latest ARM Compiler supports GCC up to 4.2 and implements some of the long-standing GCC extensions, such as:

- relaxed ref/def model
- label variables, a.k.a computed/assigned goto ← *observed in XULRunner*
- `__attribute__((constructor))` ← *observed in XULRunner, DirectFB*

Latest ARM Compiler supports GCC up to 4.2 and implements some of the long-standing GCC extensions, such as:

- relaxed ref/def model
- label variables, a.k.a computed/assigned goto ← *observed in XULRunner*
- `__attribute__((constructor))` ← *observed in XULRunner, DirectFB*

Latest ARM Compiler supports GCC up to 4.2 and implements some of the long-standing GCC extensions, such as:

- relaxed ref/def model
- label variables, a.k.a computed/assigned goto ← *observed in XULRunner*
- `__attribute__((constructor))` ← *observed in XULRunner, DirectFB*

In general porting the above applications to ARM's Compiler, was easy since from v4.0 it operates as a drop-in replacement for CodeSourcery GCC. This resulted in smaller memory footprint and better performance, especially when it came to ARM technology parts such as hardware VFP.

What is NEON

ARM NEON^{TM} technology is a 128 bit SIMD (Single Instruction, Multiple Data) architecture extension for the ARM Cortex^{TM}-A series processors, designed to provide flexible and powerful acceleration for intensive multimedia applications, thereby delivering a significantly enhanced user experience.

Some of the benefits...

- Aligned and unaligned data access allows for efficient vectorization of SIMD operations.
- Support for both integer and floating point operations
- Tight coupling to the ARM core provides a single instruction stream and a unified view of memory, presenting a single development platform target with a simpler tool flow.
- The large NEON register file with its multiple views enables efficient handling of data and minimizes access to memory, enhancing data throughput performance.

NEON Instrinsics Example

This simple example shows vector division of two single-prevision floating point values in parallel using NEON reciprocal / reciprocal-estimate

```
#include <arm_neon.h>
#include <stdio.h>

void vect_divtest2(void)
{
  float32x2_t vrc, vc, vt, va, vb;
  float x;
  vb = vset_lane_f32(1.0f, vb, 0);
  vc = vset_lane_f32(2.0f, vc, 0 );

  vrc = vrecpe_f32(vc);
  vt = vrecps_f32(vrc, vc);
  vrc = vmul_f32(vrc, vt);
  vt = vrecps_f32(vrc, vc);
  vrc = vmul_f32(vrc, vt);

  va = vmul_f32(vb, vrc);

  x = vget_lane_f32(va, 0);

  printf("%f\n", x);
}
```

```
int main(void)
{
  vect_divtest2();
}
```

Hence NEON:

Significantly improves out data processing capability (32 registers each 64-bit - split into elements - great for data reuse) with compiler technology to support it.

In addition RVCT:

Offers automatic vectorization of your code.

Example

```
armcc -cpu=Cortex-A8 -O3 -Otime --vectorize myfile.c
```

ARM works hard on setting standards

- ARM Architecture standards. We don't allow people to modify the architecture (eg. add/remove instructions) and that's a good thing for application space processors where OS & binary compatibility is essential.
- Defines the ARM EABI :)

ARM regarding Compatibility

- Works on supporting GCC command line flags
- RVCT Supporting ANSI C90 and C99

ARM regarding Compiler Usability

- Supports Scratchbox
- Pulled together a set of libs/apps that can be used as a starting point for developers launching a ARM-based Linux platform. Please grab your copy from: www.linux.onarm.com

Key Points

- Being cross-compiler compatible is a *good thing*.
  Gives your project portability to hardware platforms
- To be faithful to the above and avoid too much trouble when you want to move your application to another platform/architecture, write code according to the standards and avoid using too much toolchain- specific extensions.
- Writing vectorizer-friendly C code is also *good thing*.
  To be able to take advantage of the new technology
- ARM is focusing on setting and using standards.
  Which is aids the developer
- ARM Compiler offers greats support for the existing and new ARM technology (FPU, NEON), and early access for the things yet to come :)

Thank you very much for your time!
Any questions?...