GCC 4.4.4 multilib toolchain release note

Contents

1 What new

New features of this toolchain include:

- gcc 4.4.4.
  This compiler version supports VFPv3 and NEON.
  And it is also the first gcc version supports Cortex-A9, Cortex-R4.
- Multilib cross toolchain.
  This toolchain is a multilib toolchain, library compiled separately for different CPU model.
- Optimized library.
  We optimized some common library routines to improve application performance.
- Application debug tools.
  We provide some debug tools to trace and detect application bug.

2 What inside

The whole toolchain contains:

- binutils 2.20.1
- gcc 4.4.4 with multilib support
- glibc 2.11.1.
- glibc-ports 2.11 (some routines are optimized with neon and arm instructions)
- gdb and gdbserver 7.1
- other debug tools and some companion libraries

Toolchain directory structure.

```
|-- bin      //toolchain with prefix, such as arm-none-linux-gnueabi-gcc etc.
|-- lib      //library files used for toolchain itself, not for application
`-- arm-fsl-linux-gnueabi
      |-- bin                //toolchain without prefix, such as gcc.
      |-- debug-root         //all debug tools
      `-- multi-libs         //all libraries and headers.
            |-- armv5te      //library for armv5te (i.mx 2xx). only support soft float point
            |-- armv6        // library for armv6 (i.mx 3xx), soft fpu version
            |     `-- vfp    //library for armv6, vfp fpu version
            |-- armv7-a      //library for armv7-a (i.mx5xx), hardware fpu version
            |     |-- neon   //library for armv7-a, use neon as fpu
            |     |-- thumb  //library for armv7-a, use thumb-2 instruction instead of arm.
            |     `-- vfpv3  //library for armv7-a, use vfpv3 as fpu.
            |-- lib      //default library. It can be used for armv4t and above.
            `-- usr
                  |-- include      //header files for the application development
                  `-- lib          //three-part library and static built library
```

3 Howto use

3.1 gcc

As a multilib toolchain, different gcc options link to different binary library, for example, Compile test.c with:

a) arm-none-linux-gnueabi-gcc -Wall test.c -o test
b) arm-none-linux-gnueabi-gcc -Wall -mcpu=cortex-a8 -mfpu=neon -mfloat-abi=softfp test.c -o test

case a) will link to default binary lib directory, which located at: multi-libs/lib
case b) will link to armv7-a neon binary lib directory, which locate at: multi-libs/armv7-a/neon/lib

You can check multilib toolchain information by:

a) arm-none-linux-gnueabi-gcc -print-multi-lib
b) arm-none-linux-gnueabi-gcc “your gcc options” -print-multi-directory
c) arm-none-linux-gnueabi-gcc “your gcc options” -print-search-dirs

command a) shows all the multilib support.
command b) shows the library directory corresponding to the gcc options you provided. But it cannot handle options alias. For example, We have set -mcpu=cortex-a8 has same effect to -march=armv7-a, but the result this command shows is not we expected. Although the information it shows not correct, gcc will link to correct library when you use alias options, you can confirm the behavior by command c).
command c) shows the library search patch corresponding to the gcc options you provided.

Incompatible notes:
gcc 4.4 rename armv7a architect name to armv7-a, rename vfp3 fpu name to vfpv3. So if you use -march=armv7a or -mfpu=vfp3 will not work as expect.

Gcc options for different library directory.

Directory path	Targe CPU model	Gcc options	Alias options
lib, usr/lib	armv4t		
armv5te/lib armv5te/usr/lib	armv5te, i.mx2xx	-march=armv5te	-march=armv5te alias to: -mcpu=”cpu name of armv5te” or -mtune=”cpu name of armv5te” such as: arm968e-s, arm946e-s. etc.
armv6/lib armv6/usr/lib	armv6, i.mx3xx	-march=armv6	-march=armv6 alias to: -mcpu=”cpu name of armv6 arch” or -mtune=”cpu name of armv6 arch” such as: arm1136j-s etc.
armv6/vfp/lib armv6/vfp/usr/lib	armv6, i.mx3xx	-march=armv6 -mfpu=vfp –mfloat-abi=softfp	save to above.
armv7-a/lib armv7-a/usr/lib	armv7-a, i.mx5xx	-march=armv7-a	-march=armv7-a alias to: -mcpu=”cpu name of armv7-a” or -mtune=”cpu name of armv7-a” such as: cortex-a8 cortex-a9 etc.
armv7-a/thumb/lib armv7-a/thumb/usr/lib	armv7-a, i.mx5xx	-march=armv7-a -mthumb	same as above
armv7-a/neon/lib armv7-a/neon/usr/lib	armv7-a, i.mx5xx	-march=armv7-a -mfpu=neon -mfloat-abi=softfp	same as above
armv7-a/vfpv3/lib armv7-a/vfpv3/usr/lib	armv7-a, i.mx5xx	-march=armv7-a -mfpu=vfpv3 -mfloat-abi=softfp	same as above

Notes:

“directory path” is relative path to:

“gcc-4.4.4-glibc-2.11.1-multilib-1.0/arm-fsl-linux-gnueabi/ arm-fsl-linux-gnueabi/multi-libs”

“target CPU model” means cpu model and above, not just only the specific model.

“gcc options” means the options must provided, other options can add base on this.

“alias options” means:

some “gcc options” alias to other, so if you use these alias options, has the same effect to “gcc options”

3.2 Application debug tools

strace and strace-graph:

strace used to trace system call invoked by application and library.

It shows system call parameters and results, and can also provide some performance statistic among system calls.

ltrace:

ltrace similar to strace, except ltrace trace library call.

duma:

duma is a memory access debug tools. It helps to solve memory access error(segmentation fault) and memory leak problem.

dmalloc

dmalloc can be used to debug application memory leak. It needs library(glibc) support.

You can check this reference for more:

http://dmalloc.com/docs/latest/online/dmalloc_toc.html

4 Howto build a multilib toolchain

4.1 Introduce

There are different kinds of toolchains. You can distinguish them by understand the machines involved:

1) machine used to build the toolchain: build machine
2) machine used to running the toolchain: host machine
3) machine the toolchain generating code running on: target machine

When build = host = target. native toolchain.
When build != host = target. native toolchain.
When build = host != target. cross toolchain
When build !=host != target. Canadian toolchain

In this chapter, we only describe cross toolchain be used on Linux, and suppose building machine is x86 server.

Classic cross toolchain can be build out in following steps:

1. install Linux kernel headers. The kernel header is required by binutils and glibc. Linux kernel provide “make header_install” to install headers.
2. build gcc companion library, which required by gcc to running, such as gmp, ppl etc. This step is built by x86 gcc.
3. Build cross binutils. Binutils provide assembler, linker, and some elf binary operation tools, which is the base of compiler and toolchain building. It is built by x86 gcc.
4. Build bootstrap cross gcc. This gcc only enabled minimal functions, can generate target code for ARM, but only support compiling c source code. We can only generate bootstrap gcc now, since some components need by full function gcc not ready yet, and these components require cross gcc too, this chicken egg problem can only solved by a bootstrap gcc. It is built by x86 gcc.
5. Build glibc basic files. The basic file include libc header and start files. These files required by gcc to generate cross binary. It is built by bootstrap gcc.
6. Build gcc, this gcc support shared binary library. We can’t build full function gcc without glibc, since gcc has some high level library and binaries, such as libstdc++, require libc.
7. Build libc by gcc in step 6. This step build and install glibc.
8. Build full functioned gcc. This step build gcc, g++, libstdc++ and all gcc features you specified. Built by x86 gcc. Until these step, cross toolchain built out.
9. Build target running application. Using cross toolchain to build rootfs, the rootfs contains all the application, library of the target machine. Such as kernel image, bootloader, debug tools, shell, or even native toolchain.

Multilib toolchain is more complicated than classic cross toolchain. We need to modify step 5, 6, 7, 8.
At first, you need to list all gcc options you want to use,
Then:
In step 5, build glibc start file with every gcc option, and install them in separate directory.
In step 6, modify gcc source code, specific the options combination and alias source code, and write gcc spec file.
In step 7, build glibc with every gcc option, and install in separate directory.
In step 8, the building process will build c++ library with every gcc option and install to separate directory.

4.2 Toolchain building tools

You can build a cross toolchain according to previous chapter describe, or choose to use a well-know toolchain build tools to generate.

In this section, we describe how to build toolchain by crosstool-ng.
Crosstool-ng is a menu config toolchain building tools. It mainly used to generate glibc base toolchain.
Crosstool-ng can't build multilib or native toolchain yet. We provide a simple patch to enable this feature.
Steps to build multilib cross toolchain.

1. You can download crosstool-ng from: http://ymorin.is-a-geek.org/hg/crosstool-ng/rev/3fb8b4acdc4a, or get it from our release package. and unpack it.
2. get toolchain_gcc4.4.4_community_patches_v1.0.tar.bz2, toolchain_gcc4.4.4_Freescale_patches_v1.0.tar.bz2, and toolchain_gcc4.4.4_multilib_build_script_v1.0.tar.bz2, which released by Freescale, and unpack it. Please note to unpack community patches and Freescale patches to same directory.
3. Apply "patch/crosstool-ng/000_crosstool-ng_multilib_support.patch" to crosstool-ng.
4. copy "crosstool-ng_config_gcc4.4.4" to crosstool-ng directory, and rename it to ".config"
5. goto crosstool-ng directory, configure and build it with: ./configure --local, and "make"
6. "./ct-ng menuconfig" to add you custom config to the toolchain. Be careful about following options:
   - Automatically download option. if choose not use source package we provided, enable this options.
   - enable multilib support and set multilib script path.
   - set local patch directory. Set to the directory you unpack to in 2
   - change toolchain prefix to yours.
7. Write your own multilib build script, or modified base on our release. This script will invoked by crosstool-ng after every step. And modified gcc according to "patch/gcc/4.4.4/100_fsl_multilib_support.patch". More discuss about this topic in next section.
8. "./ct-ng build" to build the toolchain. In this step, will download source tarballs automatically if you enable it, and apply patches to it.

You can get release package with this release note provided by FreeScale.

4.2.1 Howto write multilib building script

This script invoked by crosstool-ng after every step complete. And TOP_DIR environment variable saves directory of crosstool-ng.
So this script can be write as:

```
#!/bin/sh
CT_TOP_DIR=${TOP_DIR}
. ${TOP_DIR}/.config
. /home/test/toolchains/test/multilib_scripts/gcc_shared_multilib.sh
. /home/test/toolchains/test/multilib_scripts/startfile_multilib.sh
. /home/test/toolchains/test/multilib_scripts/libc_multilib.sh

if [ -z $1 ]; then return 0; fi
```

```
case $1 in
	libc_start_files)
	    startfile_multilib; break;;
	gcc_core_pass_2)
	gcc_shared_multilib; break;;
	libc)
		libc_multilib; break;;
	*)
		break;;
esac
```

libc_start_file.sh

libc_start_file.sh is used to build and install glibc start files by different gcc options. So script can be something like:

```
#!/bin/sh
startfile_multilib()
{
	local TC_PREFIX=arm-fsl-linux-gnueabi
	local build_dir=${CT_WORK_DIR}/${TC_PREFIX}/build/build-libc-startfiles/
	local src_dir=${CT_WORK_DIR}/src/glibc-2.11.1
	local dest_dir_prefix=${CT_PREFIX_DIR}/${TC_PREFIX}/multi-libs/
	local CC_PATH=${CT_WORK_DIR}/${TC_PREFIX}/build/gcc-core-static/bin

	export PATH=${CT_PREFIX_DIR}/bin:${CC_PATH}:$PATH

	local -a gcc_options=("-march=armv7-a -mfpu=neon -mfloat-abi=softfp" "")
	local -a dest_dirname=("armv7-a/arm/neon" ".")
	local -a fp_config=("--with-fp" "without-fp")

	for ((index=0; index < ${#gcc_options[@]}; index++)) {
		cd ${build_dir}
		make clean

		echo "libc_cv_forced_unwind=yes" > config.cache
		echo "libc_cv_c_cleanup=yes" >> config.cache

		BUILD_CC=${CC_PATH}/${TC_PREFIX}-gcc CFLAGS="${gcc_options[index]}"
CC="${TC_PREFIX}-gcc" AR=${TC_PREFIX}-ar RANLIB=${TC_PREFIX}-ranlib ${src_dir}/configure
--prefix=/usr --build=i686-build_pc-linux-gnu --host=${TC_PREFIX} --without-cvs --disable-profile --disable-debug
--without-gd --with-headers=${dest_dir_prefix}/usr/include --cache-file=config.cache --with-__thread --with-tls
--enable-shared ${fp_config[index]} --enable-add-ons=nptl,ports
--enable-kernel=${CT_LIBC_GLIBC_MIN_KERNEL}

		make OBJDUMP_FOR_HOST=${TC_PREFIX}-objdump ASFLAGS="${gcc_options[index]}"
PARALLELMFLAGS= -j1 csu/subdir_lib

		mkdir ${dest_dir_prefix}/${dest_dirname[index]}/usr/lib -p
		cp -fpv csu/crt1.o csu/crti.o csu/crtn.o ${dest_dir_prefix}/${dest_dirname[index]}/usr/lib
```

```
    }
}
```

gcc_shared_multilib.sh

And gcc_shared_multilib.sh, which be invoked after build gcc-shared, is important to patch gcc to enable you multilib scheme. Gcc not support multilib flexibly.

For example, if compile ARM multilib toolchain, first apply the multilib patch provided by FreeScale. then modify gcc-4.4.4/gcc/config/arm/t-arm-elf it as following according to your scheme:

```
MULTILIB_OPTIONS    = march=armv5te/march=armv6/march=armv7-a mfpu=vfp/mfpu=vfpv3/mfpu=neon mthumb
MULTILIB_DIRNAMES       = armv5te armv6 armv7-a vfp vfpv3 neon thumb
MULTILIB_EXCEPTIONS    = mfpu*
MULTILIB_EXCEPTIONS += *march=armv5te/*mfpu*
MULTILIB_EXCEPTIONS += *march=armv6/*mfpu=vfpv3*
MULTILIB_EXCEPTIONS += *march=armv6/*mfpu=neon*
MULTILIB_EXCEPTIONS += *march=armv7-a/*mfpu=vfp
MULTILIB_EXCEPTIONS += *mfpu*/*mthumb*
MULTILIB_EXCEPTIONS += mthumb*
MULTILIB_EXCEPTIONS += march=armv5te/*mthumb*
MULTILIB_EXCEPTIONS += march=armv6/*mthumb*
```

This scheme will build armv5te, armv6, armv7-a architecture, fpu can be vfp, vfpv3, neon, and instruction type can be arm and thumb

OPTIONS describe the gcc options for multilib.

DIRNAMES describe the directory name corresponding to each option

EXCEPTIONS describe the except of the option combination. For example, -march=armv5te -mfpu=neon is invalid combination.

And next, modify gcc/config/arm/sysroot_suffix.h to define your option alias. For example:

```
#undef SYSROOT_SUFFIX_SPEC
#define SYSROOT_SUFFIX_SPEC "" \
"%{march=armv5te|mcpu=arm946e-s|mtune=arm946e-s|mcpu=arm968e-s|mtune=arm968e-s|mcpu=arm926ej-s|mtune=
arm926ej-s|mcpu=arm10tdmi|mtune=arm10tdmi|mcpu=arm1020t|mtune=arm1020t|mcpu=arm1026ej-s|mtune=arm1026
ej-s|mcpu=arm10e|mtune=arm10e|mcpu=arm1020e|mtune=arm1020e|mcpu=arm1022e|mtune=arm1022e:/armv5te;" \
"march=armv6|mcpu=arm1136j-s|mtune=arm1136j-s|mcpu=arm1136jf-s|mtune=arm1136jf-s|mcpu=mpcore|mtune=mpc
ore|mcpu=mpcorenovfp|mtune=mpcorenovfp|mcpu=arm1156t2-s|mtune=ar1156t2-s|mcpu=arm1156t2f-s|mtune=arm115
6t2f-s|mcpu=arm1176jz-s|mtune=arm1176jz-s|mcpu=arm1176jzf-s|mtune=arm1176jzf-s:" \
   "%{mfpu=vfp:/armv6/vfp;" \
      ":/armv6};" \
"march=armv7-a|mcpu=cortex-a5|mtune=cortex-a5|mcpu=cortex-a8|mtune=cortex-a8|mcpu=cortex-a9|mtune=cortex-a9:
" \
   "%{mfpu=vfpv3:/armv7-a/vfpv3;" \
   "mfpu=neon:/armv7-a/neon;" \
   "mthumb:/armv7-a/thumb;" \
   ":/armv7-a};" \
   ":}"
```

lib_multilib.sh

And the last one, libc_multilib.sh, similar to libc_start_file.sh, compile glibc with different gcc options, the difference is

start file compiled by static gcc, this step compiled by shared gcc.
The script looks like this:

```
#!/bin/sh
libc_multilib()
{
    local TC_PREFIX=arm-fsl-linux-gnueabi
    local build_dir=${CT_WORK_DIR}/${TC_PREFIX}/build/build-libc/
    local src_dir=${CT_WORK_DIR}/src/glibc-2.11.1
    local dest_dir_prefix=${CT_PREFIX_DIR}/${TC_PREFIX}/multi-libs/
    local CC_PATH=${CT_WORK_DIR}/${TC_PREFIX}/build/gcc-core-shared/bin

    export PATH=${binutils dir}:${your shared gcc dir}:$PATH

    local -a gcc_options=("…"  "…"  …)
    local -a dest_dirname=("…"  "…"  …)
    for ((index=0; index < ${#gcc_options[@]}; index++)) {
        cd ${build_dir}
        BUILD_CC=${your shared arm gcc} CFLAGS=${gcc_options[index]} CC=${your shared arm gcc} $src_dir/configure --prefix=/usr/ ${other options}

        make OBJDUMP_FOR_HOST=${your arm binutils objdump} ASFLAGS=${gcc_options[index]} all

        mkdir ${dest_dir_prefix}/${dest_dirname[index]}/usr/lib -p
        make install_root=${dest_dir_prefix}/${dest_dirname[index]} OBJDUMP_FOR_HOST=${your arm binutils objdump} install
    }
}
```

We provide example scripts release package, packed with patch and crosstool-ng. But beware we not guarantee you can get correct toolchain with the scripts, because toolchain building depends on host machine, environment variable you have set and the components configurations.

5 Appendix

5.1 Toolchain test result

Some basic tests have be done to validate the toolchain, below is the result.

gcc built-in compiling test:
---------- gcc 4.4.4 multilib ----------
# of expected passes 7037
# of unsupported tests 44

memcpy() performance test result.

- memcpy: memcpy original from glibc -2.11.1
- memcpy_aroid: memcpy from "Android 2.2 CTS release 2"
- memcpy with neon: memcpy optimized with NEON instruction. Release in multi-libs/armv7-a/neon/

memcpy() performance benchmark result:
data unit(MB/s), meaning of A/B data format:
A is the average value of memcpy with different source and destination address at every test loop.
B is the average value of memcpy with same source and destination address at every test loop.

Block size	memcpy	memcpy_aroid	memcpy with neon
3 bytes	82.4 / 86.8	94.6 / 101.0	127.6 / 143.5
4 bytes	56.1 / 64.3	110.8 / 115.6	139.0 / 192.2
5 bytes	63.3 / 80.4	73.0 / 71.7	171.8 / 239.2
7 bytes	78.2 / 112.4	104.1 / 100.4	220.9 / 336.5
8 bytes	87.1 / 128.4	222.4 / 230.8	218.4 / 384.6
11 bytes	112.0 / 176.6	177.6 / 173.1	281.9 / 526.2
12 bytes	119.9 / 192.9	301.4 / 313.1	276.7 / 574.4
15 bytes	142.3 / 241.2	211.4 / 204.2	334.6 / 718.0
16 bytes	148.9 / 257.2	165.3 / 368.2	271.7 / 539.1
24 bytes	193.8 / 385.9	243.2 / 508.5	373.5 / 808.8
31 bytes	227.1 / 498.4	322.1 / 360.2	486.3 / 1044.7
4096	1360.4 / 2246.0	3394.9 / 3560.4	3551.7 / 3695.4
6144	1385.3 / 2270.4	3508.3 / 3630.9	3586.7 / 3671.9
65536	990.1 / 1298.7	1557.7 / 1369.3	1577.5 / 1545.9
98304	736.9 / 884.1	1544.2 / 1350.3	1552.9 / 1463.8

5.2 Release code directory structure

```
|-- crosstool-ng                //crosstool-ng build script, already applied multilib support patches
|-- multilib.sh                 //multilib build script entrance.
|-- multilib_scripts            //multilib build scripts
|     |-- gcc_shared_multilib.sh
|     |-- libc_multilib.sh
|     `-- startfile_multilib.sh
`-- patch                       //patch for source code and crosstool-ng
      |-- crosstool-ng
      |-- gcc                   //patches for gcc-4.4.4
      `-- glibc                 //routine optimized patch
```