Autotools and Intel® Xeon Phi™ Coprocessor

One of the strengths of the Intel® Xeon Phi™ coprocessor is the ability to build existing software to run on the Intel® Many Integrated Core (Intel® MIC) hardware with a minimum of change (in most cases, no changes to the code itself are necessary). The same cannot always be said, however, for the build systems used to compile existing software packages.

For software built with nothing more complicated than Makefiles (or even just invoking the Intel® Composer XE compiler directly), the necessary build change is often as simple as adding \texttt{--mmic} to the set of compiler and link flags. When the build scripts are generated via GNU Autotools (\texttt{automake/autoconf}), \texttt{SCons} or Kitware \texttt{CMake}, however, life becomes more interesting.

In this article we will discuss the mechanics of building existing software packages with GNU Autotools. As using CMake to build software for Intel Xeon Phi coprocessor is already handled in an existing article, we will not discuss it further here. Similarly, in this article we will not discuss using SCons to build software for the Xeon Phi coprocessor.

**Autotools for embedded systems**

When it comes to considering an autotools-based package port to the Intel Xeon Phi coprocessor, the simplest way to view the coprocessor is as an embedded system. Embedded target systems typically run different hardware than the host build platform, and this prevents the possibility of running code built for the target, on the host platform.

This becomes a potential issue for autotools-based builds, as many build setups may attempt to build and run automatically-generated code snippets on the host build platform, and this obviously won’t work when the code is built for a different hardware architecture. Autotools is not ignorant of this need – in fact, the GNU Compiler Collection (GCC) uses autotools for its build system and GCC is often cross-compiled for target architectures different than that of the build platform.

To this end, \texttt{autotools supports cross-compile}ation directly. The most visible result of using the cross-compile flags with autotools builds is suppression of any attempt to invoke code built for the target, on the host system.

Since the best teacher is often an example, let’s start with one. In order to follow along at home, you will need

1 Things become slightly more complicated when code builds involve “fat” libraries and binaries – those including code intended to run on both Intel® Xeon™ and Intel® Xeon Phi™ coprocessor hardware. Such scenarios are outside the scope of this article.
- A working copy of Intel® Composer XE 2013 or newer (for Linux x86_64)
- A Linux installation of Intel® Manycore Platform Software Stack (MPSS)
- A version of Linux supported by the Intel® MPSS (which, at the time of this writing, means 64-bit Red Hat Enterprise Linux 6.x (although equivalent versions of CentOS 6.x have been known to work as well). This example is specific to RedHat Enterprise Linux, but similar steps can be applied to other distributions supported by the Intel® MPSS.
- Standard software-development packages shipped with RHEL6/CentOS6:
  - autoconf
  - automake
  - libtool
  - m4

Note that although the version of GCC shipped with Intel® MPSS can build software to run on the Intel Xeon Phi coprocessor, it cannot produce optimized code that takes advantage of the hardware vector units, and is generally used instead to build custom Linux kernels from the source provided with the MPSS distribution. Although it is possible to build this GCC to run on the coprocessor hardware itself, code built in this manner still will not take full advantage of the hardware. For this reason, in this article we will consider only software packages cross-compiled on the host using the Intel® Composer XE 2013 or newer compiler suites.

### e2fsprogs

**e2fsprogs** is a suite of tools and libraries intended for use with the EXT2/3/4 Linux filesystems. I have needed to build e2fsprogs on more than one occasion, usually to satisfy a dependency on the `uuidgen()` API call provided by one of the e2fsprogs libraries (libuuid). I will start with e2fsprogs as the first example as it is a straightforward build, with no surprises or special considerations (and more importantly, no dependencies other than itself and a Linux system), and also provides command-line utilities that make for good tests and examples.

#### Preparing the build

The following steps assume you have already obtained a tarball of the e2fsprogs source code, and have placed it in a directory on a filesystem convenient for you. In this example, I have used a directory called `/usr/pic1/gjunker/projects` on a host named `il028` for that purpose. After that, we need to source the Composer XE 2013 compilervars.csh script (or compilervars.sh, for sh/bash shells) to add paths to the Composer XE 2013 tools and libraries to our shell environment:

```bash
il028 [/usr/pic1/gjunker/projects] 68% source <path to Composer XE 2013>/bin/compilervars.csh intel64
```

and then unpack the tarball:

```bash
il028 [/usr/pic1/gjunker/projects] 72% tar xvf e2fsprogs-1.42.8.tar.gz
```

In this example I want to show the difference between a normal host build, and a cross-compile for Intel Xeon Phi coprocessor, so I set up an out-of-source build directory to make this simpler to manage.
Building for Intel® Xeon™ and Intel® Xeon Phi™ Coprocessor

Next, I configure and build for the host (a “normal” build, in other words):

```
    % [usr/pic1/gjunker/projects/e2fsprogs-1.42.8/build/xeon] 78% env CC=icc LD=icc ../../configure --prefix=/usr/pic1/gjunker/sysroot \
    % [usr/pic1/gjunker/projects/e2fsprogs-1.42.8/build/xeon] 78% make -j12 install
```

In the above steps, I am instructing the configure script to use icc as its C compiler (CC variable) and also as its linker driver (LD variable). I am also using a local, non-system directory as the install prefix to avoid permissions issues (or the need to run the install step as root), since this is just an example. The –j12 flag allows the build to run up to 12 jobs in parallel (the build host is a dual-socket Xeon X5680). To make sure the build works, we run the uuidgen command we just built, and observe the expected output:

```
    % [usr/pic1/gjunker/projects/e2fsprogs-1.42.8/build/xeon] 79% /usr/pic1/gjunker/sysroot/bin/uuidgen
    7c6d3ae9-5a06-4bce-a20f-ec60bb8e9e0b
```

Next we repeat the same, but for the coprocessor build, and from inside the coprocessor out-of-source build tree:

```
    % [usr/pic1/gjunker/projects/e2fsprogs-1.42.8/build/mic] 82% env CC=icc LD=icc CFLAGS=--mmic LDFLAGS=--mmic ../../configure --prefix=/usr/pic1/gjunker/microot --host=x86_64-k1om-linux
```

Note the addition of the –mmic flag to the CFLAGS and LDFLAGS variables; this is how you tell an autotools build with icc that it should build and link for the Intel Xeon Phi coprocessor. This is separate from the other important extra argument in this step: --host. By providing this argument, we are telling the configure script that we are doing a cross-compile, and it should not try to run anything it builds as part of the build process. Note that you can provide virtually any value for this argument; in this case I am providing a “triple” that uses the coprocessor k1om architecture type, but you could even use the same triple as the build host if you like – it is the presence of the --host argument that is important.

Next we perform the build, installation, and try to run one of the resultant executables:

```
    % [usr/pic1/gjunker/projects/e2fsprogs-1.42.8/build/mic] 83% make -j16 install
    % [usr/pic1/gjunker/projects/e2fsprogs-1.42.8/build/mic] 84% /usr/pic1/gjunker/microot/bin/uuidgen
    /usr/pic1/gjunker/microot/bin/uuidgen: Exec format error. Wrong Architecture.
```
What went wrong here? The problem is that we tried to run the Intel Xeon Phi coprocessor executable on the host Intel® Xeon™ platform, and the OS properly informed us that was not possible. We need instead to run the executable on the coprocessor, so let’s copy the executable up to the card and run it there:

```
scp /usr/pic1/gjunker/microot/bin/uuidgen mic0:/tmp
```

```
Root@mic0:~> uuidgen
100%  54KB   53.5KB/s  00:00
```

```
ssh mic0 /tmp/uuidgen
```

```
f30b4546-781d-4309-9dd5-be1689c59d45
```

This exercise also serves to illustrate another limitation of cross-compiling (one also not unique to building for the coprocessor): it is not possible (or at least not easily possible) to run the self-tests (i.e. `make test`, `make check`, etc.) that many open-source software packages provide, when cross-compiling for the Intel Xeon Phi coprocessor. One needs either to forego the tests, or run them manually on the device.

**zlib**

Another common dependency for many open-source projects is **zlib**. This package is readily available by default in virtually every Linux distribution in existence, but if you need it for the Intel Xeon Phi coprocessor you typically need to build it yourself. However, there is a catch – while **zlib** uses a script called “configure” to set up its build, it is **not** an Autotools configure script. That said, building **zlib** for Intel Xeon Phi coprocessor is not much different than building **e2fsprogs** was; in fact, since this is not an autotools script, we do not need to provide the `--host` flag.

First we unpack the archive (note that **zlib** does not support out-of-source builds, so it’s a bit more hassle to build for both Xeon and the Xeon Phi coprocessor.)

```
tar xf zlib-1.2.8.tar.gz
cd zlib-1.2.8/
```

We supply the same `CC` and `LD` values we did for **e2fsprogs**, and perform the build and install:

```
configure --prefix=/usr/pic1/gjunker/sysroot --64
make -j16
make install
```

A simple test is to use the built **minigzip64** app to compress an arbitrary file, and verify that the result is what we expect:

```
./minigzip64 /tmp/myapp.log
```
Next we repeat for the Intel Xeon Phi coprocessor. Similar to the e2fsprogs device build, we add \texttt{--mmic} to the \texttt{CFLAGS} and \texttt{LDFLAGS} variables. We also added the \texttt{--static} flag to avoid having to copy \texttt{libz.so} to the card for testing (\texttt{--static} will ensure that the built \texttt{zlib} applications are statically linked). After building and verifying that the executable cannot be run on the host system, we copy the executable and a test file to the card and verify expected results.

\begin{verbatim}
 i1028 [/usr/pic1/gjunker/projects/zlib-1.2.8-mic] 156% env CC=icc CFLAGS=--mmic LD=icc LDFLAGS=--mmic ./configure --static --prefix=/usr/pic1/gjunker/microot --64
 i1028 [/usr/pic1/gjunker/projects/zlib-1.2.8-mic] 157% make -j16
 i1028 [/usr/pic1/gjunker/projects/zlib-1.2.8-mic] 158% make install
 i1028 [/usr/pic1/gjunker/projects/zlib-1.2.8-mic] 159% ./minigzip64 ./minigzip64: Exec format error. Wrong Architecture.
 i1028 [/usr/pic1/gjunker/projects/zlib-1.2.8-mic] 160% scp /tmp/myapp.log mic0:/tmp
   myapp.log 100%  217  0.2KB/s  00:00
 i1028 [/usr/pic1/gjunker/projects/zlib-1.2.8-mic] 161% scp minigzip64 mic0:/tmp
   minigzip64 100%  171KB 171.3KB/s  00:00
 i1028 [/usr/pic1/gjunker/projects/zlib-1.2.8-mic] 162% ssh mic0
   /tmp/minigzip64 /tmp/myapp.log
\end{verbatim}

\textbf{Conclusion}

In this article, we have demonstrated that building open-source software packages for the Xeon Phi coprocessor is really no different than building for any embedded system, and if the reader considers a coprocessor as such, much potential confusion surrounding software builds for the Intel Xeon Phi coprocessor can be eliminated. Happy building!