COPRTHR-2
Development Tools

This document describes the COPRTHR-2 development tools. Section 0 describes the compiler front-end *coprcc* used for compiling executable code for the coprocessor device. Section 2 describes the binary analysis tool *coprcc-info* used to extract information about a compiled binary. Section 3 describes the specialized shell command *coprsh* used to control the environment of the COPRTHR-2 run-time. Section 4 describes the interactive core debugger *coprcc-db*.

Contents

1 Compiling Coprocessor Binaries: coprcc

With COPRTHR-2 compiling executable code for a coprocessor device is accomplished with the compiler front-end *coprcc*. The primary purpose of *coprcc* is to provide the correct compilation environment and drive the COPRTHR-2 compilation model using the native compiler for the coprocessor. The default native compiler is GCC which may be overridden by the environment variable `COPRTHR_CC` or using the command-line flag `-mcc`.

1.1 Options Reference

`coprcc` `[options] infile... [-o outfile]`

Options:

`--extract`

Extract coprocessor binary from host program.

`-fdynamic-calls`

Enable use of dynamic calls.

`-fhost`

Generate host-executable program with embedded coprocessor binary for implicit offload. The default is to generate a coprocessor binary for explicit offload.

`--info`

Apply *coprcc-info -b* to final output file to generate brief summary information.

`--info-more`

Apply *coprcc-info -j* to final output file to generate detailed summary information.

`-k, --keep`

Keep intermediate files.

`-v`

Show details of compilation.

1.2 Compilation Model

COPRTHR-2 provides several variations to the compilation model suitable for different application development scenarios:

- Compile source to a host-executable coprocessor binary (a.out)
- Compile source to a offload-executable coprocessor binary (e.out)
- Compile source to a re-linkable object file

1.2.1 Host-Executable Coprocessor Binaries

For simple applications *coprcc* is able to generate a host-executable program for executing application code on the coprocessor. This is accomplished using the `-fhost` command-line flag for *coprcc*. This compilation model requires that a conventional main routine is provided, and which will be used as the entry point of the executed program on the coprocessor device. The coprocessor executable binary is linked into a host-executable program that will automatically perform the offload of execution to the coprocessor device.

As an example, for a conventional C main routine:

```
] coprcc -fhost main.c
```

The result will be an *a.out* file that can be executed directly on the host platform:

```
] ./a.out
```

The embedded coprocessor binary will be automatically offloaded. This compilation model requires no explicit host code and is the simplest way to target the coprocessor.

1.2.2 Offload-executable Coprocessor Binaries

The default behavior of coprcc is to generate coprocessor binaries that require explicit host code to off-load work to the coprocessor device. Two off-load call models are supported. The first is a Pthreads style interface which requires the programmer to mark the entry point for the coprocessor program using the `__entry` qualifier:

```
void __entry my_thread( void* parg ) {
	...
}
```

Assuming the code is in my_thread.c the code is compiled for the coprocessor using:

```
] coprcc my_thread.c
```

The result will be an *e.out* file that can be explicitly loaded into a host program for offload to the coprocessor:

```
int main() {
       ...
       coprthr_
       coprthr_program_t prg = coprthr_cc_read_bin("./e.out",0);
       coprthr_sym_t thr = coprthr_getsym(prg,"my_thread");
       ...
}
```

Here the C main program would be compiled for the host using the native compiler, e.g., GCC.

1.2.3 Re-linkable coprocessor binaries

As with the native compiler, *coprcc* supports generating re-linkable object files. As an example, assume that *foo.c* and *bar.c* contain routines needed by the main thread routine in *my_thread.c*. Compilation may be broken up into steps as expected:

```
] coprcc –c foo.c
] coprcc –c bar.c
] coprcc my_thread.c foo.o bar.o
```

Here *foo.o* and *bar.o* will be linked into the final *e.out* coprocessor binary.

2 Analyzing Compiled Binaries: coprcc-info

The layout in memory of COPRTHR-2 coprocessor binaries for the Epiphany architecture is segmented to address specific requirements for fast execution and the efficient use core-local memory, with many features being provided to the application developer for precise control over the compiled binaries. The *coprcc-info* tool can be used to analyze COPRTHR-2 coprocessor binaries to support diagnostic and optimization work. The tool is similar to the binutils *nm* and *readelf* commands, but provides detailed information specific to the COPRTHR-2 compilation and execution models.

As an example, below is summary information for the matrix-matrix multiply kernel obtained with the –b command line option:

```
dar@parallella3:$ coprcc-info -b cannon_tfunc.e32
file: cannon_tfunc.e32
architecture: Epiphany
ELF type: EXEC (loadable binary)
local memory:
        total size 32 KB
        syscore 1024 bytes (3.1%)
        user code 9504 bytes (29.0%)
        fragmentation 220 bytes (2.1%)
        free memory 22112 bytes (67.5%)
```

An additional reference example using *coprcc-info* to analyze the matrix-matrix multiple kernel is provided at the end of this section.

2.1 Options Reference

`coprcc-info` `[options] file`

Options:

`--base` ***`file`***

Set base *file* for delta calculation.

`-b, --brief`

Display summary information only.

`-B`

Do not display summary information.

-d

Display section for each symbol.

-D

Display dynamic calls only.

-g, --group

Group symbols by package.

-h, --help

Show usage.

-H

Display host calls only.

-j

Display segment headers.

-l, --large *bytes*

Highlight symbols larger than *bytes* in size.

-L

Highlight symbols larger than 256 bytes.

-p, --package *name*

Display symbols from specified named package only.

-P, --no-package

Do not display symbols from packages.

-s, --section *name*

Display symbols from specified named section only.

-S, --segment *segnum*

Display symbols from specified segment number only. Special keywords may be used in place of segment number (`IVT, CONFIG, SYSCORE, USRCORE, SYSMEM, USRCORE`).

--version

Print version information.

2.2 Reference example for the output from coprcc-info

The output from applying coprcc-info to the matrix-matrix multiply kernel is shown below. The output is annotated to allow the identification of various fields and symbols to describe the coprocessor binary layout and elements. These are described below.

(1) Summary information showing the byte allocation in core-local memory. The size of 'user code' is for the USRCORE segment only. Fragmentation is a measure of padding or other bytes that are unusable for instructions or data due to alignment requirements.

(2) This is a segment header inserted using the `-j` command line option. The format is (in order): segment number, starting address, ending address, size in bytes, and segment name.

(3) This is a symbol within the SYSCORE segment, specifically, the main syscore routine. The format for symbol information is starting address, size in byes, padding in bytes, symbol type, and symbol name. For symbol type, F = function and O = data object. A '~' preceding the 'F' indicates that the symbol is an alias. Here _syscore is marked with an 'F' to indicate that it is a function.

(4) An example of a data objected with symbol type marked as 'O'.

(5) This is the USRCORE segment where critical user code is placed.

(6) This is an example of a symbol alias marked as '~F'. In this case the symbol is `_MPI_Comm_rank` which is an alias for `_coprthr_mpi_comm_rank`.

(7) The symbol `__local_mem_base` is the end of the USRCORE segment, in this case core-local address 0x27d0. The memory between this address approximately 0x7000 represents core-local memory available to the application for memory allocation. This measure is approximate since the stack will grow downward from 0x8000 and care must be taken to avoid collisions.

(8) This is the USRMEM segment in off-chip DRAM where more memory is available but access is significantly slower.

```
dar@parallella3:$ coprcc-info -j cannon_tfunc.e32
file: cannon_tfunc.e32
architecture: Epiphany
ELF type: EXEC (loadable binary)
local memory:
        total size 32 KB
        syscore 1024 bytes (3.1%)
        user code 9504 bytes (29.0%)                  <---------- (1)
        fragmentation 220 bytes (2.1%)
        free memory 22112 bytes (67.5%)
Symbols:
segment 0: 0x0000-0x0004 4 (IVT)  <-------------------------- (2)
            0x0000    4       F _start
segment 1: 0x0000-0x0000 0 (CONFIG)
```

```
segment 2: 0x0058-0x03e0 904 (SYSCORE)
            0x0064         28  F init
            0x0080  132    12  F _epiphany_start
            0x0110  374     2  F _syscore         (3)
            0x0288  188        F ___sys_barrier
            0x0344         20  F fini
            0x0358   16        O ___coprthr_barrier_state
            0x0368    8        O ___coprthr_thread
            0x0370   40        O ___coprthr_proc
            0x0398   16        O _sys_barrier_state
            0x03a8    8        O _sys_thread
            0x03b0   40        O _sys_proc        (4)
            0x03d8    4     4  O _core_timer_0
            0x03e0             F __syscore_high
segment 3: 0x0400-0x2920 9504 (USRCORE)       (5)
            0x0400    4     4  F __init_tab
            0x0408 1416        F _my_thread
            0x0990 1092     4  F _MatrixMultiply
            0x0dd8   78        F ___syslog
            0x0e22    4     2  F ___wrap___syslog
            0x0e28   30     2  F _readi
            0x0e48   30     2  F _read_h
            0x0e68  342     2  F ___coprthr_mpi_init
            0x0fc0   52     4  F ___coprthr_mpi_finalize
            0x0ff8   40        F _coprthr_mpi_comm_rank
            0x0ff8            ~F _MPI_Comm_rank   (6)
            0x1020   40        F _coprthr_mpi_comm_size
            0x1020            ~F _MPI_Comm_size
            0x1048   64        F ___e_irq_set
            0x1088   30     2  F _readi
            0x10a8   30     2  F _read_h
            0x10c8  652     4  F _coprthr_mpi_cart_create
            0x10c8            ~F _MPI_Cart_create
            0x1358  146     6  F _coprthr_mpi_cart_coords
            0x1358            ~F _MPI_Cart_coords
            0x13f0            ~F _MPI_Cart_shift
            0x13f0  754     6  F _coprthr_mpi_cart_shift
            0x16e8   30     2  F _readi
            0x1708   30     2  F _read_h
            0x1728 1610     6  F _coprthr_mpi_sendrecv_replace
            0x1728            ~F _MPI_Sendrecv_replace
            0x1d78   30     2  F _readi
            0x1d98   30     2  F _read_h
            0x1db8   30     2  F _xxx_readi
            0x1dd8   40        F _coprthr_tls_brk
            0x1e00   68     4  F _coprthr_tls_sbrk
            0x1e48  218     6  F ___coprthr_dma_setup_xfer
            0x1f28  244     4  F ___coprthr_dma_setup_xfer2d
            0x2020  396     4  F ___coprthr2_memcopy_align
            0x21b0  422     2  F ___coprthr2_memcopy2d_align
            0x2358   38     2  F ___coprthr2_wait
            0x2380   52     4  F _coprthr_ctimer_reset
            0x23b8   46     2  F _coprthr_ctimer_get
            0x23e8  212     4  F ___coprthr_barrier
```

```
            0x24c0     6     2  F ___coprthr_mutex_set
            0x24c8     8        F ___coprthr_mutex_unlock
            0x24c8             ~F ___coprthr_mutex_init
            0x24d0     8        F ___coprthr_mutex_testlock
            0x24d8    16        F ___coprthr_mutex_testlock_self
            0x24e8    16        F ___coprthr_mutex_trylock
            0x24f8    20        F ___coprthr_mutex_trylock_self
            0x250c    18     2  F ___coprthr_mutex_lock
            0x2520    26     2  F ___coprthr_mutex_lock_self
            0x253c    14     2  F ___coprthr_dma_start_0
            0x254c    14     2  F ___coprthr_dma_start_1
            0x255c    14     2  F ___coprthr_dma_wait_0
            0x256c    14     2  F ___coprthr_dma_wait_1
            0x257c   390     2  F ___divsi3
            0x2704           8  F __exit
            0x270c     8        F ___esyscall_phalt
            0x2714   126     2  F ___esyscall
            0x2794     4        O ___mem_free
            0x2798              F __bss_start
            0x2798           8  F _edata
            0x27a0    24        O ___dma1_desc
            0x27b8    24        O ___dma0_desc
            0x27d0              F _end
            0x27d0   334        F __thread_init
            0x27d0              F __local_mem_base <----------- (7)
segment 4: 0x0000-0x0000 0 (SYSMEM)
segment 5: 0x8e002000-0x8e002ca8 3240 (USRMEM) <----------- (8)
        0x8e002000    58     2  F _exit
        0x8e00203c     4        O __global_impure_ptr
        0x8e002040   206     2  F _memcpy
        0x8e002110   308     4  F ___call_exitprocs
        0x8e002248     4     4  O __impure_ptr
        0x8e002250 1096         O _impure_data
        0x8e002698   334     2  F ___modsi3
        0x8e0027e8   326     2  F ___init_core_local_data
        0x8e002930   878        F _MatrixMatrixMultiply
segment 6: 0x0000-0x0000 0
```

3 Run-Time Shell Command: coprsh

The run-time shell command *coprsh* is used for setting up the COPRTHR-2 run-time environment. It may be used for the immediate execution of an application using the coprocessor, or to create a shell with a fixed environment defined. The *coprsh* command may also be used to query the current environment for information impacting the COPRTHR-2 run-time. An example of the most common use is to set the number of threads that are to be used on the coprocessor and to control the verbosity of debug message reporting.

Consider the example of the program my_program.x that we would like to execute using 8 threads on the coprocessor. This is accomplished with:

```
] ./coprsh –np 8 -- ./my_program.x
```

If we want to limit debug messages to those identified as at least as critical as an error (but excluding warnings), this can be accomplished by modifying the options:

```
] ./coprsh –np 8 –r err -- ./my_program.x
```

In some cases we would like to create a shell with the COPRTHR-2 environment already setup in order to avoid having to continually specify it. This can be accomplished with:

```
] ./coprsh –np 8 –r err -- bash
] ./my_program.x
```

In the above example, *my_program.x* will be executed using eight (8) threads and with debug messages less critical than *err* suppressed, for each time a program is executed within the bash shell that was created.

3.1 Option Reference

coprsh [options] [-- command [command-options]]

Options:

-h, --help

Show this usage information.

-nc, -np ***num***

Set number of coprocessor threads.

`-r` ***`level`***

Set clmesg report level where *`level`* may be a number from 0-7 or one of the following aliases: `emerg(0)`, `alert(1)`, `crit(2)`, `err(3)`, `warning(4)`, `notice(5)`, `info(6)`, `debug(7)` .

`-v`

Show what is being done.

`--version`

Show version information.

4 Integrated Core Debugger: coprcc-db

The integrated core debugger coprcc-db allows interactive debugging for many-core coprocessors. With the run-time core debugger enabled, at any time during the execution of code on the coprocessor (especially useful when your code is hanging due to a bug), hitting *ctrl-z* will cause the terminal to drop-down into the core debugger *coprcc-db*. This will provide a shell-like command-line that can be used to query the state of the coprocessor cores. The core debugger is designed to support many-core coprocessors and provides a simple prefix notation for applying any command to any number of selected cores. The use of a shell-based debugger extends its functionality to include any shell commands for a flexible and familiar debugging environment. As an example, the output of any debug command may be piped through UNIX commands like *grep* and also redirected to an output file for subsequent analysis.

4.1 Command Reference

Command structure:

`[core-select-prefix] command [options] [| command ...]`

Commands:

`continue`

Continue execution from halt.

`coredump`

Dump core local memory to file(s)

`help`

Print this help information.

`mem start[,end]`

Display memory content from address *start* to address *end*. If *end* address is omitted a single word is displayed.

`quit`

Quit the debugger.

`reg [`*`regname,...`*`]`

Display contents of selected registers specified as comma separated list. Available registers are: `config, status, pc, imask, ipend`.

`sh <shell-cmd>`

Execute (only) an ordinary shell command.

`state`

Display core execution state.

`sym [`*`symbol`*`]`

Display value of *`symbol`*. If no symbol is specified then all available symbols are displayed.

Syntax and conventions:

- **Core select prefix.** All commands may be prefixed to apply the command to one or more specific coprocessor cores. A comma separated list of core numbers or ranges of core numbers may be used. In the example below, the status register for cores 7,10,11,12, and 15 will be reported:

```
(coprdb) 7,10-12,15 reg status
```

- **Pipes.** The output of a command may be piped to a shell command. Useful shell commands for post-processing the output of a coprdb command are *grep*, *awk*, and *tee*. Any shell command including customer programs can be used. In the example below, the output of the *reg* command being filtered using *grep* to print only the results for cores showing an exception status, and then *more* is used to Pipe command to shell example:

```
(coprdb) reg pc,status | grep -e except | more
```

- **Redirects.** The output of a command may be redirected to a file like any shell command. In the example below, the memory contents over a specific range is being written to an output file:

```
(coprdb) mem 0x400,0x500 > logfile.txt
```

- **Multiple commands.** Multiple commands may be combined using a semicolon. In the example below, the PC register values would be reported forst followed by the STATUS register values:

```
 (coprdb) reg pc; reg status
```