CHERI
Capability Hardware Enhanced RISC Instructions
Preliminary User's Guide

Version 1.5

This interim document is not released for public consumption

Robert N. M. Watson, David Chisnall, Brooks Davis,
Wojciech Koszek, Simon W. Moore, Steven J. Murdoch, Jonathan Woodruff
SRI International and the University of Cambridge[1]

March 3, 2013

[1]Sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

Contents

Abstract

This document is the *User's Guide* for SRI International and the University of Cambridge's research prototype implementation of the Capability Hardware Enhanced RISC Instructions (CHERI) instruction set architecture (ISA). It complements the *CHERI Platform Reference Manual* in providing information required by hardware and software developers working with the prototype. The document is intended to capture our thoughts early in the research and development cycle.

The User's Guide is targeted at system developers bringing up an operating system and compiler stack for CHERI; future versions of this document will also address end users. The guide describes the CheriBSD operating system, and CHERI-adapted Clang/LLVM compiler suite, and Deimos demonstration microkernel.

Acknowledgments

The authors of this report thank other members of the CTSRD team, and our research collaborators at SRI and Cambridge:

Peter G. Neumann	Ross J. Anderson	Jonathan Anderson
Gregory Chadwick	Nirav Dave	Khilan Gudka
Steven Hand	Asif Khan	Myron King
Ben Laurie	Patrick Lincoln	Anil Madhavapeddy
Andrew Moore	Will Morland	Alan Mujumdar
Robert Norton	Philip Paeps	Michael Roe
John Rushby	Hassen Saidi	Muhammad Shahbaz
Stacey Son	Richard Uhler	

The CHERI team wishes thank its external oversight group for significant support and contributions:

Lee Badger	Simon Cooper	Rance DeLong
Jeremy Epstein	Virgil Gligor	Li Gong
Mike Gordon	Andrew Herbert	Warren A. Hunt Jr.
Doug Maughan	Greg Morrisett	Brian Randell
Kenneth F. Shotting	Joe Stoy	Tom Van Vleck
Samuel M. Weber		

Chapter 1

Introduction

This is the *User's Guide* for the Capability Hardware Enhanced RISC Instructions (CHERI) prototype that complements the *CHERI Preliminary Architecture Document*, which specifies the CHERI architecture and ISA, and *CHERI Platform Reference Manual*, which describes the hardawre prototype.

The User's Guide describes the CheriBSD prototype, a version of the Clang/LLVM/gas toolchain used with CHERI, and the Deimos demonstration microkernel. The Guide is intended to address the needs of hardware and software developers prototyping new hardware features, bringing up operating systems, language runtimes, and compilers on CHERI, rather than literal end users. Future iterations will continue to flesh out operational aspects of the CHERI processor.

1.1 Background

Capability Hardware Enhanced RISC Instructions (CHERI), developed by SRI International and the University of Cambridge, are security extensions for the MIPS64 instruction set architecture (ISA). The CHERI ISA provides direct hardware support for fine-grained compartmentalisation of (and within) system software and application software – a technique used to mitigate security vulnerabilities and enforce fine-grained security policies.

Whereas traditional CPU designs provide heavy performance and programmability penalties for employing compartmentalisation, CHERI's ISA features support fast and easy application compartmentalisation of C-language systems software. These improvements are made possible by integrated hardware support for continuous memory protection and enforcement using memory segments and the object capability model. Contemporary software trusted computing bases (TCBs) such as operating system kernels and language runtimes are of particular interest, as CHERI will allow us to improve their security and reliability – and hence the security and reliability of applications built on top of those services.

Detailed information on the CHERI ISA and possible uses may be found in the *CHERI Preliminary Architecture Document*, including new co-processor registers and instructions. The CHERI prototype is a reference implementation of the CHERI ISA, intended to allow us to validate the approach through a complete system implementation; CHERI is based on the Bluespec Extensible

RISC Implementation (BERI) FPGA soft core, and implemented as an additional coprocessor. The distinction between BERI and CHERI is evolving – however, the long-term hope is that BERI will be a reusable platform across multiple research projects in the hardware-software interface.

1.2 Version history

This is the fifth version of the *CHERI User's Guide*.

1.0 An initial version of the *CHERI User's Guide documenting* the implementation status of the CHERI prototype, including CHERI ISA and processor implementation, as well as user information on how to build, simulate, debug, test, and synthesise the prototype.

1.1 Minor refinements to the text and presentation of the document, as well as incremental updates to its descriptions of the SRI/Cambridge development and testing environments.

1.2 This version of the *CHERI User's Guide* follows a first demonstration of CHERI synthesised for the Terasic tPad FPGA platform. The Guide contains significant updates on the usability of CHERI features, the build process, and debugging features such as CHERI's debug unit. A chapter has been added on Deimos, a demonstration microkernel for the CHERI architecture.

1.3 The document has been restructured into hardware prototype and software reference material. Information on the status of MIPS ISA implementation has been updated and expanded, especially with respect to the MMU. Build dependencies have been updated, as well as information on the CHERI simulation environment. The distinction between BERI and CHERI is now discussed in detail. The Altera development environment is now described in its own chapter. A new chapter has been added detailing bus and device configuration and use of the Terasic tPad and DE4 boards, including the Terasic/Cambridge MTL touch screen display. New chapters have been added on building and using CheriBSD, as well as a chapter on FreeBSD device drivers on BERI/CHERI. A new chapter has been added on cross-building and using the CHERI-modified Clang/LLVM suite, including C-language extensions for capabilities.

1.4 This version introduces improved Altera build and Bluespec simulation instructions. A number of additional C-language extensions that can be mapped into capability protections are introduced. FreeBSD build instructions are updated for changes to the FreeBSD cross-build system. Information on the CHERI2 prototype is added.

1.5 In this version of the *CHERI User's Guide*, several chapters describe the CHERI hardware prototype have been moved into a separate document, the *CHERI Platform Reference Manual*, leaving the User's Guide focused on software-facing activities.

1.3 Licensing

CheriBSD and the CHERI-enhanced version of the Clang/LLVM have been released under BSD licenses. Modifications to the pathname gas assembler have been released under GPLv2.

The Deimos demonstration microkernel is currently proprietary, although we intend to release it under a BSD license in the future.

1.4 Document Structure

This document is an introduction to and user manual for version one of the Capability Hardware Enhanced RISC Instructions (CHERI) CPU prototype:

Chapter 2 describes how to build the FreeBSD/BERI port, as well as our CheriBSD adaptation.

Chapter 3 describes how to use FreeBSD/BERI and CheriBSD.

Chapter 4 describes CHERI extensions to Clang/LLVM and the C programming language.

Chapter 5 provides additional reference material for device driver configuration and use under FreeBSD/BERI and CheriBSD.

Chapter 6 describes Deimos, a prototype operating system software stack for CHERI. This microkernel is able to run on CHERI in simulated and synthesised forms, illustrating some of CHERI's protection and compatibility properties.

Chapter 2

Building CheriBSD

FreeBSD/BERI and CheriBSD are adaptations of the open source FreeBSD operating system to run on the Bluespec Extensible RISC implementation (BERI) and CHERI. This chapter describes how to check out and build FreeBSD/BERI and CheriBSD.

2.1 Obtaining FreeBSD/BERI and CheriBSD source code

Source code for both FreeBSD/BERI and CheriBSD is maintained in the FreeBSD Perforce repository in the following Depot locations:

```
//depot/projects/ctsrd/beribsd/...
//depot/projects/ctsrd/cheribsd/...
```

The FreeBSD/BERI development tree is branched from the FreeBSD 10.x development tree in Perforce. The CheriBSD development tree is branched from the FreeBSD/BERI tree and contains additions to the FreeBSD/BERI branch in order to support CHERI's capability coprocessor. To use Perforce, you must have an account; contact Robert Watson if you do not have one, but require one. Configure the Perforce client by creating a `.p4config` file in your home directory with the following contents:

```
P4PORT=perforce.freebsd.org:1666
P4USER=<username here>
P4PASSWD=<password here>
P4CLIENT=<client name here>
```

Replace the values for `P4USER` and `P4PASSWD` as appropriate. Perforce requires that each client be configured with a unique name; you should come up with a unique name, perhaps using the template `username_hostname`, and replace the value for `P4CLIENT` appropriately. For example, a suitable client name might be `rwatson_svr_ctsrd_mipsbuild`, reflecting a Perforce username of `rwatson` on the machine `svr-ctsrd-mipsbuild`. Use the `p4 client` command to configure the client; a typical client configuration will include fields along the following lines:

```
Client: rwatson_svr_ctsrd_mipsbuild
Host: svr-ctsrd-mipsbuild
Root: /local/scratch/rnw24/p4
View:
```

Modify the `Root:` entry as required – typically to point it at a scratch area, such as a `p4` tree under your home directory; the default Perforce choice of synchronising the depot to your home directory is not suitable for most users.

You will need to add a new entry to the `View:` section to map the appropriate areas of the Depot into your local workspace. Add the following to the `View:` section on a new line indented by a single tab.

```
//depot/projects/ctsrd/...  //<your client>/projects/ctsrd/...
```

Ensure that there is no mapping of the complete depot space (`//depot/...`) into your local client space – most users will want to check out only specific subtrees from the depot. Replicating the depot directory layout (i.e., mapping the depot's `//depot/projects/ctsrd/...` to a local subtree `projects/ctsrd/...`) makes it easy to check out further subtrees in the future.

Once you have saved the client configuration, you can use `p4 sync` to update your local check-out. For more detailed information on using Perforce, see the Perforce documentation.

2.2 About FreeBSD/BERI

The FreeBSD/BERI port is adapted from the FreeBSD/MALTA 64-bit MIPS port, which offers the closest match in terms of ISA. BERI- and CHERI-related kernel files may be found in directories listed in Table 2.2. Wherever possible, FreeBSD/BERI reuses generic MIPS platform code, and is successful in almost all cases. BERI uses flat device tree (FDT); currently, `DTS` files describing BERI hardware are stored in the FreeBSD source tree, but we hope to embed them in ROMs in BERI and CHERI bitfiles in the future. Table 2.2 lists BERI- and CHERI-specific files in the common MIPS configuration directory.

2.3 Building FreeBSD/BERI

The following sections consider how to build a FreeBSD/BERI system; examples assume the `svr-ctsrd-mipsbuild` environment, but with appropriate pathname and username substitutions, should work on other FreeBSD 9.x build hosts. This requires building a cross-build toolchain, and then using that toolchain to build FreeBSD/BERI userspace and kernel. Once these elements have been built, they must be installed into a directory tree from which disk images can be created. If you wish to build a kernel that includes a memory root file system, userspace must be built and installed, and a memory file system image created, before the kernel can be built. Where appropriate, `cheribsd` may be substituted for `beribsd`, and kernel configuration file names changed, to build CheriBSD instead of FreeBSD/BERI.

Filename	Description
sys/mips/beri/	BERI-specific processor/platform code.
sys/mips/cheri/	CHERI-specific code: coprocessor 2 initialisation and context management.
sys/boot/fdt/dts/	Home of BERI flat device tree (FDT) description files.
sys/dev/isf/	Intel StrataFlash device driver.
sys/dev/altera/atse	Altera Triple-Speed Ethernet MAC.
sys/dev/altera/avgen	Avalon "generic" driver to export I/O address ranges to userspace.
sys/dev/altera/jtag_uart	Altera JTAG UART device driver.
sys/dev/altera/sdcard	Altera University Program SD Card IP core device driver.
sys/dev/terasic/de4led	Terasic DE4 LED array device driver.
sys/dev/terasic/mtl	Terasic Multitouch LCD device driver.

Table 2.1: FreeBSD/BERI and CheriBSD files in src/sys/

Filename	Description
BERI_DE4.hints	Terasic DE4 hardware configuration hints
BERI_SIM.hints	Bluespec simulation hardware configuration hints
BERI_TPAD.hints	Terasic tPad hardware configuration hints
BERI_TEMPLATE	FreeBSD/BERI template configuration entries, included by other kernel configuration files
BERI_SIM_MDROOT	FreeBSD/BERI kernel configuration to use a memory root file system while in simulation
BERI_DE4_MDROOT	FreeBSD/BERI kernel configuration to use a memory root file system on the Terasic DE4
BERI_DE4_SDROOT	FreeBSD/BERI kernel configuration to use an SD Card root file system on the Terasic DE4
CHERI_DE4_MDROOT	CheriBSD kernel configuration to use a memory root file system on the Terasic DE4
CHERI_DE4_SDROOT	CheriBSD kernel configuration to use an SD Card root file system on the Terasic DE4

Table 2.2: FreeBSD/BERI and CheriBSD files in src/sys/mips/conf; note that hints files have been deprecated in favour of FDT DTS files for board configuration.

Note well: The details of the build process are likely to change over time as we merge changes from the upstream FreeBSD tree due to the rapid evolution of MIPS support. Users should take care to ensure that they are using a *CHERI User's Guide* that is contemporary with their source tree.

2.3.1 Configuring the build environment

By default, the FreeBSD build system will use `/usr/obj` as its scratch area. Instead, create and configure your own per-user scratch space:

```
$ mkdir -p /local/scratch/rnw24/obj
$ export MAKEOBJDIRPREFIX=/local/scratch/rnw24/obj
```

You may wish to modify your `.cshrc` or `.bashrc` to automatically configure the `MAKEOBJDIRPREFIX` variable every time you log in.

2.3.2 Building a MIPS64 toolchain

When building FreeBSD world, a MIPS64 cross-toolchain is compiled automatically as a dependency. However, if you wish to build kernels without first building world, or use the toolchain to build other software, you can independently compile a MIPS64 cross-toolchain using:

```
$ cd /local/scratch/rnw24/p4/projects/ctsrd/beribsd/src
$ make -j 8 toolchain TARGET=mips TARGET_ARCH=mips64
```

You can start a shell with the cross-tools, rather than native tools, in the path using:

```
$ cd /local/scratch/rnw24/p4/projects/ctsrd/beribsd/src
$ make buildenv TARGET=mips TARGET_ARCH=mips64
```

Note: building a toolchain is implied by the `buildworld` target; if you are using `buildworld`, you can skip building the `toolchain` target.

2.3.3 Cross-building world

In FreeBSD parlance, "world" refers to all elements of the base operating system other than the kernel – i.e., userspace. This includes system libraries, toolchain including compiler, userland utilities, daemons, and generated configuration files; it excludes third-party software such as Apache, X11, and Chrome. Cross-build a big-endian, 64-bit MIPS world with the following commands:

```
$ cd /local/scratch/rnw24/p4/projects/ctsrd/beribsd/src
$ make -j 8 TARGET=mips TARGET_ARCH=mips64 DEBUG_FLAGS=-g \
  buildworld
```

Note: the DEBUG_FLAGS=-g requests that debugging symbols be generated for all userland components.

Some utility software and software used for demos is stored in the src/ctsrd and src/ctsrd-lib directories. They can built with the world by adding the following to the make command line:

```
LOCAL_DIRS=ctsrd LOCAL_LIB_DIRS=ctsrd-lib \
  LOCAL_MTREE=ctsrd/ctsrd.mtree
```

2.3.4 Cross-building a kernel

FreeBSD kernels are compiled in the context of a configuration file; for BERI, we have provided two reference configuration files as described earlier in this chapter. In this section we consider only how to build a kernel without a memory root file system; information on memory root file systems may be found in Section 2.4.3. The following commands cross-build a FreeBSD/BERI kernel:

```
$ cd /local/scratch/rnw24/p4/projects/ctsrd/beribsd/src
$ make -j 8 TARGET=mips TARGET_ARCH=mips64 buildkernel \
  KERNCONF=BERI_DE4_SDROOT
```

Notice that the kernel configuration used here is BERI_DE4_SDROOT; replace this with other configuration file names as required.

BERI and CHERI use flat device trees (FDT) to describe most aspects of hardware configuration, including bus topology and peripheral attachments. The only significant exception to this is physical memory configuration, which is passed directly to the kernel by the boot loader; we anticipate that this will also be configured using FDT in the future. For the time being, DTS files describing hardware are embedded in the FreeBSD source tree in a manner similar to hints files used in earlier iterations of BERI and CHERI. In the future, these will be embedded in hardware and a pointer to the configuration will be passed to the FreeBSD kernel on boot by the loader.

2.4 Cross-installing FreeBSD/BERI

The install phase of the FreeBSD/BERI build process takes generated userspace and kernel binaries from the MAKEOBJDIRPREFIX and installs them into a directory tree that can then be converted into a file system image. Most make targets in this phase make use of the DESTDIR variable to determine where files should be installed to. Typically, DESTDIR will be set to a dedicated scratch directory such as /local/scratch/rnw24/root. It is advisable to remove the directory between runs to ensure that no artefacts slip from one instance into a later one:

```
# rm -Rf /local/scratch/rnw24/root
# mkdir -p /local/scratch/rnw24/root
```

2.4.1 Cross-installing world

This phase consists of two steps: first, installing "world", which installs libraries, daemons, command line utilities, and so on, and second, "distribution", which creates additional files and directories used in the installed configuration, such as `/etc` and `/var`. The following commands cross-install to `/local/scratch/rnw24/root`

```
# cd /local/scratch/rnw24/p4/projects/ctsrd/beribsd/src
# make DESTDIR=/local/scratch/rnw24/root \
  TARGET=mips TARGET_ARCH=mips64 DB_FROM_SRC=yes -DNO_ROOT \
  installworld
# make DESTDIR=/local/scratch/rnw24/root \
  TARGET=mips TARGET_ARCH=mips64 DB_FROM_SRC=yes -DNO_ROOT \
  distribution
```

To install the software in the `src/ctsrd` and `src/ctsrd-lib` directories, the following should be added to the make command line:

```
LOCAL_DIRS=ctsrd LOCAL_LIB_DIRS=ctsrd-lib \
  LOCAL_MTREE=ctsrd/ctsrd.mtree
```

2.4.2 Cross-installing kernels

As with world, kernels can be installed to a target directory tree along with any associated modules. The following commands install the `BERI_DE4_SDROOT` kernel into `/local/scratch/rnw24/root`:

```
# cd /local/scratch/rnw24/p4/projects/ctsrd/beribsd/src
# make DESTDIR=/local/scratch/rnw24/root \
  TARGET=mips TARGET_ARCH=mips64 DB_FROM_SRC=yes -DNO_ROOT \
  KERNCONF=BERI_DE4_SDROOT installkernel
```

2.4.3 Preparing a memory root file system

In order to build the `BERI_DE4_MDROOT` kernel, a memory file system image is first required. A demonstration script, `makeroot.sh`, is available via the CTSRD Subversion repository; most users will wish to customise the script based on their specific environment. The following command generates an 26 megabyte root file system image in `/local/scratch/rnw24/mdroot.img`, and depends on previous steps to install world having been completed:

```
# cd /local/scratch/rnw24/ctsrd/cheribsd/trunk/bsdtools
# sh makeroot.sh -e extras/sdroot.mtree -s 26112k -f net.files \
   /local/scratch/rnw24/mdroot.img /local/scratch/rnw24/root
```

This script requires a version of the `makefs` command only found in FreeBSD HEAD. On older versions of FreeBSD you can download and build the source from `svn://svn.freebsd.org/base/head/usr.sbin/makefs`.

You must also customise `BERI_DE4_MDROOT` in order to notify it of where to find the memory root file system image, modifying the following section of its configuration file to reflect actual image location and size:

```
options  MD_ROOT # MD is a potential root device
options  MD_ROOT_SIZE=26112
makeoptions MFS_IMAGE=/local/scratch/rnw24/mdroot.img
options  ROOTDEVNAME=\"ufs:md0\"
```

Once the root file system image is generated, and the kernel configuration file generated, you can build the kernel:

```
$ cd /local/scratch/rnw24/p4/projects/ctsrd/beribsd/src
$ make -j 8 TARGET=mips TARGET_ARCH=mips64 buildkernel \
  KERNCONF=BERI_DE4_MDROOT
```

The resulting kernel can be found in your `MAKEOBJDIRPREFIX` tree.

2.5 Preparing a FreeBSD SD Card image

Build and install "world" and "distribution" as described in earlier sections. Then build and install a kernel using the configuration file `BERI_DE4_SDROOT`. Apply a few tweaks to configuration files, then use the `makefs` command to generate a UFS image file:

```
# echo "/dev/altera_sdcard0 / ufs rw 1 1" > \
  /local/scratch/rnw24/root/etc/fstab
# RCCONF=/local/scratch/rnw24/etc/rc.conf
# echo 'hostname="cheri1"' > $RCCONF
# echo 'sendmail_submit_enable="NO"' >> $RCCONF
# echo 'sendmail_outbound_enable="NO"' >> $RCCONF
# echo 'cron_enable="NO"' >> $RCCONF
# echo 'tmpmfs="YES"' >> $RCCONF
# cd /local/scratch/rnw24/root && makefs -B big -s 1886m -d \
  /local/scratch/rnw24/beribsd-sdcard.img METALOG
```

The `makefs` syntax above requires a version of `makefs` only found in FreeBSD HEAD. On older versions of FreeBSD you can download and build the source from `svn://svn.freebsd.org/base/head/usr.sbin/makefs`.

This command creates a big-endian file system of size $1,977,614,336$ bytes – the size of the Terasic 2 GB SD Cards shipped with the tPad board. The resulting `beribsd-rootfs.img` can then be installed on an SD Card using `dd` as described in later sections.

If a smaller file system is desired (e.g. quicker to prepare and write to the sdcard) then size $1,977,614,336$ bytes (2GiB) can be replaced by $512m$.

2.6 Automated builds

The set of files described in the Using CheriBSD Chapter can be built with the help of the `Makefile` in `ctsrd/cheribsd/trunk/bsdtools/`. The template config file in `Makefile.conf.template` can be copied to `Makefile.conf` and customized to your enviornment. The `worlds` target runs the `buildworld`, `installworld`, and `distribution` for each source tree. The `images` target builds file system images from the installed root directories. The `kernels` target builds kernels including MDROOT kernels with images build by `images` target. The `flash` target build flash preparation images for each kernel. Finally, the `dated` target makes dated stamped files of each compressed kernel and image.

All of these targets support the `make` flag `-j`. Passing an appropriate value to `-j` is strongly encourage

In addition to standard FreeBSD tools, the Makefile requires the `archives/pxz` port to be installed.

Chapter 3

Using CheriBSD

This chapter is concerned with installing and using FreeBSD/BERI and CheriBSD on the Terasic DE-4 FPGA board. We have structured our modifications to FreeBSD into two development branches:

- FreeBSD/BERI is a version of FreeBSD that can run on the BERI hardware-software research platform as a general-purpose OS;
- CheriBSD is a version of FreeBSD/BERI that has been enhanced to make use of CHERI's experimental capability coprocessor features.

At the time of writing, FreeBSD has been modified to support a number of BERI features, such as peripheral devices present on the Terasic DE4 board; CheriBSD extends FreeBSD/BERI to initialise and maintain CHERI coprocessor registers.

3.1 Getting started with CheriBSD

To get started with FreeBSD/BERI or CheriBSD, you need the following:

- Ubuntu development PC
- Altera Quartus 12 tools
- Terasic DE4 board with 1GB DRAM
- 2GB SD card[1]
- CHERI bitfile targeted for the Terasic DE4
- Pre-built or custom-compiled FreeBSD kernel
- Pre-built or custom-compiled FreeBSD root file system image

[1]Note: Altera's SD Card IP Core does not support SD cards larger than 2GB.

It is important that synchronised versions of CHERI bitfiles and FreeBSD be used together: as the prototype evolves, hardware-software interfaces change, as do board configurations, and mismatched combinations will almost certainly not function correctly. It is also important to ensure that the installed Quartus toolchain matches that used to generate the bitfile being programmed, in order to avoid documented incompatibility.

The remainder of the chapter describes how to obtain FreeBSD kernels and root file system images, write the FreeBSD root file system image onto the SD card, program the Terasic DE4 FPGA with a CHERI bitfile, set up the JTAG debugging tunnel so that the `cherictl` tool can manipulate CHERI via its debug unit, connect to the CHERI console using `nios2-terminal` over JTAG, and optionally re-flash the DE4's on-board CFI flash with a bitfile and kernel to avoid needing to program them via JTAG on every boot.

3.1.1 Obtaining FreeBSD/BERI and CheriBSD

FreeBSD/BERI and CheriBSD may be built from source code using the instructions found in Chapter 2. Alternatively, pre-compiled binary distributions of key files and images may be found, for the time being, in `/usr/groups/ctsrd/cheri` in the Cambridge environment. Each file is named based on the date is was generated, consisting of `YYYYDDMMv` where `v` is an optional letter indicating further builds that occurred on the same day. Table 3.1 describes file types that may be found in the directory; all bitfiles and kernels are compressed using `bzip2` and all file system images are compressed with xz. File systems images must be decompressed before they can be used. Files passed to cherictl may be uncompressed or the `-z` flag may be used to decompress them on the fly.

3.1.2 Writing out the SD Card disk image

To use FreeBSD/BERI with an SD Card root file system, write out the file system image on an existing Mac, FreeBSD, Linux, or Windows workstation. The following command is typical of how this might be done on a UNIX system; ensure that the disk device name here is actually your SD Card and not another drive!

```
$ dd if=beribsd-sdcard.img of=/dev/disk1 bs=512
```

On Mac OS X, `diskutil list` may be used to list possible devices to write to; `DiskUtility.app` allows unmounting a mounted file system, if an existing FAT file system on the SD card mounted when it was inserted. On a Mac this command should be runnable as an ordinary user; if elevated privilege appears to be required then you a probably attempting to write to the wrong device. Note that SD cards should not be initialised with FAT or other file systems; disk images include a complete UFS file system intended to be written directly to the SD card starting at the first block.

3.1.3 Setting up the DE-4 development environment

Many commands in the chapter depend on Altera Quartus 12 tools. Specifically the `nios2-terminal` must be in the user's `PATH` and the `system-console` command must be available.

Filename	Description
`beribsd-de4-kernel-demo-mdroot`	DE4 kernel with built-in memory root file system of CTSRD demos
`beribsd-de4-kernel-net-mdroot`	DE4 kernel with built-in memory root file system with basic network tools
`beribsd-de4-kernel-sdroot`	DE4 kernel using an SD Card as a root file system
`beribsd-sim-kernel-mdroot`	Simulation kernel with build-in memory root file system
`beribsd-sdcard.img`	SD Card root file system image
`cheribsd-de4-kernel-demo-mdroot`	DE4 kernel with built-in memory root file system of CTSRD demos
`cheribsd-de4-kernel-net-mdroot`	DE4 kernel with built-in memory root file system with basic network tools
`cheribsd-de4-kernel-sdroot`	DE4 kernel using an SD Card as a root file system – includes capability support
`cheribsd-sim-kernel-mdroot`	Simulation kernel with build-in memory root file system – includes capability support
`cheribsd-sdcard.img`	SD Card root file system image
`cheri-bitfile.sof`	Altera SOF bitfile for CHERI processor
`cfi0-de4-terasic`	Vanilla CFI flash `cfi0` image for the DE4

Table 3.1: Binary files available for FreeBSD/BERI. `-dump` files will sometimes also be present, which contain `objdump -dS` output for kernels and other binaries.

In the Cambridge environment, this can be accomplished by configuring the CHERI build environment:

```
$ source ctsrd/cheri/trunk/setup.sh
```

A default user install of the Quartus 12 toolkit will also accomplish this so long as the `/bin` directory is in the user's path.

The `cherictl` command is used to control various aspects of CPU and board behaviour. For example, it can be used to inspect register state, modify control flow, and load data into memory. `cherictl` works with CHERI in both simulation and in FPGA. Build `cherictl` using the following commands:

```
$ cd ctsrd/cherilibs/trunk/tools/debug
$ make
```

For users without access to the Subversion repositry, staticaly linked versions of `cherictl` are distributed along with each CHERI/CheriBSD release.

3.1.4 JTAG

Many hardware debugging functions rely on JTAG, which allows a host Linux workstation to program the FPGA board, read and write DRAM on the board, and also interact with the CHERI debug unit for the purposes of low-level system software debugging. Using JTAG requires that a USB cable be connected from your Linux workstation to the Terasic DE4 board. In the remaining sections, JTAG will be used to access four different debugging funtions:

- programming the FPGA (via `cherictl loadsof`);
- BERI's JTAG UART console (via `nios2-terminal`);
- direct DRAM manipulation (via `cherictl loaddram`);
- and to use CHERI's debug unit (most other `cherictl` commands).

3.1.5 cherictl

`cherictl` is a front-end to a variety of development and debugging features associated with the CHERI processor, both in simulation and when synthesised to FPGA. `cherictl` will generally communicate with the CHERI debug unit over JTAG or a socket.

The `-p` argument may be used to specify a UNIX domain socket by pathname, or a TCP port number. The former configuration is used exclusively to specify the communications channel to a simulation; the latter is used to help `cherictl` find a running instance of Altera's `system-console` for the purposes of direct DRAM programming over JTAG.

3.1.6 system-console tunnel

The `system-console` program is used to program the FPGA. Additionally BERI and CHERI processors are configured so that attached DRAM can be read and written directly via JTAG though the `system-console` program. `cherictl` is able to use this feature to quickly load kernels or other data into DRAM using its `loaddram` command. To start `system-console`, run:

```
$ system-console --server
TCP PORT: 51971
System Console server started on TCP port 51971
```

As shown, `system-console` will print a TCP port number that may be passed to to `cherictl`'s `-p` argument when using `loadsof` and `loaddram`. It is command practice to store the port in an environmental variable so other commands are consistent:

```
$ export SYSTEM_CONSOLE_PORT=51971
```

3.2 Programming the DE4 FPGA

FPGA designs are encapsulated in a *bitfile*, which can be programmed dynamically using JTAG, or from the on-board CFI flash on the DE4 when the board is powered on. The former configuration will be used most frequently when developing processor or other hardware features; the latter will be used most frequently when developing software to run on CHERI, as it effectively treats the board as a stand-alone computer whose firmware (and hence CPU!) may occasionally be upgraded.

3.2.1 Programming the DE4 via JTAG

To program the DE4's FPGA, start a `system-console` instance as described above and then run the following command:

```
$ cd ctsrd/cherilibs/trunk/tools/debug
$ ./cherictl -p $SYSTEM_CONSOLE_PORT -z -f cheri-bitfile.sof.bz2 loadsof
```

Note well: you must terminate all nios2-terminal sessions connected to the DE4 before using cherictl's loadsof command. If you do not the board may not be reprogrammed or `system-console` *may crash.*

3.2.2 Writing an FPGA bitfile to DE4 flash from CheriBSD

When powered on, the Terasic DE-4 board will attempt to automatically load a bitfile from the on-board CFI flash. New FPGA bitfiles may be written to the flash from CheriBSD; they take effect when the board is next power-cycled. This can be done using `dd`:

```
# dd if=Design.bin of=/dev/map/fpga iseek=256 conv=oseek
```

To simplify the process and add reliability, a script called `/usr/sbin/flashit` performs these actions after verifying the MD5 checksum of the files and optionally decompressing bzip2 compressed images. Note that if `flashit` is writing a file `foo` a corresponding `foo.md5` file must exist. In addition to FPGA images, flashit can be used to write kernel images by replacing the `fpga` argument with `kernel`.

```
# flashit fpga Design.bin
```

It is important that power to the board not be lost during reflash, as this may corrupt the bitfile, preventing programming of the board on power-on. It is therefore recommended that battery-backed DE-4 boards only be programmed while fully charged.

3.3 Start a NIOS2 terminal

Connect to the CHERI JTAG UART, used as the FreeBSD/BERI console, using Altera's `nios2-terminal` tool:

```
$ nios2-terminal --instance 1
```

Somewhat irritatingly, `nios2-terminal` is terminated by when it receives a SIGINT signal (usually generated by Ctrl-C), meaning that it cannot be passed through to the JTAG UART console. You can work around this by changing the combination used either on the client or on the FreeBSD/BERI console. For example, the following command remaps Ctrl-O to generate SIGINT in place of Ctrl-C:

```
$ stty intr ^o
```

We may fix this in the future by extending the debug unit bridge to also bridge the JTAG UART port to a UNIX domain socket, which could then be connected to using `cherictl console`, as is the case in simulation.

3.4 Models for booting a FreeBSD/BERI kernel

FreeBSD kernels may be booted via two different means: installation on the on-board CFI flash device on the Terasic DE4, or direct insertion of the kernel into DRAM using `cherictl` via JTAG. The micro-boot loader embedded in ROM in CHERI on the DE4, `miniboot`, uses the USER_DIP1 switch to control whether a kernel is relocated from flash, or started directly. Note that DIP switches are numbered 0-7, but the physical package has labels 1-8. The proper labels can be seen in Figure 3.1.

USER_DIP0 controls whether `miniboot` runs immediately or whether it spins awaiting register 13 becoming 0 before booting the kernel, leaving time for the kernel to be injected into DRAM following programming of the FPGA. Register 13 would normally be set to 0 using the debug unit.

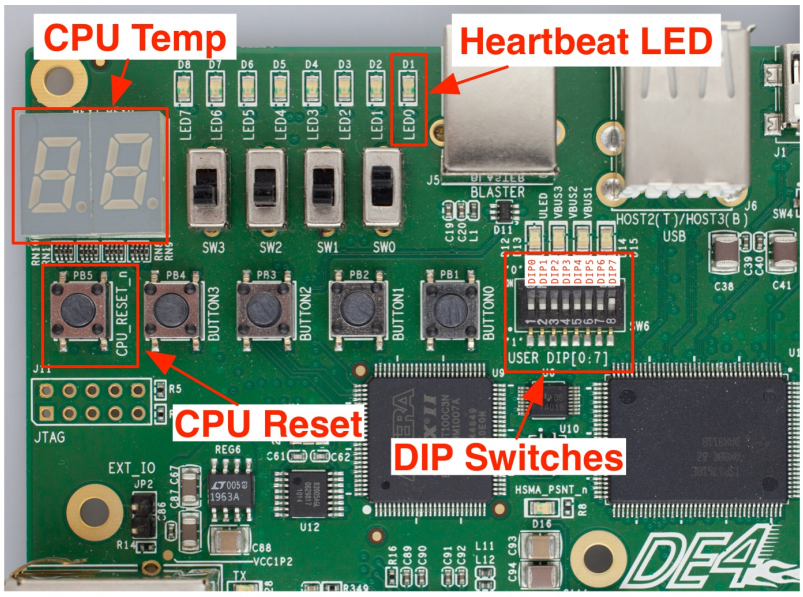

Figure 3.1: Buttons and switches on the DE4

3.4.1 Load a kernel into DRAM over JTAG

To load the kernel into DDR2 memory, it must be loaded at the physical address `0x100000` where miniboot boot loader expects to find. `miniboot` reads the ELF header in order to determine the kernel start address. You can then use `cherictl loaddram` to load the kernel to DDR2 memory starting at address `0x100000` (also see note below about USER_DIP0 and USER_DIP1):

```
$ cd ctsrd/cherilibs/trunk/tools/debug
$ ./cherictl -p ${SYSTEM_CONSOLE_PORT} \
     -z -f cheribsd-de4-kernel-sdroot.bz2 -a 0x100000 loaddram
```

To boot a kernel thus loaded you must ensure that both USER_DIP0 and USER_DIP1 are on (toward the top of the board where the USB blaster is connected). USER_DIP0 will cause the processor to spin in very early boot, waiting for register 13 to become 0. This can be accomplished through the debug command:

```
$ ./cherictl boot
```

USER_DIP1 will skip the relocation from flash routine that would over-write your freshly inserted kernel.

3.4.2 Load a kernel into flash from CheriBSD

From CheriBSD you can use `dd` or the `flashit` script to load a kernel to flash:

```
# dd if=kernel of=/dev/map/kernel conv=osync
```

The safer option using `flashit` allows the kernel to be compressed with `bzip2` or `gzip` and requires a `.md5` file to exist containing the `md5` output for the file:

```
# ls kernel.bz2*
kernel.bz2  kernel.bz2.md5
# flashit kernel kernel.bz2
```

At boot a kernel written to flash will be relocated to DRAM and executed if USER_DIP1 is set to off. This relocation will occur at power on if USER_DIP0 is off or when `cherictl boot` is run if it is on.

3.5 Start kernel execution

If USER_DIP0 is set to on, then resume the processor after power on/reset:

```
$ ./cherictl boot
```

If the DIP switch is unset, then boot will proceed as soon as the FPGA is programmed, either using JTAG or from flash. If all has gone well, you should see kernel boot messages in output from `nios2-terminal`. If you are using the `BERI_DE4_MDROOT` or `CHERI_DE4_MDROOT` kernel configuration, a memory root file system will be used and single or multi-user mode should be reached depending on the image. If you are using the `BERI_DE4_SDROOT` or `CHERI_DE4_SDROOT` kernel configuration, the SD Card should be used for the root file system, and multi-user mode should be reached. Be warned that the SD-card IP core provided by Altera is extremely slow (100KB/s), and so multi-user boots can take several minutes.

3.6 Post boot issues

After boot CheriBSD is much like any FreeBSD system with a similar set of componants. There are a few issues to keep in mind

- The MDROOT kernels are space limited and have minimal set of tools available.
- Since the root file systems of MDROOT kernels are stored in memory, all configuration including ssh keys will be lost at reboot time.
- The SD Card images are kept relatively small to speed writes to the (often very slow) cards and to allow many card sizes to be supported.
- The Ethernet controllers have no default source of unique MAC addresses and thus default to a random address that changes each boot.
- The MIPS soft reset in instruction is not currently implemented so rebooting consists of shutting down cleanly and then booting as per the instuctions above. The easiest software reboot mechanism is reprograming the FPGA.

3.6.1 Increasing the size of an SD Card root file system

After boot, you can extend the file system to the size of the SD Card using FreeBSD's `growfs` command:

```
$ growfs -y /dev/altera_sdcard0
```

Before running this command, make sure your file system is backed up or easily replacable.

3.6.2 Setting a MAC address

The Altera Triple-Speed Ethernet (ATSE) devices obtain a unique MAC address from the configuration area at the beginning of the CFI flash. Unfortunatly, all DE4 boards come from the factory with the same MAC address so it has been blacklisted by the driver and a random address is generated at boot for each interface.

An an address can be written to the DE4 using the `atsectl` command. An address derived from the factory PPR on the Intel StrataFlash on the DE4 can be written with the command:

```
$ atsectl -u
```

The default address has the locally administered bit set a and uses the Altera prefix dedicated to this purpose.

In the Cambridge environment, the decision was made to not set the locally administered bit. This can be accomplished with the command:

```
$ atsectl -gu
```

If the board was configured following the *DE4 Factory Install Guide v1.0* then an Altera prefixed MAC without the locally administred bit will have been installed on the DE4.

Chapter 4

CHERI Clang/LLVM

This chapter describes CHERI-specific modifications to the Clang/LLVM compiler suite and the GNU assembler, as well as our extensions to the C programming language to support explicit capability use.

4.1 Cross-Compiling for CHERI

For cross-compiling code that targets CHERI, we provide a modified LLVM back end and Clang front end for [Objective-]C[++]. The back end can generate CHERI assembly and object code from LLVM's intermediate representation (IR). The front end generates the IR from C family languages and supports some capability extensions to C.

For assembly language programming, we also provide a modified version of the GNU binutils, including the GNU assembler (gas), that has support for the capability instructions.

4.2 Building the Assembler

To build the assembler, you will need Git installed. Check out the source code and build it like this:

```
$ git clone git://github.com/CTSRD-CHERI/binutils.git
$ cd binutils
$ ./configure --target=mips64 --disable-werror
$ make
```

4.3 Building the Compiler

To build these, you will need Git, CMake, and Ninja installed. Check out the code and build like this:

```
$ git clone git://github.com/CTSRD-CHERI/llvm.git
$ cd llvm/tools
$ git clone git://github.com/CTSRD-CHERI/clang.git
$ cd ..
$ mkdir Build
$ cd Build
$ cmake -G Ninja -DCMAKE_BUILD_TYPE:STRING=Debug \
    -DBUILD_SHARED_LIBS:BOOL=ON ..
$ ninja
```

By default, Ninja will use 10 processes in parallel on the build. You can increase or decrease this with the -j flag. If you are on a 32-bit system, try you may want to pass `-DLLVM_ENABLE_ASSERTIONS:BOOL:` to CMake to build a release build with asserts, rather than a debug build. Linking a debug build of LLVM can run out of address space in the linker in a 32-bit system.

4.4 Using Clang

Once you have built LLVM, you (mostly) have a working cross compiler. Currently, the MIPS direct code generation part is not well tested, so you probably need to use the modified GNU assembler. You can generate CHERI assembly from [Objective-]C[++] source code with this command:

```
$ clang -S {source file} -target cheri-unknown-freebsd \
    -msoft-float
```

You can try generating native code directly, like this:

```
$ clang -c {source file} -target cheri-unknown-freebsd
    -msoft-float -integrated-as
```

The first flag specifies the target, in this case CHERI as the architecture and FreeBSD as the platform. The current version of CHERI has no FPU, so we want to emit calls to emulated FPU functions, rather than trap-and-emulate in the kernel. The final option, `-integrated-as` instructs LLVM to emit object code directly, rather than passing assembly to the GNU assembler.

There are two other arguments that you may need. `--sysroot` allows you to specify the location of a set of headers and libraries for the target. `-B` allows you to specify the location of the CHERI version of the assembler and linker.

If you do a debug build of LLVM, then clang will default to using the simple register allocator. To see significantly better code, add the following CFLAGS:

```
-mllvm -regalloc=greedy -O3 -mllvm -enable-mips-delay-filler
```

This will enable a better register allocator and will attempt to replace the nops in delay slots with instructions from before the branch. It also turns on the full set of LLVM optimisations. Note that not all of these are well tested with CHERI and so a lower optimisation level may be required to generate correct code.

4.5 Disassembling CHERI Binaries

It's common, when debugging, to want to disassemble some instructions. You can do this for individual instructions with the `llvm-mc` tool:

```
$ echo 0x48 0x02 0x08 0x02 | llvm-mc -disassemble - \
    -triple=cheri-unknown-freebsd
    .section TEXT,__text,regular,pure_instructions
    CGetType $2, $c1
```

This expects a string of hex bytes and will write out the corresponding assembly. To disassemble entire object code files, use the llvm-objdump tool:

```
$ llvm-objdump -disassemble -triple=cheri-unknown-freebsd \
    {something.o}
```

4.6 Capability Extensions to C

Clang predefines a `__capability` macro. If you want to make your code portable to non-CHERI platforms, then you can begin it with:

```
1 #if !defined(__CHERI__) && !defined(__capability)
2 #   define __capability
3 #endif
```

You can then use the `__capability` qualifier on any pointer. It will be treated as a capability with the following semantics:

- Casts to integers return the base address.
- Casts from integers are treated as C0-relative[1].
- Relative addressing will use the load/store via capability instructions
- Pointer increments will return a new capability, which may be invalid if it exceeds the upper bound of the original.
- Pointer decrements will always fail.

Clang also provides a number of built-in functions for accessing aspects of these:

[1]This means that a `__capability void*` to `uintptr_t` to `__capability void*` round trip will only work if the capability references something inside the C0 address space

```
size_t __builtin_cheri_get_cap_length(__capability void*)
size_t __builtin_cheri_get_cap_perms(__capability void*)
size_t __builtin_cheri_get_cap_type(__capability void*)
__capability void* __builtin_cheri_set_cap_length(__capability void*,
    size_t);
__capability void* __builtin_cheri_and_cap_perms(__capability void*,
    size_t);
__capability void* __builtin_cheri_set_cap_type(__capability void*,
    size_t);
_Bool __builtin_cheri_get_cap_tag(__capability void*);
__capability void*__builtin_cheri_get_cap_register(int);
void __builtin_cheri_set_cap_register(int, __capability void*);
```

Most of these correspond directly to the relevant capability inspection / modification instructions. The last two can be used for getting and setting the values of registers that are not managed by the compiler[2]. For example, you may set the value of C0 from C code like the following function. This will load the current C0 value, then set the new one, and return the old one. Note that at low optimisation levels, this will store the old value on the stack. In the current implementation, the stack is accessed via C0 (in the future this will move to a dedicated stack capability) and so this would not work if the new C0 does not have the same base as the old C0 and a length that encompasses the stack.

```
__capability void *setC0(__capability void *newC0)
{
    __capability void *oldC0 = __builtin_cheri_get_cap_register(0);
    __builtin_cheri_set_cap_register(0, newC0);
    return oldC0;
}
```

At a higher optimisation level, however, this is fine. When compiled at -O3, the following code is generated:

```
    cincbase    $c2, $c0, $zero
    cincbase    $c0, $c1, $zero
    cincbase    $c1, $c2, $zero
    jr  $ra
    nop
```

This is not quite as efficient as a hand-written assembly routine, such as the following:

```
    cincbase    $c0, $c1, $zero
    jr  $ra
    cincbase    $c1, $c0, 0
```

This is because the compiler currently treats explicit modification of capability registers as a barrier and so will not attempt reordering of these instructions.

[2]They will also work with registers that are managed by the compiler, but the contents of those registers is not necessarily predictable.

4.6.1 Const and Capabilities

In C/C++, you can explicitly cast a `const` pointer to a non-`const` pointer. This is not supported for `__capability` pointers. For example, consider the following two `__capability` pointers:

```
__capability void *a = something;
const __capability void *b;
```

This will implicitly clear the store and store-capability flags:

```
b = a;
```

So far, this is almost exactly the C behaviour, except that now we will get a hardware trap instead of just a compiler warning if you try to store through `b`. If you perform the assignment in the other direction, however, the constness is still preserved:

```
a = (__capability void)b;
```

Any attempts to store through a will cause a trap.

4.6.2 Output-only Capabilities

Clang also now supports an `__output` qualifier on capabilities. These are the twin of `const` capabilities: they can only be written to, not read. You can create a write-only capability by casting any other capability. The compiler will statically check attempts to write through an `__output` capability, and code that does so via an explicit cast will abort at run time.

```
int readFail(__capability __output int *x)
{
    *x = 12;
    (*x)++;
    return *x;
}
```

This contains two bugs: two attempts to write through the pointer passed as a parameter, even though it is declared as being solely for output. Attempting to compile this will raise the following warnings:

```
writetest.c:4:2: error: write-only variable is not readable
        (*x)++;
        ^~~~
writetest.c:5:9: error: write-only variable is not readable
        return *x;
               ^~
```

You can call this function transparently, without an explicit cast:

```
int caller(__capability int *x)
{
    return readFail(x);
}
```

In this case, the caller will automatically modify the capability to remove load and load-capability permissions before passing it.

4.6.3 Capability Implicit Range Checking

At optimisation level 1 and above, LLVM will attempt to automatically limit the range of capabilities. If static code flow analysis can determine that the capability was constructed from a global or a stack allocation, it will initially begin with its range limited to the size of that allocation. This will also work for various heap allocation functions. Any allocations that are the direct result of casting from the result of one of the standard C allocation functions. Note that this analysis is *not* interprocedural and so will not work in the general case. It is primarily intended as a way of providing a small amount of extra checking.

4.6.4 Opaque Types

CHERI-clang supports a new pragma, opaque, for linking a type to a key. This is intended to provide enforcement of opaque types in C: no code outside of a compilation unit where the key is visible is able to dereference the opaque pointers, even after casting. This is intended to be used as follows:

```
1     // In a public header (one or more of the following):
2     typedef __capability struct foo* foo_c;
3     typedef struct foo* foo_t;
4
5     // In either the implementation file or a private header:
6
7     // If we're in private header, these should be __attribute__
8     // ((visibility("hidden"))), otherwise they should be static:
9     void *ptrKey;
10    __capability void *capKey;
11
12    struct foo
13    {
14        int a, b;
15    };
16
17    #pragma opaque foo_c capKey
18    #pragma opaque foo_t ptrKey
```

In an implementation file, any function returning a `foo_t` will have its value xor'd with the value of `ptrKey` before return and any function returning a `foo_c` will have the value sealed with `capKey` before return. Similarly, any function receiving a `foo_c` or `foo_t` as an argument will have the inverse applied. Within the function, `foo_t` and `foo_c` behave exactly as they would without the pragma.

4.6.5 Stack Spills and Safety

Functions declared with the `sensitive` attribute are assume to deal with some sensitive data. C already provides tools (**`volatile`** in older versions and explicit memory operations in C11) for ensuring that sensitive data is not left in areas of memory that the programmer manages. It does not provide a mechanism for making the same guarantee for stack spills.

The traditional way of avoiding this problem is to use **`volatile`** variables everywhere and explicitly zeroing them. This, however, has the effect of effectively disabling all compiler optimisation for a function, purely to avoid the possibility of a stack spill. In general, however, higher optimisation levels are more likely to reduce the need for registers to ever be spilled to the stack. It is very hard to reason about the correctness of such code in the presence of optimisations such as common subexpression elimination. These may result in intermediary results being accidentally stored on the stack for future reuse and then persisting beyond the end of the function.

Marking a function as sensitive does not disable *any* optimisations. It does, however, ensure that any values spilled to the stack during the function are zeroed at the end. This is especially important for CHERI systems, where stack spills include capabilities. The pass does not attempt to destroy the entire capability, it simply writes a 64-bit zero value over the start. This is enough to destroy the base address (without which the rest of the information in the capability is largely useless) and, more importantly, invalidate the tag. With the tag invalidated, the capability can not be used as a capability.

Note that the current implementation may still spill values to the stack if they are left in a callee-save register across function calls and the callee spills them. The compiler will warn if you call a function that is not marked as sensitive from one that is, to allow programmers to avoid this. It would be possible to store all callee-save registers to the stack and invalidate them before every call, but this would impose a significant overhead. Instead, we aim to provide tools that allow programmers to write secure code and to decide when these trade-offs are appropriate.

In typical use, the overhead of this attribute is a few instructions in the function epilog (or epilogs, if it has multiple return paths). This, of course, depends on the complexity of the function and the size of its working set. It does, in general, improve significantly at higher optimisation levels, where the register allocator eliminates most, if not all, stack spills. Spill slots are often reused, so the number of invalidations required is often lower than the total number of spills over the program.

We do not invalidate the spills in the function prolog, because these are unlikely to contain any sensitive data. They contain the initial stack pointer, the globals pointer and the return address. Once we begin using capabilities for these, it will probably be worth invalidating them.

Chapter 5

CheriBSD device driver reference

This chapter provides reference information for BERI- and CHERI-specific device drivers and features. Most device drivers are also documented in man pages in the FreeBSD/BERI distribution.

5.1 Device drivers for Altera IP cores

FreeBSD/BERI provides device drivers for a number of useful IP cores and peripheral devices on the Terasic tPad and Terasic DE4 teaching boards. The drivers are statically linked into the reference BERI kernels, and because the Avalon bus does not support auto-configuration and device enumeration, are also statically configured into the kernel using FDT `DTS` files. `device.hints` may also be used to configure these devices, and examples of each are included in this chapter.

5.1.1 Altera JTAG UART

This device driver implements FreeBSD low-level console and TTY interfaces for the Altera JTAG UART, which can be connected to using `nios2-terminal` in order to provide a system console for FreeBSD/BERI. The driver assumes that the low-level console JTAG UART will always be configured at a fixed physical address, and so cannot be configured using FDT or `device.hints` files. However, high-level console support is entirely configurable. The device name for the first Altera JTAG UART is `/dev/ttyu0`.

Note: the Altera JTAG UART is not the same IP core as the Altera UART intended to be used with the RS232 serial port on the Terasic DE4, which is currently unsupported in FreeBSD/BERI, and not configured in our reference DE4 design.

Kernel Configuration

```
device          altera_jtag_uart
```

FDT

```
serial@7f000000 {
```

```
        compatible = "altera,jtag_uart-11_0";
        reg = <0x7f000000 0x40>;
        interrupts = <0>;
};

serial@7f001000 {
        compatible = "altera,jtag_uart-11_0";
        reg = <0x7f001000 0x40>;
};

serial@7f002000 {
        compatible = "altera,jtag_uart-11_0";
        reg = <0x7f002000 0x40>;
};
```

device.hints

```
#
# Altera JTAG UARTs configured for console, debugging, and
# data putput on the DE4.
#
hint.altera_jtag_uart.0.at="nexus0"
hint.altera_jtag_uart.0.maddr=0x7f000000
hint.altera_jtag_uart.0.msize=0x40
hint.altera_jtag_uart.0.irq=0

hint.altera_jtag_uart.1.at="nexus0"
hint.altera_jtag_uart.1.maddr=0x7f001000
hint.altera_jtag_uart.1.msize=0x40

hint.altera_jtag_uart.2.at="nexus0"
hint.altera_jtag_uart.2.maddr=0x7f002000
hint.altera_jtag_uart.2.msize=0x40
```

5.1.2 Generic Avalon device driver

This device driver exports a region of physical memory, typically representing a memory-mapped device on the Avalon bus, to userspace processes via a device node. User processes can perform I/O on the device using the POSIX `read` and `write` system call APIs, but can also map the device into virtual memory using the `mmap` API. Device instances are configured using `BERI.hints` entries, which specify the base, length, and mapping properties of the memory region, as well as any I/O alignment requirements and restrictions (e.g., a read-only, 32-bit access only).

The following example instantiates a `berirom` device node representing physical memory starting at `0x7f00a000` and continuing for 20 bytes on the Avalon bus. I/O must be performed

with 32-bit alignment; data may be both read and written using the POSIX file APIs, and may not be memory-mapped.

Currently, the `avgen` device does not support directing interrupts to userspace components, but we hope to add this in the future, in which case it will likely be exposed using the `kqueue` and `poll` APIs.

Kernel Configuration

```
device          altera_avgen
```

FDT

```
avgen@0x7f00a000 {
        compatible = "sri-cambridge,avgen";
        reg = <0x7f00a000 0x14>;
        sri-cambridge,width = <4>;
        sri-cambridge,fileio = "rw";
        sri-cambridge,devname = "berirom";
};
```

device.hints

```
hint.altera_avgen.0.at="nexus0"
hint.altera_avgen.0.maddr=0x7f00a000
hint.altera_avgen.0.msize=20
hint.altera_avgen.0.width=4
hint.altera_avgen.0.fileio="rw"
hint.altera_avgen.0.devname="berirom"
```

5.1.3 Altera University Program SD Card IP core

This device driver implements FreeBSD block storage device interfaces for the Altera University Program SD Card IP core. Currently, the driver supports only CSD structure 0 SD Cards, limited to 2 GB in size. This limitation may also apply to the Altera SD Card IP Core. SD Card devices appear in `/dev` as they are inserted, and will typically be named `/dev/altera_sdcard0`; FreeBSD's GEOM framework will automatically discover partitions on the disk, causing additional device nodes for those partitions to appear as well – for example, `/dev/altera_sdcard0s1` for the first MBR partition.

Kernel Configuration

```
device          altera_sdcard
```

FDT

```
sdcard@7f008000 {
        compatible = "altera,sdcard_11_2011";
        reg = <0x7f008000 0x400>;
};
```

device.hints

```
hint.altera_sdcardc.0.at="nexus0"
hint.altera_sdcardc.0.maddr=0x7f008000
hint.altera_sdcardc.0.msize=0x400
```

5.1.4 Altera Triple-Speed Ethernet

The `atse(4)` driver implements a FreeBSD Ethernet interface for the Altera Triple-Speed Ethernet IP core. The driver currently supports *only gigiabit Ethernet* (no 10/100). Interfaces must be configured in polling mode. In FreeBSD this may be accomplished with a line like this in `/etc/rc.conf` (this example also causes the interface to be configured using DHCP):

```
ifconfig_atse0="polling DHCP"
```

`atse(4)` devices are be discovered by FDT, but `device.hints` entries are currently required to properly configure the PHYs for each device.

Kernel Configuration

```
device          altera_atse

device          ether
device          miibus
options         DEVICE_POLLING
```

FDT

```
ethernet@7f007000 {
        compatible = "altera,atse";
        /* MAC, RX+RXC, TX+TXC. */
        reg = <0x7f007000 0x400
               0x7f007500 0x8
               0x7f007520 0x20
               0x7f007400 0x8
               0x7f007420 0x20>;
        /* RX, TX */
        interrupts = <1 2>;
};
```

```
ethernet@7f005000 {
        compatible = "altera,atse";
        /* MAC, RX+RXC, TX+TXC. */
        reg = <0x7f005000 0x400
               0x7f005500 0x8
               0x7f005520 0x20
               0x7f005400 0x8
               0x7f005420 0x20>;
};
```

device.hints

```
#
# Altera Triple-Speed Ethernet Mac, present in tPad and DE-4
# configurations
#
hint.atse.0.at="nexus0"
hint.atse.0.maddr=0x7f007000
hint.atse.0.msize=0x400
hint.atse.0.tx_maddr=0x7f007400
hint.atse.0.tx_msize=0x8
hint.atse.0.txc_maddr=0x7f007420
hint.atse.0.txc_msize=0x20
hint.atse.0.tx_irq=2
hint.atse.0.rx_maddr=0x7f007500
hint.atse.0.rx_msize=0x8
hint.atse.0.rxc_maddr=0x7f007520
hint.atse.0.rxc_msize=0x20
hint.atse.0.rx_irq=1

#
# Altera Triple-Speed Ethernet Mac, present in tPad and DE-4
# configurations
# configured from fdt(4) but PHYs are still described in here.
# Currently configured for individual tse_mac cores.
#
hint.e1000phy.0.at="miibus0"
hint.e1000phy.0.phyno=0
hint.e1000phy.1.at="miibus0"
hint.e1000phy.1.phyno=0
hint.e1000phy.2.at="miibus0"
hint.e1000phy.2.phyno=0
hint.e1000phy.3.at="miibus0"
hint.e1000phy.3.phyno=0
```

5.2 Device drivers for Terasic components

FreeBSD/BERI provides device drivers for several devices on the Terasic DE4 and Terasic tPad teaching boards. As with Altera device drivers, Terasic device drivers are statically linked into the FreeBSD kernel, and configured using `BERI.hints`.

5.2.1 Terasic DE4 8-element LED

This device driver implements FreeBSD LED interfaces for the Terasic DE4 8-element LED. These may be written to as described in the FreeBSD led(4) man page; device nodes appear in `/dev/led` with names using the scheme `de4led_0`, `de4led_1`, and so on.

The following command at the FreeBSD/BERI shell will cause DE4 LED 1 to blink roughly once a second:

```
$ echo f9 > /dev/led/de4led_1
```

Kernel Configuration

```
device          terasic_de4led
```

FDT

```
led@7f006000 {
        compatible = "sri-cambridge,de4led";
        reg = <0x7f006000 0x1>;
};
```

device.hints

```
hint.terasic_de4led.0.at="nexus0"
hint.terasic_de4led.0.maddr=0x7f006000
hint.terasic_de4led.0.msize=1
```

5.2.2 Common Flash memory Interface

FreeBSD/BERI includes an improved version of the `cfi(4)` driver supporting Intel, Sharp, and AMD NOR flash supporting the *Common Flash memory Interface* (CFI) standard. The `cfi(4)` is used to support to the Intel StrataFlash found on the DE4. The flash is both configured and partitioned using FDT via the `geom_flashmap(4)` driver. Partitions are accessable at `/dev/<device>s.<label>` for example `/dev/cfid0s.os`.

Kernel Configuration

```
device          cfi
device          cfid
options         CFI_SUPPORT_STRATAFLASH
```

FDT

```
flash@74000000 {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "cfi-flash";
        reg = <0x74000000 0x4000000>;

        /* Board configuration */
        partition@0 {
                reg = <0x0 0x20000>;
                label = "config";
        };

        /* Power up FPGA image */
        partition@20000 {
                reg = <0x20000 0xc00000>;
                label = "fpga0";
        };

        /* Secondary FPGA image (on RE_CONFIGn button) */
        partition@C20000 {
                reg = <0xc20000 0xc00000>;
                label = "fpga1";
        };

        /* Space for operating system use */
        partition@1820000 {
                reg = <0x1820000 0x027c0000>;
                label = "os";
        };

        /* Second stage bootloader */
        parition@3fe0000 {
                reg = <0x3fe0000 0x20000>;
                label = "boot";
        };
};
```

5.2.3 Intel StrataFlash

FreeBSD/BERI includes an `isf(4)` driver supporting the Intel StrataFlash found on the Terasic DE4 board; however, this driver has been deprecated in favour of enhancements to the *Common Flash memory Interface* (CFI) driver shipped with FreeBSD.

Kernel Configuration

```
device          isf
```

FDT

```
flash@0x74000000 {
        compatible = "intel,strataflash";
        reg = <0x74000000 0x2000000>;
};

flash@0x76000000 {
        compatible = "intel,strataflash";
        reg = <0x76000000 0x2000000>;
};
```

5.2.4 IntelStrata Flash partitioning

This section therefore documents only use of `device.hints` to configure regions of the flash as separate `/dev/map` partitions via the `geom_map(4)` driver. We use `/dev/map` paritions for backwards compatiblity and to provide easy access to regions not defined by the hardware.

Kernel Configuration

```
device          geom_map
```

device.hints

```
# Hardwired location of bitfile
hint.map.0.at="cfid0s.fpga0"
hint.map.0.start=0x00000000
hint.map.0.end=0x00c00000
hint.map.0.name="fpga"

# Kernel in the middle of the operating system parition
hint.map.1.at="cfid0s.os"
hint.map.1.start=0x007e0000
hint.map.1.end=0x01fe0000
hint.map.1.name="kernel"
```

5.2.5 Cambridge/Terasic Multi-Touch LCD Display (MTL)

The `terasic_mtl` device driver implements a set of device nodes representing various aspects of the Terasic MTL as interfaced with using the Cambridge IP core. These device nodes implement POSIX I/O and memory-mapped interfaces to, respectively, the register interface, text frame buffer, and pixel frame buffer.

Kernel Configuration

```
device          terasic_mtl
```

FDT

```
touchscreen@70400000 {
        compatible = "sri-cambridge,mtl";
        reg = <0x70400000 0x1000
               0x70000000 0x177000
               0x70177000 0x2000>;
};
```

device.hints

```
hint.terasic_mtl.0.at="nexus0"
hint.terasic_mtl.0.reg_maddr=0x70400000
hint.terasic_mtl.0.reg_msize=0x1000
hint.terasic_mtl.0.pixel_maddr=0x70000000
hint.terasic_mtl.0.pixel_msize=0x177000
hint.terasic_mtl.0.text_maddr=0x70177000
hint.terasic_mtl.0.text_msize=0x2000
```

Chapter 6

The Deimos demonstration operating system

Deimos is a demonstration microkernel operating system that uses the CHERI ISA's capability features to sandbox untrustworthy applications. For the purposes of this demonstration, the CHERI prototype CPU was implemented in the Terasic tPad platform using an Altera FPGA; the tPad includes a VGA touchscreen, which is used for the Deimos user interface.

6.1 Demonstration narrative

Figure 6.1 illustrates the demonstration user interface: two untrusted applications, a touchscreen drawing application and a weather status bar, as well as a trustworthy status bar across the top of the display maintained by the operating system. All applications run within the kernel ring and a single address space – traditionally reserved only for privileged applications, but using CHERI's memory capability features, partitioned into a supervisor and two sandboxes. Access to I/O devices, such as touchscreen input and portions of the frame buffer, is delegated to the sandboxes using the capability mechanism, providing hardware-enforced control over display access. Attempts to draw outside of the delegated display area trigger a hardware exception, returning control to the supervisor.

In the demonstration, all touchscreen input is sent to the on-screen drawing application – drawing outside of its delegated area using a stylus causes the application to "misbehave", attempting to write outside its screen region, triggering an exception. The supervisor displays an exception indicator showing that a disallowed access has been requested – the sandbox is then restarted on the next instruction, allowing it to continue. Over 80% of code in the Deimos kernel and applications is portable C and stock 64-bit MIPS assembly code, requiring only a very small amount of CHERI-specific assembly in order to manage additional register state, delegate capabilities, and perform I/O access via capabilities.

The demonstration shows off a number of features of the CHERI architecture:

- A hybrid capability model, allowing conventional C and MIPS assembly to coexist, transparently, with a hardware-enforced sandbox model.

Figure 6.1: The Deimos touchscreen interface

- Selective delegation and containment of I/O access within the kernel ring, illustrating how device drivers (for example) can be contained using the capability model.
- The flexibility of CHERI hybridisation: the ability to run an operating system and applications using solely capability protection features, without employing the TLB.
- How CHERI features, combined with hardware user interface elements and software UI components, can be used to implement a trusted path.

6.2 Deimos design and implementation

Figure 6.2 illustrates the rough structure of the Deimos demonstration:

- $10,500$ lines of Bluespec implementing a fully pipelined, 64-bit MIPS CPU, including a capability coprocessor as CP2.
- $2,950$ lines of C and assembly code implementing a preemptive, multi-tasking microkernel able to use capabilities for sandboxing of untrustworthy applications.
- 640 lines of C and assembly code implementing support libraries for sandboxed applications, including graphics library and string management.
- 140 and 90 lines, respectively, of C and assembly code for two touchscreen applications.

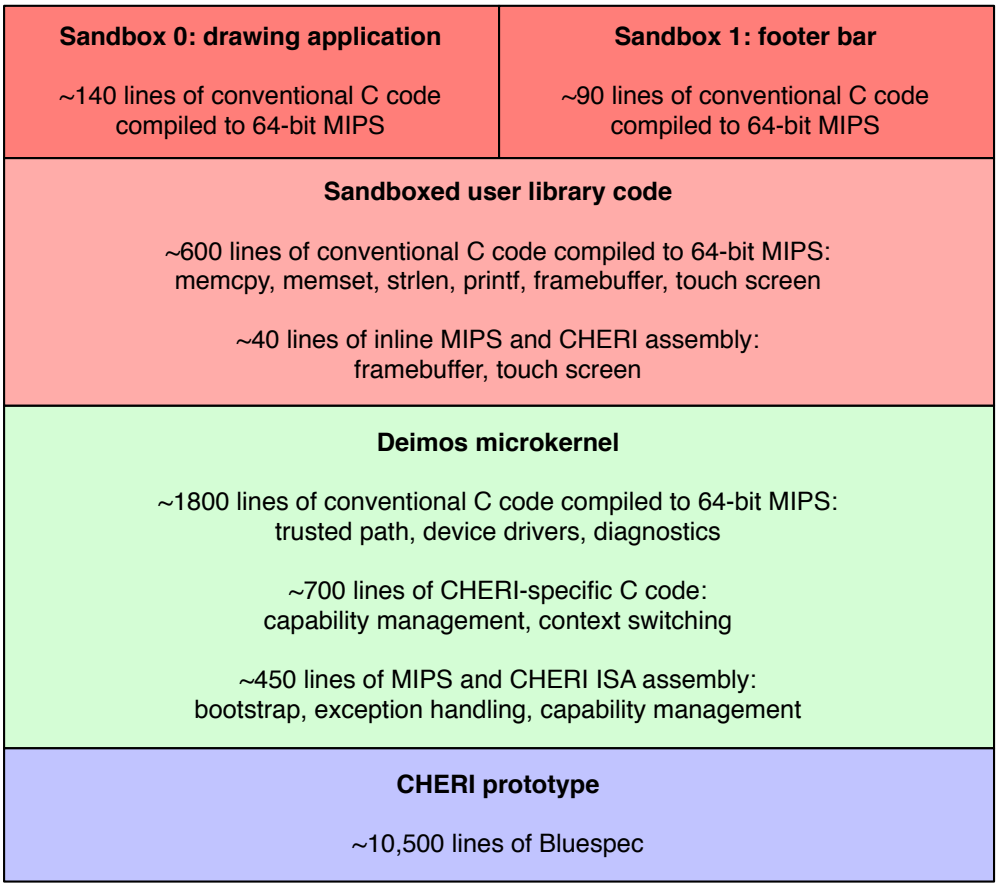

Figure 6.2: The Deimos software architecture: a blend of C code compiled to stock 64-bit MIPS machine code, stock 64-bit MIPS assembly, and a small amount of CHERI-specific assembly.

6.2.1 Supervisor

Deimos uses only the capability coprocessor for sandboxing, eschewing use of virtual addressing. The supervisor runs entirely within exception-handling context – that is, with set to $\mathbf{EXL} = 1$; a small bootstrap routine initialises the Deimos kernel stack and data structures, and then triggers an exception to enter the supervisor. The supervisor consists of an exception handler implementing a small microkernel. On an exception, the supervisor saves "userspace" general-purpose and capability registers to the "process"'s context structure; when the supervisor has completed execution, user context is restored – possibly after a context switch, selecting a new process.

Deimos handles a number of exception types, including userspace preemption using the MIPS cycle counter, explicit system calls from the user process, and capability exception handling faults. Deimos system calls include simple UART I/O routines, a voluntary yield call, and a call to retrieve capabilities using a string-based namespace; capabilities delegating access to different portions of user-addressable screen space, as well as touchscreen input, are available to processes, subject to a simple access control policy.

6.2.2 Memory

Deimos uses capability registers **PCC** and **EPC** to lay out per-sandbox address space with respect to conventional MIPS memory access instructions, providing a small dead space at address 0, code segment, heap, and stack. Using this technique, unmodified MIPS application code can be isolated transparently. Access to additional memory outside of the compatibility address space is possible through delegated capabilities – such as to the frame buffer. Application code is linked to run at alternative addresses, and some care must be taken during context switch to ensure that general-purpose and capability registers are saved and restored in appropriate order to allow both user applications and the supervisor to run correctly – especially as most of the supervisor is implemented in C code compiled to stock MIPS machine code.

6.2.3 CHERI-aware ABI

Each user process is described by context structures holding preserved general-purpose and capability register context, saved and restored when the supervisor is entered and exited. Deimos implements an enhanced Application Binary Interface (ABI) that allows capabilities to be passed to and returned from functions, including system calls. As with general-purpose registers, this calling convention designates specific capability registers for use as arguments, returned values, and temporary values. Most code in CHERI applications is not aware of the enhanced ABI and hence preserves capability registers; certain kernel and application code is aware of capability registers and manages them per these conventions. Application frame buffer library code, for example, queries frame buffer capabilities when starting, and then preserves them for later use.

Target	Description
`run`	Build Deimos and run in the CHERI simulator
`verilog`	Build Deimos and generate a memory image for FPGA use
`clean`	Clean the Deimos build; the same options must be passed to clean as were used to build Deimos.

Table 6.1: Deimos make targets

Option	Description
`CP2=YES`	Enable capability coprocessor support, sandboxing applications
`PREEMPTION=YES`	Enable cycle timer-based preemption
`SIMULATOR=YES`	Modify runtime timing parameters to improve performance in a simulated environment
`TRACE=YES`	Enable UART tracing of Deimos system events
`UART_DISABLE_FC=YES`	Disable flow control support for the Deimos console on the UART – useful for some FPGA targets where a remote endpoint may not be connected

Table 6.2: Deimos make options

6.3 Building Deimos

The Deimos source code may be found in the `deimos/trunk` tree in the CHERI Subversion repository. Table 6.1 shows a number of useful make targets for Deimos. Table 6.2 shows options that may be passed when building Deimos.

6.4 Conclusion

Deimos is a simplistic operating system, implementing an elementary process model and minimalist system services. However, it is sufficiently capable to illustrate a number of key features of the CHERI processor – especially, its support for a hybrid capability model in which some portions are explicitly aware of capabilities, but much is not. Deimos also illustrates how capability delegation can be used even within a privileged ring to contain software components, delegating limited I/O access to them, suggesting a number of applications of CHERI features within conventional operating systems. We hope to build on Deimos for further demonstration and testing purposes, and use lessons learned from the experience of building Deimos to inform our larger-scale experiments with CHERI, such as the adaptation of the FreeBSD operating system to exploit capabilities within both its kernel and userspace.