QSim Version 0.1.1 User Guide

Chad D. Kersey
Georgia Institute of Technology
Atlanta, GA 30332

January 20, 2012

Contents

Introduction

QSim is a thread safe multicore emulation library based on the QEMU emulator. QSim provides instruction-level control of the emulated environment and detailed information about the executing instruction stream. X86 Linux guests are supported with hundreds of emulated hardware contexts, each of which can run in an independent host thread. Applications of QSim include trace-gathering, multithreaded software development, and its primary target: microarchitecture simulation.

In the domain of microarchtiecture simulators, the advantage of using a library like QSim is that the instruction set is emulated with high fidelity by QEMU, so the simulator only needs to implement those aspects of the instruction set relevant to its results. This has both advantages and disadvantages that have been discussed at length elsewhere.

0.1 Requirements

The following are needed to run QSim on your machine. Before you attempt to build and run QSim, make sure these are available:

- 64-bit CPU and operating system
- 2GB of RAM and at least 4GB of swap
- Libraries and compiler to build 32-bit binaries

Chapter 1

QSim Software Architecture

QSim is an effort in combining several large and complex open source software packages into a single entity. The QSim code is very small compared to the QEMU and Linux codebases it leverages. These other projects are not included with QSim, but are downloaded and patched by the provided scripts `getqemu.sh` and `getkernel.sh`.

1.1 QSim Design

1.1.1 The QEMU Emulator

A block diagram of QEMU is shown in Figure 1.1. QEMU is an open source PC emulator that uses dynamic binary translation to translate guest instructions to a host instructions in the translation cache. Multiple guest architectures are supported, both as guests and hosts, including x86, SPARC, and PowerPC. The environment provided by QSim allows an operating system and applications to be run in an emulated environment, instruction by instruction, on multiple emulated cores. Currently only 32-bit x86 guests are supported, but other architectures are expected to become available.

Figure 1.2 illustrates the QEMU translation process. This is a two-step process in which guest instructions are translated into Tiny Code Generator (TCG) operations and then host instructions. QSim allows the creation of *helper functions*, which are functions that can be called from the translation cache. QEMU uses helper functions to implement uncommon but complex guest instructions, so that they do not have to be implemented entirely as large and complex blocks of code that are compiled at run-time.

QSim uses the helper function mechanism to instrument guest instructions. Calls to helper

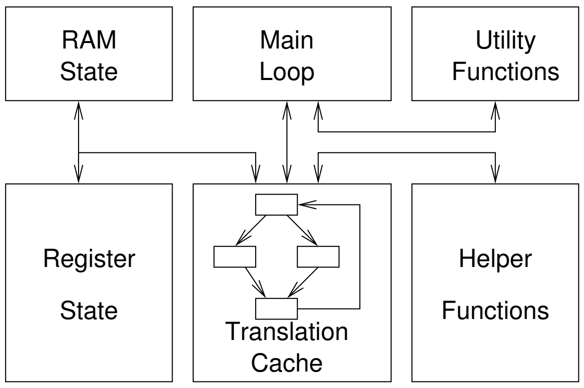

Figure 1.1: Block diagram of QEMU.

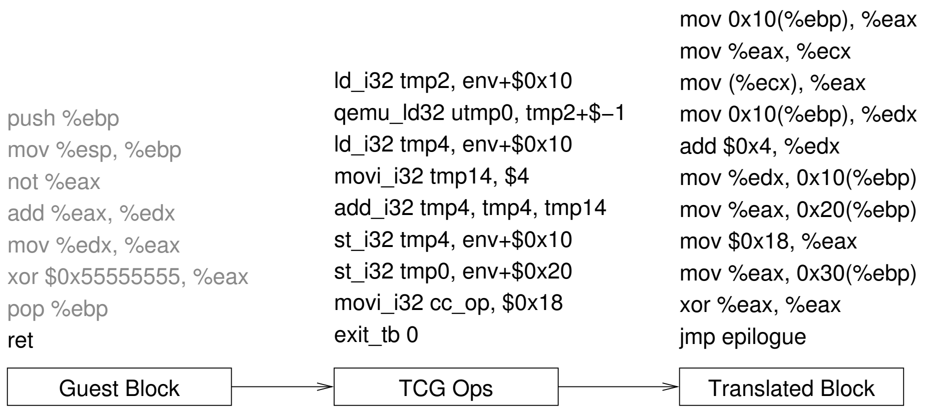

Figure 1.2: QEMU translation process with expansion of return instruction from a guest basic block.

functions are added to the translations of instrumented instructions. From these helper functions, callbacks that have been registered by the user's program are called.

1.1.2 The QSim Library

Figure 1.3 is an illustration of the fully-integrated QSim software architecture. The QEMU emulator, as seen in Figure 1.1 is present, replicated once per simulated hardware context and sharing guest RAM state. The QSim library makes calls into these QEMU CPUs on behalf of the client program, and the QEMU CPUs make the callbacks into the client programs based on events in the instruction stream. A variety of different callbacks are provided, including:

- Every instruction
- Atomic memory operations
- Memory reads/writes
- Register reads/writes
- Interrupts
- "Magic" CPUID instructions

The atomic memory operation and read/write callbacks are given in addition to the instruction callbacks for their corresponding instructions. Magic instructions are used to provide out-of-band communication between the guest and QSim, and are explained in detail in Section 1.3.

1.1.3 Guest Environment

The QSim guest environment includes the Linux kernel, the Busybox all-in-one userland, an init script, and a statically-linked application binary. Since QSim currently provides no emulation of disk or network devices, guest programs are loaded as part of a Linux inital RAM filesystem.

Porting applications to the guest environment and building the Linux kernel image is discussed at length in Chapter 4.

1.1.4 Remote QSim

For environments where detailed, distributed high-core-count simulations are performed, QSim provides a client/server version that can operate over a network. Each server is intended to support hundreds of clients, one for each simulated hardware context. Further discussion of remote QSim is in Chapter 2.

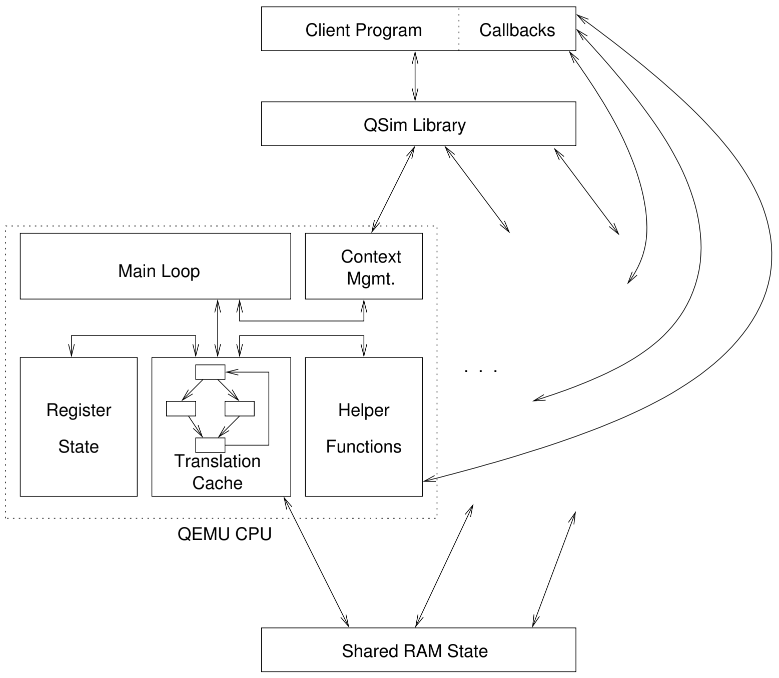

Figure 1.3: Block diagram of QSim library design with arrows representing data flow between components.

1.1.5 QDB, The QSim Debugger

QDB provides a commandline interface to QSim that can be used to debug guest applications, geust operating systems, and the QSim library itself. Compared to the rest of QSim, QDB is in an earlier state of development. It is for this reason that QDB maintains its own list of bugs and unimplemented features.

1.1.6 The Fastforwarder

The QSim fastforwarder, `qsim-fastforward` is used to generate state files that represent the state of RAM after the machine has booted, but before the application has began running. This is so that the potentially very long time spent waiting for Linux to boot can be amortized over all simulations using a given `bzImage` and number of QEMU CPUs.

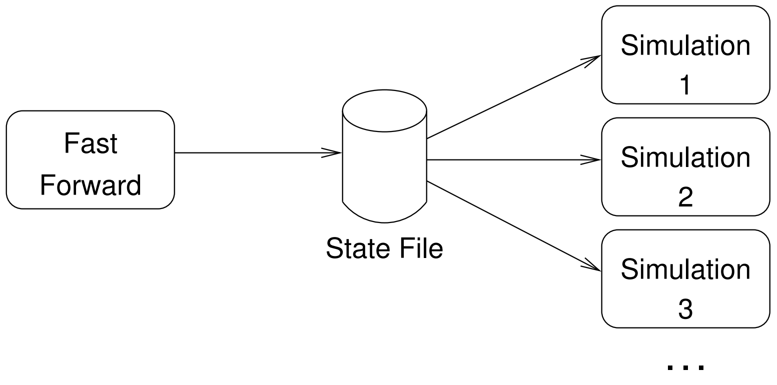

Figure 1.4: Operation of `qsim-fastforward` generates a reuseable state file.

1.2 QSim Directory Hierarchy

What follows is a tour of the directory structure of the QSim codebase. This is provided as both a reference and an introduction.

1.2.1 Documentation

```
INSTALL
README
doc/
doc/web
```

`doc/` contains this document, of which `README` is an abridged version. `INSTALL` is a simple runlog containing all of the commands needed to create a useable QSim installation.

`doc/web` is the QSim website, included here so that it may be modified under version control as needed.

1.2.2 QSim Core

```
qsim.h
qsim-regs.h
qsim-vm.h
qsim-context.h
mgzd.h
vm-func.h
qsim.cpp
Makefile
```

These files form the core of QSim. `qsim-regs.h` defines `enum regs`, an important enumeration of guest register names shared between the modifications to QEMU and QSim. `qsim-vm.h` defines the types of the basic callback functions that are called from within QEMU, as well as `qemu_ramdesc_t`, a structure used to describe guest RAM areas within QEMU to allow sharing of memory between multiple instances of QEMU. `mgzd.h` and `qsim-context.h` provide the services necessary to maintain multiple separate instances of QEMU as though they were components of the same program. `qsim.h` and `qsim.cpp` are the actual nuts and bolts of QSim. `qsim.h` must be included by any client program of QSim and `qsim.cpp` is the only source in what becomes `qsim.so`.

`Makefile` leaves a lot to be desired, but it is responsible for building QSim. For more information on building QSim, see `INSTALL` in the QSim root directory.

1.2.3 Fast Forwarder

```
fastforwarder.cpp
statesaver.cpp
statesaver.h
```

These files ocmpile to a program called `qsim-fastforwarder`, which is used as described in Section 1.1.6 to eliminate the start-up portion of long simulations, especially ones with high core counts.

1.2.4 QEMU

```
getqemu.sh
qemu-0.12.3.qsim.patch
qemu-0.12.3/
```

This directory appears after `getqemu.sh` has been run and contains the modified QEMU. The result of a successful compile of QEMU for QSim is a shared object file which will be used by the QSim library.

1.2.5 Initial Ram Filesystem

```
initrd/
initrd/init
initrd/mark_app.c
initrd/qsim_io.c
initrd/test_threads.c
initrd/busybox-1.16.1
initrd/etc
```

The `initrd` directory contains the userspace portion of the guest environment, which is packaged into a Linux initial RAM filesystem and compiled into the Linux kernel. Information on how to configure the guest system is in Chapter 4.

1.2.6 Linux Kernel Sources

```
linux/
linux/getkernel.sh
linux/linux-2.6.34.qsim.patch
linux/bzImage
linux/linux-2.6.34/
```

A modified Linux kernel is used by QSim. The `getkernel.sh` script downloads the unomdified kernel from upstream and patches it with modifications that allow it to boot in QSim guest environment, without using BIOS or usual PC hardware services that QSim does not provide.

1.2.7 Disassembler

```
distorm/
```

Several of the examples and the QSim Debugger require a disassembler. Distorm was chosen as a simple, portable, open source disassembly library. Note that it is not part of QSim proper; it is just a utility library that the bundled applications happen to require that was considered uncommon enough to be included with the source distribution.

1.2.8 Remote QSim

```
remote/
remote/README
remote/qsim-net.h

remote/client
remote/client/Makefile
remote/client/qsim-client.cpp
remote/client/qsim-client.h
remote/client/client.cpp

remote/server
remote/server/Makefile
remote/server/server.cpp
```

Remote QSim, covered in detail in Chapter 2, is a simple way to share a single instance of the QSim emulator among several remote processes. Remote QSim includes a server program (`server.cpp`), the client library (`qsim-client.h`, `qsim-client.cpp`), and a sample remote client program (`client.cpp`).

Value in `%rax`	**Function**
`0xc501e0xx`	Console output, character `x`.
`0x1d1e1d1e`	Calling CPU now executing idle loop.
`0xc75cxxxx`	Context switch to TID `x`.
`0xb007xxxx`	Bootstrap CPU `x`, `%ip` provided by subsequent magic instruction.
`0x1dxxxxyy`	Inter-processor interrupt. Interrupt CPU `x` with vector `y`.
`0xc7c7c7c7`	Request CPU count, returned in `%rax`.
`0x512e512e`	Request RAM size in MB, returned in `%rax`.
`0xaaaaaaaa`	Application start signal. Causes application start callback to be called.
`0xfa11dead`	Application end signal. Causes application end callback to be called.

Table 1.1: Magic Instructions provided by `OSDomain`.

1.2.9 Simplesim

```
simplesim/
```

SimpleSim is an example multicore out-of-order CPU simulator.

1.2.10 Examples

```
examples/
examples/qtm.cpp
examples/simple.cpp
examples/io-test.cpp
examples/vistrace.cpp
```

The “examples” are example client program using the QSim library. One of these, `simple.cpp`, is examined in detail in Chapter 3 as an example client program using QSim.

1.2.11 QDB: The QSim Debugger

```
qdb/
qdb/BUGS
```

QDB provides a commandline interface to a QSim, running one guest CPU per host thread. QDB allows users to load in symbol tables for both the kernel and individual user address spaces. It also provides dynamic instruction count based flat profiling with uniform statistical sampling. The best documentation for QDB is its own built-in help system, which can be accessed at any time by typing `help`.

1.3 Magic Instructions

QSim provides an interface for out-of-band communication between the guest environment, QSim library, and its client programs, through use of magic instructions. Magic instructions are QSim-defined regions of the CPUID address space that perform special functions. CPUID is ordinarily used by software to get information about the CPU on which it is running. A value is loaded into RAX and once the CPUID instruction has run, the registers are set according to the results. This is used ordinarily to query CPUs for features.

In QSim, the magic instructions are used to provide simplified, paravirtualized, emulated hardware. Magic instruction callbacks can also be set by client programs to extend this interface. Table 1.1 lists the magic instructions defined by QSim itself and used by the Linux kernel and some of the guest software.

Chapter 2

Remote QSim

QSim, in addition to providing a local library, also provides a remote version that can be used over a network. This is to allow the use of QSim in distributed architectural simulations where the front end emulation is not considered a bottleneck.

Figure 2.1 is a block diagram of the QSim remote architecture. The server is an ordinary client program to an instance of the local QSim library. Remote clients connect over the network using the QSim Client library, which provides an interface very similar to the local `OSDomain` to the remote clients.

2.1 Using the Server

The QSim server is in `remote/server` and can be built by running `make` in that directory. Once built, running the server with no command line options will display its usage. It takes the following command line options in the given order:

- The port to listen for incoming to connections on
- Number of QEMU CPUs (simulated hardware contexts)
- Path to kernel image file
- Optional size of guest RAM in megabytes

An example is:

```
$ ./server 1234 2 bzImage
```

The QSim server sends console output to the console on which it is started, so expect the usual boot messages. To test this server, the sample client can be started with:

```
$ ./client localhost 1234
```

This is also the port address used by the simple example program covered in Chapter 3, the remote version of which is built as `examples/simple-client`.

2.2 Writing a Client

A remote QSim client differs from an ordinary QSim client program primarily in using `Qsim::Client` instead of the usual `Qsim::OSDomain`. Many of the API calls are similar. The program listed in Chapter is a great example of the differences.

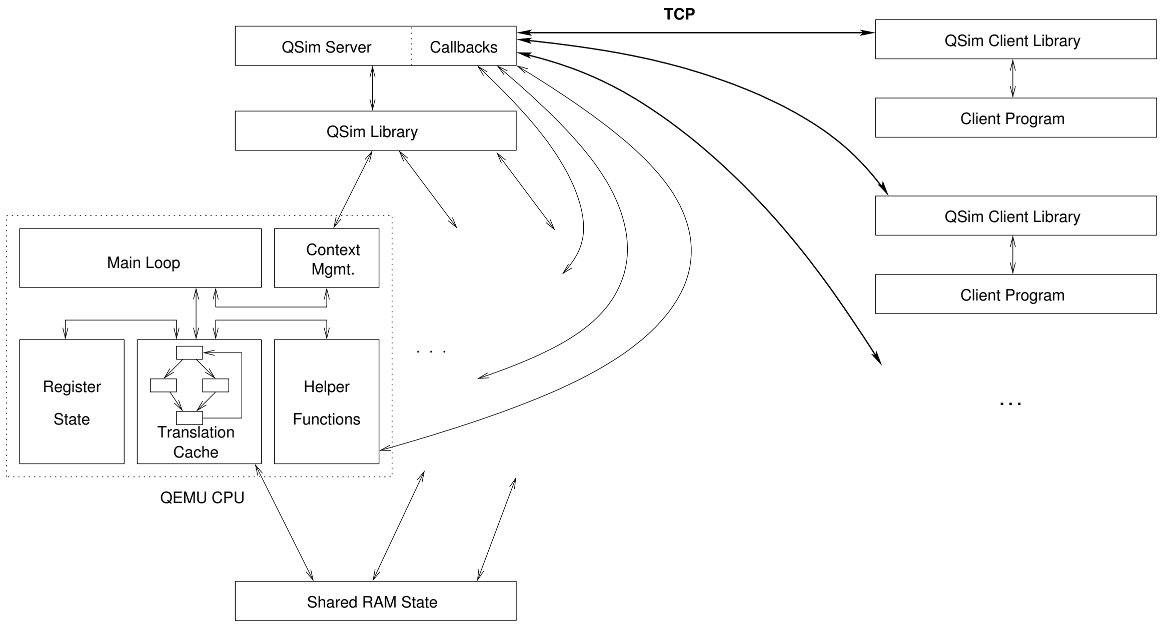

Figure 2.1: Remote QSim allows multiple clients to connect to a QSim server.

Chapter 3

A Sample Client Program

The use of QSim fundamentally boils down to a few simple steps that a client program must take:

- Instantiate one (and only one) `OSDomain`.
- Attach instrumentation to that `OSDomain`.
- Call `run()` on that `OSDomain`, ocasionally calling `timer_interrupt()`.
- Do something with the information collected by the instrumentations.

The last step is purely optional, but there would not be much reason to use QSim without the instrumentation.

For the purposes of this Chapter, we will examine `examples/simple.cpp`, the simple example; a multicore trace gathering utility.

3.1 Operation

As illustrated in Figure 3.1, `simple.cpp` is basically a trace gatherer. It installs a set of callbacks that print messages to a textual trace file whenever they are called.

3.2 Listing

```
/*****************************************************************************\
* Qemu Simulation Framework (qsim)                                            *
* Qsim is a modified version of the Qemu emulator (www.qemu.org), coupled     *
* a C++ API, for the use of computer architecture researchers.                *
*                                                                             *
* This work is licensed under the terms of the GNU GPL, version 2. See the    *
* COPYING file in the top-level directory.                                    *
\*****************************************************************************/
#include <iostream>
#include <fstream>
#include <iomanip>

#include "distorm.h"

#include <qsim.h>

#ifdef QSIM_REMOTE
#include "../remote/client/qsim-client.h"
```

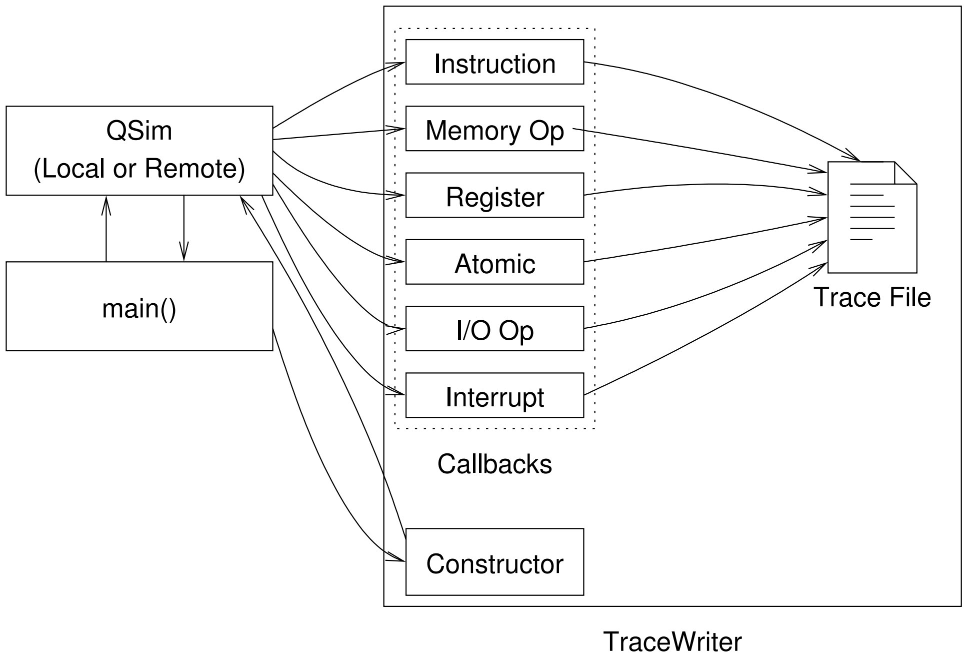

Figure 3.1: Design of the `simple.cpp` example program.

```
using Qsim::Client;
#define QSIM_OBJECT Client
#else
using Qsim::OSDomain;
#define QSIM_OBJECT OSDomain
#endif

using std::ostream;

class TraceWriter {
public:
  TraceWriter(QSIM_OBJECT &osd, ostream &tracefile) :
    osd(osd), tracefile(tracefile), finished(false)
  {
#ifdef QSIM_REMOTE
    app_start_cb(0);
#else
    osd.set_app_start_cb(this, &TraceWriter::app_start_cb);
#endif
  }

  bool hasFinished() { return finished; }

  void app_start_cb(int c) {
    static bool ran = false;
    if (!ran) {
      ran = true;
      osd.set_inst_cb(this, &TraceWriter::inst_cb);
      osd.set_atomic_cb(this, &TraceWriter::atomic_cb);
      osd.set_mem_cb(this, &TraceWriter::mem_cb);
      osd.set_int_cb(this, &TraceWriter::int_cb);
      osd.set_io_cb(this, &TraceWriter::io_cb);
      osd.set_reg_cb(this, &TraceWriter::reg_cb);
#ifndef QSIM_REMOTE
      osd.set_app_end_cb(this, &TraceWriter::app_end_cb);
#endif
    }
  }

  void app_end_cb(int c)   { finished = true; }

  int atomic_cb(int c) {
    tracefile << std::dec << c << ": Atomic\n";
    return 0;
  }

  void inst_cb(int c, uint64_t v, uint64_t p, uint8_t l, const uint8_t *b,
               enum inst_type t)
  {
    _DecodedInst inst[15];
    unsigned int shouldBeOne;
    distorm_decode(0, b, l, Decode32Bits, inst, 15, &shouldBeOne);

    tracefile << std::dec << c << ": Inst@(0x" << std::hex << v << "/0x" << p
              << ", tid=" << std::dec << osd.get_tid(c) << ", "
              << ((osd.get_prot(c) == Qsim::OSDomain::PROT_USER)?"USR":"KRN")
              << (osd.idle(c)?"[IDLE]":"[ACTIVE]")
              << "): " << std::hex;

    //while (l--) tracefile << ' ' << std::setw(2) << std::setfill('0')
    //                     << (unsigned)*(b++);

    if (shouldBeOne != 1) tracefile << "[Decoding Error]";
    else tracefile << inst[0].mnemonic.p << ' ' << inst[0].operands.p;

    tracefile << " (" << itype_str[t] << ")\n";
  }

  void mem_cb(int c, uint64_t v, uint64_t p, uint8_t s, int w) {
    tracefile << std::dec << c << ":  " << (w?"WR":"RD") << "(0x" << std::hex
              << v << "/0x" << p << "): " << std::dec << (unsigned)(s*8)
              << " bits.\n";
  }

  int int_cb(int c, uint8_t v) {
    tracefile << std::dec << c << ": Interrupt 0x" << std::hex << std::setw(2)
              << std::setfill('0') << (unsigned)v << '\n';
    return 0;
  }

  void io_cb(int c, uint64_t p, uint8_t s, int w, uint32_t v) {
    tracefile << std::dec << c << ": I/O " << (w?"WR":"RD") << ": (0x"
              << std::hex << p << "): " << std::dec << (unsigned)(s*8)
              << " bits.\n";
  }

  void reg_cb(int c, int r, uint8_t s, int type) {
    tracefile << std::dec << c << (s == 0?": Flag ":": Reg ")
              << (type?"WR":"RD") << std::dec;

    if (s != 0) tracefile << ' ' << r << ": " << (unsigned)(s*8) << " bits.\n";
    else tracefile << ": mask=0x" << std::hex << r << '\n';
  }

private:
  QSIM_OBJECT &osd;
  ostream &tracefile;
  bool finished;

  static const char * itype_str[];
};

const char *TraceWriter::itype_str[] = {
  "QSIM_INST_NULL",
  "QSIM_INST_INTBASIC",
  "QSIM_INST_INTMUL",
  "QSIM_INST_INTDIV",
  "QSIM_INST_STACK",
  "QSIM_INST_BR",
  "QSIM_INST_CALL",
  "QSIM_INST_RET",
  "QSIM_INST_TRAP",
  "QSIM_INST_FPBASIC",
  "QSIM_INST_FPMUL",
  "QSIM_INST_FPDIV"
};

int main(int argc, char** argv) {
  using std::istringstream;
  using std::ofstream;

  ofstream *outfile(NULL);

  unsigned n_cpus = 1;

#ifndef QSIM_REMOTE
  // Read number of CPUs as a parameter.
  if (argc >= 2) {
    istringstream s(argv[1]);
    s >> n_cpus;
  }
#endif

  // Read trace file as a parameter.
  if (argc >= 3) {
    outfile = new ofstream(argv[2]);
  }
```

```
129 #ifdef QSIM_REMOTE
130   Client osd(client_socket("localhost", "1234"));
131   n_cpus = osd.get_n();
132 #else
133   OSDomain *osd_p(NULL);
134   OSDomain &osd(*osd_p);

135   if (argc >= 4) {
136     // Create new OSDomain from saved state.
137     osd_p = new OSDomain(argv[3]);
138     n_cpus = osd.get_n();
139   } else {
140     osd_p = new OSDomain(n_cpus, "linux/bzImage");
141   }
142 #endif

143   // Attach a TraceWriter if a trace file is given.
144   TraceWriter tw(osd, outfile?*outfile:std::cout);

145   // If this OSDomain was created from a saved state, the app start callback was
146   // received prior to the state being saved.
147   if (argc >= 4) tw.app_start_cb(0);

148 #ifndef QSIM_REMOTE
149   osd.connect_console(std::cout);
150 #endif

151   // The main loop: run until 'finished' is true.
152   while (!tw.hasFinished()) {
153     for (unsigned i = 0; i < 100; i++) {
154       for (unsigned j = 0; j < n_cpus; j++) {
155            osd.run(j, 10000);
156       }
157     }
158     osd.timer_interrupt();
159   }
160
161   if (outfile) { outfile->close(); }
162   delete outfile;
163 #ifndef QSIM_REMOTE
164   delete osd_p;
165 #endif

166   return 0;
167 }
```

3.3 Instantiating `OSDomain`

As is often found in simple QSim client programs, the single `OSDomain` (there can only be one) is allocated on the heap in the `main()` function at lines 123 through 132. This allows the same `osd` reference to be used for either an `OSDomain` constructed with a number of CPUs and a kernel image or an `OSDomain` constructed from a state file.

If the `QSIM_REMOTE` macro is defined, this will not be done, and instead a `Qsim::Client` will be created, connecting to a predefined host and port.

3.4 The Callback System

`simple.cpp` uses instruction, atomic, memory, interrupt, and I/O callbacks and logs the occurrence of each to a trace file. Application start and end callbacks are used. `TraceWriter::app_start_cb()` sets the callbacks that perform tracing, and `TraceWriter::app_end_cb()` sets a flag that causes the main loop to exit.

`app_start_cb()` is not called in remote QSim. Instead, fast-forwarding to the application start is done automatically, before any clients are allowed to connect. If `QSIM_REMOTE` is set, the application start callback is called by the `TraceWriter` constructor instead, and no start or end callbacks are set.

Chapter 4

Porting Guest Software

Currently, there are two available methods for loading guest software into QSim. It can be compiled into the kernel image, or it can be transferred to the guest environment through the `initrd/qsim_io.c` program. With the former method, the mechanism through which they are loaded is the Linux initial RAM f ilesystem (`initramfs`). This allows a root filesystem to be included along with the kernel image and mounted at start-up, even without support for disks.

The latter method involves using the `initramfs` and the `qsim_io` program to transfer a program. The default `init` script does this. An example of a client program that transfers a file in this manner is in `examples/test-io.cpp`.

Porting an application to QSim consists of the following steps:

- Compile the application for 32-bit Linux, statically linked.
- Copy the application and all needed files into `initrd/sbin`.
- Modify the `initrd/init` script appropriately.
- Run `make` in `initrd` to build `initrd.cpio`.
- Run `ARCH=i386 make` in the Linux kernel root to create `bzImage`, the kernel image which will be loaded into the QSim address space.

4.1 Compiling for QSim

On a 64-bit host, creating 32-bit statically-linked binaries is a matter of passing the `-m32` option to the GCC compiler and the `-static` option to the linker. In a typical GNU Autotools based build environment, this can be accomplished with:

```
$ CFLAGS=-m32 CXXFLAGS=-m32 LDFLAGS=-static ./configure
$ make
```

The `file` command can be used to ensure that the resulting binary is actually a 32-bit statically-linked ELF.

4.2 Setting Up the Linux Initial Ramdisk

The `initrd` makefile is set up so that any files placed in `initrd/sbin` will be added to the generated `initrd.cpio`. User applications are not, however, included as dependencies, so when modifying them it is prudent to perform a `make clean` before running `make` to produce the ramdisk file. Typically the only actions required to set up the linux filesystem is to copy the application binary into `initrd/sbin`, modify the `initrd/init` script to launch the application, and run `make clean` followed by `make`.

The `init` script contains, out of the box, a set of sample application invocations commented out with octothorpes. If using one of the prepackaged benchmarks, it is only necessary to uncomment the corresponding line. Otherwise, the appropriate command can be added anywhere between the invocation of `mark_app` and `mark_app END`.

4.2.1 Busybox Utility Programs

The command interpreter used to execute the `init` script, as well as most of the utility programs available within it (`cat`, `grep`, `ls`) are provided by the Busybox all-in-one userland. It provides smaller-footprint versions of common Unix utilities, accessed by a set of symbolic links to the same file.

4.2.2 QSim Guest Utilities

The initial ramdisk also contains a set of QSim-specific utilities and test programs:

```
qsim-out
```

Uses the magic instruction mechanism to copy program input to the QSim console.

```
mark_app [END]
```

Uses the magic instruction mechanism to mark the start and end of the application. This causes the application start and end callbacks described on pages 29-31 to be called.

```
test-threads
```

Simply spawns 100 threads and waits for them to exit. Each thread grabs a mutex, prints a message, and exits. Used to check that atomic memory operations are being treated as such by the client program; not a simple task if this application is multithreaded.

4.3 Compiling the Kernel Image

Initially, the kernel image must be configured with the absolute path to the `initrd.cpio` file. The configuration system can be launched by changing to the kernel root and running:

```
$ ARCH=i386 make menuconfig
```

The option to configure the path to the RAM disk CPIO file is under "General Options". After this initial configuration has been performed, subsuquent builds only require the second step of the process, running `make` with `ARCH` set to `i386`:

```
$ ARCH=i386 make
```

4.4 Using the Kernel with QSim

A symlink to the kernel this produces is conveniently placed in the `linux/` directory. This is the default kernel image as used by all of the programs in `examples/`. QDB takes the path of the kernel image as a commandline option. Within client programs, the path to the kernel is an argument to the `OSDomain` constructor (see `OSDomain`, page 26 for details).

Glossary

atomic memory operation A memory operation that is required by the architecture to complete as an indivisible operation. Atomic read-modify-write operations like `XCHG` are used in the implementation of synchronization primitives.

callback A function passed as an argument to another layer of the software stack, so that it can be called by software in that layer as necessary. In QSim, callbacks are code within the *client program* that is called from *helper functions*. Callbacks are the primary way QSim allows its users to instrument the code executing in the *guest*. The word "callback" can also be used to refer to the instance of a specific call to a callback function. For example, "A memory operation callback will be received after any exception conditions on the address being accessed are cleared up." In this sentence, the "memory operation callback" is not the function itself, but a specific instance of a call to the function.

client program An application that makes use of a library; for the purposes of this document the "client program" is always an application that uses QSim.

guest The simulated system, or virtual machine, within QSim. Often used as an adjective to describe a data structure: "guest CPU state;" the registers of the virtual CPU stored in a data structure within QEMU, "guest RAM;" the memory image of the guest machine. Used as an antonym of *host*, the machine on which QSim and the client program runs.

helper function Part of QEMU, a C function intended to be called from the *translation cache*. Helper functions are used to implement *guest* functionality that is complex and not time critical. Their use results in more compact translated code and a simpler binary translator.

host The machine on which QSim and its client program runs, antonym of *guest*. Used to describe abstractions that may otherwise be confused for aspects of the guest system: "host virtual address", "host instruction set."

magic instruction An instruction that provides a service not found in current hardware, decoded using an unorthodox mechanism. In QSim, magic instructions are `CPUID` instructions initiated with invalid values in `%rax`. These can be tied to callbacks and used to implement new features or create debugging output.

OS domain A collection of CPUs among which memory is sequentially consistent and cache coherent, usually associated with a single operating system image. The `OSDomain` class gets its moniker from this concept.

QEMU CPU A hardware context within QEMU. Not to be confused with a CPU in the sense of an entire chip, or in the sense of a core within that chip that may support multiple contexts through and SMT arrangement. A four-core chip with 2-way SMT, for example, would be emulated with 8 QEMU CPUs.

remote QSim The distributed version of the QSim library. Separates functionality into an emulation server and a client which exports an API similar to locally-running QSim.

translation cache The data structure into which a dynamic binary translator like QEMU places the result of translation and from which the translated code is run. The only code emulating *guest* instructions is executed by the *host* through the translation cache, either directly or by calling *helper functions*.

Appendix A

API Documentation

What follow are descriptions of the user-facing classes of which QSim is comprised and their interfaces, member types, and member variables. By far the most important class of these is `OSDomain`, as described in Chapter 3 and documented on page 26. Entire simulators can be built interacting only with `OSDomain`. The other available classes provide various utility functions and the ability to interact directly with the QEMU CPU objects that normally comprise an `OSDomain`.

The most important utility class is `Queue`, described on page 36. It provides a way to delay the processing of execution events until after they have been emulated. Use of Queues allows all of the processing of events to be centralized within a single main loop instead of scattered through multiple callbacks. Although this results in a loss of control over the instruction-level timing in QSim, it is ideally suited for applications in which this is irrelevant or of diminished importance, such as trace generation and simple application evaluation.

A.1 `Qsim::OSDomain`

`OSDomain` is a singleton class that creates and manages a multicore processor emulator. The simplest way to use QSim is to instantiate a QSim `OSDomain`, provide it with a set of callbacks or a `Queue`, and start running instructions on it, periodically calling `OSDomain::timer_interrupt()`. This is exactly what the sample program in Chapter 3 does. `OSDomain` means "Operating System domain." It is so named because all of the CPUs in it share one cache coherent and sequentially consistent block of RAM that contains a single OS image.

A.1.1 Member Types

```
enum cpu_mode {
  MODE_REAL, MODE_PROT, MODE_LONG
};
```

Used to describe the current mode of the CPU. While 32-bit real mode and 16-bit protected mode do exist on the hardware, for modern operating systems it can be assumed that `MODE_REAL` means 16-bit default operand sizes and is only encountered during boot, and `MODE_PROT` means 32-bit default operand sizes. `MODE_LONG` only works in protected mode, and has 64-bit default operand sizes.

```
enum cpu_prot {
  PROT_KERN, PROT_USER
};
```

Describes the current protection level of the CPU. While x86 actually provides a MULTICS-like ring system complete with hardware-defined task and call gate structures and four levels of protection, these still exist only for backwards compatability. This data type is used to describe whether a given CPU within QSim is running in supervisor mode (ring 0) or user mode.

A.1.2 Member Functions

```
OSDomain(uint16_t    n,
         std::string kernel_path,
         unsigned    ram_mb = 2048);
```

This `OSDomain` constructor takes as parameters `n`, the number of CPUs, `kernel_path`, the path to an appropriate *bzImage* and optionally `ram_mb`, the size of system RAM in megabytes.

```
OSDomain(const char *state_file);
```

The other `OSDomain` constructor loads the state from a file. This allows for fast startup compared to emulating the entire boot process.

```
bool idle(unsigned i);
```

Returns true if CPU `i` is running its idle loop according to the OS, false otherwise.

```
int get_tid(uint16_t i);
```

If Linux has booted, returns the task ID of the thread currently running on CPU `i`. Otherwise returns -1.

```
enum cpu_mode get_mode(uint16_t i);
```

Returns the mode (real, protected, or long) of CPU `i` (see `enum cpu_mode`, page 26). Not capable of distinguishing 16-bit protected mode, but this is not used by any current OS.

```
enum cpu_prot get_prot(uint16_t i);
```

Returns the current protection ring (kernel or user) of CPU `i` (see `enum cpu_prot`, page 26).

```
int get_n() const;
```

Returns the number of CPUs being emulated.

```
qemu_ramdesc_t get_ramdesc() const;
```

Returns the `qemu_ramdesc_t` structure used by the instances of QEMU within this `OSDomain` to share their memory region (see `qemu_ramdesc_t`, page 38).

```
unsigned run(uint16_t i, unsigned n);
```

Run the emulator on CPU `i` for `n` instructions or until an exception condition occurs. Returns the number of instructions for which the CPU actually ran. Exception conditions are caused by callbacks that return nonzero and by the CPU being not yet booted. Since the callbacks are managed by the client program and whether the CPU has been booted can be determined through other means (see `OSDomain::booted()`, page 27), a direct method of determining which condition terminated the `run` is not necessary.

```
void connect_console(std::ostream &s);
```

The given stream is added to the list of streams to which character output from the guest software stack is delivered. This text is delivered one character at a time using a magic instruction.

```
void timer_interrupt();
```

Send a timer interrupt to all of the CPUs in the `OSDomain`. Linux expects to see these at a rate of 100Hz, or once every 10ms. Spacing calls to `timer_interrupt()` appropriately is the fundamental way the progress of simulation time can be communicated to the guest.

```
void interrupt(unsigned i, uint8_t vec);
```

Interrupt CPU `i`, causing it to run the interrupt service routine pointed to by vector `vec`.

```
bool booted(unsigned i);
```

Returns `true` if CPU `i` has been brought up by the operating system, and `false` otherwise.

```
void save_state(const char* state_file);
```

Save the state of the emulator to the given file. The OSDomain must be booted and running in protected mode, and must not be using floating point, SSE, or MMX register state when this is called. Also, unless this is called during an instruction callback, the previous instruction will be re-executed when the state is loaded. For an example demonstrating how to overcome this limitation, see `examples/fastforwarder.cpp` and its dependencies.

```
typedef int (*atomic_cb_t)(int cpu);
void set_atomic_cb(uint16_t i, atomic_cb_t cb);
void set_atomic_cb(atomic_cb_t cb);
```

Set a callback to be called from within the translation cache every time an atomic memory operation is encountered. Can be set for a given CPU `i` or, omitting this parameter, for all CPUs. This callback is of the form shown in the definition of `atomic_cb_t` and occurs after the corresponding instruction callback is made but before the instruction is executed. If the callback returns nonzero, the atomic operation will not complete and the call to `run()` (see `OSDomain::run()`, page `func:run`) will return. New callback assignments replace the previous assignment. If a null pointer is provided, the callback will be disabled.

```
typedef void (*inst_cb_t)(int            cpu,
                          uint64_t       vaddr,
                          uint64_t       paddr,
                          uint8_t        len,
                          const uint8_t *bytes);
void set_inst_cb(uint16_t i, inst_cb_t cb);
void set_inst_cb(inst_cb_t cb);
```

Set a callback to be called from within the translation cache every time a guest instruction begins executing. This callback is made prior to the instruction's execution, and upon its return, the instruction will begin. In the event of a page fault while accessing the instruction itself, the appropriate interrupt callback will be generated with no prior instruction callback for the faulting instruction, since the fetch itself and not the execution of the instruction failed, followed by the execution of the interrupt handler and finally the execution of the faulting instruction including the appropriate instruction callback.

If, however, a memory operation that is part of the instruction execution fails, the instruction callback will be seen twice, once for each attempt at executing the instruction.

Instruction callbacks provide an identifier for the CPU on which the instruction will execute, the virtual and physical address of the instruction, its size, and a pointer to the instruction code. They can be set for specific CPUs `i` or all CPUs in the `OSDomain`. New callback assignments replace the previous assignment. If a null pointer is provided, the callback will be disabled.

```
typedef int (*mem_cb_t)(int      cpu,
                        uint64_t vaddr,
                        uint64_t paddr,
                        uint8_t  size,
                        int      write);
void set_mem_cb(uint16_t i, mem_cb_t cb);
void set_mem_cb(mem_cb_t cb);
```

Set up a function to be called from within the translation cache every time a memory operation is encountered. Read operation callbacks occur immediately before the read occurs, to allow the value being read to be modified by the callback, and write operation callbacks occur immediately after the write, to allow the value just written to be read. Reads are, however, tried before the callback is called. This means that any page fault that would occur always happens before the callback is called, so that once the callback is reached the addresses are guaranteed to be reachable.

Memory operation callbacks take as arguments the CPU identifier for the CPU on which they execute, virtual and physical addresses, their size, and a flag that is 1 if the operation is a write and 0 if the operation is a read. These callbacks can be set for specific CPUs `i` or for all of the CPUs in the `OSDomain`. New callback assignments replace the previous assignment. If a null pointer is provided, the callback will be disabled.

```
typedef int (*int_cb_t)(int cpu, uint8_t vec);
void set_int_cb(uint16_t i, int_cb_t cb);
void set_int_cb(int_cb_t cb);
```

Set up an interrupt callback, either for a specific CPU `i` or all CPUs in the `OSDomain`. These callbacks are called at the moment any interrupt, hardware or software, occurs. Hardware interrupts always happen after an instruction has finished executing. Software exceptions such as page faults occur at the point in the execution of their instruction that the exception condition arises. In the case of page faults, this is prior to the receipt of a memory callback for the faulting address but after the receipt of an instruction callback for the instruction performing the access.

Interrupt callbacks take the CPU identifier and the 8-bit interrupt vector as parameters. Other related information, such as the fault address in the case of a page fault, can be retreived with `OSDomain::get_reg()` (see page 31). Like the atomic memory operation callback (see page 27), the interrupt callback can return nonzero to halt instruction execution at a given point and force `run()` to return prematurely. This can be used to provide resolution to an exception condition from within the main loop instead of a callback. New callbacks assigned with this function replace the old ones. A null assignment will disable the callback.

```
typedef void (*io_cb_t)(int      cpu,
                        uint64_t port,
                        uint8_t  size,
                        int      type,
                        uint32_t val);
void set_io_cb(uint16_t i, io_cb_t cb);
void set_io_cb(io_cb_t cb);
```

Set an I/O callback. Similar to the memory callback, but much simpler due to the simple flat I/O address space provided by the x86 architecture. Does not yet allow for modifying the value returned by an `IN` instruction, but does allow reading the value (`val`) provided by the `OUT` instruction. New callbacks assigned with this function replace the old ones. A null assignment will disable the callback.

```
typedef void (*reg_cb_t)(int     cpu,
                         int     reg,
                         uint8_t size,
                         int     type);
void set_reg_cb(uint16_t i, reg_cb_t cb);
void set_reg_cb(reg_cb_t cb);
```

Set a register callback. Same semantics as memory callback. The `reg` parameter can be interpreted as a member of `enum regs` (see page 37). In the special case that `size` is zero, the register callback can be interpreted as a flag bit read or write. In this case, `type` is the logical or of one or more elements of `enum flags` (see page **??**).

```
void set_app_start_cb(void (*f)(int));
```

A simple callback that uses a magic instruction delivered by the guest application or a three-instruction utility program to provide an indication of when this application begins running. Includes a CPU identifier, but this cannot be taken to mean the CPU on which the benchmark begins execution, since the magic instruction may have been deliviered by a utility program and not the benchmark application itself. Common uses include fast-forwarding simulators past the boot sequence in a purely emulated mode and resetting performance counters within simulators so that at application end they reflect only the operations created by the application being evaluated. New callbacks assigned with this function replace the old ones. A null assignment will disable the callback.

```
void set_app_end_cb(void (*f)(int));
```

A simple callback that uses a magic instruction delivered by a three-instruction utility program to indicate when the guest application has terminated. Typically used to end the simulation. New callbacks assigned with this function replace the old ones. A null assignment will disable the callback.

```
template <typename T>
  void set_atomic_cb(T* o,
                     int (T::*f)(int cpu));
```

Same as `set_atomic_cb()` on page 27, but instead of operating on a static or global function is capable of making calls to arbitrary member functions of any instance `o` of any class `T`, that fit the prototype. Also capable of making multiple calls, so that functionality can be combined. New calls to this version of `set_atomic_cb()` do not replace the previous callback but instead add a new one. In this way, functionality can be combined.

It is important to note that this callback system is incompatible with the other one. Calls to this version of `set_atomic_cb()` disable static callbacks, and new static callbacks disable these class member callbacks.

```
template <typename T>
  void set_magic_cb(T* o,
                    int (T::*f)(int      cpu,
                                uint64_t rax));
```

The only way to set magic instruction callbacks in `OSDomain`, since `OSDomain`'s static magic instruction callback is already of central importance and cannot be overridden. Provides an easy way to add new magic instructions by intercepting `CPUID` instructions.

```
template <typename T>
  void set_io_cb(T* o,
                 void (T::*f)(int      cpu,
                              uint64_t port,
                              uint8_t  size,
                              int      write,
                              uint32_t value));
```

The equivalent of `set_atomic_cb()` (see page 29) for the I/O callback (see `set_io_cb()`, page 29).

```
template <typename T>
  void set_reg_cb(T* o,
                 void (T::*f)(int      cpu,
                              uint64_t port,
                              uint8_t  size,
                              int      type));
```

The equivalent of `set_atomic_cb()` (see page 29) for the register callback (see `set_reg_cb()`, page 29).

```
template <typename T>
  void set_mem_cb(T* o,
                  void (T::*f)(int      cpu,
                               uint64_t vaddr,
                               uint64_t paddr,
                               uint8_t  size,
                               int      write));
```

The equivalent of `set_atomic_cb()` (see page 29) for the memory operation callback (see `set_mem_cb()`, page 28).

```
template <typename T>
  void set_int_cb(T* o,
                  int (T::*f)(int     cpu,
                              uint8_t vector));
```

The equivalent of `set_atomic_cb()` (see page 29) for the interrupt callback (see `set_int_cb()`, page 28).

```
template <typename T>
  void set_inst_cb(T* o,
                   void (T::f)(int            cpu,
                               uint64_t       vaddr,
                               uint64_t       paddr,
                               uint8_t        len,
                               const uint8_t *bytes));
```

The equivalent of `set_atomic_cb()` (see page 29) for the instruction callback (see `set_inst_cb()`, page 27).

```
template <typename T>
  void set_app_start_cb(T* o,
                        void (T::f)(int));
```

The equivalent of `set_atomic_cb()` (see page 29) for the application start callback (see `set_app_start_cb()`, page 29).

```
template <typename T>
  void set_app_end_cb(T* o,
                      void (T::f)(int));
```

The equivalent of `set_atomic_cb()` (see page 29) for the application end callback (see `set_app_end_cb()`, page 29).

```
uint64_t get_reg(unsigned cpu, enum regs r);
```

Retrieve the value of a register on CPU `i`, referenced using one of the names from `enum regs` (see page 37).

```
void set_reg(unsigned cpu, enum regs r, uint64_t value);
```

Alter the value of a register on CPU `i`, referenced using one of the names from `enum regs` (see page 37).

```
template <typename T> void mem_rd(T& d, uint64_t paddr);
```

Read a value of arbitrary size from guest RAM at physical address `paddr`.

```
template <typename T> void mem_wr(T& d, uint64_t paddr);
```

Write a value of arbitrary size to guest RAM at physical address `paddr`.

```
template <typename T>
  void mem_rd_virt(unsigned i, T& d, uint64_t vaddr);
```

Read a value of arbitrary size from guest RAM at virtual address `vaddr`, translating the address according to CPU `i`.

```
template <typename T>
  void mem_wr_virt(unsigned i, T& d, uint64_t vaddr);
```

Write a value of arbitrary size to guest RAM at virtual address `vaddr`, translating the address according to CPU `i`.

A.1.3 Utility Functions

The following functions are not part of `Qsim::OSDomain` proper, but act on OSDomains to provide important features:

```
void load_file(OSDomain &osd, const char *filename);
```

If the `OSDomain` is running the `qsim-io` guest program, this function will provide input through this system. This is used by the default initial ramdisk to load a `tar` archive containing the application to be run, along with any data and libraries it needs. A demonstration of the use of `load-file` can be seen in `examples/io-test.cpp`.

A.2 Qsim::QemuCpu

QemuCpu is the basic CPU element in QSim, instantiated once for each CPU in the OSDomain. QemuCpu can also be instantiated by the client program directly, which could be used in the simulation of more complex distributed or non-coherent architectures. The guest RAM state illustrated in Figure 1.3 is contained inside of a "master", QemuCpu allocated by QEMU. Multiple "slave" QemuCpu objects share the RAM state allocated by the master, including the kernel image loaded by it. The various callback setters and other functions behave similarly to the way they do in OSDomain, but with less emphasis on convenience of use for the end-user.

A.2.1 Member Functions

```
QemuCpu(int id, const char* kernel, unsigned ram_mb=1024);
```

Construct a master QemuCpu with ram_mb megabytes of guest RAM and load the kernel image at the path in kernel.

```
QemuCpu(int id, QemuCpu *master_cpu, unsigned ram_mb=1024);
```

Create a slave QemuCpu, with master_cpu as its master. ram_mb must match the value of ram_mb given for the master.

```
unsigned run(unsigned n);
```

Run for n instructions or until an exception condition occurrs. Explained more fully in the description OSDomain::run() (see page 27).

```
typedef int (*atomic_cb_t)(int cpu);
void set_atomic_cb(atomic_cb_t cb);
```

Set an atomic memory operation callback for this CPU as explained in the description of the corresponding member function of OSDomain on page 27.

```
typedef void (*inst_cb_t)(int            cpu,
                          uint64_t       vaddr,
                          uint64_t       paddr,
                          uint8_t        len,
                          const uint8_t *bytes);
 void set_inst_cb(inst_cb_t cb);
```

Set an instruction callback for this CPU as explained in the description of the corresponding member function of OSDomain on page 27.

```
typedef int (*mem_cb_t)(int      cpu,
                        uint64_t vaddr,
                        uint64_t paddr,
                        uint8_t  size,
                        int      write);
 void set_mem_cb(mem_cb_t cb);
```

Set a memory operation callback for this CPU as explained in the description of the corresponding member function of OSDomain on page 28.

```
typedef int (*int_cb_t)(int cpu, uint8_t vec);
 void set_int_cb(int_cb_t cb);
```

Set an interrupt callback for this CPU as explained in the description of the corresponding member function of OSDomain on page 28.

```
typedef int (*magic_cb_t)(int cpu, uint64_t rax);
void set_magic_cb(magic_cb_t cb);
```

Set a magic instruction callback for this CPU. While this works similar to the callback setters for `OSDomain`, there is no way to set a static magic instruction callback for `OSDomain` as it already provides one. It is, however, possible to set member function callbacks as described on page 29.

```
typedef void (*io_cb_t)(int      cpu,
                        uint64_t port,
                        uint8_t  size,
                        int      type,
                        uint32_t val);
void set_io_cb(io_cb_t cb);
```

Set an I/O operation callback for this CPU as explained in the description of the corresponding member function of `OSDomain` on page 29.

```
typedef void (*reg_cb_t)(int     cpu,
                         int     reg,
                         uint8_t size,
                         int     type);
void set_io_cb(reg_cb_t cb);
```

Set a register access callback for this CPU as explained in the description of the corresponding member function of `OSDomain` on page 29.

```
uint8_t mem_rd(uint64_t pa);
```

Read byte at physical address `pa`.

```
void mem_wr(uint64_t pa, uint8_t val);
```

Write byte `val` to physical address `pa`.

```
uint8_t mem_rd_virt(uint64_t va);
```

Read byte at virtual address `va`.

```
void mem_wr_virt(uint64_t va, uint8_t val);
```

Write byte `val` to virtual address `va`.

```
int interrupt(uint8_t vec);
```

Trigger an interrupt on this CPU with vector `vec`.

```
uint64_t get_reg(enum regs r);
```

Retreive the contents of register `r`.

```
void set_reg(enum regs r, uint64_t v);
```

Set the contents of register `r` to value `v`.

```
qemu_ramdesc_t get_ramdesc() const;
```

Get the RAM descriptor for this CPU. Details of the `qemu_ramdesc_t` structure are on page A.8.

A.3 `Qsim::Client`

The client class, located in `remote/client/qsim-client.h`, provides an OSDomain-like interface to a remote QSim server.

A.3.1 Member Functions

```
Client(int socket);
```

Construct a client connected to the server on `socket`. The parameter is typically provided by the utility function `client_socket(const char *host, const char *port)`, from `remote/qsim-net.h`, which is included by `remote/client/qsim-client.h`.

```
unsigned run(unsigned short cpu, unsigned insts);
void timer_interrupt();
void interrupt(int cpu, uint8_t vec);
bool booted(unsigned i);
bool idle(unsigned cpu);
unsigned get_n();
unsigned get_tid(unsigned cpu);
Qsim::OSDomain::cpu_mode get_mode(unsigned cpu);
Qsim::OSDomain::cpu_prot get_prot(unsigned cpu);
template <typename T> void set_atomic_cb(T* p, ...);
template <typename T> void set_inst_cb(T*, ...);
template <typename T> void set_int_cb(T*, ...);
template <typename T> void set_mem_cb(T*, ...);
template <typename T> void set_magic_cb(T*, ...);
template <typename T> void set_io_cb(T*, ...);
template <typename T> void set_reg_cb(T*, ...);
```

These functions are equivalent to their `Qsim::OSDomain` counterparts, with the exception that `Qsim::Client::booted()` cannot be called from within a callback function. The final arguments to the callback setters have been removed for clarity.

A.4 `Qsim::QueueItem`

`QueueItem`s contain instruction, memory, and interrupt data that would otherwise have been provided by a callback. They allow the callback arguments to be stored in a `Queue` (see page 36) for later processing. The `QueueItem` class provides three constructors corresponding to the three different types of data a `QueueItem` can hold. These constructors should rarely, if ever, need to be called by a QSim client program.

A.4.1 Member Functions

```
QueueItem(uint64_t       vaddr,
          uint64_t       paddr,
          uint8_t        len,
          const uint8_t *bytes);
```

Creates a `QueueItem` with `type = QueueItem::INST` and populates `data` accordingly.

```
QueueItem(uint64_t vaddr,
          uint64_t paddr,
          uint8_t  size,
          int      type);
```

Creates a `QueueItem` with `type = QueueItem::MEM` and populates `data` accordingly.

```
QueueItem(uint8_t vec);
```

Creates a `QueueItem` with `type = QueueItem::INTR` and populates `data` accordingly.

A.4.2 Member Variables

```
enum { INST, MEM, INTR } type;
```

Specifies the type of a `QueueItem`. Using an `enum` and a `union` instead of inheritance makes it possible to make `Queue` a queue of `QueueItem` objects themselves instead of pointers, simplifying the allocation of these objects and improving performance.

```
union {
  struct {
    uint64_t       vaddr;
    uint64_t       paddr;
    uint8_t        len;
    uint8_t        bytes[15];
  } inst;
  struct {
    uint64_t       vaddr;
    uint64_t       paddr;
    uint8_t        size;
    int            type;
  } mem;
  struct {
    uint8_t        vec;
  } intr;
} data;
```

The data carried by a `QueueItem`. Which of the three structs to read is determined by the value of `type`.

A.5 `Qsim::Queue`

A.5.1 Inheritance

```
class Queue : public std::queue<QueueItem>;
```

`Queue` inherits from `std::queue<QueueItem>`. It automatically enqueues objects of type `QueueItem` using callbacks within QSim. User software can use the familiar standard library operations to access these objects. Additional information on the use of `Queue` can be found in the API reference entry on `QueueItem`, page 35. `Queue` objects are instantiated on a per-guest-CPU basis.

A.5.2 Member Functions

```
Queue(OSDomain &cd, int i, bool timer_int_on_hlt = true);
```

The `Queue` constructor attaches itself to CPU `i` of `OSDomain cd`. Since `Queue` sets a new instruction callback, an option is provided to have this callback generate timer interrupts whenever a CPU executes the `HLT` instruction, allowing simpler timing models that are based only on instruction counts to be used. For an in-depth discussion of the `HLT` instruction and its implications in QSim, see Section **??** on page **??**.

```
void set_filt(bool user,
              bool krnl,
              bool prot,
              bool real,
              int  tid = -1);
```

Set a filter on the queue so not all instructions executed cause items to be added to the queue. A task ID field value of -1 (the default) represents all tasks. The filter actually created will be a logical or of the options chosen. For example:

```
q.set_filt(true, true, true, false);
```

Only enqueues instructions executed in protected mode, at both user and kernel privilege levels.

A.6 `enum regs`

The register enumeration contains at least the following entries:

```
enum regs {
  QSIM_RAX, QSIM_RCX, QSIM_RDX, QSIM_RBX,
  QSIM_RSP, QSIM_RBP, QSIM_RSI, QSIM_RDI,
  QSIM_ES,  QSIM_CS,  QSIM_SS,  QSIM_DS,
  QSIM_FS,  QSIM_GS,  QSIM_ESB, QSIM_CSB,
  QSIM_SSB, QSIM_DSB, QSIM_FSB, QSIM_GSB,
  QSIM_ESL, QSIM_CSL, QSIM_SSL, QSIM_DSL,
  QSIM_FSL, QSIM_GSL, QSIM_RIP, QSIM_RFLAGS,
  QSIM_CR0, QSIM_CR2, QSIM_CR3
};
```

There is no guarantee on ordering or numerical value. All references to registers in the QSim API use these names, defined in `qsim-regs.h`. Names of segment registers followed by 'B' and 'L', such as `QSIM_CSB` and `QSIM_CSL` refer to the "base" and "length" properties of segments in 32-bit protected mode. Note that also, no matter which mode the guest CPU is in, the register names reflect their names in 64-bit long mode.

A.7 `enum flags`

The flags enumeration contains at least the following entries, set up to be used as bits in a vector:

```
enum flags {
  QSIM_FLAG_OF = 0x01, QSIM_FLAG_SF = 0x02,
  QSIM_FLAG_ZF = 0x04, QSIM_FLAG_AF = 0x08,
  QSIM_FLAG_PF = 0x10, QSIM_FLAG_CF = 0x20
};
```

This is defined in `qsim-regs.h` and used to generate the value for the `reg` parameter to the register access callback in the case that `size` is zero, indicating a flag bit access.

A.8 `qemu_ramdesc_t`

This structure represents the memory map of a QSim CPU or OS domain. It contains pointers to blocks allocated for each of three memory regions and values representing their size. It will contain at least the following fields:

```
    typedef struct {
      uint8_t *low_mem_ptr;
      size_t   low_mem_sz;
      uint8_t *below_4g_ptr;
      size_t   below_4g_sz;
      uint8_t *above_4g_ptr;
      size_t   above_4g_sz;
    } qsim_ramdesc_t;
```

These can be used to translate host addresses to guest physical addresses and vice-versa. “Low memory” is RAM below 640k; “Below 4g” is RAM starting at 1M, and “above 4g” is RAM starting at 4G, available only with PAE or 64-bit addressing.