USER'S MANUAL

PAAS
FPGAs/CPUs Heterogeneous System Simulator
Based on GEM5

Reconfigurable Computing System Lab, HKUST

Tingyuan LIANG

Oct, 2017

Revision Sheet

Release No.	Date	Revision Description
Rev. 0	13/11/16	User’s Manual
Rev. 1	02/10/17	User’s Manual

1.0 GENERAL INFORMATION

1.0 GENERAL INFORMATION

1.1 System Overview

- System Name

  PAAS: Processor Accelerator Architecture Simulator

- Major Functions

  FPGAs/CPUs Heterogeneous System Simulation

- User Access Mode

  Users are supposed to used python scripts and shell scripts to describe the feature of the model and deliver command or parameter to it. The workloads of CPUs are set by user's scripts or by operation system in full-system mode. The function of FPGAs is based on the binary file generated from verilog by verilator.

- Responsible Organization

  Reconfigurable Computing System Lab, HKUST

- System Category:

  – *Major Application*

    Evaluate the function and efficiency of FPGAs/CPUs heterogeneous system, to help system designer to test different architectures, improve their code or find appropriate parameters.

  – *General Support System*

    None but Email for Questioning: zjueelty@zju.edu.cn

- Operational Status

  – Under development and a Beta version released

- General Description

  The FPGA/CPUs Heterogeneous Simulator System is intended to help designer to evaluate the function and efficiency of their projects in which CPUs co-operate with FPGAs. Moreover, the system is based on the latest version of GEM5 so that it can support the simulation of both full-system mode and syscall-emulation mode. As for architecture, both Ruby memory system and classic memory system are included and users can design the topology or the architecture they need. The part for full-system simulation with ruby memory system is still limited.

- System Environment and Prerequisites

  Linux, Verilator, g++, Python

1.2 Project References

1. GEM5 Tutorial: http://pages.cs.wisc.edu/~david/courses/cs752/Spring2015/gem5-tutorial/index.html
2. GEM5 Class List: http://www.gem5.org/docs/html/annotated.html
3. GEM5 Main Page: http://gem5.org/Main_Page
4. Verilator Wiki: http://www.veripool.org/wiki/verilator
5. TLB Class Reference: http://www.gem5.org/docs/html/classX86ISA_1_1TLB.html
6. Linux Driver Tutorial: http://freesoftwaremagazine.com/articles/drivers_linux/

1.3 Authorized Use Permission

As the system is still under development, please feel free to use this system for academic purposes but do not use this system for any commercial purposes. Moreover, this system is based on GEM5 so that users are supposed to limit their utilization of original source code generated by GEM5 group, according to GEM5 copyright notice.

1.4 Points of Contact

1.4.1 Information

Published Paper: PAAS: A System Level Simulator for Heterogeneous Computing Architectures
@ FPL 2017

Author of PAAS :
Tingyuan LIANG, Liang FENG, Sharad Sinha, Wei ZHANG

Contact Name: Tingyuan LIANG
Department: Department of Electronic and Computer Engineering
E-mail Address: tliang@connect.ust.hk

1.4.2 Coordination

None

1.4.3 Help Desk

None

1.5 Organization of the Manual

Section1: This section explains in general terms the system and the purpose for which it is intended.

Section2: This section provides a general overview of the system written in non-technical terminology. The summary outlines the uses of the system in supporting the activities of the user and staff.

Section3: This section provides a general walk-through of the system from input and initiation to output and exit. The logical arrangement will help user to understand the sequence and flow of the system.

Section4: This section provides a detailed description about how to use the system in Ruby memory or Classic memory. Moreover, this section help you understand how the FPGA module communicate with CPU and memory.

Section5: This section demonstrates how to equip FPGA module with some unique features, like DMA and reconfigurability.

Section6: This section mainly help user build a simulation in full-system mode with GEM5, from how to make a driver to how to control a FPGA device in operation system. Moreover, this section will explain the details of address translation in this simulator.

Section7: Acknowledgements

1.6 Acronyms and Abbreviations

None

2.0 SYSTEM SUMMARY

2.0 SYSTEM SUMMARY

This section provides a general overview of PAAS written in non-technical terminology.

2.1 System Configuration

PAAS is mainly the combination of GEM5 and Verilator. GEM5 is well designed to simulate the system of CPU, memory and network and, in this simulator, GEM5 helps the module of FPGA get data from memory and communicate with CPU via bus. Verilator is used to compile synthesizable Verilog and generate a executable file which implements the function of FPGA. In this combination, two parts communates with each other via shared memory.

2.2 Data Flows

- User Input

  – *User describes the configuration of the heterogeneous system which will be simulated in Python scripts.*

  – *User describes the function of FPGA in Verilog and compile it into an executable by Verilator.*

- GEM5 Simuation

  According to configuration scripts, GEM5 simulates most parts of heterogeneous system except FPGA and sends data to the executable generated by Verilator.

- Verilator Simulation

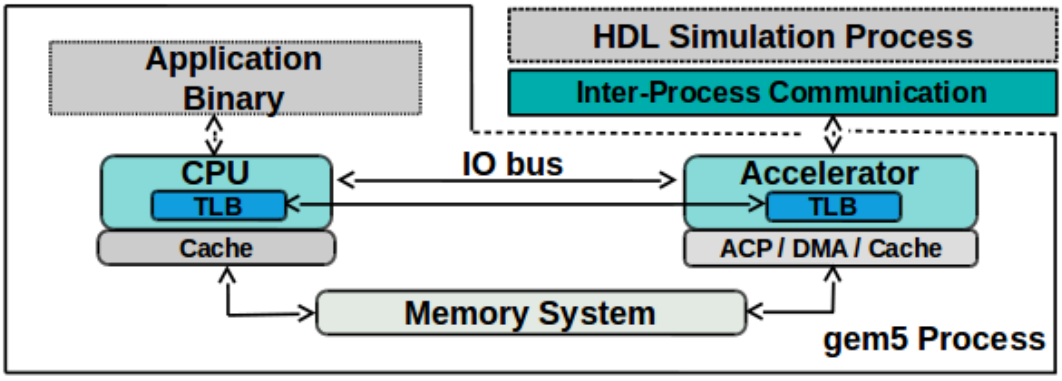
  According to data from the process of GEM5, the process of FPGA will implement the function of FPGA and send the output of FPGA back to the process of GEM5.

- Simulator Output

  User will get the output of the simulator if there is any output generated by Verilog, binary or system. Moreover, user can get the statistic from m5out.

2.3 Graphic Depiction

The following figure helps user to understand the configuration and data flows of the PAAS.



3.0 GETTING STARTED

3.0 GETTING STARTED

This section provides a general walk-through of the system from input and initiation to output and exit. The logical arrangement will help user to understand the sequence and flow of the system. As first step, user should download PAAS and Verlilator and then build them. After that, user can use Verilator to translate Verilog to C++ and build a wrapper for it. As next step, user is supposed to configure the GEM5 and specified the workload of it. Finally, simulate the system.

3.1 Download and Build

The following part will explain how to download and install this simulator. Although there is a tutorial for GEM5 online, you are strongly suggested to follow the following instructions due to the reason that there are some detailed mistakes in the out-of-date tutorial. Moreover, there are some changes in the source code of GEM5 for purpose of adding FPGA module so that there are also some changes in the configuration of GEM5.

3.1.1 GEM5

—— Download

To download PAAS, you can type the following command in the terminal of Linux:

```
git clone https://github.com/zslwyuan/PAAS_V1.0
```

—— Install

a) Before Installation, you should have all dependencies installed (please click and check carefully: http://gem5.org/Dependencies).

b) Change your working directory to /gem5

c) Execute this command line to build GEM5, please note that the configuration of PROTOCOL has to be modified to adapt to FPGA, however, only MOESI_CMP_token and MESI_Two_Level are modified, if you want to implement FPGA in other protocols, you can modify the responding procotol in the directory configs/ruby/ easily and MOESI_CMP_token.py and MESI_Two_Level.py will be good references(Attention: MOESI_CMP_token does not support multi-cpu fullsystem well) :

```
scons CPU_MODELS="AtomicSimpleCPU,MinorCPU,O3CPU,TimingSimpleCPU,FpgaCPU"
PROTOCOL=”MOESI_CMP_token” build/X86/gem5.opt
```

d) Create a new directory named *tutorial* in /*gem5*/*configs*/ and create a new file named *simple.py* in this directory. The content of *simple.py* is below.

```
import m5
from m5.objects import *
system = System()
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '1GHz'
system.clk_domain.voltage_domain = VoltageDomain()
system.mem_mode = 'timing'              # Use timing accesses
system.mem_ranges = [AddrRange('512MB')] # Create an address range
system.cpu = TimingSimpleCPU()
system.membus = SystemXBar()
```

```
system.cpu.icache_port = system.membus.slave
system.cpu.dcache_port = system.membus.slave
system.cpu.createInterruptController()
if m5.defines.buildEnv['TARGET_ISA'] == "x86":
    system.cpu.interrupts[0].pio = system.membus.master
    system.cpu.interrupts[0].int_master = system.membus.slave
    system.cpu.interrupts[0].int_slave = system.membus.master
system.mem_ctrl = DDR3_1600_8x8()
system.mem_ctrl.range = system.mem_ranges[0]
system.mem_ctrl.port = system.membus.master
system.system_port = system.membus.slave
isa = str(m5.defines.buildEnv['TARGET_ISA']).lower()
binary = 'tests/test-progs/hello/bin/' + isa + '/linux/hello'
process = Process()
process.cmd = [binary]
system.cpu.workload = process
system.cpu.createThreads()
root = Root(full_system = False, system = system)
m5.instantiate()
print "Beginning simulation!"
exit_event = m5.simulate()
print 'Exiting @ tick %i because %s' % (m5.curTick(), exit_event.getCause())
```

e) Test your GEM5 by execute the following command line.

```
build/X86/gem5.opt configs/tutorial/simple.py
```

And the output should be :

```
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 compiled Jan 14 2015 16:11:34
gem5 started Jan 15 2015 11:27:01
gem5 executing on mustardseed.cs.wisc.edu
command line: build/X86_MESI_Two_Level/gem5.opt configs/tutorial/simple.py
Global frequency set at 1000000000000 ticks per second
warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote_gdb.listener: listening for remote gdb #0 on port 7000
Begining simulation!
info: Entering event queue @ 0.  Starting simulation...
Hello world!
Exiting @ tick 345518000 because target called exit()
```

3.1.2 Verilator

For your convenience, the follow commands is a easy way to build Verilator from Git. If you fail to download Verilator, please try this link (http://www.veripool.org/ftp/verilator-3.888.tgz).

```
git clone http://git.veripool.org/git/verilator   # Only first time
unsetenv VERILATOR_ROOT  # For csh; ignore error if on bash
unset VERILATOR_ROOT  # For bash
cd verilator
git pull        # Make sure we're up-to-date
git tag         # See what versions exist
autoconf        # Create ./configure script
./configure
make & sudo make install
```

3.2 Translate Verilog

This section describes how to generate an executable file that can be added in to GEM5 as FPGA module.

3.2.1 Generate an Executable File of FPGA Module

Verilator will generate an executable file from a .cpp file and a .v file. The .cpp file act as the simulator of the .v file. For example, the following command will implement this function.

```
verilator -Wall --cc our.v --exe sim_main.cpp && make -j -C obj_dir -f Vour.mk Vour
```

In this command, *sim_main.cpp* is the simulator of *our.v* and the executable file Vour generated in diretory /obj_dir can implement the function of FPGA. More details can be found in the manual of Verilator (S*ection 6 EXAMPLE C++ EXECUTION*). When the executable file is generated, you should rename it *FpgaModule* and copy it into the directory */gem5/fpga*

Please pay attention to the permission of the executable files if you download the PAAS from Github!!!! It might lead to a fatal error call "fpga process cannot work!"

3.2.2 Content of Verilog Code and sim_main.cpp

In order to let the executable file, the FPGA module, communicate with the GEM5 module, there should be particular ports in Verilog code and functions of sharing memory in sim_main.cpp. For this consideration, here are the templates of the *.v* file and *.cpp* file and you can find them easily in the directory */gem5/fpga* .

Content of our.v

```
module our (
  input wire clk,

  input wire reset,
  input wire[63:0] read_base,
  input wire[63:0] write_base,
  input wire[63:0] num_read,
  input wire[63:0] read_size_input,
  input wire[63:0] read_ready,
  input wire[63:0] write_ready,
  input wire[31:0] read_data,
  output wire read_enable,
  output wire write_enable,
  output wire finish_read,
  output wire finish_write,
  output wire done,
  output wire[63:0] read_addr,
  output wire[63:0] write_addr,
  output wire[63:0] write_size,
  output wire[63:0] read_size_output,
  output wire[31:0] write_data,
  output wire[31:0] returnvalue
);
```

```
parameter ADDR_WID = 13;
parameter DATA_WID = 32;
reg[DATA_WID-1:0] r_data[8192-1:0];
reg[63:0] read_cnt;
reg[63:0] write_cnt;
reg[63:0] state;
reg r_read_enable;
reg r_write_enable;
reg r_finish_read;
reg r_finish_write;
reg r_done;
reg[63:0] r_read_addr;
reg[63:0] r_write_addr;
reg[63:0] r_write_size;
reg[63:0] r_read_size_output;
reg[31:0] r_write_data;
reg[31:0] r_returnvalue;
assign returnvalue=r_returnvalue;
assign read_enable=r_read_enable;
assign write_enable=r_write_enable;
assign finish_read=r_finish_read;
assign finish_write=r_finish_write;
assign read_addr=r_read_addr;
assign write_addr=r_write_addr;
assign write_size=r_write_size;
assign read_size_output=r_read_size_output;
assign write_data=r_write_data;
assign done=r_done;


parameter IDLE = 64'd0;
parameter READY_READ = 64'd1;
parameter WAIT_READ = 64'd2;
parameter DEAL_READ = 64'd3;
parameter FINISH = 64'd4;
parameter READY_WRITE = 64'd5;
parameter WAIT_WRITE = 64'd6;
parameter DEAL_WRITE = 64'd7;
parameter SUSPEND = 64'd8;
parameter LOOP = 64'd9;
parameter HANDLING = 64'd10;
parameter INIT = 64'd11;
parameter ALL_READY = 64'd12;
parameter SEND = 64'd13;
parameter ALL_PROC = 64'd14;
parameter BACK = 64'd15;


wire[ADDR_WID-1:0] addr0,addr1;
wire ce0,we0,ce1,we1;

reg r_next;
wire next=r_next;
wire next_out;
wire[DATA_WID-1:0] q0,q1,d0,d1,ret;
reg[DATA_WID-1:0] r_q0,r_q1;
assign q0=r_q0;
```

```
        assign q1=r_q1;
        reg [63:0] timer;

kernel_2mm mm(
    .ap_clk(clk),
    .ap_rst(reset),
    .ap_start(next),
    .ap_done(next_out),
    .ap_idle(),
    .ap_ready(),
    .indata_address0(addr0),
    .indata_ce0(ce0),
                .indata_we0(we0),
                .indata_d0(d0),
    .indata_q0(q0),
    .indata_address1(addr1),
    .indata_ce1(ce1),
                .indata_we1(we1),
                .indata_d1(d1),
    .indata_q1(q1)
);

integer i;

   always @(posedge clk)
   begin
                        //$display("LOOP");
                    if (reset)
                    begin
                       state<=INIT;
                       read_cnt<=0;
                       write_cnt<=0;
                       r_read_enable<=0;
                       r_write_enable<=0;
                       r_finish_read<=0;
                       r_finish_write<=0;
                       r_read_addr<=0;
                       r_write_addr<=0;
                       r_write_size<=0;
                       r_read_size_output<=0;
                       r_done<=0;
                       r_next<=0;
                       timer<=0;
                    end
                    else
                    begin
                        timer<=timer+1;
                        if (state == INIT)
                        begin
                        //          $display("INIT\n");
                                    state<=IDLE;
                        end
                        else if (state == IDLE)//START
                        begin
                                        //$display("IDLE %d\n",read_base);
                        state<=WAIT_READ;
                        r_read_addr<=read_base;
                        r_read_size_output<=read_size_input;
```

```
                    r_read_enable<=1;
                end
                else if (state == WAIT_READ)
                begin
                    //$write("WAIT_READ");
                    r_finish_read<=0;
                    if (read_ready == 1)
                    begin
                            r_data[read_cnt[ADDR_WID-1:0]]<=read_data;
                            if (read_cnt + 1 < num_read)
                            begin
                                read_cnt<=read_cnt+1;
                                r_read_addr<=r_read_addr+read_size_input;
                                state<=WAIT_READ;
                                r_finish_read<=1;
                            end
                            else
                            begin
                                r_read_enable<=0;
                                r_finish_read<=0;
                                r_next<=1;
                                read_cnt<=0;
                                state<=LOOP;
                                $display("time1: %d",timer);
                            end
                    end
                end
////////////////////////////////////////////////////
                else if (state == LOOP)
                begin
                    //$display("LOOP");
                    r_next<=0;
                    if (ce0)
                    begin
                            if (we0)
                            begin
                                r_data[addr0]<=d0;
                            end
                            else
                            begin
                                r_q0<=r_data[addr0];
                            end
                    end
                    if (ce1)
                    begin
                            if (we1)
                            begin
                                r_data[addr1]<=d1;
                            end
                            else
                            begin
                                r_q1<=r_data[addr1];
                            end
                    end
                    if (next_out)
                    begin
                            state<=READY_WRITE;
                            r_returnvalue<=ret;
```

```
                                $display("time2: %d",timer);
                            end
                        end
////////////////////////////////////////////////////
                        else if (state == READY_WRITE)
                        begin
                            //$display("READY_WRITE");
                            write_cnt<=0;
                            r_write_addr<=write_base;
                            r_write_size<=read_size_input;
                            r_write_enable<=1;
                            state<=WAIT_WRITE;
                        end
                        else if (state == WAIT_WRITE)
                        begin
                            //$display("WAIT_WRITE\n");
                            r_finish_write<=0;
                            if (write_ready == 1)
                            begin
                                    if (write_cnt + 1 < num_read)
                                    begin
                                        r_finish_write<=1;
                                        write_cnt<=write_cnt+1;
                                        r_write_data<=r_data[write_cnt[ADDR_WID-1:0]+1];
                                        r_write_addr<=r_write_addr+write_size;
                                        state<=WAIT_WRITE;
                                    end
                                    else
                                    begin
                                        r_finish_write<=0;
                                        r_write_enable<=0;
                                        state<=SUSPEND;
                                        r_done<=1;
                                        $display("time3: %d",timer);
                                    end
                              end
                        end
                        else if (state == SUSPEND)
                        begin
                                    read_cnt<=0;
                                    write_cnt<=0;
                                    r_read_enable<=0;
                                    r_write_enable<=0;
                                    r_finish_read<=0;
                                    r_finish_write<=0;
                                    r_read_addr<=0;
                                    r_write_addr<=0;
                                    r_write_size<=0;
                                    r_read_size_output<=0;
                                    r_done<=0;
                                    r_next<=0;
                        end
                    end
   end
endmodule
```

Content of sim_main.cpp

```
#include<iostream>
#include "Vour.h"
#include "verilated.h"
#include<stdio.h>
#include<ctime>
#include <string>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<fcntl.h>
#include<limits.h>
#include<sys/stat.h>
#include<sys/types.h>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <sys/shm.h>
#include <signal.h>
using namespace std;
Vour *our;

#define TEXT_SZ 20

int running = 1;
void *shm = NULL;
struct shared_use_st *shared;
int shmid,cnt;
int num_input_fpga,num_output_fpga;
int sharedid;
struct shared_use_st
{
   int written;
   unsigned long long text[TEXT_SZ];
};

void createShare()
{
   num_input_fpga = 13;
   num_output_fpga = 15;
   shmid = shmget((key_t)sharedid, sizeof(struct shared_use_st), 0666|IPC_CREAT);
   if(shmid == -1)
   {
      fprintf(stderr, "shmget failed\n");
      exit(EXIT_FAILURE);
   }
   shm = shmat(shmid, 0, 0);
   if(shm == (void*)-1)
   {
      fprintf(stderr, "shmat failed\n");
      exit(EXIT_FAILURE);
   }
   printf("\nMemory attached at %llX\n", (unsigned long long)shm);
```

```
            shared = (struct shared_use_st*)shm;
    shared->written = 0;
}

void deleteShare()
{
    if(shmdt(shm) == -1)
    {
        fprintf(stderr, "shmdt failed\n");
        exit(EXIT_FAILURE);
    }
    if(shmctl(shmid, IPC_RMID, 0) == -1)
    {
        fprintf(stderr, "shmctl(IPC_RMID) failed\n");
        exit(EXIT_FAILURE);
    }
}

int main(int argc, char** argv)
{
    int ppid=getppid();
            FILE *t0 = fopen("fpga_share.txt","r");
            fscanf(t0,"%d",&sharedid);fclose(t0);t0 = fopen("fpga_share.txt","w");fclose(t0);
            printf("ppid = %d\n",ppid);
            int cpu_running;
    static const int //FPGA Output
                        bit_ReturnValue = 4,
        bit_ReadEnable = 5,
        bit_WriteEnable = 6,
        bit_FinishRead = 7,
        bit_FinishWrite = 8,
        bit_ReadAddr = 9,
        bit_Size_ReadData_Output = 10,
        bit_WriteAddr = 11,
        bit_Size_WriteData = 12,
        bit_WriteData = 13,
        bit_Done = 14;
    static const int //FPGA Input
        bit_ReadReady = 5,
        bit_WriteReady = 6,
        bit_ReadData = 7,
        bit_ReadBase = 8,
        bit_WriteBase = 9,
        bit_Num_Read = 10,
        bit_Size_ReadData_Input = 11,
        bit_Run = 12;

    Verilated::commandArgs(argc, argv);

    our = new Vour;
    our->reset = 1;
    our->clk=0;
    int sum,clk,x,y,reset;
    createShare();
    unsigned long long *inputArray = &shared->text[0];
    unsigned long long *outputArray = &shared->text[num_input_fpga];
            shared->written = 0;
            shared->written = 0;
```

```
        shared->written = 0;
char *buffer;
if((buffer = getcwd(NULL, 0)) == NULL)
{
   perror("getcwd error");
}
else
{
   printf("%s\n", buffer);
   free(buffer);
}
while(running)
{
                cpu_running = (shared->written==0);
                //printf("cpu_running:%d\n",cpu_running);
                if (kill(ppid,0)<0) break;
                if (!cpu_running)
                {
                   if(shared->text[num_input_fpga+num_output_fpga] == 10101)
                   {
                      shared->written = 0;
                      running = 0;
                      break;
                   }
                   //OUTPUT
                   outputArray[bit_ReadEnable] = our->read_enable;
                   outputArray[bit_WriteEnable] = our->write_enable;
                   outputArray[bit_FinishRead] = our->finish_read;
                   outputArray[bit_FinishWrite] = our->finish_write;
                   outputArray[bit_ReadAddr] = our->read_addr;
                   outputArray[bit_WriteAddr] = our->write_addr;
                   outputArray[bit_Done] = our->done;
                   outputArray[bit_Size_WriteData] = our->write_size;
                   outputArray[bit_WriteData] = our->write_data;
                   outputArray[bit_Size_ReadData_Output] = our->read_size_output;
                         outputArray[bit_ReturnValue] = our->returnvalue;
                   bool Vfinish=!Verilated::gotFinish();
                   if(Vfinish)
                   {
                      our->clk=1;
                      our->eval();
                      //INPUT
                      our->read_base = inputArray[bit_ReadBase];
                      our->write_base = inputArray[bit_WriteBase];
                      our->num_read = inputArray[bit_Num_Read];
                      our->read_size_input = inputArray[bit_Size_ReadData_Input];
                      our->read_ready = inputArray[bit_ReadReady];
                      our->write_ready = inputArray[bit_WriteReady];
                      our->read_data = inputArray[bit_ReadData];
                      our->clk=0;
                      our->eval();
                   }
                   else break;
                   if (shared->written == 1)  shared->written = 0;
                }

}
our->final();
```

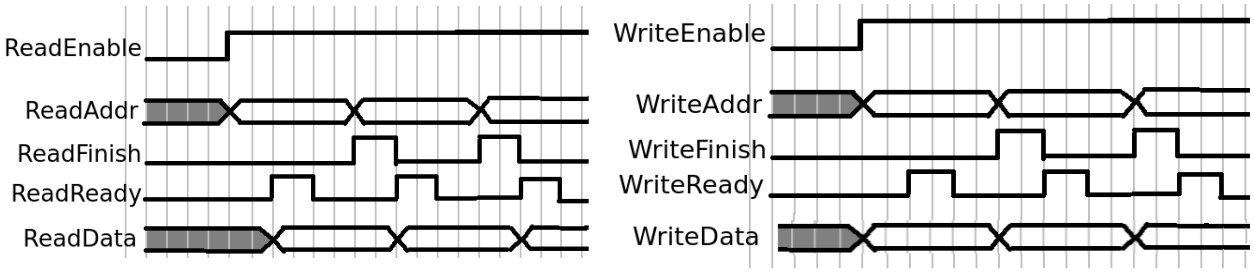
```
    delete our;
    exit(EXIT_SUCCESS);
}
```

3.2.3 Functions and States of FPGA Module

As you notice, the FPGA work in a FSM mode and the state *HANDLING* is to implement the particular function you want.

The FPGA get data from memory one after another and send data back to memory one after another after processing. If you want to change the FSM more than only the part of HANDLING, you are suggested to figure out how the FPGA module exchange data with memory in this heterogeneous system simulator. In order to help you understand the way in which FPGA accesses memory easier, the timing and the usage of various ports are provided below.

PortName	Function	I/O	PortName	Function	I/O
ReadEnable	FPGA Read	O	WriteEnable	FPGA Write	O
ReadAddr	VA of ReadData	O	WriteAddr	VA of WriteData	O
ReadFinish	FPGA gets ReadResponse	O	WriteFinish	FPGA gets WriteResponse	O
ReadReady	ReadResponse Ready	I	WriteReady	WriteResponse Ready	I
ReadData	Data from Memory	I	WriteData	Data to Memory	O



You can also add, change or remove the ports of FPGA easily if you modify several corresponding parts in *fpgacpu.cc* and *fpgacpu.hh*. These will be explained in detail later in this manual.

3.3 Configure GEM5

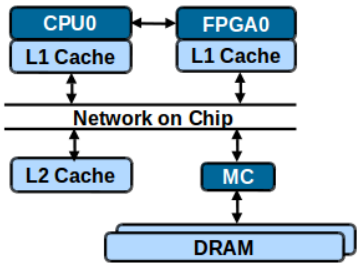
Because you want to add FPGA module(s) in GEM5, so you are supposed to configure the heterogeneous system in Python scripts. The following python script *2mm_fpga.py* set 1 CPU and 1 FPGA with ruby memory system. Please note that gem5 uses a port abstraction to connect memory system components together. Like "*system.cpu[i].icache_port = ruby_port.slave*", you can connect two

ports. By typing command: “./build/X86/gem5.opt configs/tuturial/2mm-fpga.py --ruby --num-cpus=1 –num-fpgas=1” , you can run the simulation for this example.

```
import optparse
import sys
import os
import m5
from m5.defines import buildEnv
from m5.objects import *
from m5.util import addToPath, fatal
addToPath('../')
from ruby import Ruby
from common import Options
from common import Simulation
from common import CacheConfig
from common import CpuConfig
from common import MemConfig
from common.Caches import *
from common.cpu2000 import *
parser = optparse.OptionParser()
Options.addCommonOptions(parser)
Options.addSEOptions(parser)

if '--ruby' in sys.argv:
    Ruby.define_options(parser)
else:
    fatal("This test is only for FPGA in Ruby. Please set --ruby.\n")

(options, args) = parser.parse_args()

if args:
    print "Error: script doesn't take any positional arguments"
    sys.exit(1)

numThreads = 1

process1 = LiveProcess()
process1.pid = 1100;
process1.cmd = ['tests/test-progs/polybench-c-4.2/2mm-fpga']

(CPUClass, test_mem_mode, FutureClass) = Simulation.setCPUClass(options)
CPUClass.numThreads = numThreads

np = options.num_cpus
system = System(cpu = [DerivO3CPU() for i in xrange(np)],
#system = System(cpu = [TimingSimpleCPU() for i in xrange(np)],
                mem_mode = 'timing',
                mem_ranges = [AddrRange('512MB')],
                cache_line_size = 64)

system.fpga = [FpgaCPU() for i in xrange(options.num_fpgas)]
system.voltage_domain = VoltageDomain(voltage = options.sys_voltage)
system.clk_domain = SrcClockDomain(clock =  options.sys_clock,
                          voltage_domain = system.voltage_domain)
system.cpu_voltage_domain = VoltageDomain()
system.cpu_clk_domain = SrcClockDomain(clock = options.cpu_clock,
                        voltage_domain =  system.cpu_voltage_domain)
```

```
for cpu in system.cpu:
    cpu.clk_domain = SrcClockDomain(clock = options.cpu_clock,
                          voltage_domain =  system.cpu_voltage_domain)

system.fpga[0].clk_domain = SrcClockDomain(clock = options.fpga_clock)
system.fpga[0].clk_domain.voltage_domain = VoltageDomain()
system.fpga[0].fpga_bus_addr = 1073741824*2
system.fpga[0].size_control_fpga = 29*8
system.fpga[0].ModuleName = '2mm/obj_dir/Vour'

system.cpu[0].workload = process1
system.cpu[0].createThreads()
system.piobus = IOXBar()

if options.ruby:
    if options.cpu_type == "atomic" or options.cpu_type == "AtomicSimpleCPU":
        print >> sys.stderr, "Ruby does not work with atomic cpu!!"
        sys.exit(1)
    Ruby.create_system(options, False, system, system.piobus)

    system.ruby.clk_domain = SrcClockDomain(clock = options.ruby_clock,
                                    voltage_domain = system.voltage_domain)
    for i in xrange(np):
        print len(system.ruby._cpu_ports)
        ruby_port = system.ruby._cpu_ports[i]

        system.cpu[i].createInterruptController()

        system.cpu[i].icache_port = ruby_port.slave
        system.cpu[i].dcache_port = ruby_port.slave
        if buildEnv['TARGET_ISA'] == "x86":
            system.cpu[i].interrupts[0].pio = ruby_port.master
            system.cpu[i].interrupts[0].int_master = ruby_port.slave
            system.cpu[i].interrupts[0].int_slave = ruby_port.master
        system.cpu[i].itb.walker.port = ruby_port.slave
        system.cpu[i].dtb.walker.port = ruby_port.slave

    for i in xrange(options.num_fpgas):
        ruby_port = system.ruby._cpu_ports[i+np]
        system.fpga[i].icache_port = ruby_port.slave
        system.fpga[i].dcache_port = ruby_port.slave
        system.fpga[i].itb.walker.port = ruby_port.slave
        system.fpga[i].dtb.walker.port = ruby_port.slave
        system.fpga[i].control_port = system.piobus.master

root = Root(full_system = False, system = system)
m5.instantiate()
print "Beginning simulation!"
exit_event = m5.simulate()
print 'Exiting @ tick %i because %s' % (m5.curTick(), exit_event.getCause())
```

The description file above is for the architecture shown below:



3.4 Run PAAS

Before you run the simulator, let find out how CPU control the FPGA module. You should take a look in a part of the source code of FPGA module(*/gem5/src/cpu/fpgacpu/fpgacpu.cc*) and the binary run on CPUs.

Source Code of FPGA module	Explanation

Addr offset = (pkt->getAddr() - ControlAddr)>>3;
uint64_t reg = offset;
uint64_t val = htog(getFPGAReg(reg));
if (pkt->isRead())
{
// printf("isRead\n");
pkt->setData((uint8_t *)&val);
pkt->makeResponse();
}
else if (pkt->isWrite())
{
//printf("isWrite\n");
pkt->writeData((uint8_t *)&val);
setFPGAReg(reg,val,pkt);
pkt->makeResponse();
}
return latency;
	When there is a packet from CPU sending to a range of memory start from ControlAddr, the FPGA wrapper will get it.
	
	When FPGA receive a packet from bus or network, it will set or get the registers on FPGA and in this way, CPU can control FPGA.
	
	The register ControlAddr can be set in configuration file *BaseCPU.py*.

void
FpgaCPU::setFPGAReg(uint64_t regid, uint64_t val, PacketPtr pkt)
{
switch (regid)
{
case 0: if (!OccupyFPGA)TaskHash = val; break;
// For Control Consistency
case 1: ReadBase = val;inputArray[bit_ReadBase] = val; break;
case 2: WriteBase = val;inputArray[bit_WriteBase] = val; break;
case 3: CurrentThreadID = pkt->req->contextId();
break;
case 4: MemoryRange = val; inputArray[bit_Num_Read]=val;break;
case 5: MemorySize = val;
inputArray[bit_Size_ReadData_Input] = val;
break;
case 6: RunState = val;inputArray[bit_Run] = (val==0); break;
	Register ReadBase represents the base virtual address where FPGA get date from memory.
	
	Register WriteBase represents the base virtual address where FPGA send date to memory.
	
	Register CurrentThreadID lets FPGA know which thread call it.
	
	Register RunState shows the state of FPGA.
	
	Register Terminate can shutdown FPGA and end the simulation.

case 7: Terminate = val;break;
case 8: OccupyFPGA = val; break;
}
}
	Register OccupyFPGA let system know whether FPGA is occupied by someone.

Then, according to the fundamental of communication between CPU and FPGA, you can program like the file below. There are some hints in comments.

```
//the source code of algo-fpga, in which FPGA can accelerate the revolution.
#include <new>
#include <iostream>
#include <string>
#include <cstdio>
#include <pthread.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>

unsigned long long p1[401];//FPGA Read Area
unsigned long long p2[401];//FPGA Write Area

using namespace std;
int main()
{
        int i;
        unsigned long long* p0 = new((unsigned long long *)0xc0000000) unsigned long long[10];//Contro Port
        //TLB will directly translate the virtual address starting from 0xc0000000 to the physical address of I/O
        //ports of FPGA.
        While(1)
        {
                while(p0[8]);
                p0[0] = getpid()*getpid();
                p0[8] = 1;//Try to Occupy the FPGA
                if (p0[0] == getpid()*getpid()) break;//Occupy the FPGA Successfully
                // Two processes  might try to control the FPGA at the same time which might  turn
                // to the situation that two processes occupy the FPGA at the same time. These lines will prevent
                // the simulation from this error.
         }
        //unsigned long long* p0 = new unsigned long long[10];//Contro Port
        for (i=0;i<199;i++)  p1[i]=(unsigned long long)(i*171133321)%12391231;// generate FPGA input data
        p0[1] = (unsigned long long)p1;//ReadBase, let FPGA know where to read data
        p0[2] = (unsigned long long)p2;//WriteBase, let FPGA know where to write data
        p0[3] = getpid();//CurrentThreadID
        p0[4] = 200;//Memory Range
        p0[5] = 8;//MemorySize
        p0[7] = 0;//Terminat
        p0[6] = 0;//RunState
        p0[6] = 1;//Lauch FPGA
        while (p0[6]);//FPGA finish it work
        int n;
        for (n=0;n<198;n++)printf("%llu ",p2[n]);printf("\n");//print data
        p0[8] = 0;    //Release the FPGA
        return 0;
}
```

All the registers can be modified to meet user's requirement. After you finish all the works below, you can execute this command in the directory /gem5 to run the simulator with FPGA(s).

```
./build/X86/gem5.opt configs/tuturial/ruby_fpga.py --ruby --network=garnet2.0 --num-cpus=14 --num-fpgas=2 --num-dirs=8 --topology=Mesh_XY --mesh-rows=4 --num-l2caches=8
```

4.0 USING THE SIMULATOR (WITH DIFFERENT MEMORY SYSTEM)

4.0 USING THE SIMULATOR (WITH DIFFERENT MEMORY)

This section provides a detailed description about how to use the system in Ruby memory or Classic memory. Moreover, this section help you understand how the FPGA module communicate with CPU and memory.

4.1 Using The Simulator with Classic Memory System

In classic memory system, ports support direct point-to-point connections between two `MemObjects` and buses/crossbars connect two or more `MemObjects` together. Cache coherence is maintained using an abstract MOESI snooping protocol where state-transitions due to snoops occur instantaneously.

In order to act with classic memory system, you should modify your configuration file in Python properly. Although, there are some cases online, for the sake of the addition of FPGA, here is an example to help you.

4.1.1 Example

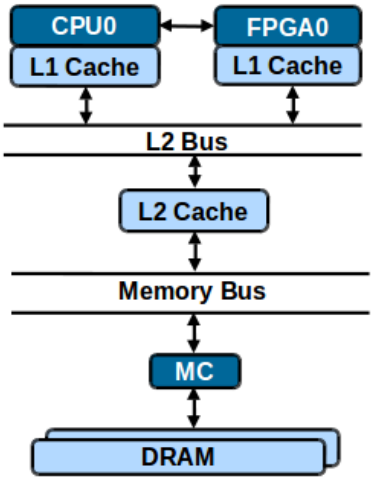
The following Python code build up a system based on classic memory. In this simulator, FPGA have the same status with CPU and the wrapper of FPGA in GEM5 is also derived from BaseSimpleCPU. Therefore, FPGA can be regarded as a special CPU here. This will be present in the following code.

```
# import the m5 (gem5) library created when gem5 is built
import m5
# import all of the SimObjects
from m5.objects import *

# Add the common scripts to our path
#m5.util.addToPath('../../')
m5.util.addToPath('../')
# import the caches which we made
from caches import *

# import the SimpleOpts module
from common import SimpleOpts

# Set the usage message to display
SimpleOpts.set_usage("usage: %prog [options] <binary to execute>")

# Finalize the arguments and grab the opts so we can pass it on to our objects
(opts, args) = SimpleOpts.parse_args()

# get ISA for the default binary to run. This is mostly for simple testing
isa = str(m5.defines.buildEnv['TARGET_ISA']).lower()

# Default to running 'hello', use the compiled ISA to find the binary
#binary = 'tests/test-progs/polybench-c-4.2/2mm_ref'
binary = 'tests/test-progs/polybench-c-4.2/2mm-fpga'
# Check if there was a binary passed in via the command line and error if
# there are too many arguments
if len(args) == 1:
```

```
    binary = args[0]
elif len(args) > 1:
    SimpleOpts.print_help()
    m5.fatal("Expected a binary to execute as positional argument")

# create the system we are going to simulate
system = System()

# Set the clock fequency of the system (and all of its children)
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '2GHz'
system.clk_domain.voltage_domain = VoltageDomain()


# Set up the system
system.mem_mode = 'timing'               # Use timing accesses
system.mem_ranges = [AddrRange('512MB')] # Create an address range

#system.piobus = IOXBar()
# Create a simple CPU
#system.cpu = DerivO3CPU()
system.cpu = DerivO3CPU()
system.fpga = FpgaCPU()
# Create an L1 instruction and data cache
system.cpu.icache = L1ICache(opts)
system.cpu.dcache = L1DCache(opts)
system.fpga[0].icache = L1ICache(opts)
system.fpga[0].dcache = L1DCache(opts)
# Connect the instruction and data caches to the CPU


system.l2bus = L2XBar()
system.l2bus.io_bypass = 1
system.l2bus.io_bypass_head = 0xc0000000
system.l2bus.io_bypass_tail = 0xc0000000+4096
system.cpu.icache.connectCPU(system.cpu)
system.cpu.dcache.connectCPU(system.cpu)
system.cpu.dcache.io_bypass = 1
system.cpu.dcache.io_bypass_head = 0xc0000000
system.cpu.dcache.io_bypass_tail = 0xc0000000+4096
system.fpga[0].icache.connectCPU(system.fpga[0])
system.fpga[0].dcache.connectCPU(system.fpga[0])
system.cpu.dtb.baseaddress_control_fpga = 0xc0000000
system.cpu.itb.baseaddress_control_fpga = 0xc0000000
system.cpu.dtb.fpga_bus_addr = 0xffff00000000
system.cpu.itb.fpga_bus_addr = 0xffff00000000

system.fpga[0].clk_domain = SrcClockDomain(clock='300MHz')
system.fpga[0].clk_domain.voltage_domain = VoltageDomain()
system.fpga[0].fpga_bus_addr = 0xffff00000000
system.fpga[0].ModuleName = './2mm/obj_dir/Vour'
system.fpga[0].control_port = system.l2bus.master
#4096

# Create a memory bus, a coherent crossbar, in this case


# Hook the CPU ports up to the l2bus
```

```
system.cpu.icache.connectBus(system.l2bus)
system.cpu.dcache.connectBus(system.l2bus)
system.fpga.icache.connectBus(system.l2bus)
system.fpga.dcache.connectBus(system.l2bus)
# Create an L2 cache and connect it to the l2bus
system.l2cache = L2Cache(opts)
system.l2cache.connectCPUSideBus(system.l2bus)

# Create a memory bus
system.membus = SystemXBar()

# Connect the L2 cache to the membus
system.l2cache.connectMemSideBus(system.membus)

# create the interrupt controller for the CPU
system.cpu.createInterruptController()

# For x86 only, make sure the interrupts are connected to the memory
# Note: these are directly connected to the memory bus and are not cached
if m5.defines.buildEnv['TARGET_ISA'] == "x86":
    system.cpu.interrupts[0].pio = system.membus.master
    system.cpu.interrupts[0].int_master = system.membus.slave
    system.cpu.interrupts[0].int_slave = system.membus.master


# Connect the system up to the membus
system.system_port = system.membus.slave

# Create a DDR3 memory controller
system.mem_ctrl = DDR3_1600_x64()
system.mem_ctrl.range = system.mem_ranges[0]
system.mem_ctrl.port = system.membus.master

# Create a process for a simple "Hello World" application
process = LiveProcess()
# Set the command
# cmd is a list which begins with the executable (like argv)
process.cmd = [binary]
# Set the cpu to use the process as its workload and create thread contexts
system.cpu.workload = process
system.cpu.createThreads()

# set up the root SimObject and start the simulation
root = Root(full_system = False, system = system)
# instantiate all of the objects we've created above
m5.instantiate()

print "Beginning simulation!"
exit_event = m5.simulate()
print 'Exiting @ tick %i because %s' % (m5.curTick(), exit_event.getCause())
```

Please note that as you can see, FPGA can be equipped with caches like what CPU can do. This description file describes the architecture shown below:



If you want to regard FPGA as a device in system, you can declare it separately (like *system.fpga = FpgaCPU()*).

4.1.2 Details in I/O Maping

There are two kinds of ports on FPGA and they are DataPort and ControlPort. DataPorts are connected to memory system while ControlPort are connected to CPU so that CPU can directly control FPGA via bus. When you write to special virtual address range in binary, the CPU will transfer the data to the physical address of I/O port of FPGA. The fundamental of address translation will be explained in detail in *Section 5* of this manual.

You can change these configuration by modifying the following part of code in both *TLB.py (in /gem5/src/arch/YOUR_ARCH/)* and *BaseCPU.py(in /gem5/src/cpu/)*.

Please note that the change in *TLB.py* will be invalidated in full-system mode because the operation system will manage the page table of memory instead of the processes. Therefore, in order to control FPGA by operation system, you have to insert a driver module in kernel and *Section 5* will help.

```
baseaddress_control_fpga = Param.UInt64(3221225472,"Base Virtual Address of FPGA Control Port")
size_control_fpga = Param.Unsigned(4095,"Size of FPGA Control Port")
fpga_bus_addr = Param.Unsigned(536870912,"Base Physical Address of of FPGA Control Port")
```

Also, as IO requests should not be cached by L1 or L2 cache, we need to let cache and bus bypass the IO requests to FPGA, by configuring the cache and bus as follow:

```
system.l2bus = L2XBar()
system.l2bus.io_bypass = 1
system.l2bus.io_bypass_head = 0xc0000000
system.l2bus.io_bypass_tail = 0xc0000000+4096
system.cpu.icache.connectCPU(system.cpu)
system.cpu.dcache.connectCPU(system.cpu)
system.cpu.dcache.io_bypass = 1
```

```
system.cpu.dcache.io_bypass_head = 0xc0000000
system.cpu.dcache.io_bypass_tail = 0xc0000000+4096
```

4.2 Using The Simulator with Ruby Memory System

Ruby memory system supports a domain specific language called SLICC, and ports only connect cpus and devices to the memory system via the `RubyPort` object. Moreover, all objects are connected to each other via `MessageBuffers, which` include a queue that stores messages. From the designer's perspective, the most important feature of Ruby is that Ruby accurately models both cache coherence and network related features in the memory system.

4.2.1 Example

The example can be found in *Section 3 Get Started*.

There are some significant differences in the example and they will be showed below.

The example declares FPGA module in different way in which FPGA module is seem relatively independent with CPU. However, this is just an example and you can still declare FPGA as a special CPU in Ruby.

```
system = System(cpu = [TimingSimpleCPU(), TimingSimpleCPU(), TimingSimpleCPU()], mem_mode = 'timing',
            mem_ranges = [AddrRange('512MB')], cache_line_size = 64)
system.fpga = [FpgaCPU()]
```

You can configure the clock source of FPGA by this way.

```
system.fpga[0].clk_domain = SrcClockDomain(clock = '300MHz')
system.fpga[0].clk_domain.voltage_domain = VoltageDomain()
```

Moreover, you are supposed to create a Ruby system when required.

```
Ruby.create_system(options, False, system, system.piobus)
```

Finally, you should connect FPGA to Ruby system.

```
  for i in xrange(options.num_fpgas):
     ruby_port = system.ruby._cpu_ports[i]
     system.fpga[i].icache_port = ruby_port.slave
     system.fpga[i].dcache_port = ruby_port.slave
     system.fpga[i].itb.walker.port = ruby_port.slave
     system.fpga[i].dtb.walker.port = ruby_port.slave
     system.fpga[i].control_port = system.piobus.master
```

The ruby_port is actually a sequencer which initiates memory requests and handles responses in Ruby memory system. More details of objects in Ruby can be found here (http://gem5.org/Ruby).

4.2.2 Detail of Memory Configuration for FPGA

Actually in order to equipped FPGA with cache and main memory in Ruby, there should be some change in the configuration of PROTOCOL. Take MOESI_CMP_token.py (*/gem5/configs/ruby/*)as example, you will find these lines for FPGA.

```
for i in xrange(options.num_fpgas):

    l1i_cache = L1Cache(size = options.l1i_size,
                        assoc = options.l1i_assoc,
                        start_index_bit = block_size_bits)
    l1d_cache = L1Cache(size = options.l1d_size,
                        assoc = options.l1d_assoc,
                        start_index_bit = block_size_bits)

    l1_cntrl = L1Cache_Controller(version=i+options.num_cpus, L1Icache=l1i_cache,
                                  L1Dcache=l1d_cache,
                                  l2_select_num_bits=l2_bits,
                                  N_tokens=n_tokens,
                                  retry_threshold=options.l1_retries,
                                  fixed_timeout_latency=\
                                  options.timeout_latency,
                                  dynamic_timeout_enabled=\
                                  not options.disable_dyn_timeouts,
                                  no_mig_atomic=not \
                                  options.allow_atomic_migration,
                                  send_evictions=send_evicts(options),
                                  transitions_per_cycle=options.ports,
                                  clk_domain=clk_domain,
                                  ruby_system=ruby_system)

    cpu_seq = RubySequencer(version=options.num_cpus+i, icache=l1i_cache,
                            dcache=l1d_cache, clk_domain=clk_domain,
                            ruby_system=ruby_system)

    l1_cntrl.sequencer = cpu_seq
    exec("ruby_system.l1_cntrl%d = l1_cntrl" % (i+options.num_cpus))

    # Add controllers and sequencers to the appropriate lists
    cpu_sequencers.append(cpu_seq)
    l1_cntrl_nodes.append(l1_cntrl)

    # Connect the L1 controllers and the network
    l1_cntrl.requestFromL1Cache = MessageBuffer()
    l1_cntrl.requestFromL1Cache.master = ruby_system.network.slave
    l1_cntrl.responseFromL1Cache = MessageBuffer()
    l1_cntrl.responseFromL1Cache.master = ruby_system.network.slave
    l1_cntrl.persistentFromL1Cache = MessageBuffer(ordered = True)
    l1_cntrl.persistentFromL1Cache.master = ruby_system.network.slave

    l1_cntrl.mandatoryQueue = MessageBuffer()
    l1_cntrl.requestToL1Cache = MessageBuffer()
    l1_cntrl.requestToL1Cache.slave = ruby_system.network.master
    l1_cntrl.responseToL1Cache = MessageBuffer()
    l1_cntrl.responseToL1Cache.slave = ruby_system.network.master
    l1_cntrl.persistentToL1Cache = MessageBuffer(ordered = True)
    l1_cntrl.persistentToL1Cache.slave = ruby_system.network.master
```

5.0 USING THE SIMULATOR (WITH FPGA FEATURES)

5.0 USING THE SIMULATOR (WITH FPGA FEATURES)

In practical application, FPGA can transfer data via DMA and also be reconfigured with a new kernel. In this section, we are going to talk about how to describe the system with DMA and how to reconfigure a FPGA.

5.1 DMA for FPGA

Usually, DMA is set up by CPU but to reduce the programming effort, we provide event-triggered DMA. When FPGA read the data from memory for the first time, the DMA controller will fill the scratchpad memory of FPGA and the following read requests will be handled by the scratchpad memory. As for FPGA to write data back to memory, a first write request will lead to the writing back of the scratchpad memory into main memory.

5.1.1 The Description File of FPGA with DMA

As shown below, to add DMA into the simulation, user need to enable the DMA option, set the size of DMA and connect
the DMA to particular bus so that the DMA engine can fetch data from memory.

```
system.fpga[0].dma_available = 1
system.fpga[0].dma_size = 5102*4
system.fpga[0].dma = system.membus.slave
```

A full description file of FPGA with DMA is shown below.

```
# import the m5 (gem5) library created when gem5 is built
import m5
# import all of the SimObjects
from m5.objects import *

# Add the common scripts to our path
#m5.util.addToPath('../../')
m5.util.addToPath('../')
# import the caches which we made
from caches import *

# import the SimpleOpts module
from common import SimpleOpts

# Set the usage message to display
SimpleOpts.set_usage("usage: %prog [options] <binary to execute>")

# Finalize the arguments and grab the opts so we can pass it on to our objects
(opts, args) = SimpleOpts.parse_args()

# get ISA for the default binary to run. This is mostly for simple testing
isa = str(m5.defines.buildEnv['TARGET_ISA']).lower()

# Default to running 'hello', use the compiled ISA to find the binary
#binary = 'tests/DMA/polybench-c-4.2/2mm_ref'
binary = 'tests/DMA/polybench-c-4.2/2mm-fpga'
```

```
# Check if there was a binary passed in via the command line and error if
# there are too many arguments
if len(args) == 1:
    binary = args[0]
elif len(args) > 1:
    SimpleOpts.print_help()
    m5.fatal("Expected a binary to execute as positional argument")

# create the system we are going to simulate
system = System()

# Set the clock fequency of the system (and all of its children)
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '2GHz'
system.clk_domain.voltage_domain = VoltageDomain()


# Set up the system
system.mem_mode = 'timing'               # Use timing accesses
system.mem_ranges = [AddrRange('512MB')] # Create an address range

#system.piobus = IOXBar()
# Create a simple CPU
system.cpu = TimingSimpleCPU()
system.fpga = FpgaCPU()
# Create an L1 instruction and data cache
system.cpu.icache = L1ICache(opts)
system.cpu.dcache = L1DCache(opts)
# Connect the instruction and data caches to the CPU

system.topbus = SystemXBar()
system.cpu.icache.connectCPU(system.cpu)
system.cpu.dcache.cpu_side = system.topbus.master
system.cpu.dcache_port = system.topbus.slave
system.cpu.dtb.baseaddress_control_fpga = 0xc0000000
system.cpu.itb.baseaddress_control_fpga = 0xc0000000
system.cpu.dtb.fpga_bus_addr = 0xffff00000000
system.cpu.itb.fpga_bus_addr = 0xffff00000000

system.fpga[0].clk_domain = SrcClockDomain(clock='300MHz')
system.fpga[0].clk_domain.voltage_domain = VoltageDomain()
system.fpga[0].fpga_bus_addr = 0xffff00000000
system.fpga[0].size_control_fpga = 29*8
system.fpga[0].ModuleName = '2mm/obj_dir/Vour'
system.fpga[0].control_port = system.topbus.master
system.fpga[0].noL1 = 0
system.fpga[0].dma_available = 1
system.fpga[0].dma_size = 5102*4


system.l2bus = L2XBar()

# Hook the CPU ports up to the l2bus
system.cpu.icache.connectBus(system.l2bus)
system.cpu.dcache.connectBus(system.l2bus)
# Create an L2 cache and connect it to the l2bus
system.l2cache = L2Cache(opts)
system.l2cache.connectCPUSideBus(system.l2bus)
```

```
# Create a memory bus
system.membus = SystemXBar()
system.membus.disenable_snoopFilter = 1
system.fpga[0].icache_port = system.membus.slave
system.fpga[0].dcache_port = system.membus.slave
# Connect the L2 cache to the membus
system.l2cache.connectMemSideBus(system.membus)
system.fpga[0].dma = system.membus.slave

# create the interrupt controller for the CPU
system.cpu.createInterruptController()

# For x86 only, make sure the interrupts are connected to the memory
# Note: these are directly connected to the memory bus and are not cached
if m5.defines.buildEnv['TARGET_ISA'] == "x86":
    system.cpu.interrupts[0].pio = system.membus.master
    system.cpu.interrupts[0].int_master = system.membus.slave
    system.cpu.interrupts[0].int_slave = system.membus.master

# Connect the system up to the membus
system.system_port = system.membus.slave

# Create a DDR3 memory controller
system.mem_ctrl = DDR3_1600_x64()
system.mem_ctrl.range = system.mem_ranges[0]
system.mem_ctrl.port = system.membus.master

# Create a process for a simple "Hello World" application
process = LiveProcess()
# Set the command
# cmd is a list which begins with the executable (like argv)
process.cmd = [binary]
# Set the cpu to use the process as its workload and create thread contexts
system.cpu.workload = process
system.cpu.createThreads()

# set up the root SimObject and start the simulation
root = Root(full_system = False, system = system)
# instantiate all of the objects we've created above
m5.instantiate()

print "Beginning simulation!"
exit_event = m5.simulate()
print 'Exiting @ tick %i because %s' % (m5.curTick(), exit_event.getCause())
```

5.1.2 Flush the Cache for DMA

Since we apply DMA in system, CPU need to handle the flush of cache to guarantee the data coherence.

5.2 Reconfiguration of FPGA

The most important feature of FPGA is reconfigurability. To meet the requirement, we implement the simulation of reconfiguration in PAAS. CPU can reconfigure the FPGA with new kernel to make different acceleration.

5.2.1 The Description File for Reconfiguration of FPGA

To make the FPGA model in PAAS is reconfigurable, user need to declare as follow:

```
system.fpga[0].ModuleName = './2mm/obj_dir/Vour;./nussinov/obj_dir/Vour'
system.fpga[0].Reconfigurable = 1
system.fpga[0].Reconfiguration_time = '1ms'
```

First, indicate the potential kernels for FPGA, then enable the reconfigurability of FPGA and define the latency of reconfiguration. The full description file as an example is shown below:

```
# import the m5 (gem5) library created when gem5 is built
import m5
# import all of the SimObjects
from m5.objects import *

# Add the common scripts to our path
#m5.util.addToPath('../../')
m5.util.addToPath('../')
# import the caches which we made
from caches import *

# import the SimpleOpts module
from common import SimpleOpts

# Set the usage message to display
SimpleOpts.set_usage("usage: %prog [options] <binary to execute>")

# Finalize the arguments and grab the opts so we can pass it on to our objects
(opts, args) = SimpleOpts.parse_args()

# get ISA for the default binary to run. This is mostly for simple testing
isa = str(m5.defines.buildEnv['TARGET_ISA']).lower()

# Default to running 'hello', use the compiled ISA to find the binary
#binary = 'tests/test-progs/polybench-c-4.2/2mm_ref'
binary = 'tests/test-progs/reconfiguration/2mm-fpga'
# Check if there was a binary passed in via the command line and error if
# there are too many arguments
if len(args) == 1:
    binary = args[0]
elif len(args) > 1:
    SimpleOpts.print_help()
    m5.fatal("Expected a binary to execute as positional argument")
```

```
# create the system we are going to simulate
system = System()

# Set the clock fequency of the system (and all of its children)
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '2GHz'
system.clk_domain.voltage_domain = VoltageDomain()


# Set up the system
system.mem_mode = 'timing'               # Use timing accesses
system.mem_ranges = [AddrRange('512MB')] # Create an address range

#system.piobus = IOXBar()
# Create a simple CPU
#system.cpu = DerivO3CPU()
system.cpu = DerivO3CPU()
system.fpga = FpgaCPU()
# Create an L1 instruction and data cache
system.cpu.icache = L1ICache(opts)
system.cpu.dcache = L1DCache(opts)
system.fpga[0].icache = L1ICache(opts)
system.fpga[0].dcache = L1DCache(opts)
# Connect the instruction and data caches to the CPU


system.l2bus = L2XBar()
system.l2bus.io_bypass = 1
system.l2bus.io_bypass_head = 0xc0000000
system.l2bus.io_bypass_tail = 0xc0000000+4096
system.cpu.icache.connectCPU(system.cpu)
system.cpu.dcache.connectCPU(system.cpu)
system.cpu.dcache.io_bypass = 1
system.cpu.dcache.io_bypass_head = 0xc0000000
system.cpu.dcache.io_bypass_tail = 0xc0000000+4096
system.fpga[0].icache.connectCPU(system.fpga[0])
system.fpga[0].dcache.connectCPU(system.fpga[0])
system.cpu.dtb.baseaddress_control_fpga = 0xc0000000
system.cpu.itb.baseaddress_control_fpga = 0xc0000000
system.cpu.dtb.fpga_bus_addr = 0xffff00000000
system.cpu.itb.fpga_bus_addr = 0xffff00000000

system.fpga[0].clk_domain = SrcClockDomain(clock='300MHz')
system.fpga[0].clk_domain.voltage_domain = VoltageDomain()
system.fpga[0].fpga_bus_addr = 0xffff00000000
system.fpga[0].ModuleName = './2mm/obj_dir/Vour;./nussinov/obj_dir/Vour'
system.fpga[0].Reconfigurable = 1
system.fpga[0].Reconfiguration_time = '1ms'
system.fpga[0].control_port = system.l2bus.master
#4096

# Create a memory bus, a coherent crossbar, in this case
```

```
# Hook the CPU ports up to the l2bus
system.cpu.icache.connectBus(system.l2bus)
system.cpu.dcache.connectBus(system.l2bus)
system.fpga.icache.connectBus(system.l2bus)
system.fpga.dcache.connectBus(system.l2bus)
# Create an L2 cache and connect it to the l2bus
system.l2cache = L2Cache(opts)
system.l2cache.connectCPUSideBus(system.l2bus)

# Create a memory bus
system.membus = SystemXBar()

# Connect the L2 cache to the membus
system.l2cache.connectMemSideBus(system.membus)

# create the interrupt controller for the CPU
system.cpu.createInterruptController()

# For x86 only, make sure the interrupts are connected to the memory
# Note: these are directly connected to the memory bus and are not cached
if m5.defines.buildEnv['TARGET_ISA'] == "x86":
    system.cpu.interrupts[0].pio = system.membus.master
    system.cpu.interrupts[0].int_master = system.membus.slave
    system.cpu.interrupts[0].int_slave = system.membus.master


# Connect the system up to the membus
system.system_port = system.membus.slave

# Create a DDR3 memory controller
system.mem_ctrl = DDR3_1600_x64()
system.mem_ctrl.range = system.mem_ranges[0]
system.mem_ctrl.port = system.membus.master

# Create a process for a simple "Hello World" application
process = LiveProcess()
# Set the command
# cmd is a list which begins with the executable (like argv)
process.cmd = [binary]
# Set the cpu to use the process as its workload and create thread contexts
system.cpu.workload = process
system.cpu.createThreads()

# set up the root SimObject and start the simulation
root = Root(full_system = False, system = system)
# instantiate all of the objects we've created above
m5.instantiate()

print "Beginning simulation!"
exit_event = m5.simulate()
```

```
print 'Exiting @ tick %i because %s' % (m5.curTick(), exit_event.getCause())
```

5.2.2 How CPU Reconfigure FPGA during Execution

As the typical control of FPGA is shown below, CPU can reconfigure FPGA by changing the register OcuupyFPGA, which is the p0[8] in the code.

```
p0 =(unsigned long long *) new((unsigned long long *)0xc0000000) unsigned long long[10];//Contro Port
p0[8] = 1;//reconfigure the FPGA with bitstream 1
while (p0[11]!=p0[8]); //wait until the reconfiguration is done.
p0[1] = (unsigned long long)indata;//ReadBase
p0[2] = (unsigned long long)indata;//WriteBase
p0[0] = 81;
p0[3] = 9;//CurrentThreadID
p0[4] = NI*NJ+NI*NK+NK*NJ+NJ*NL+NI*NL+2;//Memory Range
p0[5] = 4;//MemorySize
p0[7] = 0;//Terminat
p0[6]=0;p0[6]=0;//reset FPGA
init_array (ni, nj, nk, nl, indata);//initialization
p0[6]=1;//start
while(p0[6]);//wait until the acceleration in done
p0[6]=0;
p0[8] = 2;//reconfigure the FPGA with bitstream 2
while (p0[11]!=p0[8]); //wait until the reconfiguration is done.
p0[1] = (unsigned long long)indata;//ReadBase
p0[2] = (unsigned long long)indata;//WriteBase
p0[0] = 81;
p0[3] = 9;//CurrentThreadID
p0[4] = 64*64;//Memory Range
p0[5] = 4;//MemorySize
p0[7] = 0;//Terminat
p0[6]=0;p0[6]=0;//reset FPGA
p0[6]=1;//start
while(p0[6]);//wait until the acceleration in done
```

6.0 USING THE SIMULATOR (IN DIFFERENT MODE)

6.0 USING THE SIMULATOR (IN DIFFERENT MODE)

As known, GEM5 supports both syscall-emulation mode and full-system mode. Syscall-emulation mode makes it easier to configure GEM5 but only simulates binary workload on CPU. In full system mode, gem5 simulates an operation system to handle all of the hardware from the CPU to the I/O devices so that researchers can investigate the impacts of the operating system and other low-level details.

6.1 Syscall-Emulation Mode

All the examples above are set in syscall-emulation mode and here is no more explanation.
But as mentioned before, why should you pay attention to the address translation in this simulator? Why should you write a driver for FPGA in Linux? Because GEM5 deal with address translation in different way. In GEM5, the implementation of address translation is mainly based on the code in *tlb.cc* (*eg. /gem5/src/arch/x86/tlb.cc*). In syscall-emulation mode, TLBs deal with page table by a particular rule but in full-system mode, operation system take care of page table.

6.1.1 Address Translation in Syscall-Emulation Mode

In the function *translate(tlb.cc)*, you can find this lines. Actually, these lines force particular virtual addresses mapped to the physical address of I/O ports of FPGA.

```
if (!FullSystem)
{
  Process *pp = tc->getProcessPtr();
  if (pp->pTable->isUnmapped(baseaddress_control_fpga, size_control_fpga))
          pp->pTable->map(baseaddress_control_fpga, fpga_bus_addr, size_control_fpga,0);
}
```

6.1.2 Address Translation in Full-System Mode

Instead, in *tlb.cc* you find few line about how TLB translate I/O address of FPGA in full system mode. It's because that operation system handle it and you can find how OS do it. You can start from here.

```
if (FullSystem) {
    Fault fault;
    fault = walker->start(tc, translation, req, mode);
    if (!entry || timing || fault != NoFault) {
      delayedResponse = true;
      return fault;
}
```

6.2 Full-System Mode

Because of operation system, in order to utilize FPGA module, you should download a kernel of Linux and write a driver for FPGA. (Warn: It seems that the protocol MOESI_CMP_token does not support multi-cpu fullsystem well)

6.2.1 Full-System without FPGA

As initial step, you need to figure out how to implement full-system mode on GEM5, without FPGA. To get it, the tutorial of GEM5 gives good instructions and video to help you and you can find them here.

Instructions: (http://www.gem5.org/Running_M5_in_Full-System_Mode)

Video: (https://www.youtube.com/watch?v=gd_DtxQD5kc)
The X86 kernel used for regressions, an SMP version of it, and a disk image:
http://www.m5sim.org/dist/current/x86/x86-system.tar.bz2
x86 Config files for both of the above kernels, 2.6.25.1 and 2.6.28.4
http://www.m5sim.org/dist/current/x86/config-x86.tar.bz2

You can make good use of command mount to edit the *.img* file like below.

```
sudo mount -o loop,offset=32256 ../fullsystem/x86-system/disks/linux-x86.img /mnt/img/
```

6.2.2 Full-System with FPGA

Then, you can start to write a driver. Here is an example of FPGA driver and a tutorial of Linux driver (http://freesoftwaremagazine.com/articles/drivers_linux/). The address 0x2000000 in example is the physical address of I/O port of FPGA.

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/types.h>
#include <linux/cdev.h>
#include <linux/slab.h>
#include <asm/io.h>
#include <linux/mm.h>
#include <linux/fs.h>

#define MAJOR_NUM 990
#define MM_SIZE 4096

static char driver_name[] = "mmap_driver";
static int dev_major = MAJOR_NUM;
static int dev_minor = 0;
struct cdev *cdev = NULL;

static int device_open(struct inode *inode, struct file *file)
{
   printk(KERN_ALERT"device open\n");
   return 0;
}

static int device_close(struct inode *indoe, struct file *file)
{
   printk("device close\n");
   return 0;
}

static int device_mmap(struct file *file, struct vm_area_struct *vma)
{
   vma->vm_flags |= VM_IO;
   vma->vm_flags |= (VM_RESERVED);
   if(remap_pfn_range(vma,
                      vma->vm_start,
                      0X20000000>>PAGE_SHIFT,
                      vma->vm_end - vma->vm_start,
                      vma->vm_page_prot))
   {
```

```
        return -EAGAIN;
    }
    return 0;
}

static struct file_operations device_fops =
{
    .owner = THIS_MODULE,
    .open  = device_open,
    .release = device_close,
    .mmap = device_mmap,
};

static int __init char_device_init( void )
{
    int result;
    dev_t dev;
    printk(KERN_ALERT"module init2323\n");
    dev = MKDEV(dev_major, dev_minor);
    printk("now dev=%d", dev);
    cdev = cdev_alloc();
    printk(KERN_ALERT"module init\n");
    if(dev_major)
    {
        result = register_chrdev_region(dev, 1, driver_name);
        printk("result = %d\n", result);
    }
    else
    {
        result = alloc_chrdev_region(&dev, 0, 1, driver_name);
        dev_major = MAJOR(dev);
    }

    if(result < 0)
    {
        printk(KERN_WARNING"Cant't get major %d\n", dev_major);
        return result;
    }


    cdev_init(cdev, &device_fops);
    cdev->ops = &device_fops;
    cdev->owner = THIS_MODULE;

    result = cdev_add(cdev, dev, 1);
    printk("dffd = %d\n", result);
    return 0;
}

static void __exit char_device_exit( void )
{
    printk(KERN_ALERT"module exit\n");
    cdev_del(cdev);
    unregister_chrdev_region(MKDEV(dev_major, dev_minor), 1);
}

module_init(char_device_init);
module_exit(char_device_exit);
```

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Lty");
```

Then you should compile your driver. The most important thing in compiling your driver is that you have to keep you kernel version the same with the kernel you simulate or you will get some error, such wrong format or wrong magic number. Here is an example of MakeFile.

```
obj-m += hello.o
CURRENT_PATH:=$(shell pwd)
LINUX_KERNEL:=$(shell uname -r)
LINUX_KERNEL_PATH:=/usr/src/linux-headers-$(LINUX_KERNEL)
all:
    make -C $(LINUX_KERNEL_PATH) M=$(CURRENT_PATH) modules
clean:
    make -C $(LINUX_KERNEL_PATH) M=$(CURRENT_PATH) clean
```

As the last setp of preparing a driver, you are supposed to insert you driver into kernel and make a node for you FPGA. This step should be taken by the command line in the simulated operation system.

```
insmod mmap_driver.ko
mknod /dev/mmap_driver c 990 0
```

After all these, you can finally run a executable file in the simulated operation system. An example which controls a FPGA module via driver is below.

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <stdlib.h>
#include <unistd.h>
#include <iostream>
#include <string>
#include <pthread.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <new>
using namespace std;
unsigned long long p1[401];//Read Area
unsigned long long p2[401];//Write Area
int main( void )
{
   int fd,i;
   unsigned long long *mapBuf;
   fd = open("/dev/mmap_driver", O_RDWR);
   if(fd<0)
   {
      printf("open device is error,fd = %d\n",fd);
      perror("open erro:");
```

```
        return -1;
    }
    printf("before mmap\n");
    mapBuf = (unsigned long long*)mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    printf("after mmap\n");

    mapBuf[8]=1;
    for (i=0;i<199;i++){p1[i]=(unsigned long long)(i*171133321)%12391231;p2[i]=0;}
    mapBuf[0]=0;
    mapBuf[1]=(unsigned long long)p1;
    mapBuf[2]=(unsigned long long)p2;
    printf("read area : %llu\n",mapBuf[1]);
    printf("write area : %llu\n",mapBuf[2]);
    mapBuf[3] = getpid();//CurrentThreadID
    mapBuf[4] = 200;//Memory Range
    mapBuf[5] = 8;//MemorySize
    mapBuf[7] = 0;//Terminat
    mapBuf[6] = 0;//RunState
    mapBuf[6] = 0;//RunState

    if (mapBuf[6] == 0)
    {
        mapBuf[6] = 1;
        while (mapBuf[6]);
    }
    int n;
    for (n=0;n<198;n++)printf("%llu ",p2[n]);printf("\n");
    mapBuf[8] = 0;

    munmap(mapBuf, 4096);
    close(fd);
    return 0;
}
```

6.2.3 Command Line for Full System

```
build/X86/gem5.opt configs/example/fs.py --disk-
        image=/media/lty/share/Internship/lab/NEW/fullsystem/x86-system/disks/linux-x86.img
        --kernel=/media/lty/share/Internship/lab/NEW/fullsystem/x86-system/binaries/x86_64-vmlinux-
        2.6.22.9.smp --num-cpus=2   --ruby --network=garnet2.0 --num-l2caches=2 --num-dirs=2
        --topology=Mesh_XY --mesh-rows=2
```

6.2.4 Done

```
(none) driveg # insmod mmap_driver.ko
insmod mmap_driver.ko
module init2323
now dev=1038090240<1>module init
result = 0
dffd = 0
(none) driveg #
(none) driveg # mknod /dev/mmap_driver c 990 0
mknod /dev/mmap_driver c 990 0
(none) driveg #
(none) driveg # cd ..
cd ..
(none) / #
(none) / # chmod 0777 testo
chmod 0777 testo
(none) / #
(none) / # ./testo
./testo
device open
before mmap
after mmap
read area : 7396576
write area : 7399808
```

7.0 Acknowledgements

7.0 ACKNOWLEDGEMENTS

I really appreciate Professor Wei Zhang for her kind instruction and Liang Feng for his detailed suggestion. Moreover, I read a lot of materials on the webpage of GEM5 and communicated with GEM5 group. Thanks for their effort to build such a detailed simulator for academic community.

I have to admit that this job is not easy for me at the beginning but with their help, I make it.

The project is still under development, any suggestion will be appreciated.