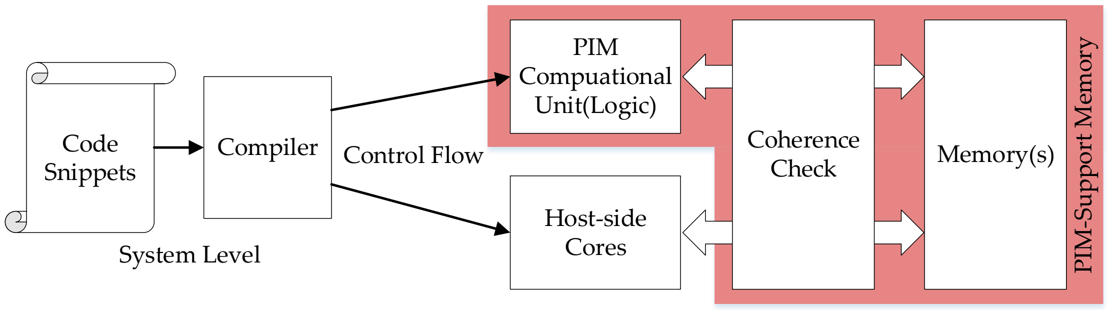
1. Why we need full-system simulation in PIM?

Although PIMSim has provided a simple and fast simulation for PIM architecture, it cannot provide exactly detailed information such as packet traffic. The fast simulation concentrates more on architecture level instead of circuit and operating system, leading to lack of such modeling. We recommend PIM researchers who focus on operating-system-level and circuit-level impacts to set a full-system simulation environment of PIM. This guide will help you establish GEM5 full system environment step by step.

2. Modifications on simulator by applying PIM.

In full-system simulation of PIMSim, PIM acts as a memory component with the capability of computing. We have to fully understand how PIM architecture works. The figure below shows general execution sequences of PIM architecture.



The frontend of PIM architecture is a compiler. It transforms computer code written in programming language (the source language) into computer language (operation code). The collection of operation codes in a particular architecture is called Instruction Set Architecture(ISA). GEM5 has implemented common ISAs like sparc, x86_64, arm and so on. However, *existed ISAs did not support any PIM operations*, which means we have to integrate our own PIM instructions into full-system simulators. Another problem is that *the existed compiler cannot recognize the added PIM instructions*, causing compiling and execution errors. PIM architecture is a highly-customized system, which made it incompatible with any current compilers.

After the compiler transferred source code into executable binaries, the host-side cores start to handle it. In this stage, PIM architecture have to *implement how cores act when execution such PIM operations* and *send the operation to PIM units, during which the coherence check is done. Then PIM Unit will start to process data.*

In PIMSim, we added some PIM instructions and compiler feature to help you establish your PIM system. If you just want to use PIM instead of studying PIM, you can follow the instruction of Section 3. If you want to focus on PIM research, we've provided a detailed method to establish your own PIM architecture in GEM5 at Section 4.

3. Getting start with PIM-supported GEM5.

This section will help you set up a basic PIM system in GEM5 using PIMSim. GEM5 is a complex cycle-accurate simulator. Each modification may result to changes complete differently. You should not avoid or ignore any steps unless you had understood what the step aimed to do and make sure you really don't need it.

Build GEM5

Before you setting up PIM architecture in GEM5, you should have an available GEM5 binary. You can follow the step of Build System and Full System and Benchmark Files at official website and get a executable GEM5 binary file. If you're not familiar with GEM5, we provide a simple guide to help you getting a Ubuntu system in GEM5 in Section 4.

After building GEM5, you can use PIMSim by simply typing:

```
> ./$GEM5_HOME/build/your_arch/gem5.opt config/example/fs.py --mem-type=PIM_DRAM
```

PIMSim supports DRAM, HMC, PCM and NVM (soon available) memory simulations. DRMSim and HMCSim are integrated into the memory. You can modify it by using –mem-type=PIM_(mem_type).

You may add debug flags to debug PIM operations and get detailed information like this:

```
> ./$GEM5_HOME/build/your_arch/gem5.opt –debug-flags=PIM config/example/fs.py --mem-type=PIM_DRAM
```

Define your own PIM Unit.

This step will help you define your own PIM Unit about what it can do. You may add two or more PIM Unit if you need.

First, you should add a python class in $GEM5_HOME/src/dev. For example, if you can create an adder with 4 inputs and 1 output. You should first create a python file named $GEM5_HOME/src/dev/PIMAdder.py

```
from m5.params import *
from m5.proxy import *
from m5.SimObject import SimObject
from m5.objects.PIMKernel import PIMKernel

class PIMAdder(PIMKernel):
        type = 'PIMAdder'
        cxx_header = "dev/pimadder.hh"
        name = Param.String("ADDer","PIM Unit name.")
        latency = Param.Int("1", "PIM Unit computation delay cycles.")
        input=Param.Int(4, "num of inputs")
        output=Param.Int(1, "num of outputs")
```

Parameters *latency* indicates the processing cycles of your PIM Unit (do not include memory access latency). Parameters *input and output* indicates the input and output variables counts. Parameters *cxx_header* indicates the cpp class of your unit's logic.

After that, you should add two cpp file to implement your PIM Unit designs. These two files should be placed in the same path as python file. In our example, you should write `$GEM5_HOME/src/dev/pimadder.cc` and place it at `$GEM5_HOME/src/dev/`. You should implement your logic by get your own functions like `void ADD();` The registers definition is in `$GEM5_HOME/src/dev/pimkernel.hh`.

Link your PIM Units in GEM5

After you create your own PIM Unit class, you should initial it in full system config file. Locate `$GEM5_HOME/configs/common/MemConfig.py, line 240.` Replace `mem_ctrl.kernels.append(PIMAdder())` with `mem_ctrl.kernels.append(Your_class_name())`. After this step, GEM5 can already recognize your customized PIM Unit and get it initialed.

Add PIM operation in your source codes

There are three ways to let GEM5 recognize your PIM operations.
You can replace your operations in source code such as

```
output_var_1=+input_var_1+input_var_2+input_var_3+input_var_4;
```

with

```
PIMKernel(&input_var_1, &input_var_2, &input_var_3, &input_var_4,
&output_var_1, PIMUnit_ID);
```

This function will use PIM to process 4 input variables and write the result to output variable using PIM Unit #(PIMUnit_ID). If you configure only one PIM Unit, just leave it 0.

For complex PIM Units, you can use #PIM CODE START and #PIM CODE END label to mark your PIM code. PIMSim will translate the code snippets automatically into kernel code and execute it on PIM CPU.

```
#PIM CODE START

#DEFINE input1 a
#DEFINE input2 b
#DEFINE input3 c
#DEFINE output1 d

            if(a!=0)
                 d=a*b+c;
          `    else
                a+=1;

#PIM CODE END
```

Another way is to use explicit address to tell PIM what to do like this. You may also use #INPUT = &variable to get it related to program variables. If you provide the detailed address, PIMSim will handle with as physical addresses.

```
#PIM START

#INPUT = 4736648
#INPUT = 3434212
#INPUT = 56345
#INPUT = 34656
#OUTPUT = 23434
#KERNEL ID = 0

#PIM END
```

Compile your codes

After you completing your code modification, you can type the following command to compiler it and get an executable binary with PIM operations. Make sure mono is installed before you compiling your codes.

```
>   mono $GEM5_HOME/tools build.exe YOUR_CODE_FILE OUTPUT_FILE_NAME
```

After done, you should copy it into your GEM5 full-system image and execute it by typing this in m5Term:

```
>   ./ OUTPUT_FILE_NAME
```

If you add PIM debug flags, you will see the PIM information in the console.

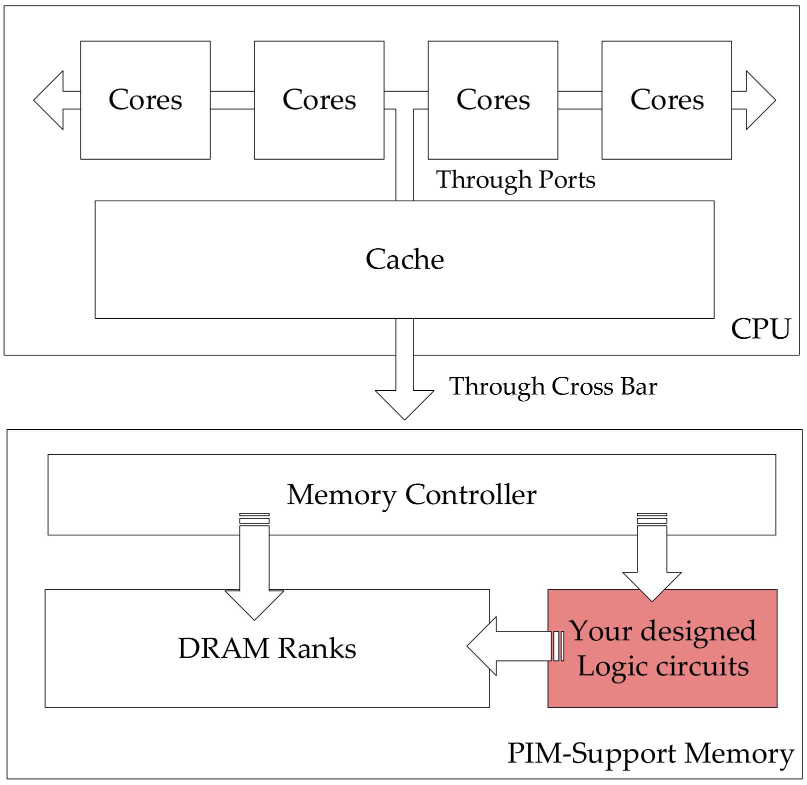
4. Going deep with advanced users

If the PIM operations provided by PIMSim cannot meet your need, you can customize your own PIM instruction. We'll show some cases to help you set up different types of PIM architecture in the following. We partitioned the hotspot of PIM researches into four main parts:

- Research on new designs of computing logic/circuit/unit in PIM.
- Research on exploring application partition methods (deicide which code snippets should be executed in PIM) to gain more performance improvement.
- Research on developing new coherence mechanism in PIM.

CASE 1: Research on new designs of computing logic/circuit/unit in PIM

In this section, we'll guide you to establish PIM architecture in the figure below. This architecture looks like Intelligent RAM (IRAM).



Build GEM5 successfully.

Before you setting up PIM architecture in GEM5, you should have an available GEM5 binary. You can follow the step of Build System and Full System and Benchmark Files at official website and get a executable binary file.
However, the full-system image provided by official website is too old and less user-friendly. We'll give a brief instruction to help you establish a latest Ubuntu system in GEM5 with newer Linux kernel. You may skip this subsection if you need. The architecture we used is x86, you can build other ISAs at the same way.

1. Creating a disk image

When using gem5 in full-system mode, the first step is to create a disk image for system installing. GEM5 had provided a python tool to make it simple to use. You can run the following command:

```
>   sudo $GEM5_HOME/util/gem5img.py init x86root.img 4096
```

This command will create a blank image file called "x86root.img" that is 4GB. You need sudo password if you don't have permission to create loopback devices. gem5img.py is a tool to manage your image file and we will be using it heavily throughout the whole process, so you may want to understand it better. If you just run $GEM5_HOME/util/gem5img.py, it displays all of the possible commands.

```
Usage: %s [command] <command arguments>
where [command] is one of
        init: Create an image with an empty file system.
        mount: Mount the first partition in the disk image.
        umount: Unmount the first partition in the disk image.
        new: File creation part of "init".
        partition: Partition part of "init".
        format: Formatting part of "init".
Watch for orphaned loopback devices and delete them with losetup -d.
Mounted images will belong to root, so you may need to use sudo to modify
their contents
```

2. Copy OS file to the disk file

Now you had an image file, we need to install Ubuntu OS files into it. We can use Ubuntu base distributes for propose (other OS is okay). Since I am simulating an x86 machine, I chose the file ubuntu-base-14.04.5-base-amd64.tar.gz downloading form Ubuntu releases page. You can download whatever image is appropriate for the

system you are simulating.
After you download the tar ball of OS, you should first mount the image file created by Subsection I.

```
> mkdir mnt
> sudo $GEM5_HOME/util/gem5img.py mount x86root.img mnt
```

Now you can copy files to your image by copying files to `mnt`.

```
> sudo tar xzvf ubuntu-base-14.04.5-base-amd64.tar.gz -C mnt
```

3. Setting up gem5-specific files

By default, gem5 uses the serial port to allow communication from the host system to the simulated system. To use this, we need to create a serial tty. Since Ubuntu uses upstart to control the init process, we need to add a file to `mnt/etc/init` which will initialize our terminal. Also, in this file, we will add some code to detect if there was a script passed to the simulated system. If there is a script, we will execute the script instead of creating a terminal.

Put the following code into a file named `mnt/etc/init/tty-gem5.conf`

```
start on stopped rc RUNLEVEL=[12345]
stop on runlevel [!12345]

console owner
respawn
script
    # Create the serial tty if it doesn't already exist
    if [ ! -c /dev/ttyS0 ]
    then
        mknod /dev/ttyS0 -m 660 /dev/ttyS0 c 4 64
    fi

    # Try to read in the script from the host system
    /sbin/m5 readfile > /tmp/script
    chmod 755 /tmp/script
    if [ -s /tmp/script ]
    then
        # If there is a script, execute the script and then exit the simulation
        exec su root -c '/tmp/script' # gives script full privileges as root user in multi-
user mode
        /sbin/m5 exit
    else
        # If there is no script, login the root user and drop to a console
        # Use m5term to connect to this console
        exec /sbin/getty --autologin root -8 38400 ttyS0
    fi
end script
```

The next step is to copy a few required files from your working system onto the disk so we can `chroot` into the new disk. We need to copy `/etc/resolv.conf` onto the new disk.

```
> sudo cp /etc/resolv.conf mnt/etc/
```

We also need to set up the localhost loopback device if we are going to use any applications that use it. To do this, we need to add the following to the `mnt/etc/hosts` file.

```
# /etc/fstab: static file system information.

# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system>      <mount point>     <type>    <options>          <dump>
<pass>
/dev/hda1                /                         ext3                  noatime
0 1
```

Finally, gem5 comes with an extra binary application that executes pseudo-instructions to allow the simulated system to interact with the host system. To build this binary, run make -f Makefile.<isa> in the $GEM5_HOME/util/m5 directory, where <isa> is the ISA that you are simulating (e.g., x86). After this, you should have an m5 binary file. Copy this file to mnt/sbin.

After updating the disk with all of the gem5-specific files, unless you are going on to add more applications or copying additional files, unmount the disk image.

```
> sudo $GEM5_HOME/util/gem5img.py umount mnt
```

By now, the OS image file has been successfully created.

4. Building Linux Kernel.

Next, you need to build a Linux kernel. Unfortunately, the out-of-the-box Ubuntu kernel doesn't play well with gem5.

First, you need to download latest kernel from kernel.org. I use Kernel version 3.2.1 which can be found here. Then, to build the kernel, you are going to want to start with a known-good configure file. The configure file that I'm used for kernel version 2.6.28.4 can be found in full-system files provided GEM5 official website.

Then, you need to move the good config to .config and the run make oldconfig which starts the kernel configuration process with an existing config file.

```
> tar xvf config-x86.tar.bz2
> cp configs/linux-2.6.28.4 /where/your/linux/kernel/source/.config
```

At this point you can select any extra drivers you want to build into the kernel. Note: You cannot use any kernel modules unless you are planning on copying the modules onto the guest disk at the correct location. All drivers must be built into the kernel binary.

Finally, you need to build the kernel.

5. Test you GEM5 simulation.

After you have done all above steps, you can now boot your Ubuntu system by typing this.

```
> $GEM5_HOME/build/x86/gem5.opt configs/example/fs.py --kernel=your_compiled_kernel
```

6. Install new applications

The easiest way to install new applications on to your disk, is to use chroot. This program logically changes the root directory ("/") to a different directory, mnt in this case. Before you can change the root, you first have to set up the special directories in your new root. To do this, we use mount -o bind.

```
> sudo /bin/mount -o bind /sys mnt/sys
> sudo /bin/mount -o bind /dev mnt/dev
> sudo /bin/mount -o bind /proc mnt/proc
```

After binding those directories, make sure you had mount-ed your image file, you can now chroot:

```
> sudo /usr/sbin/chroot mnt /bin/bash
```

At this point you will see a root prompt and you will be in the / directory of your new disk.

You should update your repository information.

```
> apt-get update
```

Now, you are able to install any applications you could install on a native Ubuntu machine via apt-get.

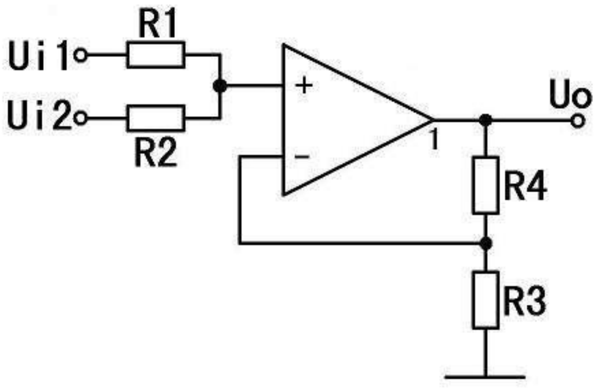
Remember, after you exit you need to unmount all of the directories we used bind on.

```
> sudo /bin/umount mnt/sys
> sudo /bin/umount mnt/proc
> sudo /bin/umount mnt/dev
```

Let GEM5 understand your PIM operation.

Naive GEM5 had no recognition of PIM operations, in this step we'll help you customize your own PIM operations. We assume we designed an adder circuit which can read the data of a given address in memory, then apply Increment operations and write results back like ++ operations in c++ below.

```
//X= [Address];
//X++;
//[Address] = X;
X++;
```



1. Define PIM operations in GEM5

The first step is to let GEM5 know our customized operation. You can either modify GEM5 reserved instructions or add new instructions. If you want to add new instructions, you should find an un-used operation code. Open $GEM5_HOME/src/arch/x86/isa/decoder/two_byte_opcodes.isa (you can also modify one-byte operation code in $GEM5_HOME/src/arch/x86/isa/decoder/one_byte_opcodes.isa or three in $GEM5_HOME/src/arch/x86/isa/decoder/three_byte_0f38_opcodes.isa or $GEM5_HOME/src/arch/x86/isa/decoder/three_byte_0f3a_opcodes.isa).

For example, we can change this

```
0x55: m5reserved1({{
            warn("M5 reserved opcode 1 ignored.\n");
}}, IsNonSpeculative);
```

Into :

```
0x55: m5reserved1({{
      warn("PIM_Add operation.\n");
      PseudoInst::PIM_ADD(xc->tcBase(), Rdi, Rsi);
}}, IsNonSpeculative);
```

The m5 reserved operation is a pseudo instruction reserved by GEM5, we can use it to understand the logic how GEM5 recognizes instructions and what will GEM5 do if decoding such new instructions. We can add more instructions the same ways as this.

In $GEM5_HOME/src/sim/pseudo_inst.hh, you should add function declaration.

```
void PIM_ADD(ThreadContext *tc, uint64_t pa1);
```

In $GEM5_HOME/src/sim/pseudo_inst.cc, modify here

```
case M5OP_ANNOTATE:
        PIM_ADD(tc, args[0], args[1]);
        break;
```

And add function implement. Now we just test PIM operation execution, so we only print strings on console at this stage.

```
void
PIM_ADD(ThreadContext *tc, uint64_t pa1, uint64_t pa2)
{
        std::cout<<"PIM_ADD Operation executed"<<std::endl;
        //tc->getCpuPtr()->sendCommandtoPIM();
}
```

Next, rebuild GEM5 by typing:

```
> scons $GEM5_HOME/build/x86/gem5.opt
```

Now GEM5 can understand your added PIM operations.

2. Compiler codes with PIM operations.

GCC cannot understand our customized PIM operation, so have to use a simple tool to compile our source code.

In $GEM5_HOME/include/gem5/m5ops.h, add this in the end.

```
m5_PIM_ADD(uint64_t pa1);
```

In $GEM5_HOME/include/gem5/m5ops.h, add this in the end.

```
TWO_BYTE_OP(m5_PIM_ADD, M5OP_ANNOTATE)
```

Now we can create a cpp file to test our PIM operation.

```
> cd $GEM5_HOME/tests/test_progs/
> mkdir pim_test
> cd pim_test
> mkdir src
> cd src
```

Now create a text file named "pim_test.c", add below content:

```
#include <gem5/m5ops.h>

int main()
{
        m5_PIM_ADD(0);
        return 0;
}
```

You can compile your code through:

```
> gcc -o pim_test pim_test.c -I ../../../../include/ ./../../../../util/m5/m5op_x86.S
```

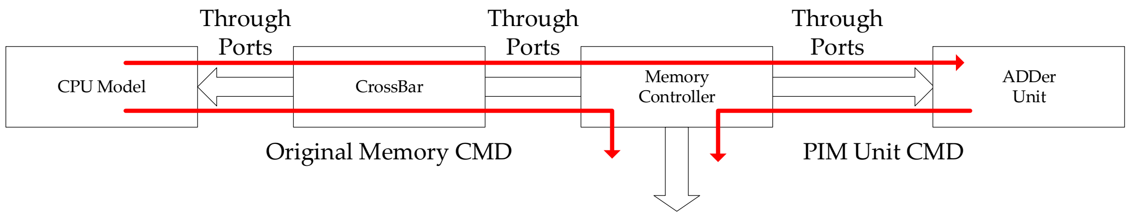
Now you can copy this binary to GEM5 image file using gem5img.py and execute it in full system mode, you can see your output in the console.

```
PIM_ADD Operation executed
```

3. Control your PIM unit.

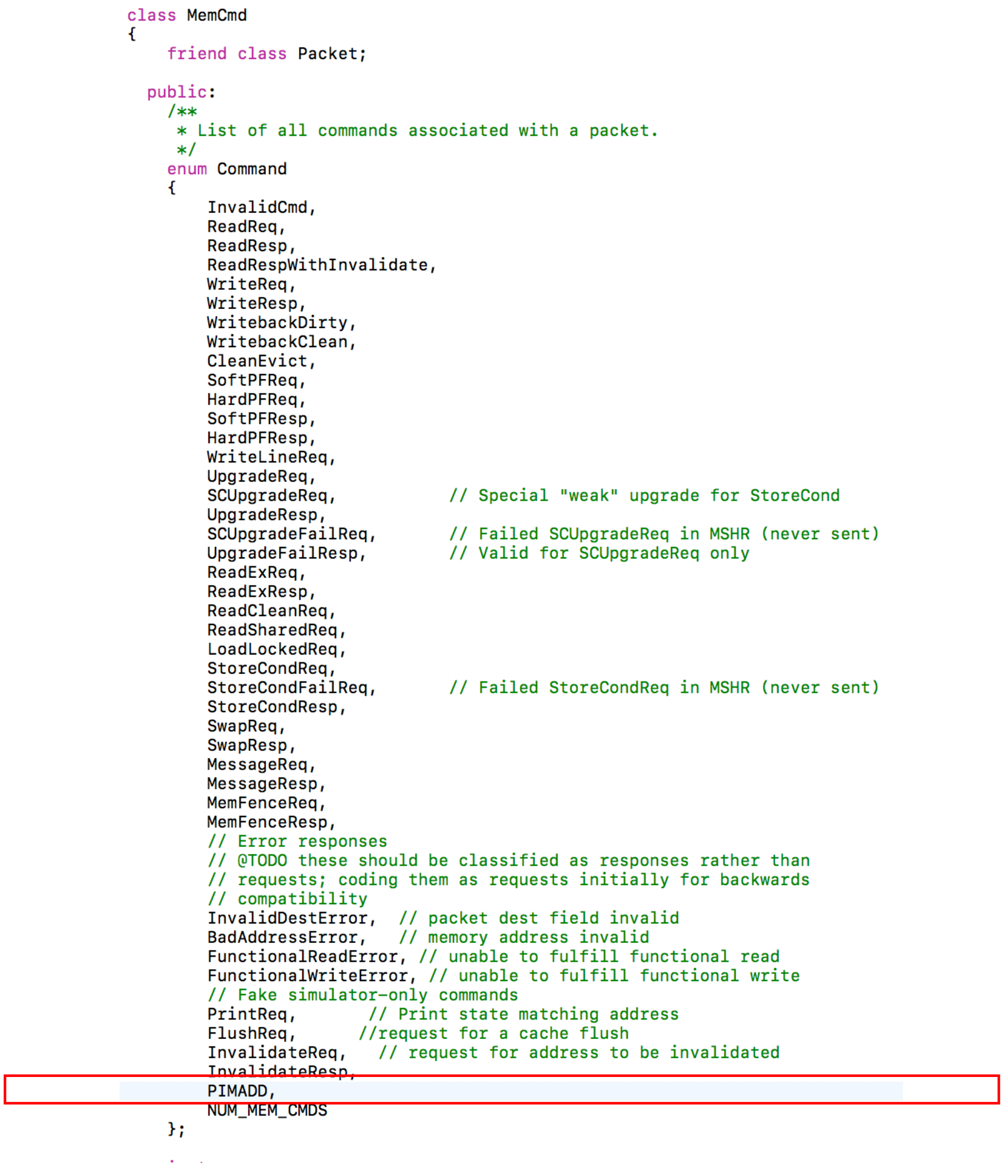
In this section, we are going to help you control your PIM circuits. In step C, we had confirmed GME5 can decode and execute what we want while executing our customized PIM operation. We'll establish control flow to control our designed PIM circuit.

As we all known, GEM5 use Port mechanism to transfer data through objects. So we should establish the connection between different objects. The figure below shows three connections between different objects. Original Memory stands for the regular memory read/write operation sent by host-side CPUs. PIM Unit CMD stands for operations on memory required by your ADDer unit. The straight right arrow stands for the control command sent from host-side CPU while PIM operation decoding is done. All the command is sent in the form of Packets.



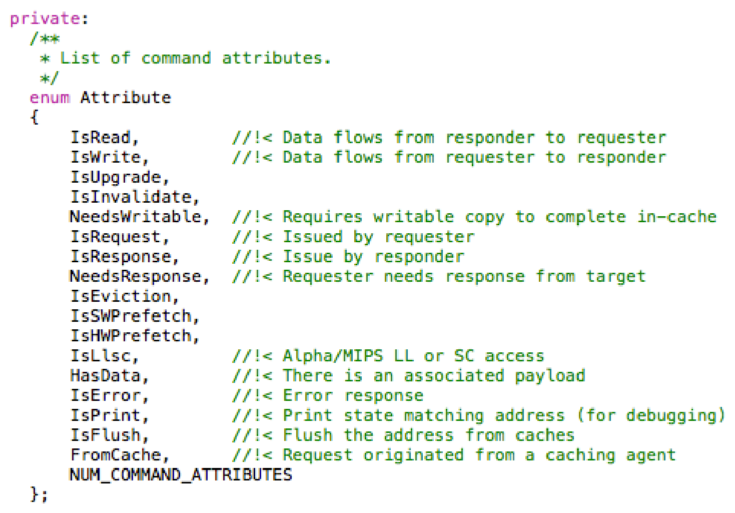
3.1 Customize your own command.

You have to define your unique command first to support different functions operated by your designed PIM Units as an identity distinct from original memory commands. The define file of memory command is in $GEM5_HOME/src/mem/packet.hh.



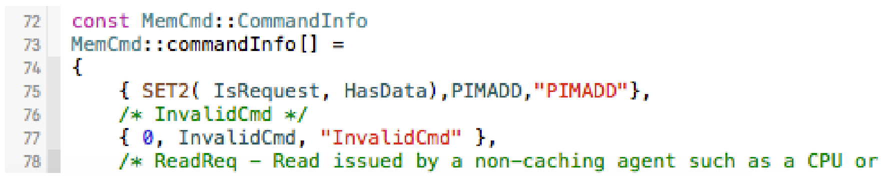
```
class MemCmd
{
    friend class Packet;

  public:
    /**
     * List of all commands associated with a packet.
     */
    enum Command
    {
        InvalidCmd,
        ReadReq,
        ReadResp,
        ReadRespWithInvalidate,
        WriteReq,
        WriteResp,
        WritebackDirty,
        WritebackClean,
        CleanEvict,
        SoftPFReq,
        HardPFReq,
        SoftPFResp,
        HardPFResp,
        WriteLineReq,
        UpgradeReq,
        SCUpgradeReq,           // Special "weak" upgrade for StoreCond
        UpgradeResp,
        SCUpgradeFailReq,       // Failed SCUpgradeReq in MSHR (never sent)
        UpgradeFailResp,        // Valid for SCUpgradeReq only
        ReadExReq,
        ReadExResp,
        ReadCleanReq,
        ReadSharedReq,
        LoadLockedReq,
        StoreCondReq,
        StoreCondFailReq,       // Failed StoreCondReq in MSHR (never sent)
        StoreCondResp,
        SwapReq,
        SwapResp,
        MessageReq,
        MessageResp,
        MemFenceReq,
        MemFenceResp,
        // Error responses
        // @TODO these should be classified as responses rather than
        // requests; coding them as requests initially for backwards
        // compatibility
        InvalidDestError,  // packet dest field invalid
        BadAddressError,   // memory address invalid
        FunctionalReadError, // unable to fulfill functional read
        FunctionalWriteError, // unable to fulfill functional write
        // Fake simulator-only commands
        PrintReq,       // Print state matching address
        FlushReq,      //request for a cache flush
        InvalidateReq,   // request for address to be invalidated
        InvalidateResp,
        PIMADD,
        NUM_MEM_CMDS
    };
```

You should add your customized PIM operation in the code location of above figure such as a PIMADD command. Except that, you should add the attributes of your command for further analysis of your packets. All of packets attributes are listed below.



```
private:
  /**
   * List of command attributes.
   */
  enum Attribute
  {
      IsRead,         //!< Data flows from responder to requester
      IsWrite,        //!< Data flows from requester to responder
      IsUpgrade,
      IsInvalidate,
      NeedsWritable,  //!< Requires writable copy to complete in-cache
      IsRequest,      //!< Issued by requester
      IsResponse,     //!< Issue by responder
      NeedsResponse,  //!< Requester needs response from target
      IsEviction,
      IsSWPrefetch,
      IsHWPrefetch,
      IsLlsc,         //!< Alpha/MIPS LL or SC access
      HasData,        //!< There is an associated payload
      IsError,        //!< Error response
      IsPrint,        //!< Print state matching address (for debugging)
      IsFlush,        //!< Flush the address from caches
      FromCache,      //!< Request originated from a caching agent
      NUM_COMMAND_ATTRIBUTES
  };
```

The PIMADD command we added is aimed to let PIM Units start an atomic add operation on a specified address, so we can set its attributes to IsRequest, HasData in $GEM5_HOME/src/mem/packet.cc like this.



```
const MemCmd::CommandInfo
MemCmd::commandInfo[] =
{
    { SET2( IsRequest, HasData),PIMADD,"PIMADD"},
    /* InvalidCmd */
    { 0, InvalidCmd, "InvalidCmd" },
    /* ReadReq - Read issued by a non-caching agent such as a CPU or
```

Or you can add new attributes if you need.

I. CPU -> Memory connection

First, we should add a function in $GEM5_HOME/src/cpu/base.hh in BaseCPU.

```
void sendCommandtoPIM(Addr addr_);
```

And add virtual implements in $GEM5_HOME/src/cpu/base.cc.

```
Fault
BaseCPU:: sendCommandtoPIM(Addr addr_)
{
        Return NoFault;
}
```

After that you should override this function in codes of your CPU model. For example, if we use AtomicSimpleCPU, we should add declaration of this function in $GEM5_HOME/src/cpu/simple/atomic.hh.

```
void sendCommandtoPIM(Addr addr_) override;
```

And add implements in $GEM5_HOME/src/cpu/simple/atomic.cc.

```
Fault AtomicSimpleCPU:: sendCommandtoPIM(Addr addr_)
{
        SimpleExecContext& t_info = *threadInfo[curThread];
        SimpleThread* thread = t_info.thread;
        static uint8_t zero_array[64] = {};
        assert(data != NULL);
        Request *req = &data_write_req;
        int fullSize = size;
        dcache_latency = 0;
        req->taskId(taskId());
        // translate to physical address if you use & operations
        //Fault fault = thread->dtb->translateAtomic(req, thread->getTC(),
BaseTLB::Write);
                if (fault == NoFault) {
                        MemCmd cmd = MemCmd::WriteReq; // default
                        bool do_access = true;    // flag to suppress cache access
                        cmd = MemCmd::PIMADD;
                        Packet pkt = Packet(req, cmd);
                        pkt.dataStatic(addr_);
                        if (fastmem && system->isMemAddr(pkt.getAddr()))
                                system->getPhysMem().access(&pkt);
                        else
                                dcache_latency += dcachePort.sendAtomic(&pkt);
                        threadSnoop(&pkt, curThread);

                        assert(!pkt.isError());
                }
                return NoFault;
        }

}
```

II. Memory -> PIM Unit connection

When memory received Memory command, it should first identify the destiny of this command. If the destiny is PIM Unit, transmit it. We use default dram controller as an example.

In $GEM5_HOME/src/mem/dram_ctrl.cc, add transmit mechanisms. Locate the following code snippets:

```
Tick
DRAMCtrl::recvAtomic(PacketPtr pkt)
{
       DPRINTF(DRAM, "recvAtomic: %s 0x%x\n", pkt->cmdString(), pkt->getAddr());

        panic_if(pkt->cacheResponding(), "Should not see packets where cache "
                      "is responding");

        // do the actual memory access and turn the packet into a response
        access(pkt);
```

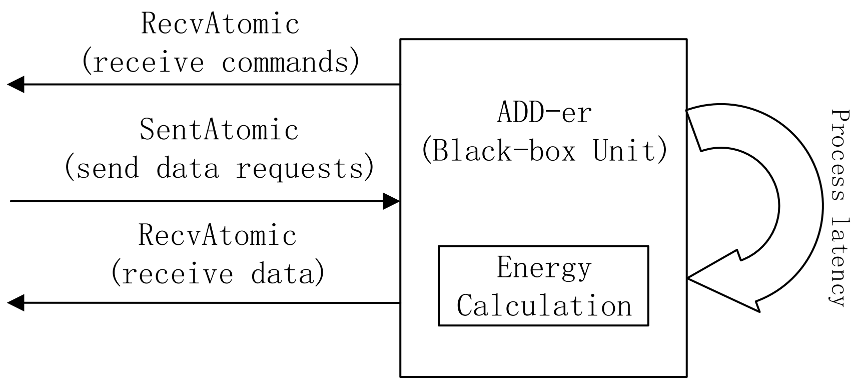
This function is called when DRAM get a memory command to access data. DRAM controller only receives command from CPU in state-of-the-art architecture, but in PIM, it will also receive commands from PIM Unit (you may also implement your circuits here instead of creating an individual competent in GEM5). Now you should add determine statements before the function access(pkt); You can do like this:

Add two ports in $GEM5_HOME/src/mem/dram_ctrl.hh, the add codes in $GEM5_HOME/src/mem/dram_ctrl.cc. You may include our provided coherence header file, or you can write your own coherence mechanism.

```
Tick
DRAMCtrl::recvAtomic(PacketPtr pkt)
{
        if(pkt->cmd==MemCmd::PIMADD )
        {
        // Distribute PIM operation to PIM Unit when receiving PIMADD command.
                pimport.sendAtomic(pkt);
                return;
        }else{
                coherence.check(pkt->getAddr());
        }
```

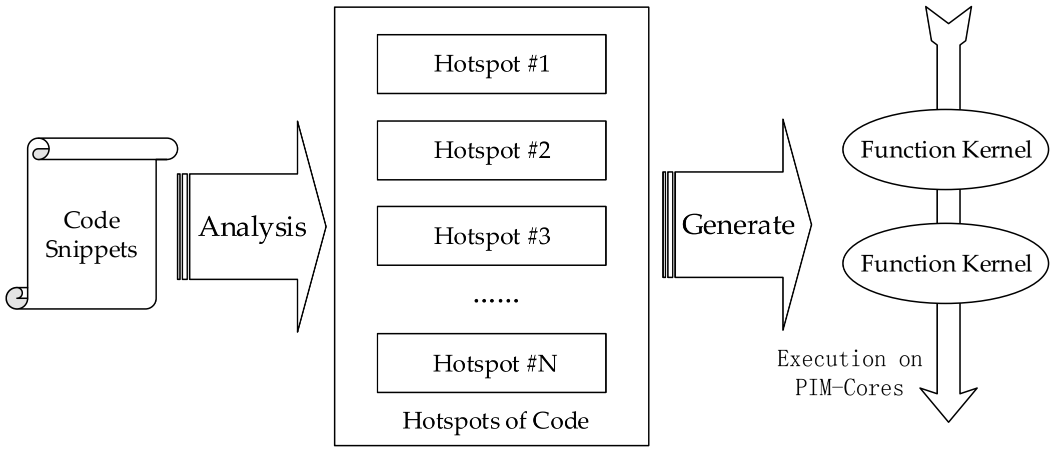
Create your PIM Unit Component.

You can use circuit-level simulator to get detailed characteristic parameters and implement it as a black-box unit. You may require latency, energy parameters and establish the connection mechanism follow the rules of Ports definition in $GEM5_HOME/src/mem/ports.hh. If you're not familiar with GEM5, you can simply implement your circuit through c/c++ in $GEM5_HOME/src/mem/dram_ctrl.cc.



CASE2: Researchers on discovering PIM's potential and gain better performance.

If you're researchers with little circuits knowledge but more coding ability, you may care less about how the PIM Units are designed but how to take advantages of PIM. For architectural researchers, they may also extract the hotspots of a popular application to figure out a way to accelerate the whole system. In this case, the logic unit in memory is assumed to be several general-propose CPU cores.



The general sequences of such research start from application analysis. This step will get the hotspots of a kind of applications. By distilling the critical path of target hotspots with the consideration of PIM architecture, researchers can get individual kernels that may benefit from PIM architecture. When host-side CPU encountered such code snippets, it will inform memory-side cores to start the kernel.

Next, we'll show how to establish such PIM system in GEM5. This architecture looks like Practical Near-Data Processing for In-Memory Analytics Frameworks and PEI.

I. Boot your GEM5 simulator.

You should first follow the steps A in CASE 1 to get an executable binary of GEM5 and get it boot. When you're in the bash shell, you should stop simulations and see how many instructions had been simulated. Then you can use --at-instruction --take-checkpoints=N to reboot your GEM5 system and get an initial GEM5 checkpoints in bash shell.

```
> ./build/x86/gem5.opt configs/examples/fs.py --at-instruction --take-
checkpoints=50000000 --max-checkpoints=1
```

The aim of this step is to get a fixed start point and prevent data changed by system boot before application analysis starts. You can boot GEM5 through checkpoints by applying --at-instruction -r N to fs.py or se.py.

```
> ./build/x86/gem5.opt configs/examples/fs.py --at-instruction -r 50000000
```

II. Analysis your applications.

In this step, you should distill the kernels that you want to execute at PIM Units. You can use profiling tools to get detailed information about hotspots.

Note that you have to build newer Linux kernel following the guide of Compiling a Linux Kernel and apply it to GEM use --kernel= command. The pre-build kernel provided by GEM5 full-system image file is too old to support such profiling software. You should always start from a checkpoint to invoke another profiling in order to prevent data changed by the boot of operating system.

III.Pass the kernel to PIM Cores

After you selected the kernels you want to executed at memory-side PIM cores, you should provide the information fetched by step B to PIM cores such as kernel stating PC and ending PC. You should first follow all the guide in CASE 1, where you only need to pass an address to PIM Units. In this case, you should provide a class named ThreadContext defined in $GEM5_HOME/src/cpu/thread_context.hh. This definition is overridden in detailed implement of different CPU model. You can define your function to pass the ThreadContext of current CPU to PIM Cores in $GEM5_HOME/src/sim/pseudo_inst.cc like this:

```
void
PIM_Kernel_Pass(ThreadContext *tc, uint64_t pa1, uint64_t pa2)
{
        BaseCPU* cpu=(BaseCPU*)tc->getCpuPtr();

        sendContexttoPIM(cpu->threadInfo[cpu->curThread]);
        ThreadID id= cpu->contextToThread(cpu->curThread);
        cpu->haltContext(id);
}
```

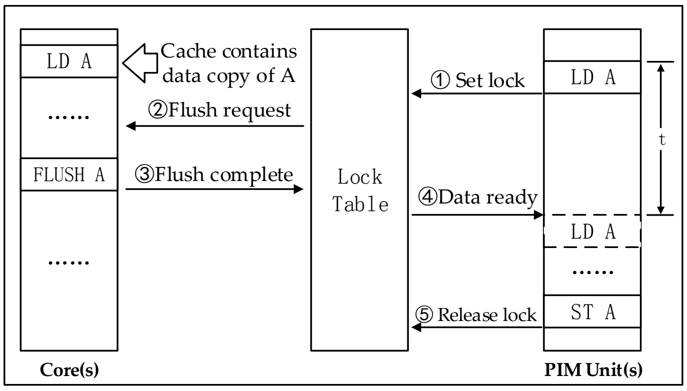
You may also call takeOverFrom function defined in

$GEM5_HOME/src/cpu/thread_context.hh to let PIM cores take over all the content from host-side CPUs.

CASE3: Researches on PIM architecture.

For researcher on PIM architecture, you have to implement all the feature like CASE1 and CASE2, then modify what you need. In this case, we use PIM coherence as an example to show how to modify PIM coherence.

Coherence is definitely an important part in PIM architecture due the limited share in memory between host-side CPUs and memory-side PIM Units. The general solution is lock mechanism. Whenever host-side cores or PIM Units require data, it should look up LockTable to ensure the operation atomicity.



The simplest way is to check Lock Table before the actual access operation happened. You should follow the steps in CASE1 and add judgment statement before the doaccess() function is called.