A Full-System Simulation Environment for the Smart Memory Cube

Erfan Azarkhish (erfan.azarkhish@unibo.it)

October 12, 2016

Partners: Davide Rossi (davide.rossi@unibo.it)
Igor Loi (igor.loi@unibo.it)
Supervisor: Professor Luca Benini (luca.benini@iis.ee.ethz.ch)

Abstract

The recent technological breakthrough represented by the Hybrid Memory Cube is on its way to improve bandwidth, power consumption, and density. This is while heterogeneous 3D integration has provided another opportunity for revisiting near memory computation to fill the gap between the processors and memories even further. We have taken the first step towards a "Smart Memory Cube (SMC)", a fully backward compatible and modular extension to the standard HMC, supporting near memory computation on its Logic Base (LoB), through a high performance interconnect designed for this purpose. In this report, a high-level model for the Smart Memory Cube is presented, which is designed in the gem5 [Binkert et al., 2011] simulation environment. This model has been validated against the previously developed cycle-accurate model of SMC, and is ready for full system simulation. Moreover, an ARMv7 core has been added to model a Processor in Memory (PIM), and a basic software stack has been developed for communication of the host with PIM.

Contents

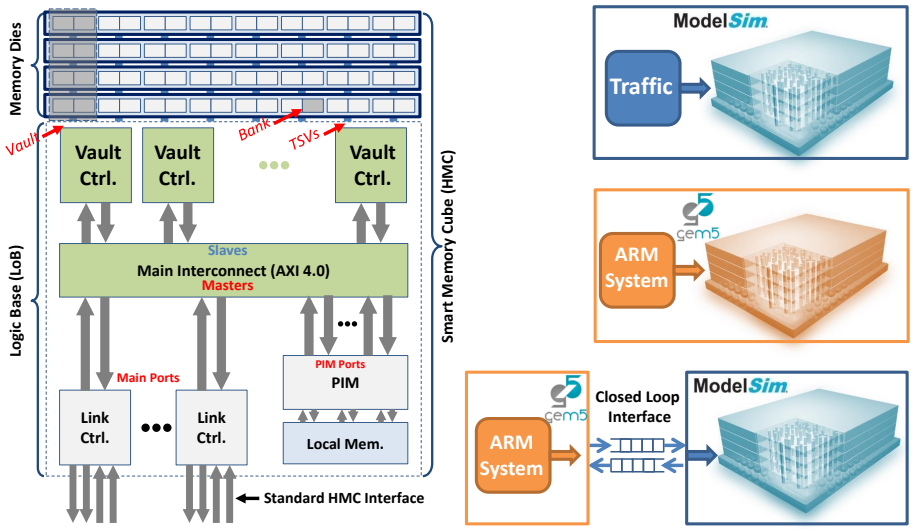

Figure 1: Overview of the Smart Memory Cube (SMC), as well as the three methodologies devised by ETHZ to study SMC: 1) ModelSim standalone simulation, 2) gem5 standalone simulation, and 3) gem5-ModelSim closed-loop simulation

1 Introduction

An overview of the Smart Memory Cube (SMC) is illustrated in Figure1.left. In this project, two independent SMC models have been developed. The first one is a cycle-accurate model using SystemVerilog language to be simulated inside ModelSim. The second one, however, is a high-level gem5-based model of SMC, suitable for full-system simulations. Moreover, a closed-loop simulation infrastructure has been developed providing the possibility to instantiate the cycle-accurate SMC model inside gem5 (See Figure1.right).

The high-level model has been designed based on the new features of the gem5 Architectural Simulation Platform [Binkert et al., 2011], enabling a full-system design space exploration and analysis of different parameters in presence of the dynamic effects of the operating system, memory management, and cache coherence. This model can be attached to an ARMv8 system with multiple processing cores, as well as the peripheral devices and the interconnects. Figure2 demonstrates an overview of the platform which can be currently modelled in this gem5-based platform.

Section 2 describes the simulation environment and how to run it for the first time. Section 3 explains how to perform a full system simulation in the gem5 environment. And finally Section 4 explains the PIM model.

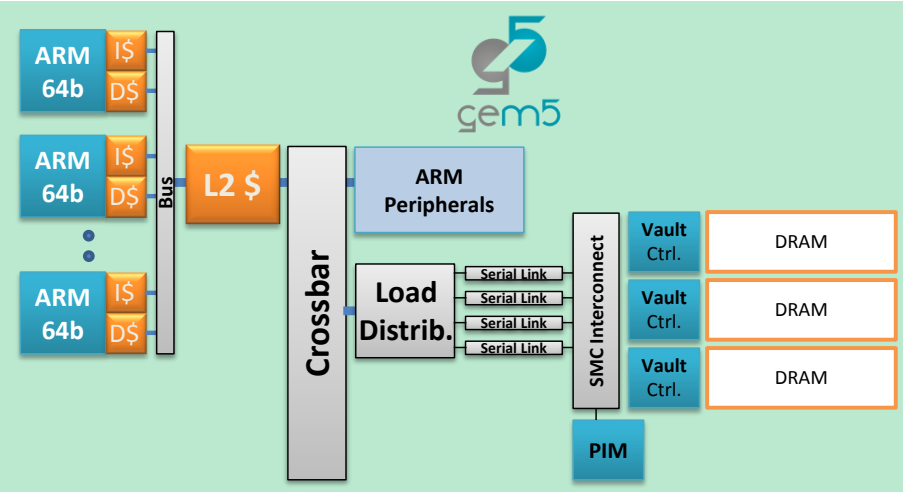

Figure 2: An overview of the Smart Memory Cube attached to an ARM system in gem5

2 The Simulation Environment

Figure3 shows the global directory structure of the SMC simulation environment. As can be seen, GEM5/ and HDL/ directories contain all the source files related to gem5 and the cycle-accurate model, respectively. All simulation scenarios are placed in scenarios/, and SW/ contains the software stack. UTILS/ consists of a set of predefined bash utilities, as well as preconfigured models which can be loaded in the simulation scripts. A list of all parameters which can be overridden by the user reside in "default_params.sh".

In this section, first we will describe how to setup and run the environment for the first time, then we will describes the implementation details of this simulator including the SMC model.

2.1 Setup and First Run

Along with this document, two packages are delivered: SMC.tgz and SMC-WORK.tgz. First, the contents of these packages should be extracted:

```
tar xvfz SMC.tgz
```

```
tar xvfz SMC-WORK.tgz
```

Next the tarballs inside the SMC-WORK/ directory should be extracted and two new directories should be created in it, as well:

```
cd SMC-WORK/
```

```
tar xvf checkpoints.tar
```

```
tar xvf linux_kernel.tar
```

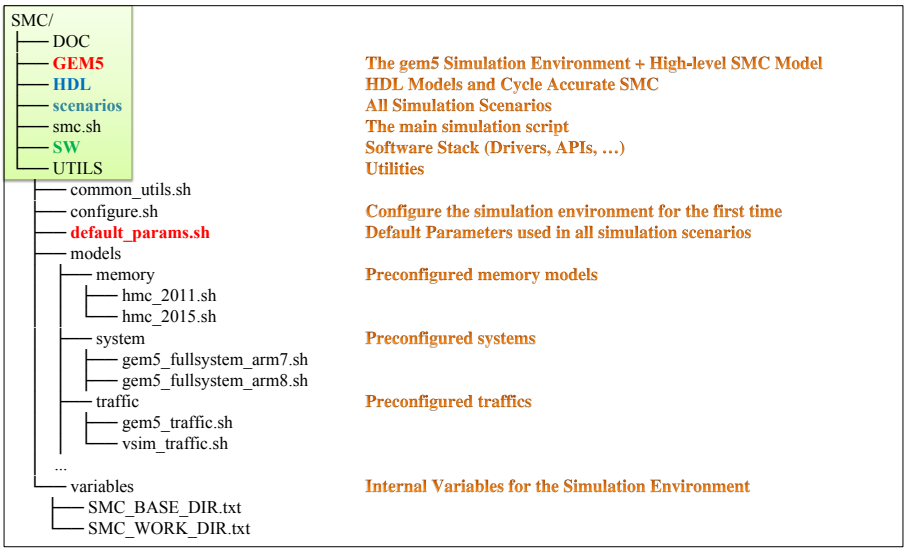

Figure 3: Directory structure of the SMC Simulation Environment

```
tar xvf gem5-images.tar
```

```
mkdir scenarios
```

```
mkdir gem5-build
```

```
cd ../SMC
```

An overview of SMC-WORK/ directory has been shown in Figure4. The main directory which users need to work with is the **scenarios/** directory. Based on the name of the running scenario, a subdirectory will be created in this location and all the logs and temporary files will be stored in it. **gem5-build** is the build directory of gem5; **gem5-images** holds the pre-built images and binaries required for full system simulation in gem5; and **linux_kernel** contains the kernel source codes of the pre-built images, required for building the device drivers.

Next, UTILS/variables/SMC_WORK_DIR.txt should be edited and the full path of the SMC-WORK/ directory should be written in it.

```
echo "/home/foo/SMC-WORK/" > ./UTILS/variables/SMC_WORK_DIR.txt
```

Now, to make sure that the tool is configured correctly and all required packages are available the following utility should be executed:

```
./UTILS/configure.sh
```

Upon execution, the messages illustrated in Figure5 should be seen on the screen. The configuration should finish successfully, displaying "Configuration finished!"". A list of the special libraries required by gem5 will be shown as ""REQUIRED LIBRARIES". These libraries should be installed prior to building gem5. Moreover, a list of the required tools should be seen in the "REQUIRED VERSIONS". It should be noted that the version of these tools must

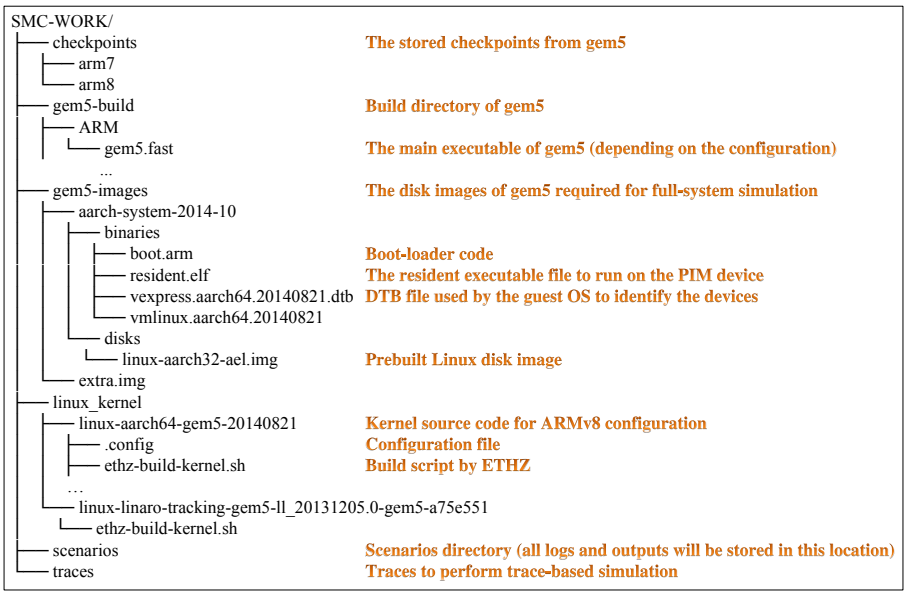

Figure 4: Directory structure of SMC-WORK/

be equal or higher than the ones used to prepare the release package. This environment has been developed using "**ubuntu 14.04 LTS**" and all the required packages and libraries can be easily installed on this platform using "apt-get" command.

Another important point to mention is that, this environment requires sufficient permissions to mount and manipulate disk images. In default_params.sh an environment variable called SUDO_COMMAND is set to "sudo" by default and should be modified based on the host environment. The utilities which require superuser privileges are mount_disk_image() and copy_to_disk_image() in UTILS/common_utils.sh. After this step, the tool is ready for building gem5 and running the first scenario. For this purpose, run the following script with *-b* switch from the base directory (SMC/):

```
./scenarios/c-gem5/a-tests/0-arm-hmc.sh 7 -b
```

This command automatically builds an optimized version of gem5 for the ARM platform. This process will take some time. No errors should be faced in this phase, and finally, "**ARM/gem5.fast was built successfully**" should be displayed on the screen. The details of the build procedure are explained in the following link:

`http://www.m5sim.org/Build_System`

Moreover, all external dependencies of gem5 are reported in:

`http://www.m5sim.org/Dependencies`

To test this build, the same script should be executed with *-o* switch (to view the output logs).

```
Info:   Configuring SMC Simulator …
Info:  Number of Colors                    > 8 RGB
Info:  arm-linux-gnueabihf-gcc             > [   OK  ]  /usr/bin/arm-linux-gnueabihf-gcc
Info:  aarch64-linux-gnu-gcc               > [   OK  ]  /usr/bin/aarch64-linux-gnu-gcc
Info:  CWD                                 > /home/erfan/projects/SMC
Info:  SMC_BASE_DIR                        > /home/erfan/projects/SMC
Info:  SMC_WORK_DIR                        > /home/erfan/projects/SMC-WORK/
Info:  TRACE_FOLDER_LOCATION               > /home/erfan/projects/SMC-WORK/traces
…
Info:   REQUIRED LIBRARIES ...
            zlibc
            zlib1g
            zlib1g-dev
            libprotobuf-dev
            libgoogle-perftools-dev

Info:   REQUIRED VERSIONS ...
            gcc (Ubuntu 4.8.2-19ubuntu1) 4.8.2
            arm-linux-gnueabihf-gcc (Ubuntu/Linaro 4.8.2-16ubuntu4) 4.8.2
            …
Info:   AVAILABLE VERSIONS ...
            gcc (Ubuntu 4.8.2-19ubuntu1) 4.8.2
            arm-linux-gnueabihf-gcc (Ubuntu/Linaro 4.8.2-16ubuntu4) 4.8.2
            …
Info:  Checking GNUPlot functionalities …
Info:  GNUPlot terminal set to wxt

Info:   CHECKING WORK SUBDIRECTORIES .
Info:  scenarios                           > OK
Info:  linux_kernel                        > OK
Info:  gem5-images                         > OK
Info:  gem5-build                          > OK
Info:  checkpoints                         > OK
Info:   Configuration finished!
```

Figure 5: An example of a successful configuration of the environment

```
./scenarios/c-gem5/a-tests/0-arm-hmc.sh 7 -o
```

This command should start the full system simulation of an ARMv7 system in gem5. The simulation should not stop or crash for any reason. To connect to the operating system running on this environment one of the following commands can be executed in another terminal:

```
./smc.sh -t
```

```
telnet localhost 3456
```

After the boot procedure of the guest OS has been finished, there will be a prompt for login. User **root** without any password can be used to log-in. Pressing Ctrl+C can close and safely stop the simulation. Lastly, the simulation logs and results for this simulation should be stored in "SMC-WORK/scenarios/ca07/". At this point the environment is ready for running different simulation scenarios.

2.2 Simulation Scenarios, Results, and Output Logs

As shown in Figure6 the scenarios are categorized in 3 groups: ModelSim only simulations, Accuracy comparison, and gem5 full system simulations (Running a scenario with -h switch will display the options available in that scenario).

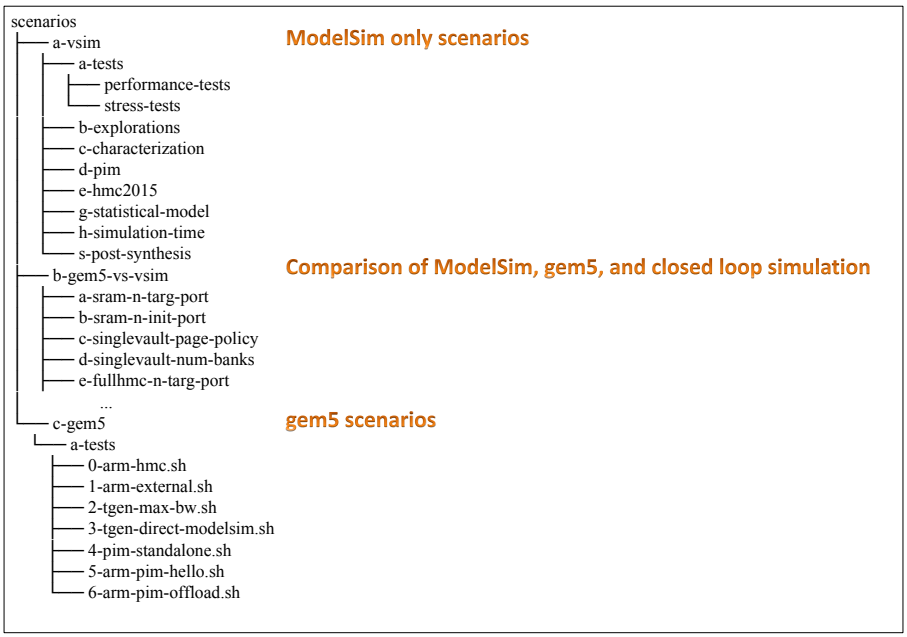

Figure 6: An overview of the available simulation scenarios

The scenarios in the first group (a-vsim) have been previously described in [**?**]. In this group of scenarios the cycle-accurate model of SMC is simulated in standalone mode inside the ModelSim environment. Traces can be applied and performance can be measured.

However, in the next two groups, the structure of the simulation scripts is slightly different from the first group. Figure7 illustrates a sample scenario script running gem5 with a synthetic traffic generator and altering the number of serial links attached to the SMC interconnect from 1 to 4.

Contrary to the script described in Section 3.2 of [**?**], in this script the statistics should be gathered from gem5. GEM5_STATISTICS keeps a list of such statistics. Moreover, a new utility called *create_scenario* has been added to all simulation scenarios (including the first group), to create a scenario location automatically based on the name of the scenario script. Here is an example:

./scenarios/**c**-gem5/**a**-tests/**0**-arm-hmc.sh **7** ⇒ Scenario Location: Scenarios/**ca07**/

After loading the default values for the simulation environment, the parameters should be overridden either directly or using the preconfigured models present in UTILS/models/ directory. *load_model* command can be used to load these models. At the end of each simulation case the statistics must be reported using *report_gem5_stats*, and finally, the results will be organized using *finalize_gem5_simulation*, and plots can be generated same as before.

The structure of the result directory has been shown in Figure8. Among the many intermediate files reported, **stats.txt** contains all gathered statistics from

```
GEM5_STATISTICS=(                                          Statistics to Gather inside gem5
"system.monitor.averageReadBandwidth"
"system.monitor.readLatencyHist::mean"
"final_tick"
)

VALUES1=( 1 2 4 )      # N_TARG_PORT                       Parameter to Explore

for V1 in ${VALUES1[*]}
do

      create_scenario "$0/$*" "$V1" "Your comment here ..."   Create a scenario automatically
      source UTILS/default_params.sh                          Load the default parameters

      load_model memory/hmc_2011.sh                           Load HMC-2011 memory model
      load_model traffic/gem5_traffic.sh -1 1000 100000 10000 1000   Load the traffic generation engine in gem5
      export N_TARG_PORT=$V1                                  Alter the parameter under study

      source ./smc.sh $1 $2 $3 $4                             Run the actual simulation
      report_gem5_stats                                       Report statistics in gem5
done

finalize_gem5_simulation                                      Finalize and generate the reports

plot_bar_chart "system.monitor.averageReadBandwidth" 0 "Bytes/S"   Generate a bandwidth plot
```

Figure 7: A sample scenario script to run gem5

```
scenarios/
└── my_scenario          Current Scenario
        └── my_case          Current Case
                └── m5out            gem5 outputs
                        ├── config.dot.pdf          Visualization of the system using GraphViz
                        ├── ethz_configs.py
                        ├── ethz_utils.py           Configurations and Utilities
                        ├── ethz_pimconf.py
                        ├── _gem5_params.py         Parameters to be passed to gem5
                        ├── _gem5_run.sh
                        ├── stats.txt               All statistics reported by gem5
                        ├── system.bridges.params.txt
                        ├── system.memobjects.clock.txt
                        ├── system.ports.names.txt
                        ├── system.terminal         Contents of the attached terminal
                        ├── system.vaults.params.txt   Parameters of the memory vaults
                        ├── system.xbars.addrmap.txt   All address maps in the system
                        └── system.xbars.params.txt
```

Figure 8: The structure of the output directory (focusing on gem5's results)

gem5 and **config.dot.pdf** visualizes the system using GraphViz. The other files can be used to understand the internals of the simulation and to debug it easier.

2.3 A Brief Overview of gem5

The gem5 simulator is a modular platform for computer system architecture research, encompassing system-level architecture as well as processor microarchitecture (`http://www.gem5.org/Main_Page`). Traditionally, gem5 has been

a glued version of two existing simulators: M5 and GEMS, with M5 being responsible for the CPU models and GEMS for the memory and cache coherence protocols. The key features of this simulator include:

- Pervasive object orientation
- Multiple interchangeable CPU models
- Event-driven memory system
- Multiple ISA support: Alpha, ARM, SPARC, MIPS, POWER and x86 ISAs
- Full-system capability with Alpha, ARM, SPARC, and x86
- Multiprocessor / multi-system capability.

An introduction to this environment can be found in `http://www.m5sim.org/Introduction`. gem5 can be executed in two modes: System Call Emulation (**SE**), and Full System (**FS**) simulation. Among these two, FS mode is more desirable because of the ability to study the interaction of hardware and the software stack. On the other hand gem5 supports two memory systems: **General Memory System**, and **Ruby**. Since Ruby is not compatible with the standard memory system, we won't focus on it.

gem5's General Memory System consists of a comprehensive set of building blocks ranging from caches and crossbars to DRAM controllers; and is suitable for loosely-timed, approximately-timed and untimed transaction-level modelling. What should be noted is that, three types of accesses are supported by the ports inside this memory system:

- **Timing**: Timing accesses are the most detailed access. They reflect modeling of queuing delay and resource contention. Once a timing request is successfully sent at some point in the future the device that sent the request will get a response. Timing and Atomic accesses can not coexist in the memory system. This is similar to the TLM *nb_transport* interface.
- **Atomic**: Atomic accesses are a faster than detailed access. They are used for fast forwarding and warming up caches and return an approximate time to complete the request without any resource contention or queuing delay. When an atomic access is sent the response is provided when the function returns. Atomic and timing accesses can not coexist in the memory system. This is similar to the TLM *b_transport* interface (without any blocking).
- **Functional**: Like atomic accesses functional accesses happen instantaneously, but unlike atomic accesses they can coexist in the memory system with atomic or timing accesses. Functional accesses are used for things such as loading binaries, examining/changing variables in the simulated system, and allowing a remote debugger to be attached to the simulator.

Therefore any standalone component which is meant to communicate with other, must implement three functions: **recvTimingReq**, **recvAtomic**, **recvFunctional**. The class diagram of gem5 is available in:

`http://www.gem5.org/docs/html/annotated.html`

Also, for more information the following links can be looked up:

`http://www.m5sim.org/Documentation`.

`http://www.m5sim.org/General_Memory_System`.

Next point to mention is that, the gem5 simulator itself is basically passive; on invoking gem5, it simply executes the user's simulation script, and performs actions only when called by the script. Simulation scripts written in Python control the configuration and execution of gem5 simulations. Simulation scripts are executed by the Python interpreter. As an example a CPU can be instantiated in the python scripts as follows:

```
cpu = SimpleCPU(clock = '2GHz', width = 2)
```

A typical simulation script has two phases: a configuration phase, where the target system is specified by constructing and interconnecting a hierarchy of Python simulation objects; and a simulation phase, where the actual simulation takes place. Simulation scripts can also define command-line options which allow users to control either or both of these phases. For more information about gem5's Python scripts please refer to:

`http://www.m5sim.org/Configuration_/_Simulation_Scripts`

Figure9 illustrates the directory structure of gem5 highlighting the main modifications performed in it. All heavily modified files start with a prefix of "**ethz_**". Moreover, gem5 allows multiple build targets depending on user's needs. According to gem5's online documentation here are the most useful targets: (The build target of gem5 has been defined in UTILS/default_params.sh as GEM5_SIM_MODE and can be easily overridden in the simulation script. The default value for this variable is **fast**.)

- **debug**: gem5.debug has optimizations turned off. This ensures that variables won't be optimized out, functions won't be unexpectedly inlined, and control flow will not behave in surprising ways. That makes this version easier to work with in tools like gdb, but without optimizations this version is significantly slower than the others. You should choose it when using tools like gdb and valgrind and don't want any details obscured, but otherwise more optimized versions are recommended.

- **opt**: gem5.opt has optimizations turned on and debugging functionality like asserts and DPRINTFs left in. This gives a good balance between the speed of the simulation and insight into what's happening in case something goes wrong. This version is best in most circumstances.

- **fast**: gem5.fast has optimizations turned on and debugging functionality compiled out. This pulls out all the stops performance wise, but does so at the expense of run time error checking and the ability to turn on debug output. This version is recommended if you're very confident everything is working correctly and want to get peak performance from the simulator.

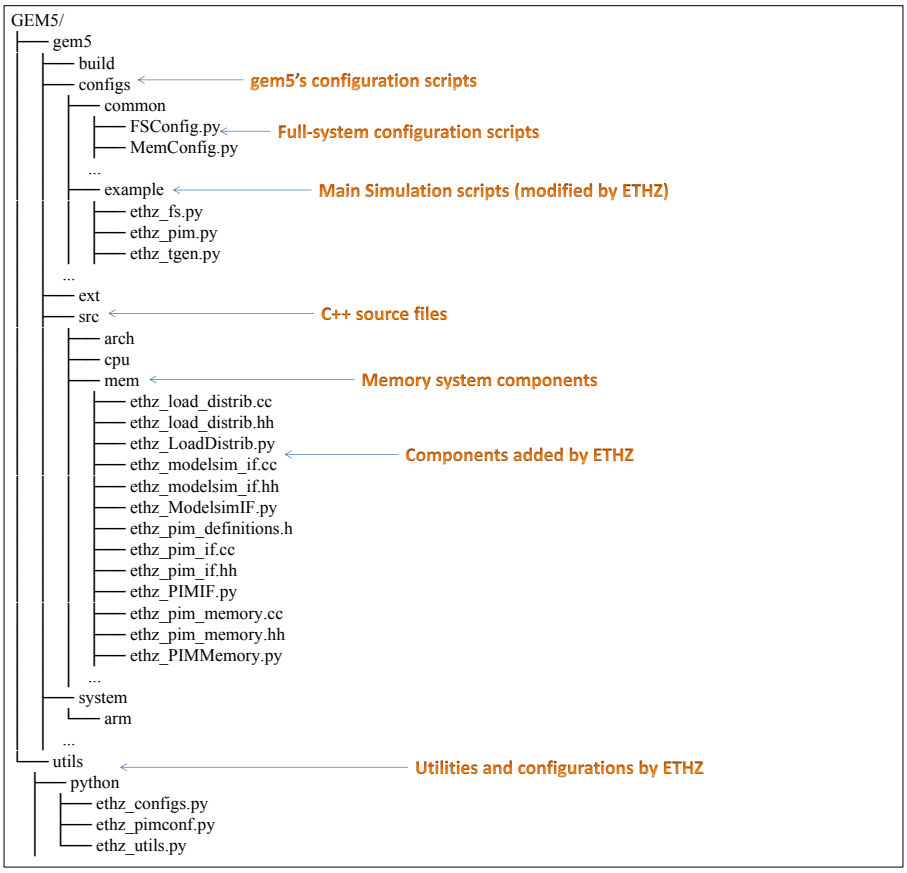

Figure 9: Directory structure of gem5 focusing on the files added by ETHZ

One last point to mention is that, gem5 is open-source and all source codes, scripts, and disk images can be obtained from its main web site. In the SMC simulation environment, however, all required files are already shipped within the two archives (SMC.tgz and SMC-WORK.tgz) and no additional package is required to be downloaded from gem5's website. Anyway, here is an explanation of how we have obtained gem5 related files:

The stable version of gem5 can be downloaded from:

http://repo.gem5.org/gem5-stable

ARM full system images can be downloaded from:

http://www.gem5.org/dist/current/arm/aarch-system-2014-10.tar.xz

A list of all available downlods is present in:

http://www.m5sim.org/Download

Compatible linux kernel sources and an explanation of how to build them is available in:

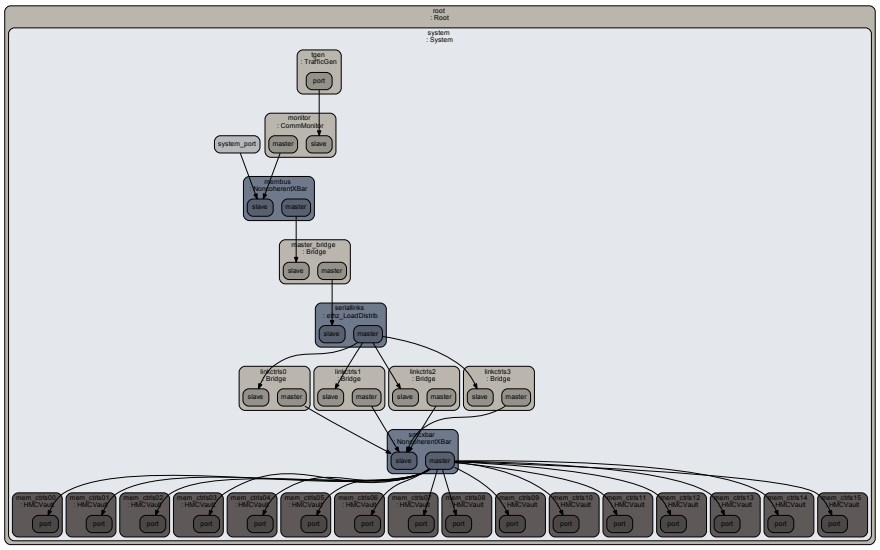

Figure 10: Modelling the Hybrid Memory Cube in gem5

http://www.m5sim.org/ARM_Linux_Kernel

2.4 The Smart Memory Cube (SMC), Modelled in gem5

A high level model of the Hybrid Memory Cube has been designed in gem5 based on the gem5's general memory system. Figure10 illustrates the baseline model inside gem5 where a synthetic traffic generator is connected to the hybrid memory cube using four serial links and through a load distributor. Each *HMCVault* is a DRAM channel controller and *smcxbar* is the main interconnect connecting the serial links to the vault controllers. The python script describing the SMC model can be found here: (./GEM5/utils/python/ethz_configs.py) This section briefly describes each component and the modifications performed in them.

2.4.1 Vault Controllers

The vault-controllers are DRAM controllers inherited from gem5's DRAMCtrl class. Here are the main source files for this class:

./GEM5/gem5/src/mem/DRAMCtrl.py

./GEM5/gem5/src/mem/dram_ctrl.cc

./GEM5/gem5/src/mem/dram_ctrl.hh

DRAMCtrl.py has been modified and a new subclass has been added to it called the HMCVault. The parameters of this subclass have been adjusted to

match the a simple vault controller in HMC (Previously described in [**?**]).

2.4.2 The Interconnect

The main interconnect in SMC is a direct instantiation of the class **NonCoherentXBar** in gem5. This class supports advanced features such as address interleaving and automatically identifies the address ranges of the slave components connected to it. Here are the related source codes for this class:

./GEM5/gem5/src/mem/XBar.py

./GEM5/gem5/src/mem/noncoherent_xbar.cc

./GEM5/gem5/src/mem/noncoherent_xbar.hh

In the meeting on October 15, 2014 it was discussed that maintaining hardware managed coherence between host and the memory stack is not suitable. (Meeting Summary-141015 - Point 2). This is the reason why the main interconnect of SMC has been instantiated from **NonCoherentXBar**.

2.4.3 Serial Links

Serial links have been modeled by a fixed latency using gem5's **Bridge** class. A load distributor dispatches received packets among the serial links and receives back the responses.

2.4.4 Load Distributor

According to the HMC Standard [HMC, 2014] the serial links in HMC behave similarly and can accept packets from the whole address range. For this reason a load distributor on the host side is required to dispatch request packets to these serial links and utilize their bandwidth efficiently. For this reason a class called **ethz_LoadDistrib** has been inherited from **NonCoherentXBar** with two main modifications:

First, it accepts the same address range on all its master ports (this is contrary to the standard behavior of gem5's interconnects). This happens in *recvRangeChange()* method. second, upon receiving of request packets in *recvTimingReq()*, it chooses one of the serial links in a round-robin fashion and transmits the packet to it. This way all links will be equally utilized. The related source codes for this class are the followings:

./GEM5/gem5/src/mem/ethz_LoadDistrib.py

./GEM5/gem5/src/mem/ethz_load_distrib.cc

./GEM5/gem5/src/mem/ethz_load_distrib.hh

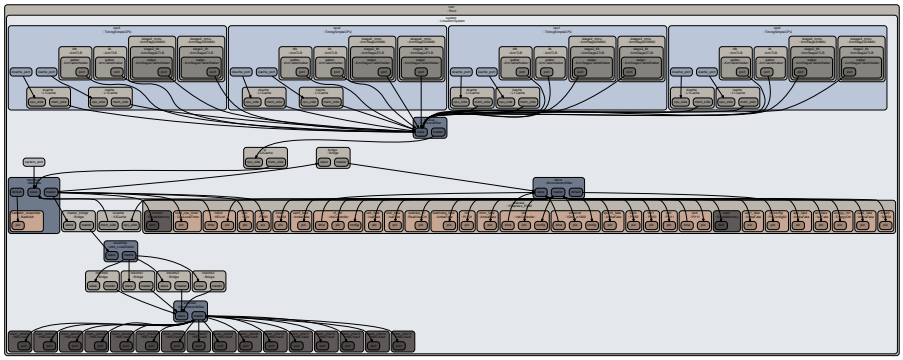

Figure 11: Full system simulation of ARM and the SMC model

3 Full-System Simulation of SMC in gem5

This section describes how to attach the SMC model to a complete ARM system or a traffic generator inside gem5. The following command should start a full system simulation of four ARM8 processors connected to the SMC model as well as the peripheral devices:

```
./scenarios/c-gem5/a-tests/0-arm-hmc.sh 8 -o
```

The resulting system is illustrated in Figure11. A look at **system.xbars. addrmap.txt** in the work directory reveals that the 16 memory vaults are mapped in the range [0x80000000, 0x9FFFFFFF] summing to a total of 512MB. The main script describing this full system simulation is:

```
./GEM5/gem5/configs/example/ethz_fs.py
```

3.1 Communication with the Guest OS

In order to run benchmarks on the guest operating system and send and receive files to/from it a communication mechanism is required. We should remind that the main simulation disk images are protected using a **COPY-ON-WRITE** layer for improved simulation performance and to avoid disk image corruption by the user. Therefore, by default the changes of the guest OS to the disk images are simply lost by closing the simulation.

To enable explicit and easy transfer of files between guest and host (regardless of the OS type), we have created an empty disk image (extra.img) which can be mounted by both guest and host enabling file transfer between them. Here is how the procedure works:

A specific bash utility for this purpose has been devised called **copy_to_ extra_image**. This utility will mount the extra image (extra.img), copy the files to it and then unmount it. Inside the guest OS, another utility has already been placed on the disk image called **/get**. Calling this script will mount the extra image, move all the data on it to /work/ directory, and then unmount it.

The important point is that, never host and guest should mount the disk image at the same time. This will result in corruption of the image. In case of such event, simply delete the image file. The environment will automatically create a new one. A reverse procedure should be followed to transfer files from guest to host (no automatic utility is available for it in this version).

Lastly, instead of full system simulation, it is possible to simply connect the SMC model to a traffic generator. This can be achieved using the following scenario script: (Resulting system will be similar to Figure10)

```
./scenarios/c-gem5/a-tests/2-tgen-max-bw.sh
```

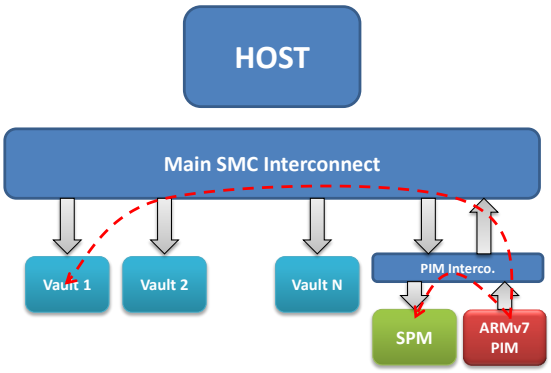

Figure 12: An ARMv7 based PIM device attached to the main SMC interconnect

4 Adding a Processor-in-Memory (PIM) to the SMC Model

It has been previously shown [Azarkhish et al., 2016] that it is possible and safe in terms of traffic interference to attach a PIM device to the main interconnect, allowing for global visibility to the whole address space. This PIM device can access the main memory periodically (e.g. using a double buffering DMA engine), carry out computations, and store back the results. This section will describe a simple ARMv7 based PIM.

4.1 Simple ARMv7 based PIM device

An ARMv7 core has been attached to the main SMC interconnect in gem5 (See Figure12). Similar to the host processors, the simulation mode of this core can be chosen among **Atomic**, **Functional**, and **Timing**. Moreover, a small Scratchpad Memory (SPM) has been added to this device to store the codes running on this device. Besides, this SPM is mapped in the physical memory map of the whole system and can be accessed by the host processors. PIM device, also, has a global visibility of the whole memory space. Figure13 depicts a screenshot of the PIM device attached to the main interconnect of SMC. Here is the main script describing this PIM device:

`./GEM5/utils/python/ethz_pimconf.py`

In this file, a new class called **PIMSystem** is inherited from **ArmSystem** which includes an interconnect (pimbus), an interrupt controller (intrctrl), and a memory (pim_memory). The python function **build_pim_system()** is responsible building this subsystem and passing the required parameters such as clock frequency and voltage to it. These modifications are enough to have a bare-metal PIM device without any operating system. Next section describes the required software stack to enable communication between host and PIM.

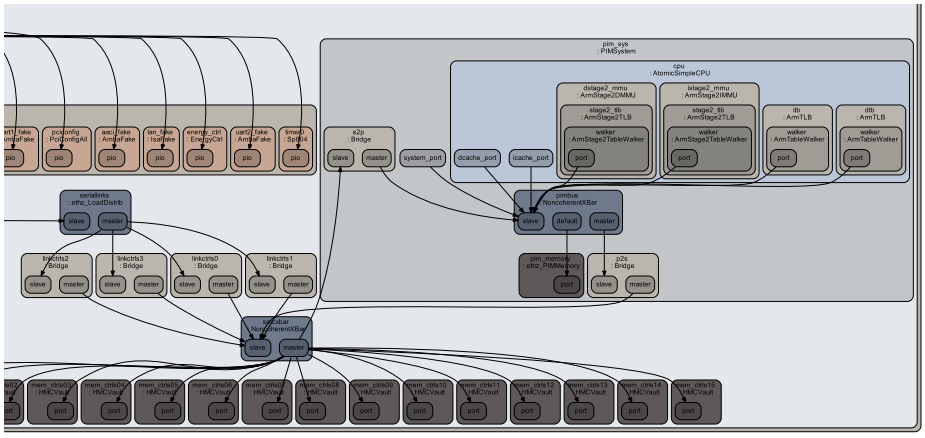

Figure 13: A screenshot of the ARMv7 based PIM device inside gem5

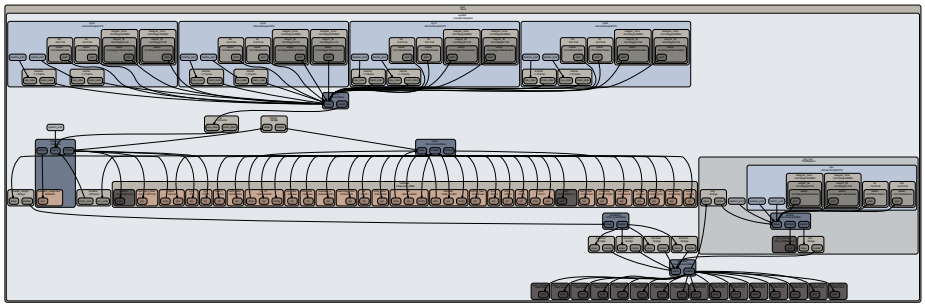

Figure 14: Full-system simulation of the ARM system along with the ARMv7 PIM device

Figure14 depicts the whole system composed by the host processors, the interconnects and the memory system, as well as, the ARMv7 PIM device.

4.1.1 PIM Memory

The class ethz_PIMMemory has been inherited from gem5's SimpleMemory class with slight modifications to accommodate the special registers of the PIM device. Here are the related source codes for this class:

./GEM5/gem5/src/mem/ethz_PIMMemory.py
./GEM5/gem5/src/mem/ethz_pim_memory.cc
./GEM5/gem5/src/mem/ethz_pim_memory.hh

Inside ethz_PIMMemory.py, the special registers such as PIM_STATUS_REG have been defined, and it is the responsibility of the configuration script (ethz_pimconf.py) to set the correct physical address previously extracted from the PIM ELF file to these variables (Please refer to Section 4.2.1 for more in-

formation). The functionality of these registers has been implemented in in ethz_pim_memory.cc, inside the **recvAtomic()** method. Here is a short explanation:

- **PIM_DEBUG_REG**: This register will simply echo the received characters to the screen. This can be used as a debugging facility for the PIM device which does not have any peripheral devices such as UART.
- **PIM_ERROR_REG**: A write to this location which is not equal to PIM_ERROR_NONE will exit the whole simulation. This can be used by the PIM device to kill the simulation in case of a catastrophic failure. The pim_error() method in the resident program of the PIM device performs this operation.
- **PIM_INTR_REG**: Writing to this register will send an interrupt to the PIM processor and will wake it up from the sleep mode. Therefore, the host processor can wake PIM up by simply writing a value to this register. This register is simply wired to the external IRQ0 of the PIM processor. No interrupt handling routine is required because there is nothing to be done and the PIM processor can return to normal execution after waking up.

4.2 Basic Software Stack

The structure of the SW/ directory has been shown in Figure15.

4.2.1 The Resident Program

The PIM device should be able to react to the commands sent from the host processor. Therefore a resident program is required on it to receive the commands and perform the required operation. Figure16 illustrates the main loop of this program. As can be seen, after initialization, the PIM device goes into sleep mode by calling the assembly instruction "WFI". The device remains in this state until it is waken up by host writing to PIM_INTR_REG register. After waking up from sleep, the PIM device checks if a new command has arrived, and after execution of this command, again will go to sleep. A group of memory mapped registers have been defined in **registers.h**, which are mapped in the scratchpad memory starting from address 0x70000000. The host processors can write to or read from these registers. A list of these registers is shown in Figure17. Also a list of the legal values to be written to these registers are present in **definitions.h**. The resident code will be compiled and linked statically using the link script present in:

`UTILS/models/system/bare_metal_link_arm7.sh`

This script adjusts the sizes in the ELF code (stack, BSS, ...), and specifies 0x70000000 as the base address for loading. All these parameters are present in UTILS/default_params.sh and can be modified in the scenario script. To be able to run this code on the bare-metal ARM core, a boot loader is required. **boot.S**

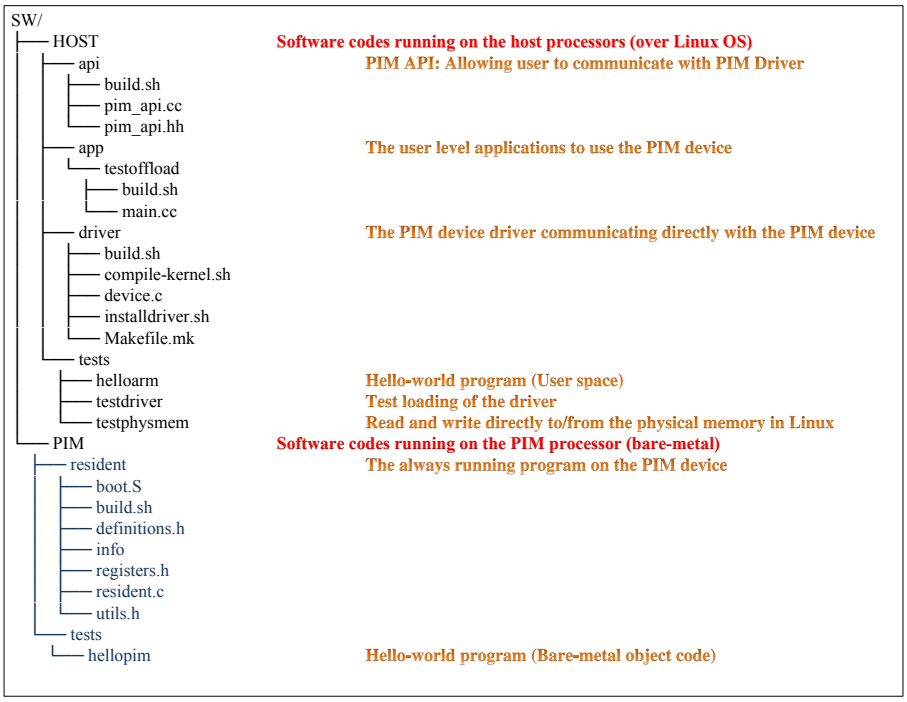

Figure 15: The structure of the SW/ directory containing the software codes

is a very simple boot-loader which sets the stack pointer and jumps to the entry point of the program (c_entry). The build script (build.sh) compiles the resident code along with the boot loader and links them using the link script. The resulting ELF file can be executed directly on the ARMv7 PIM device. After building, the symbol addresses inside this ELF code are automatically extracted using the utility *get_symbol_addr* written for this purpose. An example of these addresses is shown in Figure18. Next section describes how the driver and API communicate with this device.

4.2.2 Driver and API

A user program running on the host operating system can only access authorized virtual memory locations. To be able to communicate with the PIM device in a memory-mapped fashion, an intermediate software layer is required (See Figure19) to first translate the user space virtual addresses to kernel space virtual addresses, and then to the physical addresses. The device driver simply maps physical address to kernel space using the **ioremap_nocache()** function. Building the driver produces an object file called **device.ko** which can be installed in the guest OS using the **insmod** command. A node is then created for the device in /dev/DEVICE to be able to access it from the API. An ex-

```
int c_entry() {
        // Initialize device registers
        pim_print_msg("initializing registers ...");
        initialize_registers();                          Initialization

        // Main loop
        pim_print_msg("Going to sleep ...");
        while (1)                                        Infinite Loop
        {
                pim_print_msg("[SLEEP]");
                __asm__("wfi");                          Sleep
                pim_print_msg("[WAKE-UP]");
                if ( new_command_received() )            Process received command
                {
                        pim_print_msg("Start execution of the kernel ...");
                        pim_wait(100000);                Dummy Execution
                        pim_print_msg("Finished execution of the kernel ...");
                        command_done();
                }
                else
                        pim_error("Woke up but no new command has arrived!");
        }
        pim_exit(PIM_ERROR_FAIL);
        return 0;
}
```

Figure 16: Main loop of the resident program on the PIM device

```
/* The registers of the PIM device */
uint8_t     PIM_STATUS_REG;                  // The status of the PIM device
uint8_t     PIM_COMMAND_REG;                 // Command given from host to the PIM
uint8_t     PIM_ERROR_REG;                   // Error code is written here when an error happens
uint8_t     PIM_INTR_REG;                    // Interrupt register to wake-up the PIM device
uint8_t     PIM_DEBUG_REG;                   // For debug purpose only

/* Vectors of the PIM device */
uint8_t     PIM_VECTOR_A[PIM_VECTOR_SIZE];   // Vector A
uint8_t     PIM_VECTOR_B[PIM_VECTOR_SIZE];   // Vector B
uint8_t     PIM_VECTOR_C[PIM_VECTOR_SIZE];   // Vector C
```

Figure 17: Memory mapped register to enable host-PIM communications

planation of scenario script to perform these operations is presented in Section 4.3.

The API simply opens the memory mapped device and maps the whole range to the user space. Moreover, to hide the details of communication from user, it exposes a set of high-level methods and protects unwanted operations. Figure20 shows an overview of the API code. This code can be linked as a static library and the pimAPI class can be instantiated easily in the user level code.

```
700001e8   t   _entry
700001e8   T   _start
7000020c   t   bootldr
700007f4   T   c_entry                  The entry point: int c_entry()
70009bb0   B   PIM_ERROR_REG
70009bc4   B   PIM_COMMAND_REG          Command Register
70009bc5   B   PIM_STATUS_REG           Status Register
70009bc6   B   PIM_INTR_REG             Interrupt register
70009bc7   B   PIM_DEBUG_REG
70009bb4   B   PIM_VECTOR_A
70009bc8   B   PIM_VECTOR_B             The vector registers
70009bbc   B   PIM_VECTOR_C
7000abd0   B   stack_top
...
```

Figure 18: Symbols extracted from the resident ELF code

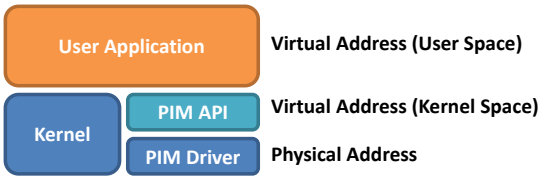

Figure 19: An overview of how the user level application should communicate with the PIM device

4.2.3 Sample User Level Application

An example user level application is available in SW/HOST/app/testoffload. The purpose of this program is only to test the communication mechanisms its performance may not be optimal. This code simply sends two vectors to the PIM device, asks it to add them (start_computation()), and after the results are ready (wait_for_completion()), prints them on the screen.

An important point to mention here is that, in the current HMC specification [HMC, 2014] no interrupt or messaging mechanism from the memory cube to the host is devised. Therefore, the only way for the host to know the status of the PIM device is to keep polling it. This version of the API implements wait_for_completion() with a polling of the status register. An overview of the operation has been shown in Figure21.

4.3 Running the PIM Scenario

Prior to running PIM scenarios, the kernel source code provided in **SMC-WORK/linux_kernel** should be built successfully. This is required for the PIM device driver to be built correctly. For this purpose change the directory to the 32-bit Linux kernel:

```
cd SMC-WORK/linux_kernel/linux-linaro-tracking-gem5-ll_20131205.0-gem5-a75e551
```

Then run the following script:

```
./ethz-build-kernel.sh
```

```
class pimAPI
{
public:
        pimAPI();
        void write_vector( uint32_t offset, uint8_t* value );
        void read_vector ( uint32_t offset, uint8_t* value );
        void start_computation( uint8_t command );
        void wait_for_completion();
        void offload_kernel(string name);

protected:
        inline void wake_up();
        inline void give_command(uint32_t command);
        inline uint32_t check_status();
        // Write and Read to/from the PIM device's registers
        inline void write_byte(uint32_t offset, uint8_t data);
        inline uint8_t  read_byte(uint32_t offset);
        // pointer to the memory mapped location
        char* deviceptr;
};
```

Figure 20: An overview of the pimAPI class

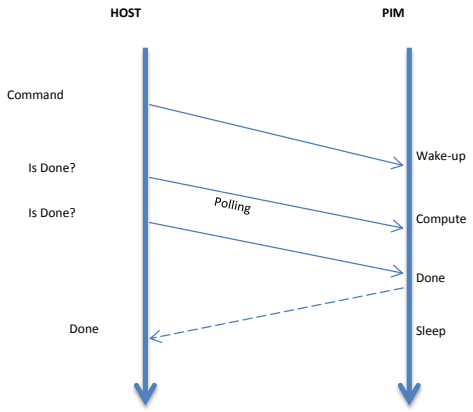

Figure 21: An overview of communication of host with PIM

This procedure must finish without any errors. Similarly, to make sure the 64-bit version of Linux can be built correctly:

```
cd SMC-WORK/linux_kernel/linux-aarch64-gem5-20140821
./ethz-build-kernel.sh
```

The scenario which performs all the operations mentioned in the previous

subsections is:

```
./scenarios/c-gem5/a-tests/5-arm-pim-offload.sh 7 -o
```

This scenario script builds the driver, the API, the resident program, and the sample user level application; then copies all required files into the extra disk image to be used in the guest OS. Next, it executes an ARMv7 system simulation running a Linux OS, along with an ARMv7 PIM device attached to the SMC model. The number of CPUs and vaults have been reduced to speed-up the simulation, and the caches have been removed for now. Looking inside this script reveals that a utility called **clonedir** copies the software sources into the scenario location, and then build operation is performed in there. This is to maintain SMC/ clean from object and temporary files.

After attaching to the guest OS using a terminal and logging in as root, the user should run the following commands to get the required files from the host system: (Files will be moved to /work/ directory)

```
cd /
```

```
./get
```

Then the driver can be installed as follows:

```
cd /work/
```

```
./installdriver.sh
```

At this point the driver should be installed successfully returning the major and minor numbers. Moreover the device should be mapped successfully in the kernel space. Now the user level program can be executed:

```
./main
```

The source code for this program is available in "SW/HOST/app/testoffload". As explained before, this program writes two vectors to the PIM registers, and asks the PIM to add them, and then reads the results. A screenshot of both gem5's debugger and the attached terminal can be seen in Figure22.

One final point to mention is that, in order to test the PIM device in standalone mode (without attachment to the main interconnect), the following scenario can be executed:

```
./scenarios/c-gem5/a-tests/3-pim-standalone.sh -o
```

This scenario allows for executing a hello-world C program on the PIM device. In order to test this device while attached to the main system, the following scenario can be executed:

```
./scenarios/c-gem5/a-tests/4-arm-pim-hello.sh 7 -o
```

Taking a look at the intermediate file (system.xbars.addrmap.txt) in the work directory reveals that the S2P bridge declares [0x70000000 : 0x700FFFFF] to the host system as the address of the SPM, and the bridge P2S declares [0x80000000 : 0x9FFFFFFF] to the PIM device as the address of the main memory.

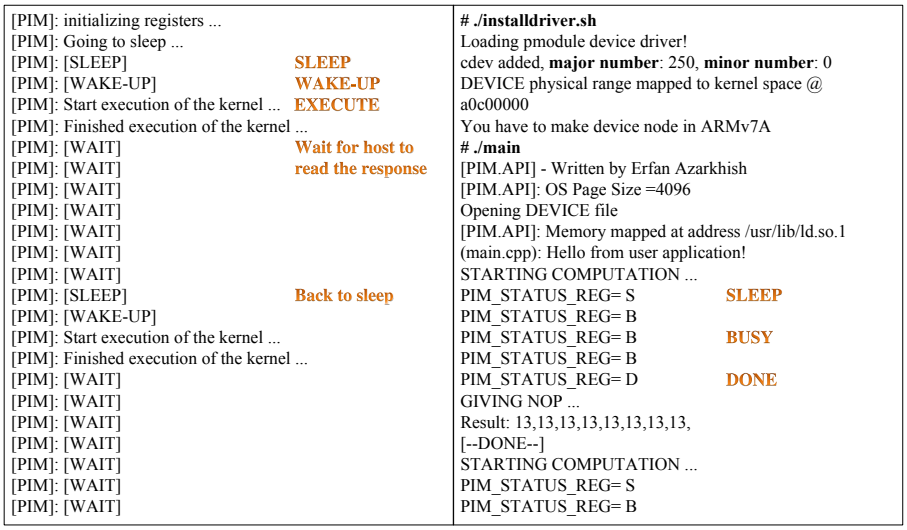

Figure 22: Running the sample user level application. Left: logs from the gem5 simulator, Right: logs from the guest OS

5 Conclusions

In this report, a high-level model for the Smart Memory Cube designed in gem5 was presented. This model has been validated against the previously developed cycle-accurate model of SMC, and is ready for full system simulations. Moreover, an ARMv7 core has been added to model a Processor in Memory (PIM), and a basic software stack has been developed for communication of the host with this PIM.

References

[HMC, 2014] (2014). Hybrid memory cube specification 1.1.

[Azarkhish et al., 2016] Azarkhish, E., Pfister, C., Rossi, D., Loi, I., and Benini, L. (2016). Logic-base interconnect design for near memory computing in the smart memory cube. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, PP(99):1–14.

[Binkert et al., 2011] Binkert, N. et al. (2011). The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7.