**Processing-In-Memory Emulator Software Usage Guide**

June 18, 2013

Executive Summary

Processing-in-memory (PIM), which moves computation closer to where the data resides, has long been recognized as having the potential to improve both the performance and energy costs associated with memory access. Previous research in this area focused on implementing processing capabilities directly on DRAM memory chips. This has yielded poor results due to incompatibilities between the CMOS processes for DRAM and high performance processors logic. With the advent of 3D stacking, it has become possible to implement processors in high performance CMOS as one layer in a larger 3D memory stack.

Rather than focus on details of a hardware design proposal, our PIM research takes a programmability-centric view. We believe that rational choices about future hardware implementations for PIM first require a more thorough understanding of the programmability requirements, coupled with research into what features of a PIM device lead to the most cost effective improvements in performance and energy efficiency for targeted applications. In support of our research, we have developed a PIM user-level application programming interface (API) and a PIM emulator software package.

This report describes the operation and internal functioning of the PIM emulator software, which has the following high level objectives: Allow applications that use the PIM user-level API described in *FastForward Milestone M1: Processing-in-Memory Baseline Architecture and API Report* to run correctly; Present an emulated machine with PIMs for application analysis; Run these applications as fast as possible, for quick turnaround; Gather useful performance and power information as the applications run; Use the information to scale performance and power information to future PIM architectures.

This report is provided in partial fulfillment of the M1 milestone described in the *Extreme-Scale Computing Memory Research and Development Statement of Work*. To complete the fulfillments of the M1 milestone, a PIM emulator software package and a PIM baseline architecture and API report are provided separately.

Table of Contents

1 Introduction 1

1.1 Objectives 2

1.2 Methodology 2

1.3 Conventions 3

2 Platform 4

2.1 Hardware Configuration 4

2.2 Operating System, Compilers, and OpenCL Runtime 5

2.3 Profiling Tools 5

2.4 CPU Frequency 5

3 Getting Started 7

3.1 Source Package 7

3.2 Contents 8

3.3 Prerequisites 8

3.4 Compilation 8

3.4.1 Linkage 9

3.5 Installation 9

3.6 Execution 10

3.6.1 Run Script 10

3.6.2 Test Setup 11

3.6.3 CPU Test 13

3.6.4 GPU Test 15

3.6.5 Combined CPU+GPU 18

4 Internal Function 20

4.1 API Emulator and Runtime 20

4.1.1 Performance Monitor Data 21

4.1.2 Trace Ordering Information 21

4.2 PIM Analytic Model 25

4.2.1 Performance Scaling 26

4.2.2 Trace Finalizer 26

4.3 Model Description Files 27

4.3.1 Format 28

4.3.2 Examples 30

5 PIM Emulator-Specific User API 33

5.1 Motivation 33

5.2 API Specification 33

5.2.1 pim\_emu\_begin 33

5.2.2 pim\_emu\_end 34

5.2.3 PIMEMUENV 36

6 Limitations 37

7 References 39

# Introduction

This report is provided in partial fulfillment of the M1 milestone described in the *Extreme-Scale Computing Memory Research and Development Statement of Work*. The milestone is reproduced below, and the operation and internal function guide is given in italics.

**Milestone M1 (Architected Programming Interface & Emulator for PIM)**

The Subcontractor shall develop an architected programming interface for PIM. This programming interface shall be in the form of a report setting forth the following:

* Methods to initiate operations on a PIM and receive results consistent with the HSA architecture
* Methods to report various types of completion codes and errors
* Methods to integrate PIM with system management, checkpoint restart, debug, and instrumentation
* Cost/benefit analysis of cache-coherent PIM implementations

Additionally, the Subcontractor shall develop PIM emulator software to emulate APUs-in-memory on standard APU hardware.

**Deliverables**:

* A report documenting newly created architecture and interface descriptions with supporting cost/benefit analysis
* *APU-in-Memory Emulator Software A report that documents the operation and internal functioning of the Emulator Software*.

This report is a companion to the *FastForward Milestone M1: Processing-in-Memory Baseline Architecture and API Report*, which is also part of the Milestone M1 milestone.

This report describes the current (initial) release of the PIM emulator software environment. This effort is focused on functionality that enables users to run what we envision to be the most relevant workload types for early PIM evaluations. These include execution of GPU kernels, which are useful for throughput-oriented computing and are relevant for proxy applications. Additionally, support for CPU threads lets users explore the offload of entire kernel graphs to PIM. It also lets users explore other computations that touch large amounts of data but do not have fine-grained synchronization with a host CPU at a fine grain. It should be understood that this report describes only the initial emulator software environment. Refinement of the emulator software and its internal function are expected to continue throughout the remainder of the FastForward research effort. **[\*]**

## Objectives

The PIM emulator software has the following objectives:

* Allow applications that use the PIM User-level API to run correctly
* Present an emulated machine with PIMs for application analysis
* Run these applications as fast as possible, for quick turnaround
* Gather useful performance and power information as applications run
* Use this information to scale performance and power information to future PIM architectures

## Methodology

To achieve these objectives, the PIM emulator uses a two-phased approach, as illustrated in Figure 1.

**Phase 2**

**Phase 1**

**Commodity Hardware (CPU/GPU/APU)**

**Linux**

**Event Trace & Stats**

**User Application**

**PIM API Emulator**

**Model Description**

**Performance and Energy Models**

**PIM Analytic Model**

**Host Tasks**

**PIM Tasks**

Figure 1: Processing “phases” in the PIM emulator

The first phase relies on native execution of the application-under-test on commodity hardware, capturing an event trace and performance statistics during the run. The second phase post-processes this information to estimate overall performance on future memory and processors.

## Conventions

Throughout this report, we will use the following typographic conventions.

1. Directory paths and file names will be given in the following typographic format:  
     
   **/path/to/file**
2. Command line invocation and output will be given in the following typographic format:  
     
   **$ command [arg0 [arg1 …]]**
3. Source code listings will be given in the following typographic format:

void\* pim\_map(void \*handle, pim\_platform\_type platform);

# Platform

The PIM emulator model runs an application-under-test on a “real system”. In other words, the binary of the application-under-test executes on existing hardware, and “sees” a PIM-enabled platform through the emulator. This real system ideally has at least a few CPU cores and a GPU, and it must allow hardware performance monitoring data to be gathered on these devices.

The emulator itself is a shared library that should be linked against any application that wishes to use the PIM user-level API. Calls to the PIM user-level API functions, such as pim\_spawn() (a function that launches work onto a PIM from the host processor), will be seen by the library and emulated. In this way, the emulator yields the same application results that would be obtained if the code were run on real PIM hardware.

The PIM performance model estimates performance for a “modeled system” based on the performance data gathered on the real system. The real system does not need to look like the hardware being modeled; it obviously will not have a PIM. In fact, the real system does not need to have the same number of GPUs or CPUs as the PIM-enabled system being modeled. This performance model is useful when analyzing whether it is beneficial to run an application on PIMs or whether parts of the application would be better suited for execution on the host processor.

The following sections describe the primary system where the PIM emulator software has been successfully deployed, including a brief description of the hardware configuration and software packages.

## Hardware Configuration

The hardware components required by the PIM emulator software are: An AMD “Bulldozer” based Opteron CPU, or newer; AMD Radeon “Southern Islands” discrete GPU. The hardware components used in our primary system are described in more detail here.

The primary system is based on dual-socket AMD Opteron 4274 HE processors, each with 8 cores at a 2.5GHz rated frequency, 4x2MB L2 cache, and 8MB L3 cache. An AMD ‘Pence 2’ reference board is used with six 1GB DDR3-533 DIMMs, with three DIMMs attached to each socket in a single-channel configuration and a 250GB Seagate 7200.10 SATA hard disk drive that operates at 7200 RPM. This system is capable of approximately 13 GB/Sec running Stream Triad. The test system also includes an AMD Radeon HD 7950 discrete GPU. This GPU is based on AMD’s Graphics Core Next (GCN) architecture, and includes 28 compute units running at a rated frequency of 800MHz, as well as 3GB of GDDR5 with a 1.25GHz memory clock.

## Operating System, Compilers, and OpenCL Runtime

The OS used on the test system is Community Enterprise Operating System (CentOS) 6.3, 64-bit (which uses version 2.6.32-279 of the Linux kernel). Version 13.4 of the proprietary AMD Catalyst GPU drivers is used to enable the GPU. The AMD ‘APP SDK 2.8’ is used to provide OpenCL1.2 support for both the CPUs and GPUs. The bulk of the OpenCL runtime functionality is in the GPU drivers (e.g. OpenCL compiler, finalizer). The GNU Compiler Collection (GCC) 4.7 is used to compile all host-side code. Only basic compiler optimization flags (e.g. -O3) are used.

## Profiling Tools

A variety of internal and publicly available tools are used for profile data collection and hardware monitoring. These include: AMD CodeXL-Linux v1.1.1537.0 and Linux PAPI v5.1.0.2. These tools are used at runtime by the PIM emulator to gather GPU and CPU hardware performance monitor information, respectively.

## CPU Frequency

The PIM performance model requires information about CPU frequency on the test machine where profiles are to be collected. The PIM run script described in section 3 allows the user to pass this information via command line parameters.

There are some additional CPU frequency settings that the user needs be aware of. Modern processors may *boost* the frequency of the CPU when in certain operating states. However, the PIM performance model requires that CPU frequency on the test machine be kept constant during test execution. The user needs to disable turbo boost and disable any power governor daemons that change the P-state of the processor.

The CPU frequency of the test machine can be controlled by various mechanisms. Below are the guidelines that have been used on the primary test system.

1. Turn off turbo boost in the BIOS on boot. This feature is normally called AMD Turbo Core or Core Performance Boost. This needs to be done just once with this option. ***OR***
2. Turn off manually turbo boost every boot time for every cpu{number} in the system, as follows:  
     
   **# echo 0 > /sys/devices/system/cpu/cpu{number}/cpufreq/cpb**

# Getting Started

This section describes the operation of the PIM emulator software, including the organization of the source package, its compilation, installation, and execution of applications under emulation.

## Source Package

The PIM emulator software package comes archived and compressed in a file named

**amd-ff-pimemu-m1.tar.gz**

The PIM emulator software package may be extracted into any directory where the user has write-permission. By way of example, suppose the PIM emulator software package is to be extracted into the following directory path:

**/pimemu**

The package can be extracted as follows:

**$ cd /pimemu**

**$ tar –zxvf amd-ff-pimemu-m1.tar.gz**

This will result in PIM emulator software being extracted into the following location:

**/pimemu/lib**

Unless otherwise noted, all path and file names in following sections are relative to this location.

## Contents

The PIM emulator software package includes source code implementing the emulator, as well as the User-level API documented in the *FastForward Milestone M1: Processing-in-Memory Baseline Architecture and API Report*. The software package also includes test programs and shell scripts aimed at demonstrating sample usage of the emulator.

The source structure for the PIM emulator software package is described in the README.txt file located in the top-level directory, as follows:

* **src/model/** all of the performance model source files, including CPU and GPU analytical performance models and the execution segment reordering tools.
* **src/pimemu/** all of the PIM emulator files.
* **src/utils/** source files for machine specification XML file parsing and validation.
* **config/** all XML machine description files.
* **test/** all test files (source, headers) used for emulator validation.
* **prereqs/** link to roxml source, patch, and installation instructions.
* **lib/** The PIM emulator library is copied here after compilation.
* **bin/** The PIM emulator run script and the Python graph script.

## Prerequisites

In addition to the hardware and software platform described in section 2, the PIM emulator software also requires that the open-source libroxml-2.2.1 package be installed on the system. This library is used to help parse the XML files that are used to describe the PIM-enabled machine that the emulator will present to software.

The file prereqs/README includes detailed instructions for installation of libroxml-2.2.1. On CentOS 6.3 systems, a patch is needed to ensure that libroxml-2.2.1 compiles and installs correctly. A patch is included in the prereqs directory. The user should install libroxml-2.2.1 as described before proceeding.

## Compilation

The PIM emulator software package can be compiled from top-level directory by typing:

**$ cd /pimemu/lib**

**$ make**

### Linkage

In order for a program to use the PIM emulator software package, it must add the following to the list of linker dependencies:

**-lpim –lpthread –rdynamic**

The program must additionally remove dependencies to –lOpenCL

The PIM emulator software package includes an example Makefile in the top-level directory, which contains linkage rules for test programs that are described in Section 3.6 of this report.

## Installation

Previously compiled PIM emulator software can be installed by typing:

**$ make install**

By default, this will place emulator header files, libraries, and binaries into the system directory:

**/usr/local/include - files installed here are**

**pim.h**

**/usr/local/lib – files installed here are**

**libpim.so.\***

**libpim.a**

**/usr/local/bin - files installed here are**

**ordering\_grapher.py**

**pim\_amodel.exe**

**pim\_perf\_model.sh**

**run\_pim\_emulation.sh**

Note: The default installation path requires that the user have permissions to write to /usr/local. Alternatively, the user can install the PIM header files, libraries, and binaries into another location, for example

**/piminstall/include**

**/piminstall/lib**

**/piminstall/bin**

By typing:

**$ make install prefix=/piminstall**

After installation the user can compile then run test programs by typing:

**$ make test**

The individual test programs will produce appropriate error messages if there are failures.

## Execution

The following discussion illustrates and describes the outputs of running three sample test applications that are included with the PIM emulator software package.

Prior to running the sample applications described, it is required to have properly compiled then installed the PIM emulator library and its associated test programs, as described in the preceding sections. For the purpose of this discussion, we assume the PIM emulator software package and install directories are:

**/pimemu/lib**

**/piminstall**

### Run Script

The PIM emulator run script can be used to execute an application that has been ported to the PIM API. This utility script sets the PIMEMUENV environment variable (which is described in Section 5 of this report), executes the application, generates CPU and/or GPU performance profiles, executes the PIM analytic performance model to generate time ordered output in ASCII format and graphically plots the results in a PDF image.

The PIM emulator run script itself is installed into:

**/piminstall/bin/run\_pim\_emulation.sh**

The PIM emulator run script can be invoked with the ‘-h’ option to print a help message:

**$ /piminstall/bin/run\_pim\_emulation.sh –h**

This produces the following help message:

**Usage: run\_pim\_emulation.sh [Optional flags] --run <command to run with its arguments>**

**Specify Optional flags before Required flags**

**Required:**

**--run : The command/program to run in PIM emulation**

**Optional Flags:**

**-h : Print this help message**

**-c : Path to XML configuration file for PIM, defaults to PIM.xml in current working directory: /pimemu/lib/test**

**-e : Path to ROOT of PIM emulator installation directory, defaults to: /usr/local**

**-f : CPU core freq in MHz of the machine where profile was collected, defaults to: 2500**

**-l : memory access latency in nanoseconds of the machine where profile was collected, defaults to: 52.5**

**-o : Path to store result of emulation run, defaults to current working directory: /pimemu/lib/test**

**following result files are generated: emulation\_result.csv, emulation\_result.pdf**

**-p : Path to Papi library, defaults to: /usr/local/lib/**

**-r : Path to ROXML library, defaults to: /usr/lib64/**

**-s : Path to Sprofile binary, part of AMD CodeXL tool, defaults to: /opt/AMD/CodeXL/bin/x86\_64**

**-t : Path to store temporary files used for tracing and modeling, defaults to current working directory: /pimemu/lib/test**

**following temporary files are generated on full tool run: pimemu\_cpucounterdata.csv, pimemu\_gpucounterdata.csv, model\_cpu.csv, model\_gpu.csv, pimemu\_timelinedata.csv**

**-x : Without -x option, by default the tool runs all commands in order : Profile, Modeling, and Timeline Ordering.**

**Use -x with arguments to run only one of them :**

**1 - run profiler, this runs the application and generates performance profiles: pimemu\_cpucounterdata.csv, pimemu\_gpucounterdata.csv, pimemu\_timelinedata.csv**

**2 - run models, reads peformance profiles and generates performance predictions: model\_cpu.csv, model\_gpu.csv**

**3 - run timeline ordering, reads performance predictions and generates timeline ordering results: emulation\_result.csv, emulation\_result.pdf**

**Commands 1 and 2 assume read/write from TMPDIR, change it using -t option**

**Command 3 reads predictions from TMPDIR change with -t option, and writes results to OUTDIR change with -o option**

**Example: run\_pim\_emulation.sh -c config.xml -t /tmp -o /result --run my\_prog my\_args**

Figure 2: Help message from the PIM emulator run script.

### Test Setup

The source code for the sample test programs is located in the PIM emulator software package under the subdirectory:

**/pimemu/lib/test**

For the remainder of this discussion, we will use this as our current working directory.

**$ cd /pimemu/lib/test**

The PIM emulator run script will require a subdirectory called tmp/ to store intermediate files and a directory called result/ to store the final result of the emulation.

**$ mkdir tmp**

**$ mkdir result**

The PIM emulator run script requires an XML model description file as input. Here, we copy a sample XML model description file from the config/ subdirectory of the PIM emulator package.

**$ cp /pimemu/lib/config/PIM.xml .**

### CPU Test

The CPU test program computes a prefix sum in parallel using POSIX threads. The program source is called test\_cpu\_thread.c, the compiled binary executable is called test\_cpu\_pthread.exe, and the test can be run as follows:

**$ /piminstall/bin/run\_pim\_emulation.sh –c 2.xml \**

**–e /piminstall \**

**–t /pimemu/lib/test/tmp \**

**–o /pimemu/lib/test/result \**

**--run ./test\_cpu\_pthread.exe**

The following temporary files are produced:

**$ ls -1 /pimemu/lib/test/tmp**

**model\_cpu.csv**

**pimemu\_cpucounterdata.csv**

**pimemu\_timelinedata.csv**

The file pimemu\_cpucounterdata.csv holds the hardware performance counter data for each segment of the pthreads-enabled program. These values are later used as input to the analytical performance-scaling model, and are transformed into the values held in model\_cpu.csv. The latter file holds the performance of each segment in the program as it is estimated to be on the modeled PIM-enabled system. Finally, the file named pimemu\_timelinedata.csv holds information about the required order of the segments in the POSIX thread program. Because the modeled machine can look quite different than the machine on which the program was run, the required ordering of each segment in the program is needed so that a legal ordering can be regenerated on the modeled machine.

In the next step of the performance modeling process, the following result files are produced:

**$ ls -1 /pimemu/lib/test/result**

**emulation\_result.csv**

**emulation\_result.pdf**

The ordering information between program segments and their scaled runtimes are used to build a program trace as a final output result. These files contain this output trace represented in ASCII and PDF formats, respectively. The file emulation\_result.csv is primarily a textual representation of the runtime of each segment and a description of hardware it runs on. However, the last line in the file also includes a summary that describes the total runtime of the program. A sample of the graphical output contained in the file emulation\_result.pdf is illustrated in Figure 3.



Figure 3: Graphical output from running the simple CPU test.

This graph in Figure 3 shows five segment intervals, which correspond to the estimated program execution times on the modeled system described in the configuration file 2.xml. Note that the CPU test includes an initialization stage that is not relevant to the performance model for this application. The initialization stage is eliminated from captured output by calling *emulator-specific APIs* that are described in Section 5 of this report.

The runtime of each segment is generated by first gathering hardware performance monitor values as the emulated program runs on the real system. These are gathered from both the CPU and GPU, and are associated with the events that occurred while a segment was executing. The gathered events vary depending on the device being monitored, since the events that cause execution inefficiencies are dependent upon the microarchitecture of the processor. On the CPU, for example, we measure events such as cache misses and pipeline stall cycles, since CPUs are vulnerable to stalls due to long-latency operations. On the other hand, we measure values on the GPU such as the total amount of data transferred to DRAM and number of wave fronts executed, since these devices are often compute-resource or bandwidth limited.

These values are then fed into analytical performance prediction models that estimate the runtime of each segment when executed on the emulated PIM hardware. Performance prediction of this nature is an active area of research, because accurate performance prediction from online hardware analysis is useful in techniques such as dynamic voltage and frequency scaling (DVFS) and for accurately estimating benefits from potential algorithmic changes [1, 2, 3]. As such, we are in the process of analyzing the events and analytical formulae to understand which models will most accurately describe the performance of a future PIM system.

Currently, we do not expose these individual events to users in our final output file (model\_cpu.csv). We list the runtime of each segment based on analyzing these counters, but we do not directly display the effect of any individual counter on the total performance. Instead, our simulator displays the final estimated aggregate performance. We are currently researching methods of presenting useful performance-enhancing recommendations to PIM developers based on the statistics we gather, and are actively soliciting input for statistics that expert users may find useful. **[\*]**

### GPU Test

The GPU test program implements OpenCL compute kernels for a simple finite-element simulation. The program source is called test\_gpu\_opencl.c, the compiled binary executable is called test\_gpu\_opencl.exe, and the test can be run as follows:

**$ /piminstall/bin/run\_pim\_emulation.sh –c 2.xml \**

**–e /piminstall \**

**–t /pimemu/lib/test/tmp \**

**–o /pimemu/lib/test/result \**

**--run ./test\_gpu\_opencl.exe ./ \**

**-ccounters –loop 1500**

The following temporary files are produced:

**$ ls -1 /pimemu/lib/test/tmp**

**model\_cpu.csv**

**model\_gpu.csv**

**pimemu\_cpucounterdata.csv**

**pimemu\_gpucounterdata.csv**

**pimemu\_timelinedata.csv**

Like the CPU performance modeling example shown above, this program produces the files model\_cpu.csv, pimemu\_cpucounterdata.csv, and pimemu\_timelinedata.csv. Because this example program also uses the GPU, two extra files are generated. The file pimemu\_gpucounterdata.csv is a collection of hardware performance counters gathered on the GPU that describes how the program executed on the underlying real hardware. These numbers are then put through an analytical scaling model and used to generate model\_gpu.csv, which shows the estimated runtime of each of these GPU-based segments on the modeled PIM-enabled system.

The following result files are produced:

**$ ls -1 /pimemu/lib/test/result**

**emulation\_result.csv**

**emulation\_result.pdf**

The results files contain time ordered output represented in ASCII and PDF formats respectively. A sample of the graphical output contained in the file emulation\_result.pdf is illustrated in Figure 4.



Figure 4: Graphical output from running the simple GPU test.

This graph shows four intervals, which correspond to the estimated program execution times on the modeled system described in the configuration file 2.xml. This program has three major areas of computation:

1. The EHP CPU performs serial computation on the CPU, preparing data to be sent to the GPUs of PIMs 0-1 (green, lower left).
2. The EHP CPU spawns parallel work on the GPUs of PIMs 0-1 (orange).
3. The EHP CPU waits on completion events from the PIMs GPU, then verifies the results, and exits. (green, lower right).

### Combined CPU+GPU

This test program is a combined CPU+GPU test that exercises both devices and is essentially the combination of the Simple CPU and Simple GPU tests described in previous sections. The test program source is called test\_cpu\_gpu.cc, the compiled binary executable is called test\_cpu\_gpu.exe, and the test can be run as follows:

**$ /piminstall/bin/run\_pim\_emulation.sh –c 2.xml \**

**–e /piminstall \**

**–t /pimemu/lib/test/tmp \**

**–o /pimemu/lib/test/result \**

**--run ./test\_cpu\_gpu.exe ./ \**

**-ccounters –loop 500**

The following temporary files are produced:

**$ ls -1 /pimemu/lib/test/tmp**

**model\_cpu.csv**

**model\_gpu.csv**

**pimemu\_cpucounterdata.csv**

**pimemu\_gpucounterdata.csv**

**pimemu\_timelinedata.csv**

The following result files are produced:

**$ ls -1 /pimemu/lib/test/result**

**emulation\_result.csv**

**emulation\_result.pdf**

The results files contain time-ordered output represented in ASCII and PDF formats respectively. A sample of the graphical output contained in the file emulation\_result.pdf is illustrated in Figure 5.



Figure 5: Graphical output from running the combined CPU+GPU test.

This graph shows the program intervals, which correspond to the estimated program execution on the modeled system described in the configuration file 2.xml. This program has five major areas of computation.

1. The EHP CPU spawns parallel work on the CPUs of PIMs 0-2 (blue).
2. The EHP CPU joins with the CPUs of PIMs 0-2.
3. The EHP CPU spawns parallel work on the CPUs of PIMs 1-2 (blue).
4. The EHP CPU spawns parallel work on the GPUs of PIMs 0-1 (orange).
5. The EHP CPU joins with the CPUs of PIMs 1-2, and the GPUs of PIMs 0-2, verifies results, and exits.

# Internal Function

This section describes internal function of the PIM emulator software.

## API Emulator and Runtime

The PIM emulator implements the user-level APIs, in order to allow a PIM application to execute on a real machine, as illustrated by Figure 6.

**Phase 1**

**Commodity Hardware**

**(CPU/GPU/APU)**

**Linux**

**Event Trace & Stats**

**User Application**

**Model Description**

**Host Tasks**

**PIM Tasks**

PIM API

Wrappers

Emulator Runtime

Figure 6: PIM API and Emulator Runtime

The PIM emulator instruments calls to the PIM user-level APIs. This is done in order to collect performance monitor and event trace information during the application’s execution. The emulator additionally provides wrappers for system threading APIs.[[1]](#footnote-1) Together, the APIs and wrappers allow the emulator to collect performance monitor and event information, as illustrated by Figure 7.

**Phase 1**

**Event Trace & Stats**

**User Application**

**PIM API Emulator**

**Host Tasks**

**PIM Tasks**

**CPU Performance Monitor Data**

**GPU Performance Monitor Data**

**Trace Ordering Information**

Figure 7: Data gathered by the PIM Emulator.

The PIM emulator also implements a set of *emulator-specific APIs* that allow the application programmer to enable and optionally disable statistics gathering around code sections not of interest, e.g. file IO, etc. The emulator-specific APIs are described in Section 5 of this report.

### Performance Monitor Data

CPU performance monitor data is gathered by instrumenting API wrappers with calls to Linux Performance Application Programming Interface (or, PAPI). AMD CodeXL is used to gather GPU Performance monitor data. The resulting “.CSV” files are parsed during phase 2 of the emulator.

### Trace Ordering Information

The execution of a parallel application can be represented as a series of events on a timeline. Some events may occur in parallel, while others must happen sequentially (due, for instance, to logical constraints of the program). At runtime, the emulator gathers performance monitor data for all of the sequential and parallel portions of the application, and additionally records the event-ordering information. To illustrate this point, we consider an execution timeline for a simple parallel program in Figure 8.

CPU 0

CPU 1

GPU

pim\_spawn(pthread)

pthread\_join()

pim\_spawn(opencl)

ocl\_wait()

pthread\_exit()

kernel end

Figure 8: Execution timeline for a simple parallel application.

This timeline illustrates:

* A main CPU program (CPU 0), which launches work on a modeled PIM CPU (CPU 1) by calling pim\_spawn().
* The PIM CPU does some work, then launches work on a modeled PIM GPU by calling pim\_spawn().
* The PIM CPU waits for the PIM GPU’s work to complete, does a small amount of work, and then exits.
* CPU 0 waits for PIM CPU to finish, then continues its work until completion.

The **critical path** through the execution will define the total runtime for a program. In this example, the main thread stalls waiting for the second thread to complete. So even if the first thread ran much faster after creating the second thread, the total runtime of the program would not change. However if the GPU executed much faster, the runtime of the program would decrease by some amount since the GPU is on the critical path.

For the purpose of estimating power and performance on a modeled system with PIMs, it is important to note that there are multiple, semi-independent segments in the timeline. Figure 9 shows the same timeline, except with each semi-independent segment labeled with a number. There are seven program segments, each of which requiring some time to run, and which must occur in a specific order. The solid lines represent work that must be performed to complete the program, while the dashed lines represent time that a thread spends waiting. The arrows indicate communication between parallel tasks.

CPU 0

CPU 1

GPU

Figure 9: Execution timeline with labels for the semi-independent parallel sections.

The PIM emulator outputs performance monitor information for each semi-independent segment as it runs on the “real system”. The PIM emulator also outputs the series of relationships between segments that must hold, regardless of by how much the runtime for each segment may be scaled.

Figure 10 illustrates the program segments requiring performance trace information for our example parallel program. By collecting data for each segment, we enable the PIM analytic model to scale performance of each program segment according to the parameters of the modeled system.

GPU Performance

Traces

CPU Performance

Traces

Figure 10: Performance trace data for segments of the example parallel program.

Figure 11 illustrates trace-ordering information for the example parallel program. The ordering information will enable the PIM analytic model to rebuild the trace from disparate segments of performance data, while maintaining correct relationships between the segments.

Trace Ordering Information

after

after

after

after

after

after

after

after

Run on Host CPU

Run on PIM CPU

Run on PIM GPU

Execution Location

Figure 11: Trace ordering and execution location information for the example parallel program.

## PIM Analytic Model

The PIM analytic model implements the second phase of the PIM emulator.

**Event Trace & Stats**

**CPU Performance Monitor Data**

**GPU Performance Monitor Data**

**Trace Ordering Information**

**Phase 2**

**Model Description**

**PIM Analytic Model**

**Performance Trace Finalizer**

**Performance Scaling Model**

Performance

Estimate

Figure 12: PIM Analytic model, implementing Phase 2 of emulation.

The analytic model combines the event trace and statistics with the PIM model description in order to estimate the performance of the application running on the modeled system, as illustrated in Figure 12.

### Performance Scaling

The first step in the PIM analytic model scales performance data for each of the program task segments. Returning to our example from the previous section, Figure 13 illustrates how the semi-independent program segments might scale:

Figure 13: Per-task performance scaling.

The blue lines in Figure 13 indicate original performance monitor data that was collected during Phase 1. Below these are scaled estimates for the EHP CPU (red lines), the PIM CPU (orange lines) and the PIM GPU (green line).

### Trace Finalizer

The next step in the PIM analytic model is to reassemble the scaled performance data, subject to trace ordering constraints discovered during the first phase.

EHP CPU

PIM CPU

PIM GPU

Original

Figure 14: Scaled performance data, reassembled by trace finalizer.

The blue line in Figure 14 indicates overall performance for the application as measured on the real system. The red line indicates estimated time on the EHP CPU, the orange line indicates estimated time on the PIM CPU, and the green line indicates estimated time on the PIM GPU.

## Model Description Files

The PIM emulator uses XML formatted files to represent information about the modeled system.

**XML Model Description File**

**Phase 2**

**PIM Analytic Model**

**Phase 1**

**User Application**

**PIM API Emulator**

**Host Tasks**

**PIM Tasks**

Figure 15: XML model configuration file is input to both Phase 1 and Phase 2 of the emulator.

This information can be queried at runtime by calling the PIM user-level APIs. The PIM analytic model also uses this information to estimate performance on the modeled system.

### Format

The format described below is an initial format for PIM experimentation, the details of which may be modified as the project progresses. At the highest level, every XML model description file must contain a node with a single EHP (host processor) and at least one PIM:

<node>

<ehp>

<!-- … -->

</ehp>

<pim>

<!-- … -->

</pim>

</node>

Each EHP must contain at least one CPU core and can also contain zero or more GPU cores. Each PIM can contain zero or more CPU cores and GPU cores, but must contain at least one DRAM.

<node>

<ehp>

<cpu\_core>

<!-- … -->

</cpu\_core>

<gpu\_core>

<!-- … -->

</gpu\_core>

</ehp>

<pim>

<cpu\_core>

<!-- … -->

</cpu\_core>

<gpu\_core>

<!-- … -->

</gpu\_core>

<dram>

<!-- … -->

</dram>

</pim>

</node>

The values currently associated with <cpu\_core> are:

* <isa>: The ISA of the CPU core. Current supported value is “x86-64”.
* <ordering>: Whether the CPU core is in-order or out-of-order. Options are “inorder” and “ooo”.
* <freq>: Frequency of the CPU core, measured in MHz.
* <width>: The maximum number of instructions committed per cycle.
* <num\_hwthreads>: The number of simultaneous hardware threads on the CPU core.

The values currently associated with <gpu\_core> are:

* <isa>: The ISA of the GPU core. Current legal value is “si”.
* <freq>: Frequency of the GPU core, measured in MHz.
* <cus>: The number of compute units (CUs) on the GPU core.
* <width>: The number of execution units in a CU.
* <num\_instr\_perclck>: The maximum number of instructions that can be completed by one lane of a SIMD unit per clock.

The values currently associated with <dram> are:

* <size>: Size of the DRAM stack, in MB.
* <freq>: Frequency of the memory interface clock, in MHz.
* <num\_channels>: Number of memory channels that are available on this stack.
* <channel\_width>: Number of wires used for data transfer in each memory channel.
* <bits\_percycle>: Number of bits transferred across a wire on each clock.
* <latency>: Average latency in nanoseconds required for a device on the PIM stack to access memory within the stack.
* <off\_stack\_latency>: Percentage increase in time required for external (host) access to this DRAM stack.
* <off\_stack\_bw>: Percentage of total peak bandwidth available for the host when it accesses this memory stack.
* <bw\_between\_pims>:Percentage of total peak bandwidth available to PIMs that are not part of this memory stack.
* <utilization>: Percentage of total peak bandwidth that’s sustainable.

The PIM emulator software package contains config/README.txt, which describes the format of a model description file in further detail.

### Examples

A set of model configuration files is provided with the PIM emulator software package under the config/ directory.

The XML code fragment listed in Figure 16 provides an example model configuration file, describing a host EHP with 4 CPU cores (lines 10-25), 192 GPU cores (lines 27-30) and 2 PIM stacks (lines 33-58, 59-84).

1 <?xml version="1.0" encoding="ISO-8859-1"?>

2 <!-- Example PIM Emulation setup file -->

3 <!-- The devices described in this -->

4 <!-- XML file are the devices that -->

5 <!-- software sees when using the -->

6 <!-- PIM user-level API -->

7 <node>

8 <ehp>

9 <!— EHP: 4 core CPU, 192 core GPU -->

10 <cpu\_core>

11 <freq>4000</freq>

12 <num\_hwthreads>1</num\_hwthreads>

13 <!cpu\_core>

14 <cpu\_core>

15 <freq>4000</freq>

16 <num\_hwthreads>1</num\_hwthreads>

17 <!cpu\_core>

18 <cpu\_core>

19 <freq>4000</freq>

20 <num\_hwthreads>1</num\_hwthreads>

21 <!cpu\_core>

22 <cpu\_core>

23 <freq>4000</freq>

24 <num\_hwthreads>1</num\_hwthreads>

25 <!cpu\_core>

26 <!-- EHP GPU model -->

27 <gpu\_core>

28 <cores>192</cores>

29 <freq>1000</freq>

30 <!gpu\_core>

31 <!ehp>

32 <!-- 2 PIMs per node -->

33 <pim>

34 <!-- PIM CPU model - 2 cores -->

35 <cpu\_core>

36 <freq>2000</freq>

37 <num\_hwthreads>1</num\_hwthreads>

38 <!cpu\_core>

39 <cpu\_core>

40 <freq>2000</freq>

41 <num\_hwthreads>1</num\_hwthreads>

42 <!cpu\_core>

43 <!-- PIM GPU model -->

44 <gpu\_core>

45 <cores>16</cores>

46 <freq>800</freq>

47 <!gpu\_core>

48 <!-- PIM DRAM stack model -->

49 <dram>

50 <freq>2000</freq>

51 <num\_channels>8</num\_channels>

52 <channel\_width>256</channel\_width>

53 <bits\_percycle>1</bits\_percycle>

54 <latency>100</latency>

55 <utilization>80</utilization>

56 <off\_stack\_bw>25</off\_stack\_bw>

57 <!dram>

58 <!pim>

59 <pim>

60 <!-- PIM CPU model - 2 cores -->

61 <cpu\_core>

62 <freq>2000</freq>

63 <num\_hwthreads>1</num\_hwthreads>

64 <!cpu\_core>

65 <cpu\_core>

66 <freq>2000</freq>

67 <num\_hwthreads>1</num\_hwthreads>

68 <!cpu\_core>

69 <!-- PIM GPU model -->

70 <gpu\_core>

71 <cores>16</cores>

72 <freq>800</freq>

73 <!gpu\_core>

74 <!-- PIM DRAM stack model -->

75 <dram>

76 <freq>2000</freq>

77 <num\_channels>8</num\_channels>

78 <channel\_width>256</channel\_width>

79 <bits\_percycle>1</bits\_percycle>

80 <latency>100</latency>

81 <utilization>80</utilization>

82 <off\_stack\_bw>25</off\_stack\_bw>

83 <!dram>

84 <!pim>

85 <!node>

Figure 16: Sample XML model description file for EHP + 2 PIM stacks.

# PIM Emulator-Specific User API

This section describes the user-level programming interfaces that are specific to the PIM emulator.

## Motivation

The PIM emulator captures performance and event information for an application run on a real system. Some sections of an application relating to initialization or shutdown may not be relevant to the performance model, e.g. sections requiring file IO, etc. For these reasons, the emulator-specific APIs provide a mechanism to turn on/off performance information gathering at any point inside an application.

## API Specification

On a properly installed system, function prototypes for the PIM emulator-specific APIs can be found in:

**include/pim.h**

### pim\_emu\_begin

Signal that the PIM emulator should begin gathering performance and event information.

int pim\_emu\_begin (void);

Parameters

This routine takes no parameters.

Return Value

PIM\_SUCCESS on success.

PIM\_GENERAL\_ERROR on all other errors.

Notes

To start capturing emulation events, pim\_emu\_begin() ***must*** be called by the main thread of the application at least once to take effect.

### pim\_emu\_end

Signal that the PIM emulator should stop gathering performance and event information.

int pim\_emu\_end (void);

Parameters

This routine takes no parameters.

Return Value

PIM\_SUCCESS on success.

PIM\_GENERAL\_ERROR on all other errors.

Notes

The use of this interface by the programmer is optional. If pim\_emu\_begin() has been called but pim\_emu\_end() has not, the PIM emulator will stop gathering performance and event data when the program exits. Both interfaces must be called from the application’s main thread to take effect. Any threads spawned between calls by the main thread to pim\_emu\_begin() and pim\_emu\_end() will continue to gather information until the end of their lifespan.

A user of the PIM emulation environment may chose not to include in the performance estimate for some parts of the application code that she might consider non-essential or just pure instrumental and not related to the estimated performance load. Alternatively, it might be interesting to concentrate on performance/energy characteristics of a small part of the entire application.

A pair of pim\_emu\_begin()/pim\_emu\_end() calls provides such flexibility to a user. Each pair defines a program interval during which the PIM emulator captures timing events. We call it an active interval. Only active intervals are going to be included into the combined EHP-PIM performance model.

Figure 17 illustrates sample usage within a program. The sample program performs some basic initialization (init\_1(), init\_2()) and cleanup (fini\_1(), fini\_2()) that are non-essential to the program’s performance on the modeled system, for example network or file IO. The only segments of interest for the performance model are the computational kernels (compute\_kernel\_1(), compute\_kernel\_2()).

1 init\_1 ();

2

3 **pim\_emu\_begin ();**

4 compute\_kernel\_1 ();

5 **pim\_emu\_end ();**

6

7 fini\_1 ();

8 init\_2 ();

9

10 **pim\_emu\_begin ();**

11 compute\_kernel\_2 ();

12 **pim\_emu\_end ();**

13

14 fini\_2 ();

15

16 /\* ... \*/

Figure 17: Paired calls to enable/disable emulator trace gathering.

In this example, the application may perform steps, that are not relevant to performance of the application on the modeled system (lines 1,7,8,14).

With regard to threading, all children of the master thread spawned inside an active interval take part in the performance emulation as do their own children do and so on. They continue to capture timing events until the end of their time span even if master thread calls pim\_emu\_end() while they are still alive.

If a programmer wishes to turn off the capturing entirely the next pim\_emu\_end() call has to be preceded by an application-wide barrier that might be implemented as set of join and/or WaitForEvent calls. It is recommended to make master thread synchronization calls – join, WaitForEvents – inside an active interval to guarantee a correct performance evaluation.

### PIMEMUENV

PIMEMUENV is the emulator’s environment variable to enable/control performance gathering by the PIM emulator. It is used to pass settings to the PIM emulator without requiring specific arguments to be passed through the user program itself.

PIMEMUENV=<PIM model .xml configuration file’s location>**:**<1/0>**:**<PIM model temporary files' location>

Parameters

The PIMEMUENV environment variable may contain up to 3 colon-separated arguments.

The first argument is the location of the XML configuration file that is used to describe the system that the PIM emulator should make visible to the program running under emulation.

The second argument, when set to 1, allows turning on the performance information gathering. At least one call to pim\_emu\_begin() has to be made to begin performance gathering. When set to 0, performance information gathering will not be enabled even when pim\_emu\_begin() is called.

The third argument is the location of temporary files generated by the performance gathering. If the argument is omitted then the directory of the PIM model configuration file is used.

Return Value

None.

Notes

The PIMEMUENV environment variable ***must*** be set before gathering any information with the PIM performance model occurs.

The PIM emulator run script described earlier in this report will set PIMEMUENV, as a convenience to the user.

# Limitations

The initial release of the PIM emulator software has limitations that should be taken into consideration by the user. Several of these represent potential areas for improving the function, usability or accuracy of the PIM emulator software. **[\*]**

1. The PIM emulator is limited to modeling performance for C/C++ language programs.
   1. Calls to system threading APIs (POSIX threads, OpenCL 1.2) are supported.
   2. OpenMP and OpenACC programs are not supported in the initial release.
2. The PIM emulator does not attempt to discover parallelism where it does not already exist in the application. For example, if the program only launches four threads, the emulator will not split the work across more than four “emulated” CPUs.
3. The PIM emulator does not consider application parallelism limits (e.g. # of work-groups) when scaling to future architectures in the GPU model. We envision early evaluations to focus on highly parallel applications where this is not a significant limiting factor.
4. The PIM emulator does not track memory capacity. We expect this to be a lower priority in our early-stage studies for studying candidates for, and the viability of, PIM.
   1. Reported errors from PIM user-level APIs may not be accurate.
   2. Dynamic queries of capacity may return incorrect information.
5. The PIM emulator does not model differences in the cache hierarchy between the test system and the modeled system.
6. The PIM emulator does not track non-local memory access by PIM. For early studies, we rely on the programmer to ensure PIM computation operates predominantly out of the local memory stack.
   1. For early studies, the PIM emulator assumes all PIM memory accesses are local, and does not estimate the performance impact for non-local accesses (i.e. applications that cache well are often a good fit for today’s CPUs and GPUs; PIM is primarily of interest for applications that do not cache well).
   2. The programmer must guarantee this by writing the program to use PIM APIs for memory allocation and task invocation.
7. The PIM emulator does not track fine-grained synchronization such as locks from POSIX threads.
8. The PIM emulator does not currently account for bandwidth contention between overlapping parallel computations.
9. The PIM emulator is currently only tested for modeling systems with a single level memory of PIM.
10. The PIM emulator does not implement an energy model.
11. The XML configuration file format allows more flexibility in defining core attributes than is exposed through the user-level API or implemented in the PIM emulator
    1. When defining attributes for cores within a CPU, GPU, or PIM, the user should specify the same attribute value for all of these cores.

Further, the current (initial) release of the PIM emulator implements the user-level API features necessary for fundamental application studies and evaluations related to PIM. However, the following forward-looking features of the user-level API documented in *FastForward Milestone M1: Processing-in-Memory Baseline Architecture and API Report* are currently unsupported:

1. Only the PIM\_CLASS\_0 device type is supported for pim\_get\_device\_ids().
2. The following device query parameters are unsupported for pim\_get\_device\_info(): PIM\_CPU\_ISA, PIM\_GPU\_ISA, PIM\_MEM\_LEVEL, PIM\_MEM\_TECHNOLOGY, PIM\_HOST\_COHERENT, PIM\_PEER\_COHERENT, HOST\_CPU\_ISA, HOST\_GPU\_ISA.
3. As noted above, memory capacity is not dynamically tracked and, as a result, reporting of errors for pim\_malloc() may be inaccurate.
4. The pim\_reloc() call is unimplemented.
5. Returning error codes (via errno) may not be implemented for errors encountered during pim\_map() (error conditions will still be detectable by a NULL return value from pim\_map()).
6. The pim\_unmap() call only returns PIM\_GENERAL\_ERROR on all errors (i.e. other error returns are unimplemented).
7. Calling get\_pim\_id() from GPU kernels is unsupported.

# References

1. Wim Heirman, Trevor E. Carlson, Shuai Che, Kevin Skadron, Lieven Eeckhout, “Using Cycle Stacks to Understand Scaling Bottlenecks in Multi-Threaded Workloads,” IEEE Int’l Symposium on Workload Characterization (IISWC), 2011.
2. Barry Rountree, David K. Lowenthal, Martin Schulz, Bronis R. de Supinski, “Practical Performance Prediction Under Dynamic Voltage Frequency Scaling,” 4th International Green Computing Conference (IGCC), 2011.
3. Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, Richard Vuduc, “A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications,” 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2012.

1. The PIM emulator provides wrappers for POSIX threads and OpenCL 1.x APIs. The application source code does not require modification in order for the PIM emulator to collect performance information about calls to these APIs, as they are redirected to the wrappers when the application links with the PIM emulator library. [↑](#footnote-ref-1)