NORTHWESTERN UNIVERSITY

PIM Profiling and Offlading Project Report

MASTER PROJCT REPORT

SUBMITTED TO THE MS PROJECT COMMITTEE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

MASTER OF SCIENCE

Field of Computer Engineering

By

Haoxuan Zhang

EVANSTON, ILLINOIS

March 2021

ABSTRACT

PIM Profiling and Offlading Project Report

Haoxuan Zhang

With increasingly larger data set for software workloads, data movement has become major performance and energy bottleneck. Processing in Memory has been a hot topic in recent years to solve this problem. The basic idea of processing in memory is to push some computation units into the memory. Therefore, instead moving large amount of data into the processors, PIM system can send instructions or commands to the memory and ask the smaller processing units in the memory to do the computation near the data. This modification can bring significant performance and energy advantage.

In this report, a design which can offload loops in programs according to their load/-store ratio and data locality is proposed. A profiler to detect loops and their load/store ratio is built. Based on profiling result as well as data size and cache sizes, the proposed architecture can switch between host mode and PIM mode with non-significant overhead. The design is simulated using Gem5 simulator. The evaluation shows that the proposed

architecture can have 41% speedup for low locality workload, 35% speedup for high locality workload but with small cache size and 32% speedup for mixed locality workloads against the baseline pure host processing system.

Acknowledgements

I would like to thank my supervisor and MS project committee chair Prof. Gokhan Memik for his continuous support and advice. My journey during Master's program and the project is always enjoyable under instruction of Prof. Gokhan Memik. While doing the project, Prof. Gokhan Memik can always come up with clear and feasible directions, which are greatly helpful for my project progress. I have learned a lot about the tools and strengthened knowledge learnt in classes during the project thanks to Prof. Gokhan Memik's great help.

I would like to thank Prof. Stephen Tarzia who is also in the MS project committee for his help and support during my master program. I would also like to express my gratefulness to the Electrical and Computer Engineering department and Northwestern University which provide great platform for me to make improvement.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

This project is conducted by Haoxuan Zhang under supervision of Professor Gokhan Memik. The goal of this project is to evaluate effect of Processing in Memory(PIM) for data intensive workloads.

The design proposed in this project offloads loops with low time locality of data in the program into the in-memory processor which has access to the DRAM with higher bandwidth. The in-memory processor will have smaller functional unit pool, reorder buffer size and caches. The goal is to offload the memory-bound, low time locality loops into smaller CPU which has higher memory access speed while those compute-bound, high time locality loops remains in the larger host CPU.

Gem5 simulator is used in the project. The effort in this project could be illustrated as followings:

A profiler is firstly developed based on timing simple CPU model to identify loops in programs and record the load/store to total operation ratio. This could be used along with the data size and cache size to determine whether the loop has low or high time locality and therefore should be offloaded or not. After running the profiler, a list of loops to be offloaded is generated.

Secondly, a dynamic PIM architecture is created to offload loop workloads into memory according to the time locality indicated by the profiling result as well as data size and cache size.

Thirdly, a modified O3 CPU model and a modified DRAM memory controller model are created within Gem5 which can switch between host and PIM mode to simulate the system running in these two different modes. The simulation system is built by connecting these modified simulation objects with memory hierarchy system to imitate the proposed dynamic PIM architecture behavior.

Fourthly, test programs with various locality workload are created and tested on the simulation system. The simulation result is used in analyzing the performance of the proposed design.

The Github repository of this project could be found at `https://github.com/haoxuan1080/gem5.git` and the branch containing the works of this project is `PIM_Draining`.

The report is organized in the following manner: In section 2, Background of PIM and motivation of the design will be introduced. Section 3 describes the profiling process and the profiling result generation. Section 4 present the proposed dynamic PIM offloading architecture. Section 5 explain the simulation setup and process. Section 6 present the evaluation results and provide analysis of the design. Section 7 summarize the project and discuss potential future works.

CHAPTER 2

Background

As workloads of today's computer system becoming more and more data-intensive with the emergence of 3D stacking integrated circuits, PIM architecture shows its great potential to achieve better performance and power efficiency because it can significantly reduce unnecessary data movements. The basic idea of processing in memory is to push some computational unites into the memory. Therefore instead of moving large amount of data into the processors, PIM system can send instructions or commands to the memory and ask the smaller processing units in the memory to do the computation near the data.

PIM was firstly proposed in 1990s. one of the representitve design to explore pushing computational units into memory is Intelligent RAM(IRAM)[**1**]. However, due to high cost of integrating computing units into memory, PIM didn't gain popularity until the the last decade. In fields demanding high throughput such as parallel computing and artificial intelligence, there has been lots of effort dedicated to transporting data to computation units more efficiently. However, the problem of inefficiency to transport data to computing units still remains not resolved. The gap between memory and cpu speed has been growing every year in conventional architectures, which lead to the prevalent usage of cache. However, in today's big data workloads, data locality could be very hard to utilize because of large amount of random data access as well as the larger data set size but relatively smaller amount of computation for each data element. Thus, people are adopting PIM again to remedy this problem.

A number of novel PIM designs are proposed. RowClone[**2**] proposes to offload bulk data copy and initialization into the memory with relatively trivial hardware modification. In the papers selected in this survey, TOP-PIM[**3**] firstly proposed to build processing in memory system based on Hybrid Memory Cube(HMC) using the same proessing units as the host processor but with smaller size. NDA[**4**] places data flow processors into the logical die of the HMC. Tesseract[**5**] push general-purposed core into HMC to conduct graph workload locally in each bank and the cores inside the memory use message passing to cooperate with each other. PIM-Enabled Instructions[**6**] proposed a locality-aware architecture which selectively offload computation into the memory based on their locatlity. Practical Near-Data Processing for In-Memory Analytics Frameworks[**7**] embedded general purpose processors into logic die of HMC, also utilizing flexibility of virtual memory. PRIME[**8**] leverages the crossbar structure of Resistive random-access memory(ReRANM) to compute multiplication and accumulation which is the dominated computation in today's neuronetwork. Neurocube[**9**] places data-flow processor into logic die of HMC and uses data-centric computation to simplify scheduling. GraphPIM[**10**] spare a non-cacheable area in the virtual memory space as PIM only area and offload any operation to the memory if the data is in the separated partition. PipeLayer[**11**] also uses ReRAM for PIM Neural Network processing but added features to support training process of Neural Networks. TETRIS[**12**] places previous field specific accelerator called Eyeriss[**13**] into logic die in HMC to process Neural Network. Neural Cache[**14**] which uses bit-serial computation inside the cache to increase the parallelism of multiply accumulation computation.

Among the works mentioned above, the PIM-Enabled Instructions[**6**] can be applied to general-purpose computing and assign PIM-enabled instructions to host CPU or PIM processing units based on the operands' locality. However, by doing so, extra logic for monitoring data locality needs to be added including a table with the same size as last-level cache to record the data locality. In addition, the tracking of locality also increase the latency of determining whether to offload instructions to memory. Moreover, offloading instructions on the granularity of instructions could add huge overhead because there could be large number of consecutive PIM-enabled instructions within a chunk of program and each of the instruction needs to check the monitor which is as slow as last level cache before assigned to host CPU or the PIM processing units.

In this project, a PIM architecture is proposed so that it can dynamically assign general purpose workload to PIM units without adding too many logics. The design proposed will not determine PIM offloading on the granularity of individual instructions. Instead, the design in this project will offload loops which has low time locality into PIM units and utilize profiling to determine whether a loop in the program possess low time locality with high load/store to total operation ratio and hence should be offloaded to PIM execution.

CHAPTER 3

Profiling

3.1. PIM Profiling Process

In Gem5 TimingSimpleCPU model, the struct of BranchNode is created to store loop information and a loop list is stored in each TimingSimpleCPU. Whenever the cpu sees a branch instruction and the target is before the current PC and the result is taken, then the list will be checked. With the current PC to be the end point and the target PC to be the start point of the loop, if the loop already exists in the list, all the statistics related to that loop will be incremented accordingly. Otherwise, if the loop is a newly recognized loop, a new BranchNode will be created to track that loop and be pushed to the list.

After the simulation, all the information about the recognized loops will be shown in m5out/stats.txt. The name of each stats of the loop will be `thread_name` + `.loop_#` + `specific stats`. The "`specific stats`" information collected for each loop are as follows:

1. TargePC: The starting point of the loop.
2. BranchPC: The ending point of the loop.
3. iterNum: The iteration number of the loop.
4. TotalInstNum: The total operation number in the loop.
5. LoadNum: The total load operation
6. ldstRatio: The total operation to memory access ratio of the loop.

3.2. PIM Profiling Result Generation

After profiling, the PIM loops that should be offloaded will be extracted using the script: `get_pim_loop`. The usage of `get_pim_loop` is as follows:

```
get_pim_loop -f input_file
    -t iterNum -r ratio
```

Where the `input_file` is the stats.txt that is containing the loop information, `iterNum` is the iteration number threshold above which a loop must have to be considered to be offloaded to PIM and `ratio` is the total operation to memory operation ratio threshold below which loop must have to be considered to be offloaded to PIM.

After running the script, a file called `PIM_loops.txt` will be created containing all the information of the loops that are supposed to be offloaded. Each offloaded loop takes three lines in `PIM_loops.txt`. The format is as follows:

```
Loop number
Start PC
End PC
```

In `PIM_loops.txt`, all the loops selected are having high load/store to all operation ratio, which means the loop is memory intensive. However, in a cached system, this doesn't mean that the memory request will always get to the memory controller. If the data size is small and fits in the caches and the data is reused multiple times, then the requests to memory controller will be a lot fewer than the requests made by the CPU. Such workload still shouldn't be offloaded into PIM because it still possesses high time locality.

Therefore the criteria for PIM offloading should also include low time locality. With high memory access to total operation ratio and low locality, it will be very inefficient to bring the data into the caches with only a few operations on it and then store it back without accessing in the near future. Therefore, such workload should be offloaded into PIM.

The profiling data will be loaded into the CPU when the program starts. The detail of this process will be introduced in the next section.

CHAPTER 4

PIM Offloading Architecture

Figure 4.1 shows the architecture of the diagram of the proposed architecture. The system has a host CPU which is a larger out-of-order CPU with first level instruction and data cache connected to its instruction fetch port and data port. The first level caches are connected to a bus called "L2 Bus" and then connected the unified L2 Cache(Last level cache). The L2 Cache will be connected to the System Bus which communicates with the memory system through the Memory Data Port. The memory hierarchy between the host CPU and the memory system is the same as that of a conventional two-level cached architecture.

Inside the memory system, the Memory Data Port is further connected to the Memory bus. The Memory bus master port is connected to the Memory Controller which communicates with the DRAM blocks.

On the right side of the diagram shown in Figuire 4.1, the PIM processing system is having a similar 2-level memory hierarchy as that in the host system. However, the caches and CPU in PIM system will be smaller than those in the host system due to limited logic resources. The PIM system is connected to a innerBus which is further connected to the PIM memory controller. The PIM memory controller communicates with the DRAM blocks. The PIM bus and the PIM memory controller is a lot faster than the system bus and memory controller because the PIM system has the advantage of being nearer to the data and able to access more interface resources to the Memory blocks such as through

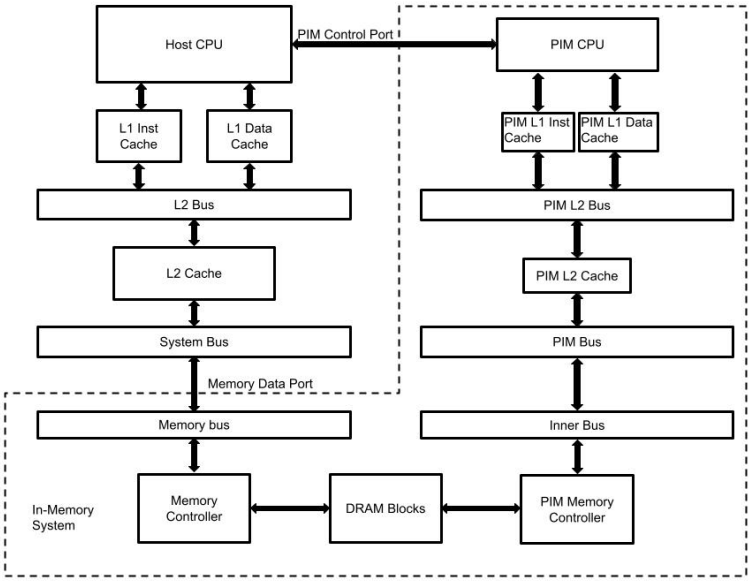


Figure 4.1. Architecture of Dynamic PIM System

silicon via in Hybrid Memory Cube(HMC)[**15**]. Therefore, PIM system has much higher memory bandwidth but less powerful CPU with smaller caches.

The reason why PIM system still needs caches is because the workloads offloaded could still posses some level of spacial locality. Therefore a cache which will bring in a whole block of data from memory can speed up the processing by utilizing the spacial locality.

There is also an additional PIM control Port between the host CPU and the PIM CPU. The host CPU can send the information needed by the PIM CPU to execute the offloaded workloads. The information would contain the start and end program pointer as well as the register file content.

The program run on this system will firstly start in host CPU and its memory hierarchy by default. When the CPU encounters the offloaded loops in the program marked in the profiling process, the host CPU will drain it's piplines and inform all the caches including

first level and last level caches to write back all the dirty data and invalidate all the cache line. All the uncommitted instruction after the switching point will be squashed during the draining. Then it will send offloaded workload information to PIM CPU and the PIM CPU would start to execute from the start PC until it finishes the loop with the next program counter being out of the loop enclosed by start and end PC. The cache write back and invalidation process in this project is simulated functionally which don't affect the timing stats. This could be changed to include the time spent on write back and invalidation into consideration to obtain more accurate simulation result. However, due to the switching event not happening frequently in this project's testbenches, we used the functional model instead of a timing model for cache writing back and invalidation.

Here we have several assumptions for the workload offloading process:

1. The profiling information will be loaded into host CPU so the system will be aware of the offloaded workloads. This could be achieved in the following way: the profiling information is embedded into the binary file of the program by the compiler so that it will be loaded into the memory when the program starts running. Then at the begining of the program, there will be some instruction writing the address and length of the profiling information into a set of special registers in host CPU. Therefore the host CPU will be aware of the profiling result. By sending these information to the PIM CPU, the PIM CPU also know when the work is finished and it will also drain itself, write back dirty data inside caches and invalidate all the cache lines. By doing this the ISA will be extended with more registers but the types of instructions won't be changed.

2. The host CPU and PIM CPU will test the PC during commit stage where instructions are reordered. if the PC reaches the switching point, the subsequent instruction will be squashed after committing the current instruction and the CPU will be drained.
3. As mentioned before, when switch from host mode to PIM and from PIM mode to host mode, the CPUs needs to drain themselves and write back dirty blocks and invalidate all the cache lines before the other system will start executing. Also given that there is only one CPU in host system and in PIM system respectively, there won't be multiple copies of data in the cache system across the host and PIM system or any same level caches. Therefore, cache coherence problem is simplified.
4. We assume that all the loops generated by the compiler will only exit at "End PC", which means the branching instruction that will result in getting out of the loop can only reside at the "End PC". Therefore, no matter what the loop content is, the hardware will only test the loop exiting condition at the "End PC" which could lead to switching between PIM mode and host mode. This simplifies the logic needed to detect switching point in both host CPU and PIM CPU.

The architecture proposed in project will have following physical overheads:

1. New registers added into host CPU and PIM CPU and therefore the ISA is extended.
2. Extra logic in host and PIM CPUs will be added to determine switching point and do the switching from host mode to PIM mode or from PIM mode to host mode.

3. PIM CPU and it's memory hierarchy including buses and caches needs to be added into the memory logical portion.
4. additional port "PIM Control Port" should be added between the host CPU and the memory system for communication while switching.

CHAPTER 5

Simulation Setup

5.1. Assumptions

We have the following assumptions for setting up the simulator to evaluate the proposed architecture.

1. The PIM CPU will have half the fetch size of the host CPU
2. The ROB size of the PIM CPU will be a quarter of the size of the host CPU
3. All types of functional units in PIM CPU will be reduced to only 1.
4. The PIM memory controller has all the memory timing parameter half as large as those of memory controller
5. PIM Bus will be faster than System Bus. The system bus has front delay of 100 system cycles while the PIM bus only has delay of 10 system cycles.
6. the delay of memory bus and inner bus inside the memory system are the same and set to 10 system cycles.

5.2. Testbench Program

We built four testbench programs which will be used to evaluate the proposed architecture.

The first one is a program with low locality loops. This testbench will be referred as LLOC in the rest of the report. The code is shown below:

```c
#include <stdio.h>
#include <stdlib.h>

#define M (65536 * 1000)

int main() {
    printf("Low Locality loop!\n");
    //low locality
    int* m = malloc(M*sizeof(int));
    int* n = malloc(M*sizeof(int));
    long int* l = malloc(M*sizeof(long int));

    for (int i = 0; i < M; i++) {
        m[i] = i;
        n[i] = M-i;
    }

    for (int i=0; i < M; i++) {
        l[i] = m[i] * n[i];
    }

    printf("%d * %d = %ld\n", m[2], n[2], l[2]);

    return 0;
}
```

The second one is a program with high locality loops. This testbench will be referred as HLOC in the rest of the report. The code is shown below:

```
#include <stdio.h>
#include <stdlib.h>

#define L 1024
#define N 65536
#define M (65536 * 1000)

int main() {
    printf("High Locality loop!\n");
    //hgigh locality, not get evicted
    int* x = malloc(L*sizeof(int));
    int* y = malloc(L*sizeof(int));
    long int* z = malloc(L*sizeof(long int));
    for (int i = 0; i < L; i++) {
        x[i] = i;
        y[i] = N-i;
    }

    printf("repeat %d * %d times\n", M/L, L);

    // data will be in the cache, look at the CPI,
    // IPC is high then compute bounded.
    for (int j =0; j < (M/L); j++)
```

```
24          for (int i = 0; i < L; i++) {
25          //what if a call here?
26          //in the report discussion
27              z[i] += x[i]+y[i];
28      }
29
30      printf("(%d + %d) * %d = %ld\n", x[2], y[2], M/L,z[2]);
31
32      return 0;
33  }
```

In the HLOC program, the array size is small and can fit in the caches and the data will be reused so that the time locality is high.

The third one is a program with high locality loops but the data array is larger than the L2 cache capacity so the system cannot fully utilize the time locality due to eviction. This testbench will be referred as HLE in the rest of the report. The code is shown below:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N 65536
5 #define M (65536 * 1000)
6
7 int main() {
8     printf("High Locality get evicted!\n");
9     //high locality, get evicted
```

```
10      //increase N to get evicted
11      int* a = malloc(N*sizeof(int));
12      int* b = malloc(N*sizeof(int));
13      long int* c = malloc(N*sizeof(long int*));
14      for (int i = 0; i < N; i++) {
15          a[i] = i;
16          b[i] = N-i;
17      }
18
19      printf("repeat %d * %d times\n", M/N, N);
20
21      // data will be in the cache, look at the CPI,
22      // IPC is high then compute bounded.
23      for (int j =0; j < (M/N); j++)
24          for (int i = 0; i < N; i++) {
25              c[i] += a[i]+b[i];
26          }
27
28      printf("(%d + %d) * %d = %ld\n", a[2], b[2], M/N, c[2]);
29
30      return 0;
31  }
```

The last test bench will be a program which is a combination of the previous 3 test-benches. This testbench will be referred as COMB.

5.3. Simulator setup

Gem5 is used in the simulation of the proposed architecture. Figure 5.1 shows the system built and run on Gem5 simulator. We used the O3CPU model to simulate both the host and PIM CPU. The configuration file used in the repository is the modified `configs/learning_gem5/part1/two_level.py`.

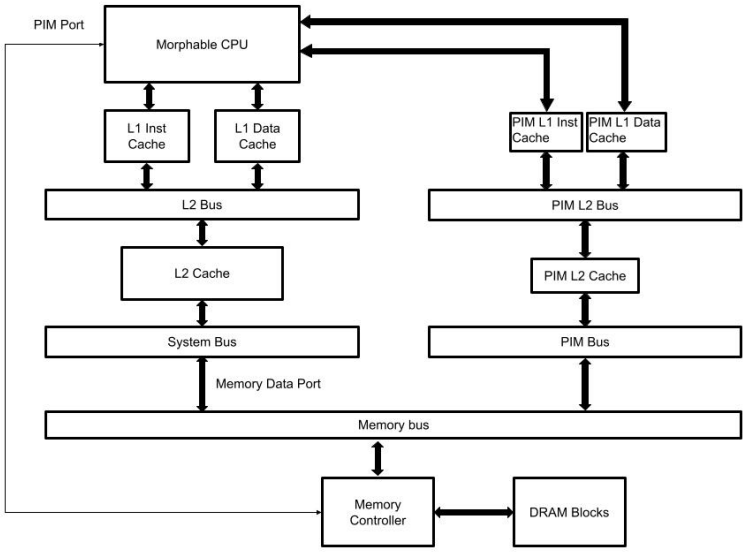


Figure 5.1. Diagram of Simulation System

There is only one CPU and memory controller in the memory system. Ports and morphability are added to these models so that they can switch between host mode and PIM mode. In different modes, the CPU is using different ports and hence different memory hierarchy.

On the left side of the diagram in Figure 5.1, the CPU is connected to the host system memory hierarchy and on the right side, it is the PIM memory hierarchy. When the CPU detects the switching PC either to PIM mode or from PIM mode according to the

profiling information in commit stage, the CPU will squash uncommitted instructions after the switching point in the program order and drain the CPU.

There is a PIM Port from the CPU to the Memory controller. Note that this port isn't corresponding to the "PIM Control Port" in Figure 4.1 which is not needed in simulation because the host and PIM CPU will be all run within the single Morphable CPU model in the modified Gem5 simulator. The "PIM Port" in Figure 5.1 is only used to let the Morphable CPU to tell the memory controller to switch when switch point is detected after CPU itself having been drained and let the memory controller to tell the CPU when the switch is finished. The PIM port is only used in the simulation and don't have physical meaning because in the real architecture, host CPU and PIM CPU are separated and memory controller and PIM memory controller are also separated.

There is another simplification in the simulation system against the real architecture apart from the combined CPU and memory controller model: The switching of memory controller will happen immediately without costing any time. However, the memory controller will delay 1000 ns after its switch before it send the response back to the CPU telling it that the memory controller has finished switching and the CPU will be resumed. This delay can imitate the information sending process delay from the host CPU to the PIM CPU. The two processes are all happening after the CPU draining and they all need some time to process. In addition, after these two processes the following workload will be resumed in the new operating mode. Therefore, the delay of memory controller switching in simulation system can imitate the delay for host and PIM CPU sending information to each other.

The baseline system(BLS) against which the proposed architecture will be compared is a two level cached system with only the host CPU and memory controller. Apart from the baseline system, 3 system are simulated: PIM dynamic offloading system(PDOS), PIM dynamic offloading low time locality only system(PDOLOS) and Pure PIM system(PPS). The configurations of the PIM dynamic offloading system are shown below:

- system clock frequency: 1 GHz
- cpu clock frequency: 1 GHz
- host cpu functional unit pool:

Functional unit	count
IntALU	6
IntMulDiv	4
FP_ALU	4
FP_MultDiv	2
SIMT unit	4
RdWrPort	4

- PIM CPU functional unit pool:

Functional unit	count
IntALU	1
IntMulDiv	1
FP_ALU	1
FP_MultDiv	1
SIMT unit	1
RdWrPort	4

- host CPU ROB size: 192 instructions
- PIM CPU ROB size: 48 instructions
- memory size: 2 GB
- L2 Bus and PIM L2 Bus front end delay: 1 cycle
- L2 Bus and PIM L2 Bus response delay: 1 cycle
- System Bus front end delay: 100 cycles
- PIM Bus front end delay: 10 cycles
- Memory Bus front end delay: 10 cycles
- L1 inst cache size: 16 kB
- PIM L1 inst cache size: 4 kB
- L1 data cache size: 64 kB
- PIM L1 data cache size: 16 kB
- L2 cache size: 256 kB
- PIM L2 cache size: 64 kB

- all L1 caches(PIM and host) association: 2
- all L2 caches(PIM and host) association: 8
- cache line size: 64 bytes

The configurations of all other three systems(BLS, PDOLOS and PPS) are all the same to the corresponding ones in PDOS. The BLS only have the host portion of the PDOS system. The PDOLOS has identical architecture introduced in section 4 with PDOS but only loops with high load/store to total operation ratio and low time locality are offloaded, while PDOS system offload any loop with high load/store to total operation ratio. The PPS system only has the PIM portion of the PDOS system.

To show the necessity to include PIM caches, a system was built which is the same as PDOS but without any PIM caches nor PIM L2 Bus or PIM Bus. This system is simulated against BLS running COMB test bench. The runtime of BLS is: 9.268701s while the runtime of PDOS without PIM caches is: 27.138331s which is much slower than that of BLS. This is because although COMB is dominated with loops having high load/store instruction to total operation ratio with some of them even having low time locality, all the loops are still possessing spacial locality. The cache line size is 64 bytes so BLS can utilize spacial locality while PDOS without PIM caches cannot. Therefore cached memory hierarchy in the PIM system is needed to exploit spacial locality.

5.4. Unexpected Draining Behavior in Simulation

For the draining process of CPUs in the simulation, Gem5 `Drainable` model and it's helper function are used. However, it is worth mentioning that while testing the draining process of the modified CPU model and the Gem5 `Drainable` model, the simulator shows

CPU model	start to 1st	1st	1st to 2nd	2nd	2nd to 3rd	3rd	3rd to 4th	4th	4 to end
drain	9295	43(9338)	8391(17729)	8 (17737)	30691(48428)	285(48713)	2235(50948)	684 (51632)	24029922 (24081554)
no drain	9295	-	8434(17729)	-	30555(48284)	-	2454 (50738)	-	24031964(24082702)

Table 5.1. Draining Test Result Comparison between Draining and Non-Draining Model

the system is running faster with draining than without draining running the same workload. The details are as follows: A program is created in which the first two loops are offloaded and hence should make the modified CPU model to drain and switch between CPU and PIM mode. There should be 4 draining and switching in total. In this small test, actual switching, cache write back and cache invalidation are not implemented except the draining process. The system will resume immediately after completion of draining. Table 5.1 shows the simulation result of this test comparing time line of systems with the modified draining CPU and the unmodified CPU.

In table 5.1, first column is the CPU model with the second row being the modified CPU model and the third row being the unmodified model. From the second column, each column shows the tick count of each time period being either from start to first draining, draining or the period between two drainings(switchings). The numbers in parenthesis are the tick counts elapsed from the starting point in total.

From table 5.1, we can observe that, firstly, the draining model system is running faster than the non-draining model system from the completion of the first drain to the start of the second drain. Secondly, the draining model is running faster than the non-draining model from the completion of 3rd draining to the start of 4th draining. Thirdly, The draining model is running faster than the non-draining mdel after the completion of the 4th draining until the end of the test program. This is not the same as expected

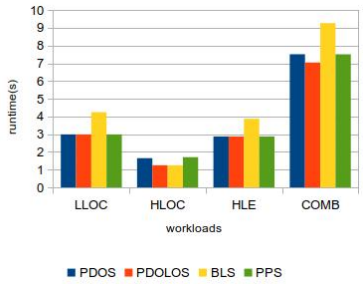
draining timing behavior. Typically, after draining, the CPU need more cycles to load the pipline before the first instruction reaches the commit stage, which will make it run for longer time than a fully loaded CPU. However, in the test result shown in table 5.1, within 3 in total 4 time period after draining point, the modified draining model is running faster than the unmodified model. Due to limit of time, the reason for draining model to be unexpectedly faster is not known. However, the difference between the runtime of the modified draining model and unmodified model are not significant. Besides, the draining is not happening frequently in the testbenches used in this project, therefore such difference won't affect the simulation result of this project significantly.

In the next section, we will demonstrate the simulation result for the four systems: BLS, PDOS, PDOLOS and PPS.

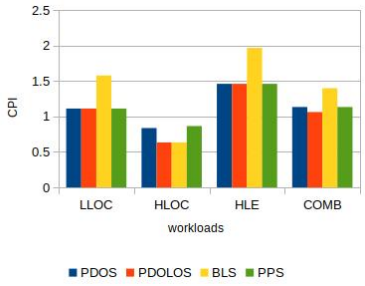
CHAPTER 6

Simulation Result and Analysis

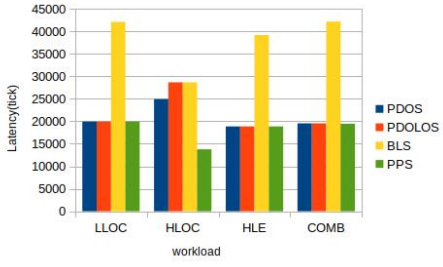
The simulation result comparison is shown in Figure 6.1. Figure 6.1a shows the runtime of workloads with various locality running on different systems. Figure 6.1b shows cycle per instruction of the workloads running on each system. Figure 6.1c shows the time spent from burst creation until serviced by the DRAM in terms of number of ticks(1000 ticks equals to 1 ns).



(a) Runtime Comparison



(b) CPI Comparison



(c) Latency Comparison

Figure 6.1. Comparison of Simulation Result Between Baseline system and Dynamic PIM Offloading Systems

From Figure 6.1a and Figure 6.1b, we can see that the baseline system BLS is only running faster with high locality workload. While in other workloads either low locality, high locality with data size larger than cache size and mixed locality, dynamic PIM system PDOS is always having better performance over the baseline system BLS. For either LLOC or HLE workloads, PDOLOS is having similar performance to PDOS. While for HLOC workload, PDOLOS's performance is similar to that of BLS which is only using host cpu. When running mixed locality workload COMB, PDOLOS shows the highest performance because it can utilize the larger CPU when all the data can fit in the caches and utilize the higher memory bandwidths by switching to PIM mode whenever the data cannot fit in the caches or shows low time locality.

In terms of the memory access latency, as shown in Figure 6.1c, systems with workload offloaded always show better memory access latency. For LLOC, HLE and COMB, PDOS, PDOLOS and PPS are having similar latency. Take COMB as an example, the memory access latency on PDOS, PDOLOS, BLS and PPS are 19556.75, 19554.3, 42189.05 and 19475.47 ticks on average. BLS has the highest latency while PPS has the lowest. PDOS and PDOLOS are having latency in between but very similar to that of PPS. This is because PDOS and PDOLOS are using mix of host system and PIM system while the majority of the memory access is dominated by the low locality workloads or the workloads with high locality but large data size which cannot fit into the caches. Therefore the memory accesses are dominated by PIM system memory accesses. However, for HLOC workload, PDOS is still having latency between BLS and PPS but more similar to BLS. This is because with the high locality workloads, the data could fit in not only host system caches but also PIM system caches. Therefore, the memory accesses are no longer

workload	PDOS	PDOLOS	BLS	PPS
LLOC	876710684	876710684	617674010	876751591
HLOC	56440	55993	55993	39831
HLE	547605901	547605901	406530306	547643437
COMB	558558003	595707272	452663448	558752052

Table 6.1. bandwidth utilization(bytes/s)

dominated by the offloaded workloads. For PDOS, the total number of access to the offloaded workload data is significantly reduced compared to LLOC, HLE and COMB while the host system memory access is not significantly reduced leading to larger portion of the total memory accesses happening in the host system. Therefore the PDOS system is having memory access latency similar to the BLS system.

Table 6.1 shows the bandwidth utilization of the DRAM for different workloads and systems. It can be observed that the workload is having the largest influence on the bandwidth utilization. For the same workload except for HLOC, PDOLOS always reveal the highest bandwidth utilization which is also very similar to that of PPS. This is because the PDOLOS can switch to PIM mode whenever the time locality is low and utilize the higher bandwidth of the PIM system. However, for HLOC workload, PDOS is having the highest bandwidth with PDOLOS very similar to it but PPS's bandwidth is far less than that, even lower than that of BLS. This is because the HLOC workload is no longer a memory bounded workload with caches and the CPU speed would be the bottleneck of the performance. The memory accesses in HLOC are not dominated by the offloaded portion of the program because the data can fit in the caches in both host and PIM system. However, the CPU in PPS is slower than the host CPU and cannot make memory request

as fast as the host CPU in the workload portion which is not offloaded in PDOS. With HLOC workload, PDOS is having the highest memory bandwidth because it also utilizes the PIM memory hierarchy which is fast. However, the bandwidth of PDOS is not much larger than that of BLS or PDOLOS because the memory accesses operations in HLOC has higher time locality and majority of them don't have to go to the memory. Thus, in this case, the benefit of faster memory interface won't have large influence.

Having the simulation result stated above, we can conclude that the PDOLOS design can utilize faster host CPU when locality is high, and take advantage of higher memory bandwidth available in PIM system when time locality is low. The proposed PDOLOS design can have 41% speedup for LLOC workload, 35% speedup for HLE workload and 32% speedup for COMB against BLS system.

CHAPTER 7

Conclusion and Future Work

In this project, we proposed a dynamic PIM offloading architecture which can send the workload with high load/store to total operation ratio and low time data locality into computing unit in memory which has faster access to the memory but slower processing speed. We modified the Gem5 simulator to create the profiler which can identify the loops in the workload having high load/store to total operation ratio and to simulate the proposed architecture.

The simulation result shows that our proposed system can achieve upto 41% speedup against the conventional architecture with two-level cached memory hierarchy.

The limitation and potential future works related to this project are listed below:

First, in the project, the PIM system only has one CPU. However, with the smaller size of the PIM CPU, more than one CPUs could fit into the memory logic layer. Therefore a potential future work could be to extend this design into multicore PIM system.

Second, in this project, we mainly focused on the hardware architecture, while we didn't discussed much about various workload with support of the compiler. For example, the loops used in the simulation of this project are not including function calls inside. In fact, this could be very prevalent in real-world tasks. Therefore, potential future works could be to extend the simulation of the design with more diverse testbenches with compiler support.

Third, the simulation used in this project is not full system simulation. Future works could be to extend the simulator to support full system simulation to acquire more accurate simulation result.

Fourth, the simulation in this project is not using specific novel memory structures such as HMC nor utilizing specific process library to support the simulation object timing parameters. The parameters are set based on assumptions. Future works could be to extend the simulator to have specific memory architecture such as HMC and real memory controller timing parameter to acquire more accurate evaluation result.

References

[1] D. Patterson et al., *"A case for intelligent RAM,"* in IEEE Micro, vol. 17, no. 2, pp. 34-44, March-April 1997.

[2] V. Seshadri et al., *"RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization,"* 2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Davis, CA, 2013, pp. 185-197.

[3] Zhang, D., Jayasena, N., Lyashevsky, A., Greathouse, J. L., Xu, L., & Ignatowski, M. (2014, June). *TOP-PIM: throughput-oriented programmable processing in memory.* In Proceedings of the 23rd international symposium on High-performance parallel and distributed computing (pp. 85-98).

[4] A. Farmahini-Farahani, J. H. Ahn, K. Morrow and N. S. Kim, *"NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules,"* 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), Burlingame, CA, 2015, pp. 283-295.

[5] J. Ahn, S. Hong, S. Yoo, O. Mutlu and K. Choi, *"A scalable processing-in-memory accelerator for parallel graph processing,"* 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), Portland, OR, 2015, pp. 105-117.

[6] J. Ahn, S. Yoo, O. Mutlu and K. Choi, *"PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture,"* 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), Portland, OR, 2015, pp. 336-348

[7] M. Gao, G. Ayers and C. Kozyrakis, *"Practical Near-Data Processing for In-Memory Analytics Frameworks,"* 2015 International Conference on Parallel Architecture and Compilation (PACT), San Francisco, CA, 2015, pp. 113-124.

[8] P. Chi et al., *"PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory,"* 2016 ACM/IEEE 43rd

Annual International Symposium on Computer Architecture (ISCA), Seoul, 2016, pp. 27-39.

[9] D. Kim, J. Kung, S. Chai, S. Yalamanchili and S. Mukhopadhyay, "Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory," 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), Seoul, 2016, pp. 380-392.

[10] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar and H. Kim, *"GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks,"* 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), Austin, TX, 2017, pp. 457-468.

[11] L. Song, X. Qian, H. Li and Y. Chen, *"PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning,"* 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), Austin, TX, 2017, pp. 541-552.

[12] Gao, M., Pu, J., Yang, X., Horowitz, M., & Kozyrakis, C. (2017, April). *"Tetris: Scalable and efficient neural network acceleration with 3d memory."* In Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 751-764).

[13] Y.-H. Chen, J. Emer, and V. Sze. *"Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks."* In 43rd Annual International Symposium on Computer Architecture (ISCA), pages 367379, 2016.

[14] C. Eckert et al., *"Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks,"* 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), Los Angeles, CA, 2018, pp. 383-396.

[15] J. Jeddeloh and B. Keeth, *"Hybrid memory cube new DRAM architecture increases density and performance,"* 2012 Symposium on VLSI Technology (VLSIT), Honolulu, HI, USA, 2012, pp. 87-88, doi: 10.1109/VLSIT.2012.6242474.