Multi-core Processor Learning Using a Simulator and Pin Tools

Dr. Yul Chu, University of Texas, Rio Grande Valley

Dr. Yul Chu is an Associate Professor in the Department of Electrical Engineering at the University of Texas Rio Grande Valley. He received his Ph.D. in Electrical and Computer Engineering from the University of British Columbia, Canada in 2001 and MS in Electrical engineering from Washington State University in 1995. His current research interests lie in the area of low-power embedded systems, high-performance computing, parallel processing, cluster and high-available architectures, computer networking, digital system design, etc.

©American Society for Engineering Education, 2019

Multi-core Processor Learning using a Simulator and Pin Tools

Abstract:

In this paper, we propose a simple simulation model for multi-core processor learning, which will provide students effective ways of learning cache memory architecture through computer architecture labs including new cache designs. The proposed pedagogical approach is based on the Kolb experiential learning cycles. In our approach, it is recommended to use the Simple Simulator and Pin Tool. We developed the Simple Simulator, while the Pin Tool is open-source (from Intel) to build a trace file for any benchmark programs. The Simple Simulator can implement any detailed characteristics for a cache scheme, such as replacement policy, mapping function, average memory access time, coherence protocol, amount of bus traffics, power consumption, etc. After PIN Tool builds trace files, those files will be inserted into the Simple Simulator to collect the outputs to measure performance of cache scheme.

Introduction:

For a computer architect, cache memory is a key functional unit to consider in both increasing system performance and lowering power consumption for multi-core processors [1]. Therefore, multi-core cache scheme has been a popular research and teaching topic in computer architecture communities. In this paper, we present how to design and implement a multi-core cache memory using the Simple Simulator and Pin Tool for senior- and/or graduate-level computer architecture labs and designs. Until now, many open-source cache memory simulators (such as SimpleScalar [2], Multi2Sim [6], or Gem5 [4]) have been used to design and evaluate performance for single-core and multi-core cache memories. For example, SimpleScalar (open-source) has been popular to perform accurate simulation for a modern processor, such as RISC architecture [2, 7], using benchmark programs. For reference, SPEC benchmark programs [3] have been popular to evaluate CPUs and cache memories in the computer architecture community. However, those simulators are very complicated to implement multi-core cache schemes for students. Therefore, there have been reasonable demands to develop a flexible and simple multi-core cache memory simulator, which can design and implement any cache schemes without learning how to use those simulators in detail. To meet such demands, this paper proposes the

Simple Simulator to implement a multi-core cache scheme for students, which can design it through simple running methods using Pin Tool. The proposed Simple Simulator is a trace-driven simulator since it needs to read traced instructions and data along with memory addresses and functions, such as load, store, and other instructions. The Pin Tool collects the traced instructions and data by executing real application programs (or benchmark programs) [5, 8]. However, the cache scheme design is not easy since students need to know both CPU and cache memory in detail, such as how to access and process data in a cache scheme. Therefore, it can be a challenge for a student to learn, design, and implement a multi-core cache memory. In addition, in the aspect of teaching computer architecture, a practical teaching methodology is needed. Since the traditional classroom lecture-based teaching has many limitations on the learning capability, team-based learning methodologies have been adapted in many disciplines in the areas of science and engineering. For the disciplines of computer science and computer engineering, project-based learning has been used as a popular methodology in helping students to understand course materials and to apply theoretical knowledge to solve real-world problems effectively.

However, it is still difficult for students to implement their computer architecture labs with their own design specifications, such as replacement policy, writing policy, etc. This brings about the issue of how to guide students in the right direction. Based on our long-term experience of conducting research and teaching in the field of computer architecture, we developed and proposed a new multi-core processor learning methodology, which elevates students' knowledge and training based on the Kolb experiential learning cycles to complete their cache memory design labs.

The rest of this paper is organized as follows. First, we describe the proposed multi-core processor learning through the Kolb experiential learning cycles. Second, simulating conventional cache schemes for experience-based learning is introduced. Third, designing a new cache scheme and cache coherency for design-based learning is introduced. Then, we present a simulation methodology, benchmark programs, and experimental results.

Multi-core processor learning using the Simple Simulator and Pin Tool through the Kolb Experiential Learning Cycles:

Figure 1 introduces four modified steps (2 steps for experience-based learning and 2 steps for design-based learning) of Kolb experiential learning cycles [9], which are highly effective pedagogies in teaching multi-core processor learning using several labs as follows:

Two steps for experience-based learning:

- Concrete observation consists of running the Simple Simulator and Pin Tool to get some results as an experience; and
- Reflective observation consists of reviewing the simulation results and reflecting on how to design a cache memory with a new idea.

Two steps for design-based learning:

- Abstract conceptualization consists of designing a new idea by porting the code into the Simple Simulator to implement; and
- Active experimentation consists of finalizing the design after analyzing the simulation results.

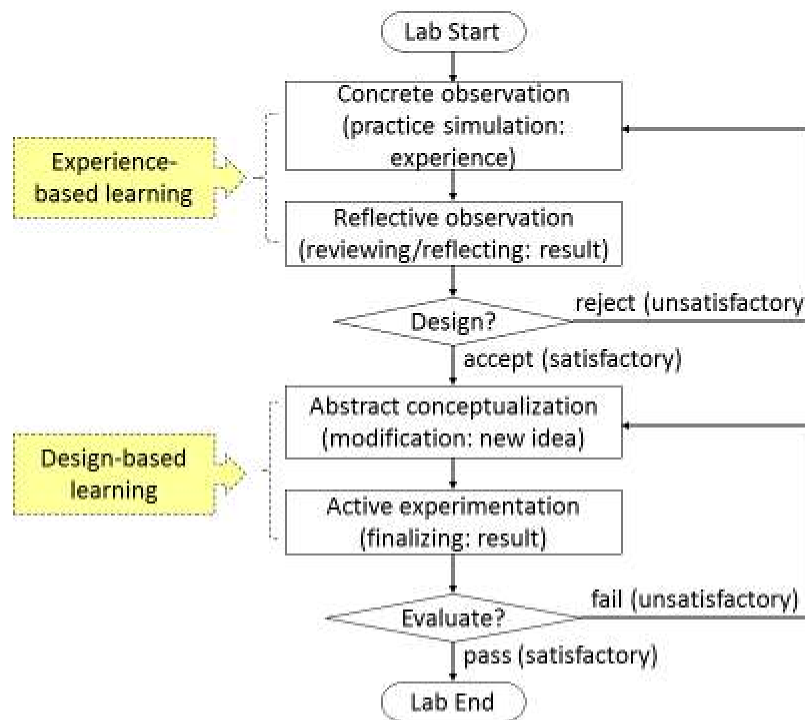

Figure 1. Flowchart of Kolb Experiential Learning Cycles.

Simulating conventional cache schemes for experience-based learning:

The Simple Simulator's purpose is to design and implement multi-core cache memory schemes for computer architecture labs for senior- and/or graduate-level students. Figure 2 shows the major steps to implement the Simple Simulator, which has five stages as follows: 1) Getting input/output parameters from a command line; 2) Fetching a line from a trace file; 3) Decoding the data and

calculating memory address; 4) Implementing cache hit and cache miss; and 5) Collecting output results.
For the Kolb experimental learning cycles, experience-based learning needs to use the entire 5 stages by using the Simple Simulator and Pin Tool, which already have the functions for conventional cache memory schemes (e.g. direct-map and n-way set-associative). Through these stages, students will learn how to simulate conventional cache schemes using the Simple Simulator and Pin Tool to get outputs in stage 5 in Figure 2.

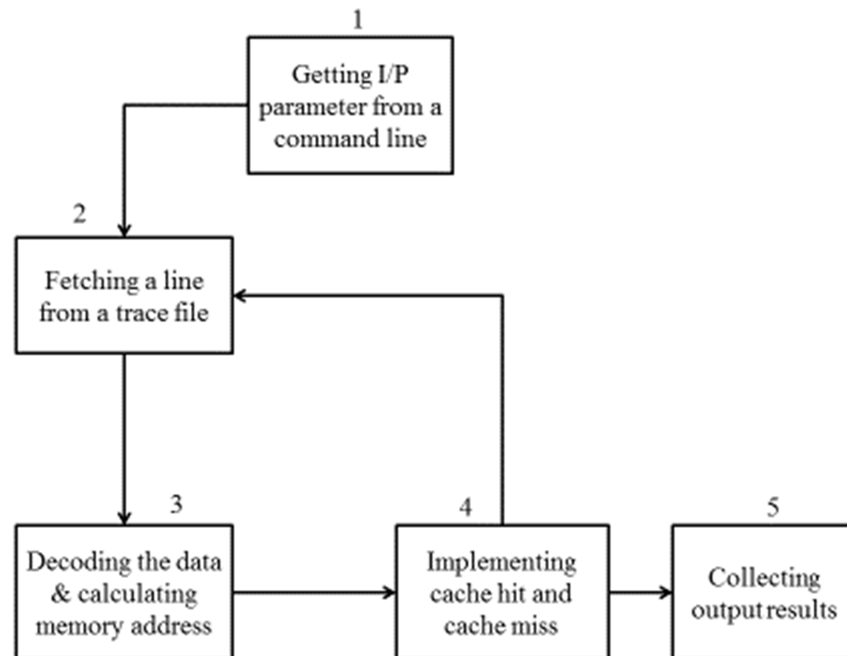

Figure 2. Five steps to implement cache memory using the Simple Simulator and Pin Tool

The five stages in Figure 2 are as follows:

- Getting I/P Parameters from a Command Line (1st step): The Simple Simulator reads the cache memory parameters from the command line. The parameters are the cache size, block size, set-associativity, write policy, number of cores, coherence protocol, etc.
- Fetching a Line from a trace file using Pin Tool (2nd step): Figure 3 shows an example of a trace file: The first column represents a number of thread or core, such as '0' for single core and other numbers for cache coherence (i.e., multi-core). The second column shows the type of address, such as 'r' for load memory address (reading), 'w' for store memory address (writing), and 'O' for other instructions. In addition, the third column shows memory addresses to access for each instruction. For reference, the Pin Tool (version 3.2) is an open-source

software tool (provided by Intel) to generate a trace file to feed into the proposed cache simulator using benchmark programs.

- Decoding the data & calculating memory address (3rd step): After decoding a fetched data, the memory address should be placed or updated into cache memories, such as L1 and L2 cache memories. In order to update cache memories, it is necessary to check cache hit/miss before the update according to the following four cases: 1) cache hit for instructions except Store; 2) cache hit for Store instruction; 3) cache miss for instructions except Store; and 4) cache miss for Store instructions.

raytrace (~/TraceFiles) -

Open Save

raytrace ×

```
0 w 0xbfd00dcc
0 w 0xbfd00dc8
0 w 0xbfd00db8
0 r 0xbfd00db8
0 w 0xbfd00d88
0 w 0xb7701cd4
1 w 0xab59010c
3 r 0xb10dd1b0
2 r 0xb2dbfe74
3 w 0xb10dd180
1 r 0xab58ff30
7 r 0xadcb1170
1 r 0xab58ff18
5 r 0xaf6c720c
```

Figure 3. A trace file generated by the Pin-tool.

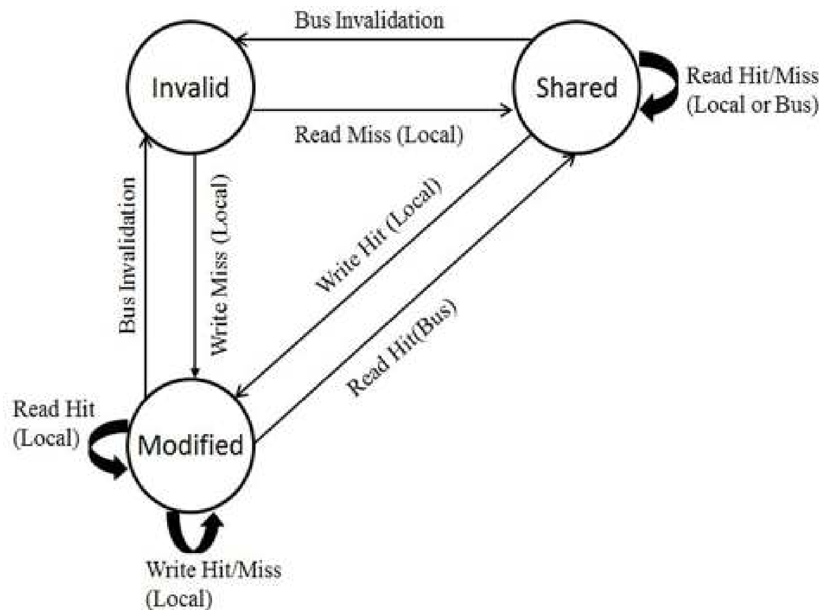

Figure 4. Snoopy coherence transition diagram [10].

- Implementing cache hit and cache miss (4th step): Cache hit/miss is determined based on a cache-coherence protocol (e.g. Snoopy) for multi-core cache schemes. Figure 4 shows the transition diagram for Snoopy cache coherence protocol [10] as an example. Snoopy protocol contains three different states, such as Modified, Shared, and Invalid.
- Collecting output results (5th step): After running a trace file, the simulator produces the following results per core like outlined in Figure 5, which provides a total number of load, store and other instructions, number of cache hits, cache miss rates, etc.

```
rano@ubuntu: ~
rano@ubuntu:~$ ./FM-SIM -l1 13:5:4:2:1: -C4 -P2 raytrace

CacheSize                8192Bytes
BlockSize                32Bytes
Assoc:                   4
Protocol:                MESI
        Result Core0
#READ                    1.01998e+08
#RDmiss                  4.64831e+06
#Write                   5.4437e+07
#WTmiss                  711184
Missrate                 3.42601 %
#BusRds                  4.64831e+06
#BusRdxs                 711184
#BusUpgrs                75772
#Cach-to-Cache           1.45654e+06
        Result Core1
#READ                    7.77976e+07
#RDmiss                  3.77248e+06
#Write                   4.07092e+07
#WTmiss                  597070
Missrate                 3.68717 %
#BusRds                  3.77248e+06
#BusRdxs                 597070
```

Figure 5. Output results.

After completing the above 5 steps, students learn how to simulate conventional cache memory schemes using the Simple Simulator and Pin Tool. Therefore, we expect they can design and implement their own cache scheme by porting their cache code into the Simple Simulator at the design-based learning stage, such as the 'Abstract conceptualization and Active experimentation' stages.

Designing a new cache scheme and cache coherence for design-based learning:

For experience-based learning, students just use the conventional cache schemes and Snoopy cache coherency for learning how to simulate cache schemes for multi-core processor through 5 steps using the Simple Simulator and Pin Tool. Then, at the design-based learning stage, students will design their own cache scheme and cache coherency (option) for multi-core processor using the Simple Simulator and Pin Tool. For doing this, students need to

design (and program using C++) their cache scheme for a new replacement policy, mapping function, and write policy. Then, they need to port their code into the Simple Simulator. There are four cases for implementing cache hit or miss: 1) cache hit for instructions except Store; 2) cache hit for Store instruction; 3) cache miss for instructions except Store; and 4) cache miss for Store instructions. Figure 6 shows the Simple Simulator coding for cache scheme along with a replacement policy (e.g., Least Recently Used or LRU) and a write policy (e.g., write through or write back). For example, if students want to design a new cache scheme, such as 2-way skewed-associative, then they need to port mapping function and replacement policy into the Simple Simulator. The red boxes in Figure 6 show the porting codes for 2-way skewed-associative cache on top of the Simple Simulator.

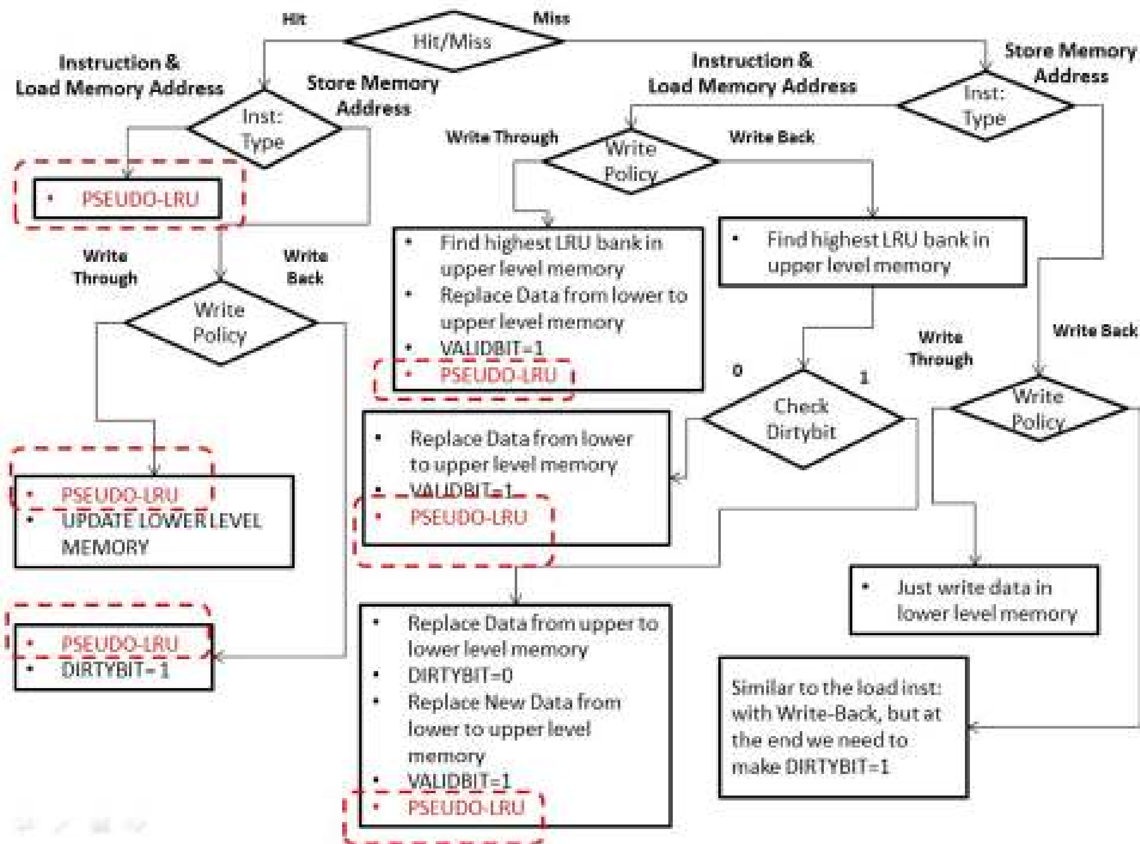

Figure 6. Four cases for implementing cache hit or miss including 2-way skewed-associative cache scheme.

The red boxes in Figure 6 are for Pseudo LRU (PLRU, pseudo least replacement used) policy and the PLRU works as follows: The 2-way skewed-associative uses an XORing mapping function for 2 banks; for Bank 0, it uses un-hashed indexed; for Bank 1, it uses a hashed indexed function. It employs Pseudo-LRU replacement policy [11]. The replacement control bit is located at

each cache line in Bank 0; if the data is missed in Bank 0, then control bit value will be set to ‘1’ after replacement from the lower memory. The ‘1’ means, for the next cache miss in that line, the data will be updated to Bank 1 (then, the control bit is updated to ‘0’).

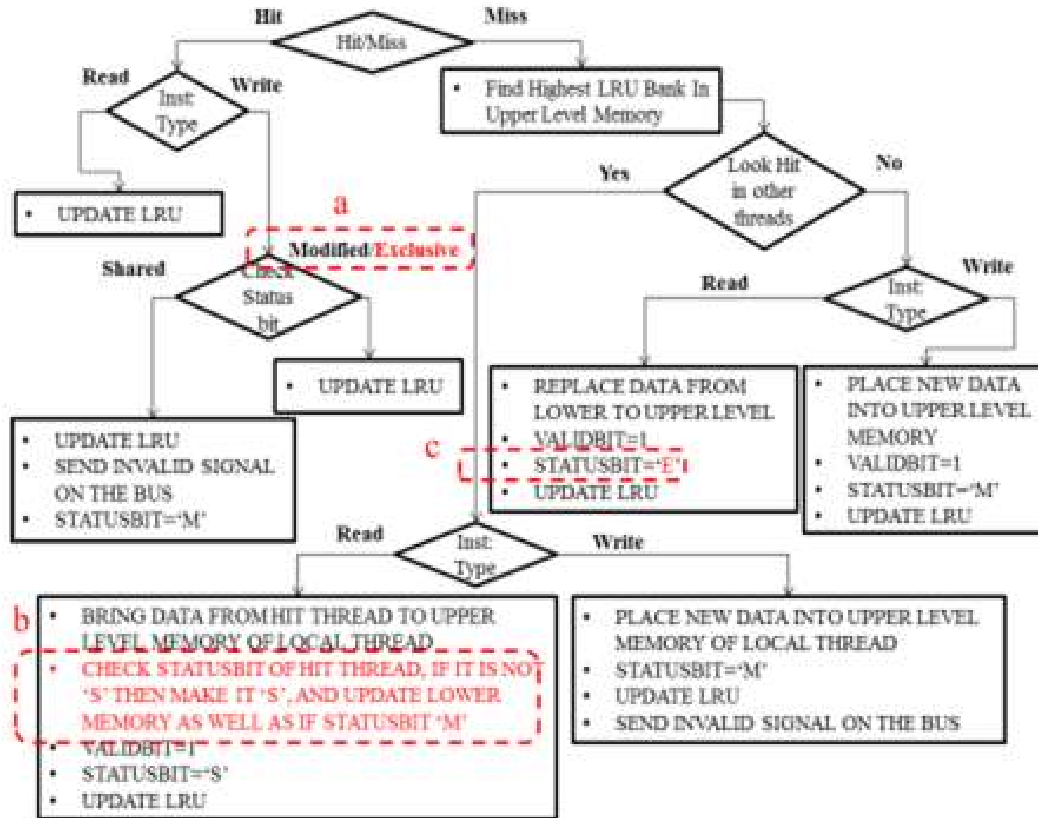

Figure 7. Snoopy coherency protocol including MESI protocol (red-boxes)

Like the same way, students can port their cache coherency protocol for multi-core shared memory as well. For example, if students want to port MESI (Modified-Exclusive-Shared-Invalid) protocol, they can port four states for MESI into the Simple Simulator as follows: 1) Modified state: For read miss, the current state will be changed into Exclusive state. For read hit (bus), the current state will be moved to Shared state. In the case of ‘Bus invalidation signal,’ the state will be transferred to Invalid state. For read/write hit (local), the state will stay at the current state; 2) Exclusive state: For read hit (bus), the current state will be changed into Shared state. For write hit (local), the state will be changed into Modified state, and for ‘Bus invalidation signal,’ the state will be changed into Invalid state, and for read hit(local), the state will remain the same state; 3) Shared state: For write hit (local), the current state will be moved to Modified state. For ‘Bus

invalidation signal,' it will be changed into Invalid state. For read hit (local or bus), the state will remain at the current state; and 4) Invalid state: For read miss (ex), the current state will be moved to Exclusive state. For read miss (sh), the current state will be moved to Shared state. For write miss (local), it will be moved to Modified state.

Figure 7 shows the flowchart for the Snoopy cache coherency protocol, which is a conventional protocol, and the Simple Simulator already has this protocol as a default one. The red boxes in Figure 7 have the ported codes for MESI protocol with Snoopy. The red boxes mean: 1) Check the Status bit whether it is Modified or Exclusive. If then, the MESI will do 'UPDATE LRU'; 2) Doted box 'b' is for 'STATUS BIT is EXCLUSIVE,' when the fetched instruction is 'read miss and not found in other threads'; and 3) Doted box 'c' is for the case that the instruction is 'read miss but hit in other thread'. If then, check the STATUSBIT OF HIT THREAD. If it is not SHARED, then update it to SHARED, and update lower memory as well such that the status bit is modified.

Simulation methodology, benchmark programs, and experimental results:

For the experiments, SPEC 2006, Parsec, and SPLASH2x benchmark programs [3, 12] are used to evaluate performance for various multi-core cache memories. The steps for experimental methodology are as follows: 1) SPEC, Parsec and SPLASH2x benchmark programs are compiled by using GCC compiler (or other compilers) to make execution file (binary code or machine language); 2) the execution file will be run on a Linux machine (Ubuntu-15.04) using the Pin Tool to collect trace files; 3) Then, trace files are inserted into the proposed cache simulator to get the outputs. Here, students can port their designed multi-core cache scheme into the Simple Simulator; and 4) they can get the outputs for their performance analyses.

Table 1: List of Trace files for SPEC2006 Benchmark Programs.

Programs	Instruction # (Input: Train)	Instruction # (Input: ref)
401.bzip2	142 BILLIONS	348 BILLIONS
456.hmmer	314 BILLIONS	2,164 BILLIONS
465.tonto	1,351 BILLIONS	3,466 BILLIONS
471.omnetpp	265 BILLIONS	701 BILLIONS

Table 1 and Table 2 show the SPEC-2006, Parsec, and SPLASH2x benchmark programs [3, 13] with the input data (train, ref, simsmall, simmedium, and simlarge) and number of instructions during run time. The Pin Tool was used to collect the following

data, which are listed in Table 1 and 2. For example, 401.bzip2 is a SPEC2006 benchmark program, which has 142 BILLIONS dynamic instructions (train input).

Table 2: List of Trace files for Parsec and SPLASH2x Benchmark Programs.

Programs	Instruction # Input: simsmall	Instruction # Input: simmedium	Instruction # Input: simlarge
bodytrack	1,415 M	4,520 M	15,873 M
freqmine	993 M	3,026 M	9,259 M
raytrace	2,151 M	13,926 M	55,082 M
fmm	2,541 M	10,890 M	44,336 M
radiosity	1,438 M	1,438 M	1,438 M

Figures 8 to 9 show the experimental results from the following multi-core cache parameters: 1) Multi-core parameters: number of cores (8 cores), set-associative (4-way and 8-way), block size (32 bytes and 64 bytes), cache sizes (2KB, 4KB, 8KB, 16KB, 32KB, and 64KB), and MESI cache coherence protocol; 2) Results: Miss rates (read and write cache misses), WHP (write-hit private), RHP (read-hit private), WHS (write-hit shared), RHS (read-hit shared), and RM (read miss). Figure 8 shows the miss rates based on the cache sizes, such as from 4 KB to 256 KB using 4 benchmark programs and the results are similar to the ones on [13, 14]. In general, cache miss rates would be reduced according to the cache memory sizes. In Figures 9, all the number of local hits are counted as 'private.' In addition, all the number of 'Bus' hits are counted as 'shared.' From the figures, we can claim that the Simple Simulator works appropriately since the experimental results are similar to the results on [13, 14].

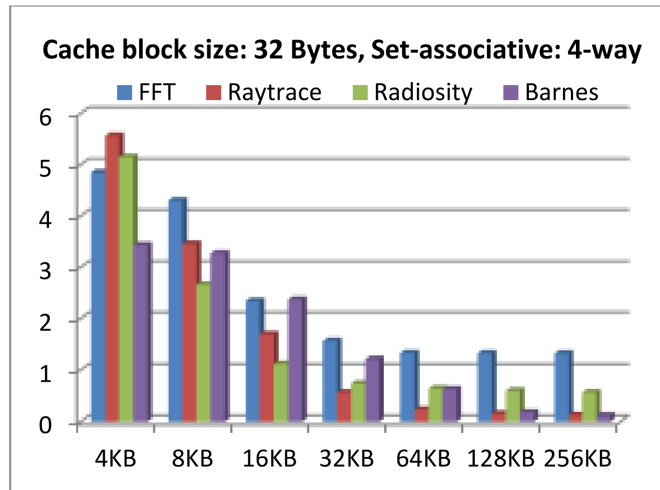

Figure 8. Miss rate (MESI)

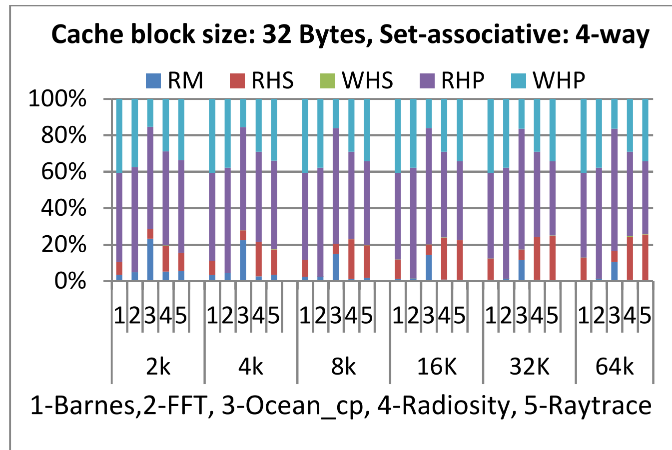

Figure 9. WHP, RHP, WHS, RHS, and RM for Multi-Core cache schemes (8 cores)

Conclusion:

We proposed a multi-core processor learning model using the Simple Simulator and Pin Tool for teaching computer architecture labs. The pedagogical approach for this paper is based on the Kolb experiential learning cycles, which consist of four modified steps as follows: 1) two steps for experience-based learning are 'Concrete observation' and 'Reflective observation'; and 2) two steps for design-based learning are 'Abstract conceptualization' and 'Active experimentation.' The Simple Simulator has 5 stages of operational flow as follows: 1) Getting input/output parameters from a command line; 2) Fetching a line from a trace file; 3) Decoding the data and calculating memory address; 4) Implementing cache hit and cache miss; and 5) Collecting output results. From the experience-based learning stages, students learn how to simulate conventional cache schemes using the Simple Simulator and Pin Tool. Then, from the design-based learning stages, students can design their own cache scheme and cache coherency protocol for multi-core processor using the Simple Simulator and Pin Tool. Students (graduate or senior) have used the Simple Simulator and Pin Tool for doing their team projects at the contemporary microprocessor design (CMD, advanced computer architecture course) class since fall 2016. The project used to design a low-power cache memory for a single-core or multi-core. Actually, before 2016 (such as fall 2015), there were five teams for doing their projects in the CMD class using two traditional open-source simulators, such as Simplescalar and Modelsim2. At that time, four teams failed to complete their designs since those simulation programs were difficult to implement their designs. However, after fall 2016, most teams could implement their low-power cache designs without any problems by using the Simple Simulator and Pin Tool. Therefore,

we could say that this new learning model is very effective since most students in the class clearly understood the concept of low-power cache memory design through the modified Kolb experiential learning cycles. In addition, from the experimental results, we would like to conclude that the Simple Simulator is functionally correct and works well as a simple and effective tool for computer architecture labs.

References

[1] M. A. Vega Rodriguez, et al., "Simulation of Cache Memory Systems on Symmetric Multiprocessors with Educational Purposes, " Proc. of the I International Congress in Quality and in Technical Education Innovation, vol. III, pp. 47-59. Donostia-San Sebastián, Spain. 4-6 September 2000.

[2] D. Burger, and T. Austin, "The SimpleScalar Tool Set, Version 2.0," University of Wisconsin-Madison Computer Sciences Department Technical Report #1242, June 1997.

[3] Standard Performance Evaluation Corporation, "SPEC CPU 2006 [online]," Available: https://www.spec.org/cpu2006.

[4] M. Al-Manasia and Z. Chaczko Z, "An Overview of Chip Multi-Processors Simulators Technology," Advances in Intelligent Systems and Computing, vol 366. pp. 877-884, Springer, Cham, 2015

[5] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation, " ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Chicago, IL, USA, 2005.

[6] R. Ubal, et al., "Multi2Sim: A Simulation Framework for CPU-GPU Computing," the 21st IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT), Minneapolis, MN, USA, pp. September 2012.

[7] C. Price, *MIPS IV Instruction Set, revision 3.1*. MIPS Technologies, Inc., Mountain View, CA, January 1995.

[8] C. McCurdy and C. Fischer, "Using Pin as a memory reference generator for multiprocessor simulation," SIGARCH Computer Architecture News 33, 5 (Dec. 2005), 39-44.

[9] D. A Kolb, "Experiential Learning: Experience as the Source of Learning and Development," Upper Saddle River, NJ: Pearson Education, 2015.

[10] J. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Waltham, MA, USA: Morgan Kaufmann, Elsevier, 2012.

[11] A. Seznec, A case for 2-way skewed-associativity cache, the 20th International Symposium on Computer Architecture (ISCA), San Diego, USA, May 1993.

[12] C. Bienia, S. Kumar, and K. Li, ″PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors, ″ In Proceedings of IISWC 2008, pages 47–56, Sept. 2008.

[13] M. Elver and V. Nagarajan, "RC3: Consistency directed cache coherence for x86-64 with RC extensions," Proceedings of 2015 IEEE International Symposium on Parallel Architecture and Compilation Techniques (PACT), Washington, DC, U.S.A., 2015.

[14] M. Elver and V. Nagarajan, "TSO-CC: Consistency directed cache coherence for TSO," Proceedings of 2014 IEEE 20th International Symposium on high Performance Computer Architecture (HPCA), Orlando, Florida, U.S.A., 2014.