An Execution-Driven Simulation Tool for Teaching Cache Memories in Introductory Computer Organization Courses

Salvador Petit, Noel Tomás
Computer Engineering Department
Faculty of Computer Science
Polytechnic University of Valencia
spetit@disca.upv.es, noetoar@eui.upv.es

Julio Sahuquillo, and Ana Pont
Computer Engineering Department
Technical School of Applied Computer Science
Polytechnic University of Valencia
{jsahuqui, apont}@disca.upv.es

Abstract

Cache memories are the most ubiquitous mechanisms devoted to hide memory latencies in current microprocessors. Due to this importance, they are a core topic in computer architecture curricula, both in graduate and undergraduate courses. As a consequence, traditional literature and current educational proposals devote important efforts to this topic. In this context, exercises dealing with simple algorithms, also known as code-based exercises, have a good acceptance among instructors because they permit students to realize how the accesses generated by the programs affect the cache's state.

From about one decade ago, simulators have been extensively employed as a valuable pedagogical tool as they enable students to visualize how computer units work and interact each other. Unfortunately, there is no simple simulator allowing to perform code-based exercises for cache memories. Hence, students perform these exercises by means of the classic "paper and pencil" methodology.

In this paper we introduce Spim-cache, a simple execution-driven cache simulator to carry out such experiments, intended to use in undergraduate courses. The tool allows, in an intuitive and easy way, to select a given cache organization and run step-by-step the code proposed while visualizing dynamic changes in the cache's state.

1 Introduction

The ever increasing gap between the memory and the microprocessor speeds has encouraged microprocessor architects across several decades to provide mechanisms in order to hide the long memory latencies. Cache memories have become the basic and ineludible mechanism implemented in current processors to hide these latencies and reduce the average data access time. Furthermore, the importance of caches grows as the memory-processor gap widens, which is the current trend; for instance, in 1980 some microprocessors were designed without caches while current microprocessors include two or even three levels of caches [1] on chip. These memory structures base their effectiveness in the exploitation of the *principle of locality* (i.e., temporal and spatial) that data exhibit, and have been employed by computer architects from about four decades ago [2].

Consequently, a large amount of research works in the computer architecture field have focused on cache memories and related issues. This research has provided efficient mechanisms to manage and exploit caches, and some of them have been implemented in modern microprocessors. For instance, the AMD Ahtlon [3] includes a victim cache [4], the Itanium 2 [1] incorporates a prevalidated tag structure to provide fast accesses, the HP 7200 implements the Assist Cache [5], a particular type of split data caches [6].

Concerning to international curricula recommendations, the joint IEEE Computer Society and ACM Computer Engineering Task Force has identified cache memories as a core topic in the Computer Organization and Architecture [7] area of knowledge. This fact is because the processor performance strongly depends on the cache performance. As a consequence, cache memories play an important role in Computer Organization/Architecture Courses mainly offered at Computer Engineering schools, although concepts concerning cache memories are also studied in a more simplified way in other Technical Schools, like Electrical Engineering or Computer Science. Computer Organization courses cover the study of the basic functional units of the computer and the way they are interconnected to form a complete computer. Computer Architecture courses mainly cover advanced processor architectures, advanced memory mechanisms, and multiprocessor systems; as well as the corresponding performance evaluation issues. Concerning to the study of caches, they involve a wide variety of con-

cepts and functionalities ranging from the basic functionality to advanced mechanisms implemented in modern microprocessors. The basis of caches, covered in initial Computer Organization courses, deals with concepts like cache organization, mapping functions, and replacement algorithms. Advanced mechanisms, like way prediction or the trace cache, are covered in a posterior Computer Architecture course.

To teach how caches work some widely referenced books, e.g., [8], [9], [10], and [11], use simple algorithms, for instance, the sum of the elements of an array (see Figure 1). For these algorithms, students must obtain some metrics concerning cache behavior, e.g., the hit ratio. These examples have a great pedagogical value to reinforce the learning process and have been commonly adopted by instructors around the world. Nevertheless, despite the undoubtable pedagogical value, instructors train students performing these exercises using the classic *paper and pencil* method.

```
sum = 0;
for (i = 0; i <100; i++)
  {
    sum = sum + A[i];
  }
```

Figure 1. Simple algorithm example

The massive use of computers in classrooms and laboratories has resulted in new teaching methodologies using computers as a pedagogical tool. In this context, the use of simulators is highly recommended as they enable students to visualize how the modelled system operates. A wide set of these simulators has been developed to teach the basic functional units of the computer in Computer Organization/Architecure courses, e.g., simple processors, computer arithmetic units, or cache memories. Concerning to cache memories there are simple simulators dealing with the basis of caches. Unfortunately, and to the knowledge of the authors, none of these basic simulators can be used to perform code-based exercises, i.e., the same kind of exercises that have been traditionally proposed in the teaching literature. In this paper Spim-cache is proposed, as a pedagogical tool intended to be used in undergraduate courses, attempting to fill the existing gap between current simulators and the reference guides.

Spim-cache has been developed by students and staff of the Computer Engineering School at the Polytechnic University of Valencia. In this University, computer organization subjects are split in two annual courses [12], namely Computer Fundamentals (first year) and Computer Organization (second year). In these courses, as in a high number of universities around the world, the MIPS approach is used to illustrate the corresponding topics, e.g., the processor pipeline, the memory hierarchy, and the input-output system, are studied using the same machine model. This means that students are trained with the MIPS assembly language and the corresponding Spim [13] simulator. This teaching context led us to extend the Spim simulator to implement our proposal.

Spim-cache provides support to perform code-based exercises detailing cache events in a friendly and easy way. Spim-cache allows students to run programs while visualizing, step-by-step, how the cache controller works, e.g., fetching memory blocks, or dealing with write policies. The simulator code is open source and can be found at http://www.disca.upv.es/spetit/spim.htm.

The remainder of this paper is organized as follows. Section 2 summarizes the main features of current simulators and the reasons that encouraged us to deal with this work. Section 3 describes the proposed pedagogical training tool. Finally, Section 4 presents some concluding remarks.

2 Background and motivation

An interesting laboratory using real microprocessor's caches is described in [14]. In this laboratory, students run a small benchmark on a given computer. The benchmark is mainly composed by a nested loop that reads and writes an array of data using different strides and array sizes. The benchmark measures the average data access time, which varies according to several factors, e.g., if this array is larger or smaller than the cache size. Analyzing the times provided by the benchmark execution, students must deduce the cache geometry (i.e., the cache size, the line size, and the associativity degree). Further details can be found in [10]. This kind of laboratory, really encourages students as they work directly on the real hardware. Unfortunately, in order to ease the analysis of the results, the benchmark must run in a relatively *old* processor like the Pentium II which includes a simple cache organization. In more recent microprocessors including advanced cache mechanisms, for instance, some kind of prefetching like the Intel Pentium 4 [15], or victim caches like the AMD Ahtlon, the cache geometry could not be deduced from the results provided by the benchmark.

The mentioned drawback jointly with the pedagogical value that simulators provide, led instructors to train students by using cache simulators. Table 1 shows a subset of current simulators dealing with caches that are being currently used for educational or research purposes, or both. An interesting survey of simulators can be found in [16]. Simulators can be classified in two main groups attending to the way they are fed: trace-driven simulators and execution-driven simulators. The first ones are fed by simple traces

Table 1. Educational simulators: an overview

Simulator	Complexity	Driven	Graphic Interface	Core Details
DCMSim	Low	Trace Driven	Yes	No
Dinero	Medium	Trace driven	No	No
mlcache	Medium	Trace or execution driven	No	No
Simplescalar	High	Execution driven	No	Yes

while the second ones, much more complex, are fed by the binary of a given benchmark.

To teach students in undergraduate courses instructors use trace driven simulators, e.g., [17]. A trace is composed by a set of lines, where each line represents an specific event, i.e., a write or read operation in a given memory address. The simulator shows how the cache contents change each time a new event occurs. In this way, students can follow how memory accesses miss or hit into the cache. Some trace driven simulators have been also used for research purposes like [18] and [19]. Nevertheless, as this kind of simulators are fed by traces they are not appropriate to perform code-based exercises.

Advanced execution-driven simulators are mainly used for research purposes. These sophisticated tools may concentrate only on cache related issues, e.g., the mlcache simulator [18], or in the complexity of the entire microprocessor, e.g., the simplescalar toolset [20], which models an aggressive out-of-order processor. This kind of tools can provide detailed information cycle-by-cycle, showing how the machine units interact each other. Due to the huge complexity involving current microprocessors, this kind of tools are organized in a structured way, including a specific cache memory module. The main advantage of advanced simulators is that they permit to know the details on each instruction in the pipeline, however, they usually fail in that they do not provide a friendly graphic interface and are quite difficult to use. This fact jointly with the complexity of the modelled processor make advanced execution-driven simulators unsuitable to perform code-based exercises in undergraduate courses.

In this paper we propose a simulator intended to use in undergraduate courses which provide support to perform code-based exercises. The tool attempts to get the best of both kinds of simulators, i.e., a friendly and easy-to-use simulator which visualizes, cycle-by-cycle, the architectural state of the machine. From the pedagogical point of view, the proposal provides a framework which permits to perform the typical exercises based on simple algorithms proposed in the literature.

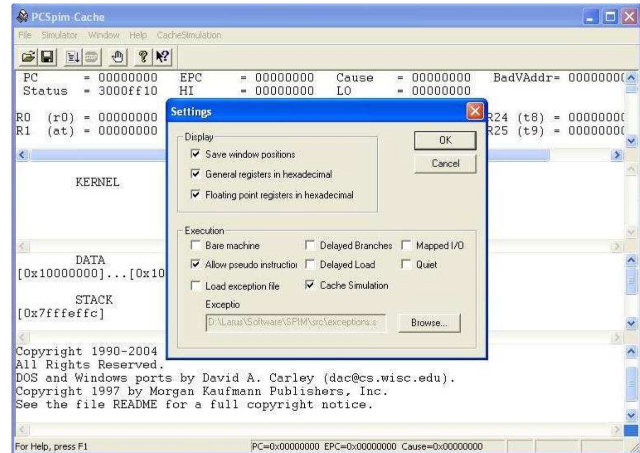

Figure 2. Cache simulator option

3 Spim-cache

Spim is a simulator that runs MIPS32 assembly language programs. It has a wide acceptance among instructors of introductory Computer Organization courses and it has been widely referenced in [8]. UNIX, Linux, MS-DOS, and Windows versions are available. The user's interface for Microsoft Windows platforms, also known as PCSpim, consists of a window that is composed by four frames, from up to bottom: the contents of the register file, the program being simulated (assembler and machine codes), the memory, and a list of log messages. This simulator permits to illustrate the instruction set architecture, i.e., the processor as seen by the programmer.

The proposed tool extends the Windows version providing support to simulate cache memories. The tool also visualizes step-by-step how the cache contents change. In addition, important statistics, e.g., number of hits or misses, are also provided on-the-fly. The tool supports the simulation of both data and instruction caches.

In order to start with the cache simulation, user should firstly select the *Cache Simulation* option in the *Cache Settings* dialog, which pops up after clicking on the *settings* entry of the simulator menu (see Figure 2). After doing so, a dialog with the different cache organization configuration options is displayed (see Figure 3). Users can select among

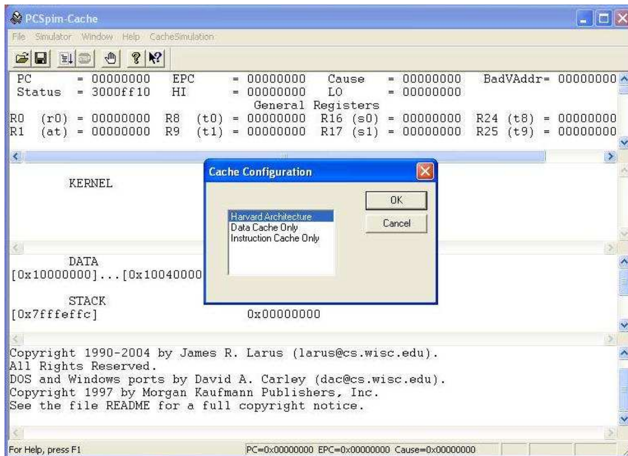

Figure 3. Cache configuration dialog

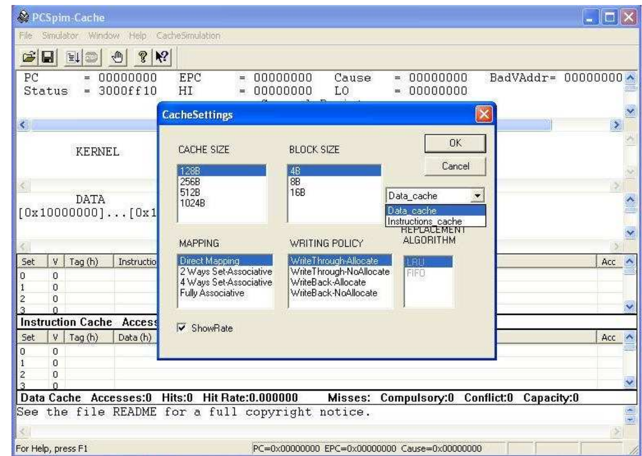

Figure 4. Cache settings dialog

simulate a data cache, an instruction cache, or both, as explained below.

3.1 Cache Architecture Configuration

Current processors implement independent cache organizations to store instructions and data. In this context, Spim-cache permits to simulate only a data cache, only a instruction cache, or both together (i.e., Harvard Architecture) as implemented in modern processors.

The default configuration (i.e., the Harvard Architecture) can be changed through the proper dialog (see Figure 3), by clicking on the *Cache Architecture Configuration* entry of the *Cache Simulation* menu option. The data and instruction caches are displayed as two independent frames inside the main PCSpim window. Both frames are only displayed if the Harvard Configuration is selected; otherwise, only one frame is shown.

3.2 Cache settings

To display the *Cache Settings* dialog, users should click on the *Cache Settings* entry of the *Cache Simulation* menu option. This dialog allows users to change the configuration of the caches. The same dialog is used to change both data and instruction cache configuration. To change between them, select the proper option from the *Cache combo_box* in the dialog as shown in Figure 4.

The dialog shows the different options available. This interface allows users to choose the cache size (ranging from 128B to 1024B), the block size (4, 8 or 16B), the mapping function (direct mapping, 2 or 4 ways set associative or fully associative), the writing hit and miss policies (write-through or write-back, and write-allocate or write-no-allocate), and the replacement algorithm (LRU, FIFO). As instruction caches are not allowed to be modified by user programs, the writing hit and miss policies are only available for data caches. In addition, in order to provide pedagogical feedback, if the *show rate* option is selected, Spim-cache displays some statistics, like the number of misses, in a small frame below the cache frame.

The cache frame visualizes the cache contents and its layout, depending on the selected configuration. The direct-mapped is the default configuration and its layout is organized as a matrix. Each row in the matrix represents a cache line and the columns represent all the information associated with it. The first column contains the line identifier (or line number) while the remaining columns contain data and control information. The control information displayed for this configuration is the valid bit and the tag bits (in hexadecimal notation). In addition, for pedagogical purposes, a field *Acc* showing the result of the cache access (i.e., hit or miss) has been included.

When adding associativity to the cache (i.e., increasing the number of ways) the layout of the cache frame changes. To this end, Spim-cache replicates the columns of a direct-mapped cache as many times as the number of ways selected. Proceeding in this way, each row of the window represents a single set of the cache (i.e., the lines belonging to the set) and the first column states the set identifier. To support the replacement policy, an additional column *(LRU/FIFO)* is added for each way, to represent the counter value of the replacement algorithm. In addition, if the user selects the write-back policy, a new column *(M)* is added for each way to represent the modified or *dirty* bit of each block.

In order to facilitate the user interaction and offer a better visualization, the basic strategy of replicating columns to provide a higher number of ways to the cache is not followed by a fully associative cache. On such case, the basic layout is transposed. In this configuration, the first column indicates the number of way and each row shows the con-

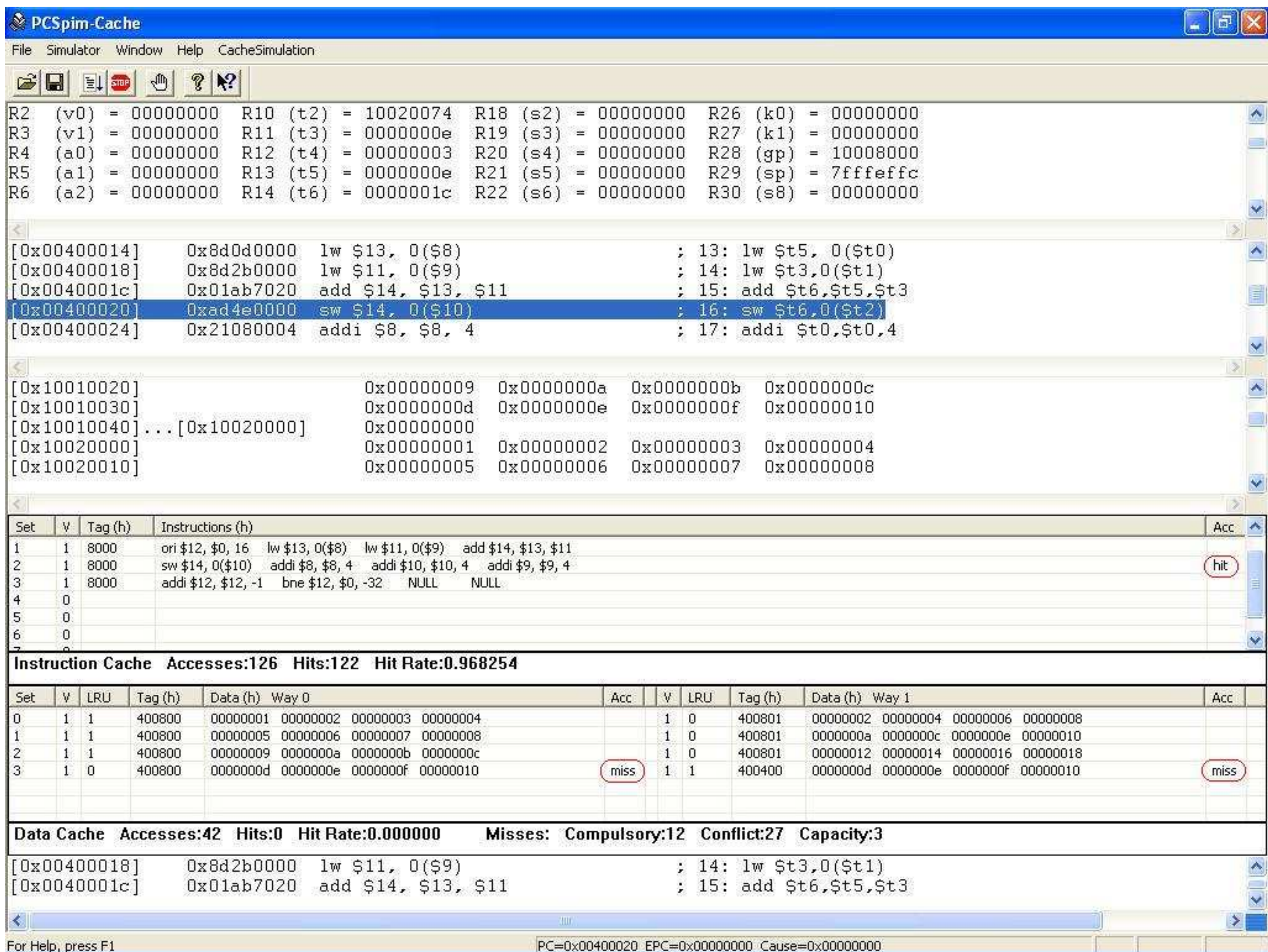

Figure 5. First step of execution

tents of the corresponding line.

3.3 Running a program

Spim-cache allows the user to visualize how the processor's state, the main memory, and the cache interact while a given program runs step-by-step. In addition, statistic results about the cache behavior are also presented. To run a program step-by-step, the user should load it and click on the *single step* menu option of the simulator menu (or use the F10 key function).

In the PCSpim simulator, a step of execution covers the entire execution of a single instruction. The cache simulation extension modifies this semantic in order to help students to follow how the cache memory works. In this sense, memory reference instructions (loads and stores) can take several steps to execute when the extension is activated. In this case, if the access hits into the cache, it takes one step to execute as the remaining instructions. Nevertheless, if the access misses, the number of steps depends on the type of instruction and the write policies.

For the sake of clarity, load and store misses are handled in a different number of steps. On one hand, a load miss is handled in three steps: a) detect and mark the miss in the corresponding set (all lines in the set are marked), b) fetch the block from main memory to the cache, and c) load the desired data into the corresponding register. On the other hand, a store miss is handled in two or three steps depending on the write miss policy (allocate or no-allocate) selected. If the policy is no-allocate, the steps are a) detect and mark the miss, and b) store the content of the register in main memory. Otherwise, the steps are a) detect and mark the miss in the corresponding set, b) fetch the block from main memory, and c) store the corresponding data into the cache. Finally, with respect to the instruction cache, a miss is handled analogously to a load miss in the data cache. In this case the third step consists in the execution of the instruction.

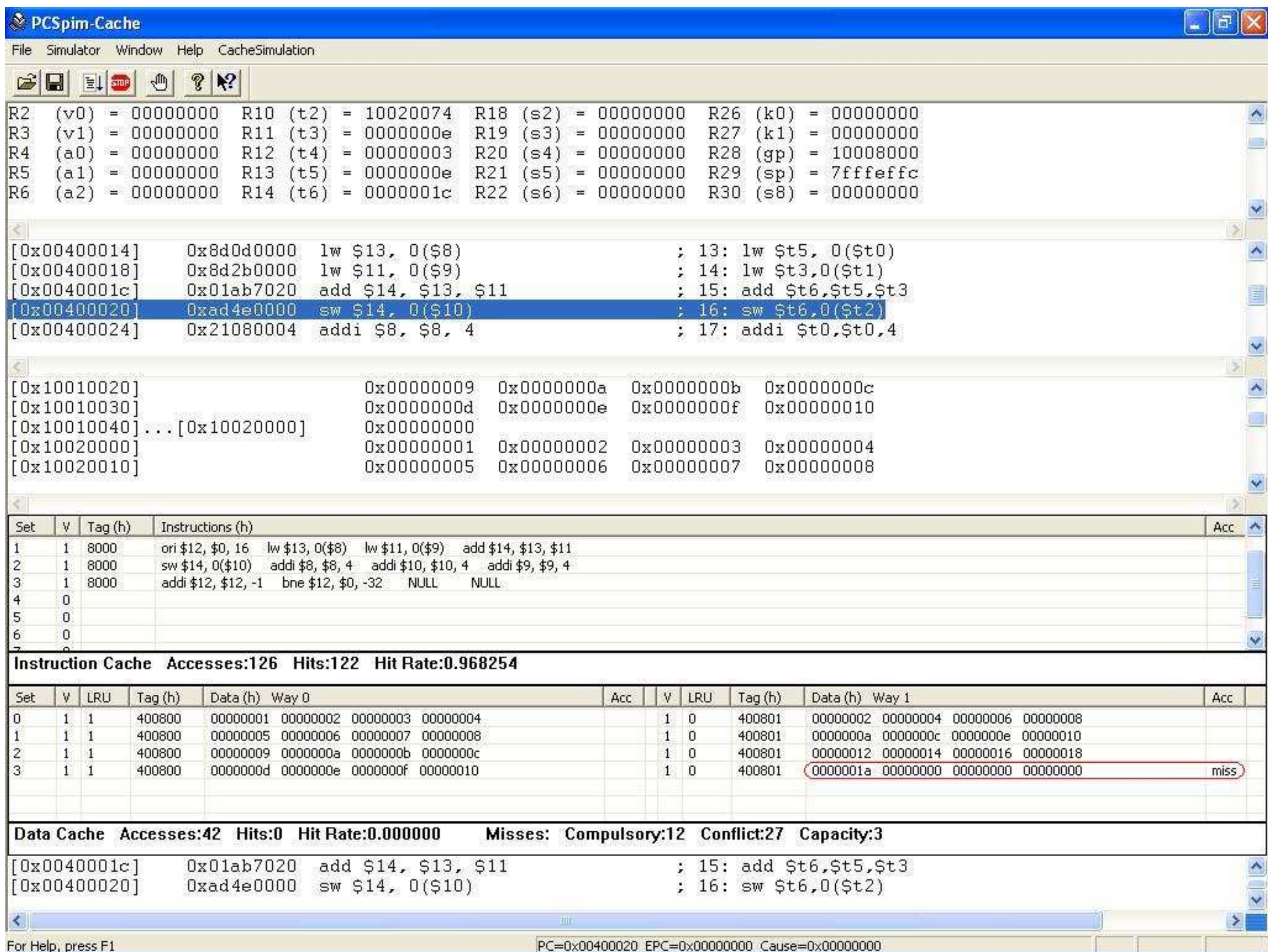

Figure 6. Second step of execution

Let us see a working example in order to illustrate how Spim-cache handles cache misses. Assume a 128B size, 16B line size, direct-mapped instruction cache and a 128B size, 16B line size, 2-way set associative data cache with LRU replacement algorithm, and write-through write-allocate policies. Let us consider the execution of the store instruction highlighted in Figure 5. Notice that the execution of a memory reference instruction involves two cache accesses, one to the instruction cache to read the corresponding instruction and the other one to the data cache.

In the first step, there is instruction cache hit as the store is already in cache (see Figure 5). Also, in the first step when the store is executed and accesses to the data cache this access results in a miss. Since both ways in the set contain a valid line, the LRU algorithm selects the line in way 1 to replace as it contains the LRU line (higher LRU counter value). Hence, in the second step, the missing block is fetched from main memory to the data cache (see Figure 6). Finally, the content of register $14 is written to the data cache. In parallel, it is also written to main memory because of the write-through selected policy.

Notice that on each step of the execution, if the *show rate* option is selected, Spim-cache visualizes the number of accesses, the number of hits and the cache's hit rate. For the data cache the Spim-cache also visualizes the number of the different misses. Misses are classified as compulsory misses (when the block is accessed for the first time), conflict misses (when the target set is full but the cache is not full), and capacity misses (when the cache is full). The obtained statistics would allow the student to realize how the configuration of the cache affects performance.

4 Conclusions

Educational books and instructors use small fragments of code implementing simple algorithms that include memory references to teach cache memories. Students learn caches identifying these instructions and following, using *paper*

and pencil, the sequence of accesses to the cache. In this context, they observe when these instructions miss or hit into the cache.

This paper has presented Spim-cache as a pedagogical tool to perform this kind of exercises in undergraduate courses. The proposed tool has been implemented as an extension of the Windows version of the MIPS R2000 Spim simulator, because this framework is well known and widely used in a high number of universities around the world. Nevertheless, the same idea could be implemented in other basic tools.

From a pedagogical point of view, the proposed tool benefits the learning process, since it permits students to observe how cache information dynamically changes as the processor runs instructions, helping students to learn how the processor-memory work as a whole.

The current version of Spim-cache has been thoroughly tested on the Windows operating system and Linux operating system using the wine library (http://www.winehq.org/). The Spim-cache source code is publicly available at http://www.disca.upv.es/spetit/spim.htm. Users can directly either run the tool by using its binary file or modify its source code to add new functionalities.

Acknowledgements

This work has been partially supported by the Generalitat Valenciana under grant GV06/326 and by Spanish CICYT under Grant TIN2006-15516-C04-01.

References

[1] C. McNairy and D. Soltis, "Itanium 2 Processor Microarchitecture", *IEEE Micro*, Vol. 23, No. 2, March-April 2003, pp. 44-55.

[2] M. V. Wilkes, "Slave Memories and Dynamic Storage Allocation", *Transactions of the IEEE*, vol. EC-14, 1965, pp. 270.

[3] J. Huynh, "The AMD Athlon MP Processor with 512KB L2 Cache", *AMD White Paper*, May 2003.

[4] N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers", *Proceedings of the ISCA-17*, June 1990, pp. 364-373.

[5] K.K. Chan, C.C. Hay, J.R. Keller, G.P. Kurpanek, F.X. Schumacher, J. Zheng, "Design of the HP PA 7200 CPU", *Hewlett-Packard Journal*, February 1996, pp. 1-12.

[6] J. Sahuquillo, A. Pont, "Splitting the Data Cache: A Survey", *IEEE Concurrency*, September 2000.

[7] *Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering*, Joint Task Force on Computer Engineering Curricula, IEEE Computer Society– Association for Computing Machinery, December 2004.

[8] D. A. Patterson, J. L. Hennessy, *Computer Organization: the Hardware/Software Interface*, Morgan Kaufmann 2005 (3rd edition).

[9] W. Stallings, *Computer Organization and Arquitecture: Designing for Performance*, McGraw Hill 2002 (6th edition).

[10] J. L. Hennessy, D. A. Patterson, *Computer Architecture: a Quantitative Approach*, Morgan Kaufmann 2003 (3rd edition).

[11] C. Hamacher, Z. Vranesic, and S. Zaky, *Computer Organization*, McGraw Hill 2002 (5th edition).

[12] J. Real, J. Sahuquillo, A. Pont, L. Lemus, and A. Robles, "A lab course of Computer Organization in the Technical University of Valencia", *Proceedings of the Workshop on Computer Architecture Education*, May 2002, pp. 119-125.

[13] J. Larus, "SPIM: a MIPS32 Simulator", *http://www.cs.wisc.edu/˜larus/spim.html.*

[14] "Machine Structures", *CS61C, UC Berkeley, http://inst.eecs.berkeley.edu/˜cs61c/labs/lab12.txt*, Spring 2005.

[15] G. Hinton, D. Sager, M. Upton, D. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Rousell, "The Microarchitecture of the Pentium 4 Processor", *Intel Technology Journal*, Q1, 2001.

[16] W. Yurcik, G.S. Wolffe, M.A. Holiday, "A Survey of Simulators Used in Computer Organization/Architecture Courses", *Proceedings of the Summer Computer Simulation Conference*, July 2001.

[17] E. Cordeiro, I. Stefani, T. Soares, C. Martins, "DCMSim: Didactic Cache Memory Simulator", *Proceedings of 33rd ASEE/IEEE Frontiers in Education Conference*, November 2003.

[18] E. S. Tam, J. A. Rivers, G. S. Tyson, and E. S. Davidson, "mlcache: A flexible multilateral cache simulator", *Proceedings of MASCOTS'98*, 1998, pp. 19-26.

[19] *http://www.cs.wisc.edu/˜markhill/DineroIV/*

[20] D.C. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0", *Computer Architecture News*, 25 (3), June 1997, pp. 13-25.