Spim-Cache: A Pedagogical Tool for Teaching Cache Memories Through Code-Based Exercises

Julio Sahuquillo, Noel Tomás, Salvador Petit, and Ana Pont

***Abstract*—Cache memories represent a core topic in all computer organization and architecture courses offered at universities around the world. As a consequence, educational proposals and textbooks address important efforts to this topic. A valuable pedagogical help when studying cache memories is to perform exercises based on simple algorithms, which allow the identification of cache accesses, for instance, a program accessing the elements of an array. These exercises, referred to as code-based exercises, have a good acceptance among instructors of computer organization courses. Nevertheless, no tool (e.g., simulator) has been developed to be used in undergraduate courses working with this kind of exercises; therefore, students perform such exercises by means of the classic *paper and pencil* methodology. To fill this gap, this paper proposes a new pedagogical tool, namely Spim-cache. A laboratory example is also presented for illustrative purposes.**

***Index Terms*—Cache memories, code-based exercises, computer curricula, laboratory sessions, processor-cache simulation.**

I. Introduction

THE ever increasing gap between the memory and the microprocessor speeds has encouraged microprocessor architects for several decades to provide mechanisms in which to hide the long memory latencies. Cache memories have become the basic and ineludible mechanism employed by processors to hide these latencies and reduce the average data access time [1]. 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 on chip [2].

A large amount of research work on computer architecture has focused on cache memories and related issues. As a consequence, efficient mechanisms to manage and exploit caches have been devised, some of them successfully implemented in modern microprocessors. For instance, the advanced micro devices (AMD) Athlon [3] includes a victim cache [4]; the Itanium 2 [2] incorporates a prevalidated tag structure to provide fast accesses; and the HP 7200 implements the assist cache [5], a particular type of split data cache [6].

Concerning international curricula recommendations, the joint IEEE Computer Society and Association for Computing Machinery (ACM) Computer Engineering Task Force has identified cache memories as a core topic in the computer organization and architecture area of knowledge [7]. As a consequence, cache memories play an important role in computer organization/architecture courses. 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. On the other hand, computer architecture courses primarily cover instruction set architectures, advanced processor architectures, advanced memory issues, and multiprocessor systems.

The study of caches comprises a wide range of concepts. The basics of caches, usually covered in an initial computer organization course, involves concepts, such as mapping functions or replacement algorithms. Advanced mechanisms, some of which have been implemented in modern microprocessors (e.g., way prediction or trace cache), are typically covered in a posterior computer architecture course.

A valuable pedagogical help when studying cache memories is to perform exercises based on simple algorithms, which allow identification and analysis of cache accesses, as described in some widely referenced books [8]–[11]. These exercises, referred to as code-based exercises, have a good acceptance among instructors of computer organization courses.

Because of the increasing use of computers in the classroom, nowadays new teaching methodologies employ them as pedagogical tools. The use of simulators is highly recommended since they enable students to visualize how the modelled system operates. Concerning cache memories, simple simulators are used to manage the basics of caches. Unfortunately, and to the knowledge of the authors, none of them can be used to perform code-based exercises.

This paper proposes Spim-cache, which extends the basic Spim [12] simulator. The proposed tool has two main pedagogical strengths over existing tools aimed at studying cache memories in undergraduate courses. First, the tool permits instructors to cover a wider range of code-based exercises; therefore, the learning process is accelerated. Second, unlike current simulators showing how caches work in an isolated way, Spim-cache displays how the processor and caches interact with each other, thus reinforcing the learning process.

The remainder of this paper is organized as follows. Section II summarizes the reasons that encouraged this work. Section III describes the proposed pedagogical training tool. Section IV

Manuscript received July 20, 2006; revised March 28 2007. This work was supported in part by the Generalitat Valenciana under Grant GV06/326, by the Spanish CICYT under Grant TIN2006-15516-C04-01, and by CONSOLIDER-INGENIO 2010 under Grant CSD2006-00046.

The authors are with the Polytechnic University of Valencia, 46071 Valencia, Spain (e-mail: jsahuqui@disca.upv.es; noetoar@fiv.upv.es; spetit@disca.upv.es; apont@disca.upv.es).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TE.2007.900021

TABLE I
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

discusses the laboratory example. Section V describes the assessment study. Finally, Section VI presents some concluding remarks.

II. BACKGROUND AND PEDAGOGICAL MOTIVATION

An interesting laboratory session where real caches are used is described in [10]. In this laboratory students run a small benchmark on a given computer. The benchmark consists of 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., whether this array fits into the cache. By analyzing this time, students must deduce the cache geometry (i.e., cache size, line size, and cache association). Further details can be found in [10]. This laboratory really encourages students when they work directly on the hardware. Unfortunately, the results can only be easily analyzed when the benchmark is executed in a relatively *old* processor (e.g., the Pentium II) with a simple cache organization. In more recent microprocessors, including some kind of prefetching (e.g., the Intel Pentium 4 [13]) or victim caches (e.g., the AMD Athlon), the cache geometry can not be deduced from the benchmark execution.

The mentioned drawbacks, jointly with the flexibility of software tools, led instructors to train students by using cache simulators. Table I shows a subset of simulators that are currently used for educational or research purposes. An interesting survey of simulators can be found in [14]. Simulators can be classified in two main groups according to the way they are driven: trace-driven and execution-driven. The first ones are fed by simple traces, while the second ones, much more complex, are fed by a given benchmark.

With the aim of teaching caches in undergraduate courses, instructors generally use trace-driven simulators, e.g., DCMSim [15]. A trace is composed of a set of lines where each line represents an specific event, that is, a write or read operation in a given memory address. The simulator shows how the cache contents change when a new event occurs. Thus, students can follow how memory accesses miss or hit the cache. Some trace-driven simulators have also been used for research purposes [16], [17]. However, since simulators are fed by traces, they are not suitable to perform code-based exercises.

Advanced execution-driven simulators are primarily used for research purposes. These sophisticated tools concentrate only on cache-related issues, e.g., the mlcache simulator [16], or on the complexity of the entire microprocessor, e.g., the simple/scalar toolset [18], which models an aggressive out-of-order processor. The main advantage of these simulators is that they provide, cycle by cycle, the details about each instruction in the pipeline. Nevertheless, they do not usually offer a friendly graphic interface. This fact, jointly with the complexity of the modelled processor, makes advanced execution-driven simulators inappropriate for undergraduates.

This paper proposes a processor-cache simulator intended to be used in undergraduate courses. This simulator attempts to get the best of both kinds of simulators, that is, a friendly and easy-to-use simulator which allows visualization, cycle-by-cycle, of both the cache and processor architectural states.

III. SPIM-CACHE

Spim is a simulator that runs MIPS32 assembly language programs. This language is used in important textbooks on computer organization, such as the book by Patterson and Hennessy [8]. The user interface for Microsoft Windows platforms, also known as PCSpim, consists of a window split in four frames which display the contents of: 1) the register file; 2) the program code (assembler and machines); 3) the data memory; and 4) the log messages. These four frames, represented in lowercase letters in Fig. 1, remain unchanged in the proposed tool. As observed, registers are 32 b, while memory is organized in 32-b words and displayed in 16-B chunks.

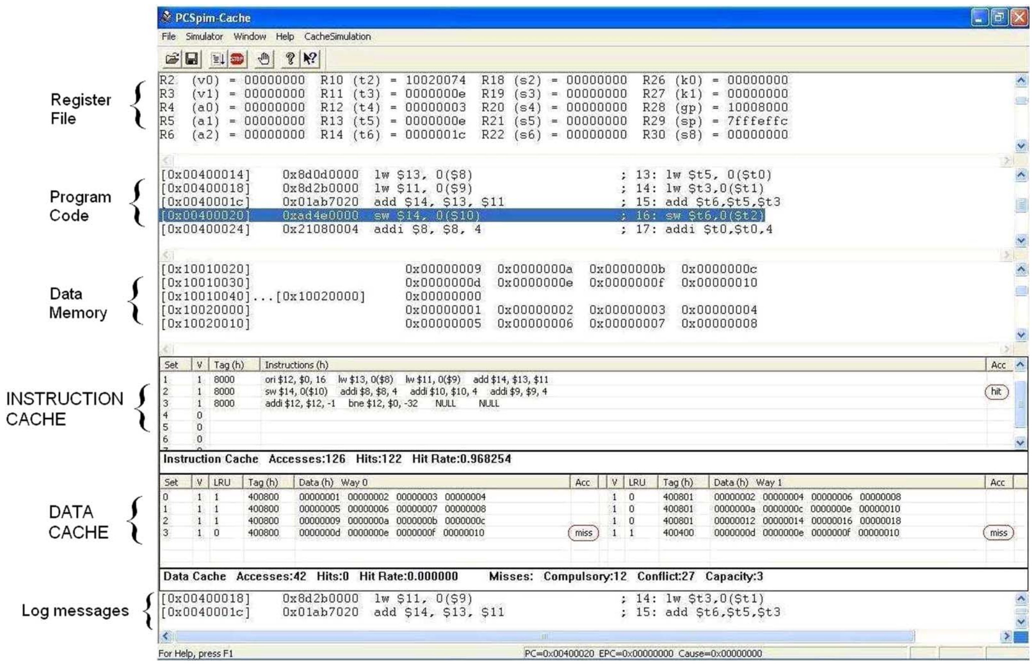
A. Configuring a Cache

Spim-cache permits the simulation of a data cache, an instruction cache, or both (i.e., Harvard Architecture). To display the contents of the cache or caches being simulated, the user interface extends the main window of PCSpim by adding one or two new frames (represented in capital letters in Fig. 1). The tool displays, step-by-step, the cache contents and statistics, such as the number of hits or misses. To start the cache simulation, users must select the *Cache Simulation* option in the *Cache Settings* dialog, which pops up after clicking on the *Settings* entry of the simulator menu (Fig. 2). Then, a dialog with different cache configurations is displayed.

The *Cache Settings* dialog, shown in Fig. 3, allows users to choose the cache configuration, that is, cache size, block size, mapping functions, etc. This dialog is accessible from the *Cache Simulation* menu option. If the *Show Rate* option is selected, some statistics are displayed in a small frame below the cache frame. For the data cache, the number of misses is broken down according to the *three Cs* categories, i.e., Compulsory, Capacity, and Conflict.

B. Running a Program

In the basic PCSpim simulator a step of execution covers the entire execution of a single instruction. The cache simulation



```
R2  (v0) = 00000000  R10 (t2) = 10020074  R18 (s2) = 00000000  R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 0000000e  R19 (s3) = 00000000  R27 (k1) = 00000000
R4  (a0) = 00000000  R12 (t4) = 00000003  R20 (s4) = 00000000  R28 (gp) = 10008000
R5  (a1) = 00000000  R13 (t5) = 0000000e  R21 (s5) = 00000000  R29 (sp) = 7fffeffc
R6  (a2) = 00000000  R14 (t6) = 0000001c  R22 (s6) = 00000000  R30 (s8) = 00000000

[0x00400014]    0x8d0d0000  lw $13, 0($8)               ; 13: lw $t5, 0($t0)
[0x00400018]    0x8d2b0000  lw $11, 0($9)               ; 14: lw $t3,0($t1)
[0x0040001c]    0x01ab7020  add $14, $13, $11           ; 15: add $t6,$t5,$t3
[0x00400020]    0xad4e0000  sw $14, 0($10)              ; 16: sw $t6,0($t2)
[0x00400024]    0x21080004  addi $8, $8, 4              ; 17: addi $t0,$t0,4

[0x10010020]                    0x00000009  0x0000000a  0x0000000b  0x0000000c
[0x10010030]                    0x0000000d  0x0000000e  0x0000000f  0x00000010
[0x10010040]...[0x10020000]     0x00000000
[0x10020000]                    0x00000001  0x00000002  0x00000003  0x00000004
[0x10020010]                    0x00000005  0x00000006  0x00000007  0x00000008
```

Set	V	Tag (h)	Instructions (h)	Acc
1	1	8000	ori $12, $0, 16 lw $13, 0($8) lw $11, 0($9) add $14, $13, $11	
2	1	8000	sw $14, 0($10) addi $8, $8, 4 addi $10, $10, 4 addi $9, $9, 4	hit
3	1	8000	addi $12, $12, -1 bne $12, $0, -32 NULL NULL	
4	0			
5	0			
6	0			

Instruction Cache Accesses:126 Hits:122 Hit Rate:0.968254

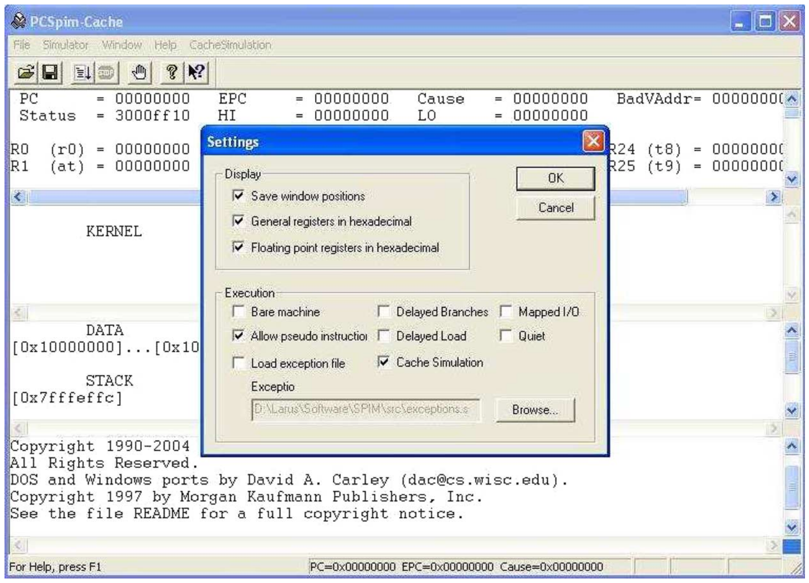
Set	V	LRU	Tag (h)	Data (h) Way 0	Acc	V	LRU	Tag (h)	Data (h) Way 1	Acc
0	1	1	400800	00000001 00000002 00000003 00000004		1	0	400801	00000002 00000004 00000006 00000008	
1	1	1	400800	00000005 00000006 00000007 00000008		1	0	400801	0000000a 0000000c 0000000e 00000010	
2	1	1	400800	00000009 0000000a 0000000b 0000000c		1	0	400801	00000012 00000014 00000016 00000018	
3	1	0	400800	0000000d 0000000e 0000000f 00000010	miss	1	1	400400	0000000d 0000000e 0000000f 00000010	miss

Data Cache Accesses:42 Hits:0 Hit Rate:0.000000 Misses: Compulsory:12 Conflict:27 Capacity:3

```
[0x00400018]    0x8d2b0000  lw $11, 0($9)               ; 14: lw $t3,0($t1)
[0x0040001c]    0x01ab7020  add $14, $13, $11           ; 15: add $t6,$t5,$t3
```

For Help, press F1   PC=0x00400020 EPC=0x00000000 Cause=0x00000000

Fig. 1. First step of execution of the store instruction.



```
PC     = 00000000   EPC   = 00000000   Cause   = 00000000   BadVAddr= 0000000(
Status = 3000ff10   HI    = 00000000   LO      = 00000000

R0  (r0) = 00000000                                    R24 (t8) = 0000000(
R1  (at) = 00000000                                    R25 (t9) = 0000000(

       KERNEL

       DATA
[0x10000000]...[0x10

       STACK
[0x7fffeffc]

Copyright 1990-2004
All Rights Reserved.
DOS and Windows ports by David A. Carley (dac@cs.wisc.edu).
Copyright 1997 by Morgan Kaufmann Publishers, Inc.
See the file README for a full copyright notice.
```

Settings

Display
- [✓] Save window positions
- [✓] General registers in hexadecimal
- [✓] Floating point registers in hexadecimal

Execution
- [ ] Bare machine
- [ ] Delayed Branches
- [ ] Mapped I/O
- [✓] Allow pseudo instructio
- [ ] Delayed Load
- [ ] Quiet
- [ ] Load exception file
- [✓] Cache Simulation

Exceptio: D:\Larus\Software\SPIM\src\exceptions.s

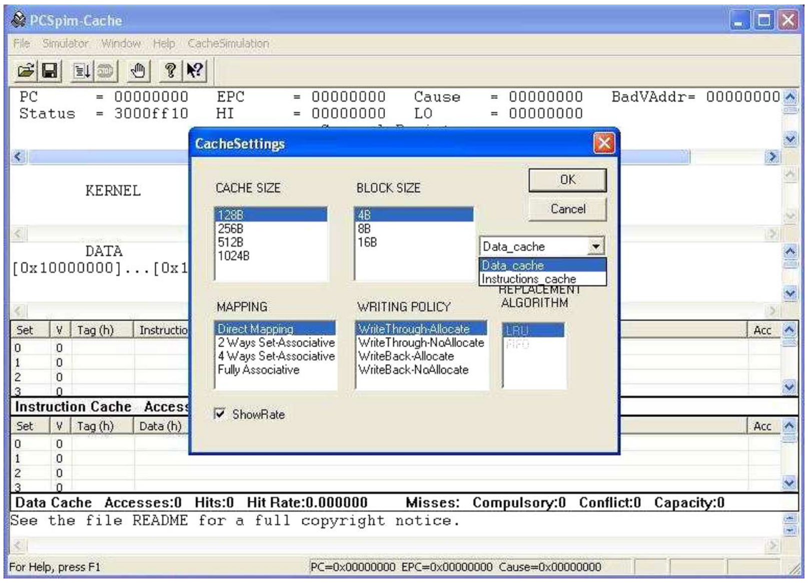
For Help, press F1   PC=0x00000000 EPC=0x00000000 Cause=0x00000000

Fig. 2. Cache simulation option.

extension modifies the semantic of memory reference instructions, i.e., loads and stores. A memory reference instruction that hits into the cache takes only one step (as a normal instruction) to execute. Nevertheless, because of pedagogical reasons, load and store misses are handled with a different number of steps.

A load miss is handled in three steps: 1) detecting and marking the miss in the corresponding set (all lines in the set are marked); 2) fetching the block from main memory; and 3) loading the requested data into the corresponding register. A store miss is handled in two or three steps depending on the selected write miss policy (allocate or no-allocate). With the no-allocate policy the steps are: 1) detecting and marking the miss; and 2) storing the content of the register in the main memory. With the allocate policy the steps are: 1) detecting and marking the miss in the corresponding set; 2) fetching the block from main memory; and 3) storing 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 is the execution of the instruction.

A working example illustrates how Spim-cache handles memory reference instructions. Assume a 128-B direct-mapped instruction cache with a 16-B line size, and a 128-B two-way set associative data cache with a 16-B line size, least recently used (LRU) replacement algorithm, and write-through write-allocate policies. These cache configurations and the program code can be seen in Fig. 1. Consider the execution of the store instruction



```
PC      = 00000000   EPC    = 00000000   Cause   = 00000000   BadVAddr= 00000000
Status  = 3000ff10   HI     = 00000000   LO      = 00000000
[0x10000000]...[0x1
```

CACHE SIZE: [✓ 128B] 256B 512B 1024B

BLOCK SIZE: [✓ 4B] 8B 16B

Data_cache: [✓ Data_cache] Instructions_cache

MAPPING: [✓ Direct Mapping] 2 Ways Set-Associative 4 Ways Set-Associative Fully Associative

WRITING POLICY: [✓ WriteThrough-Allocate] WriteThrough-NoAllocate WriteBack-Allocate WriteBack-NoAllocate

REPLACEMENT ALGORITHM: [✓ LRU] FIFO

[✓ ShowRate]

Set	V	Tag (h)	Instructio	Acc
0	0			
1	0			
2	0			
3	0			

Instruction Cache Access

Set	V	Tag (h)	Data (h)	Acc
0	0			
1	0			
2	0			
3	0			

Data Cache Accesses:0 Hits:0 Hit Rate:0.000000 Misses: Compulsory:0 Conflict:0 Capacity:0

```
See the file README for a full copyright notice.
```

For Help, press F1 PC=0x00000000 EPC=0x00000000 Cause=0x00000000

Fig. 3. Cache settings dialog box.

(the one shadowed in the code frame). The execution of this instruction involves two cache accesses, one to the instruction cache and the other to the data cache.

Regarding the access to the instruction cache, the store instruction is located at the address 00400020_{16}, which has 25 b for tag, 3 b for index, and 4 b for byte offset. Using this field layout, the address has 0000000001000000000000000 as tag (i.e., 8000_{16} if leading zeros are neglected), 010 as index (set 2), and 0 as byte offset. Since the tag of the line in set 2 matches the obtained tag, this access results in a hit.

With respect to the access to the data cache, the target memory address is 10020074_{16}, which has 26 b for tag, 2 b for index, and 4 b for byte offset. Using this field layout, the address has 00010000000000100000000001 as tag (i.e., 400801_{16} if leading zeros are neglected), 11 as index (set 3), and 4 as byte offset. Since no tag in set 3 matches the obtained tag, the data cache access results in a miss. Since the two ways of the set contain a valid line, the LRU algorithm selects the line placed in way 1 to be replaced, since that line has the higher LRU counter value. Therefore, in the second step the missing block is fetched from the main memory. Finally, in the third step the content of register \$14 is written to the data cache and also to main memory because of the write-through policy.

IV. Achieving the Learning Outcomes: A Laboratory Session Example

This section presents a laboratory example composed of a set of exercises that follows the pedagogical guidance of the current teaching books. This laboratory session was performed by the students of the computer organization course at the Polytechnic University of Valencia, Valencia, Spain, during the 2005-2006 academic year. The laboratory requires the use of MIPS R2000 assembly programs. Because of time restrictions, students are provided with small code fragments for guidance purposes.

The main goal at this level of training is to work on the basics of caches. Because of pedagogical reasons, the difficulty level of the exercises increases gradually so that students can reach, step-by-step, the learning outcomes. For the sake of clarity, all the exercises assume that variables containing nonarray elements are not located in the cache but in the register file.

A. Mapping Functions

In this exercise students work on how memory blocks map to cache. Students must deduce how the cache controller interprets the memory address by identifying which bits correspond to the tag, the set, and the offset fields. Textbooks use either the term set [11] or the term index [8] with the same meaning, that is, this field refers to the address bits used as an index to access the corresponding set.

To perform this exercise, students are supplied with the algorithm and the assembly code shown in Fig. 4. Two eight-word arrays (A and B) are defined and allocated in the main memory. The code implements the function $\mathrm{sum} = \sum(A[i])$ and, therefore, reads and adds the elements of A.

First, students work with mapping functions for a given cache geometry (256-B size, 16-B line size, four-way set associative), using *paper and pencil*. They must obtain the cache sets where the elements of the array will map and the corresponding tag values. Fig. 5 shows the address fields. The low-order 4 b of a memory address represent the offset, since the line size is $16 = 2^4$ B. The next low-order 2 b provide the set identifier as $2^2 = 256/(16 \times 4)$. The remaining 26 b $(32 - (4 + 2))$ provide the tag field. On the other hand, since lines are 16-B large, two lines are required to store an eight-word vector. In this cache, the four low-order elements of vector A are placed in the first way of set 0, and the next four elements in the first way of set 1. Both lines are tagged with a 400012_{16} value. Vector B also maps to the same sets and is stored in the following ways, with a 400033_{16} tag value (Fig. 6). After obtaining these values, students must extend the program in Fig. 4 to implement $\mathrm{sum} = \sum(A[i] + B[i])$. In this way, when the program is executed, the elements of both arrays will be written into the cache. Once the program

		.data 0x10000480	
	Array_A:	.word 1,1,1,1,2,2,2,2	
		.data 0x10000CC0	
	Array_B:	.word 3,3,3,3,4,4,4,4	
		.text	
sum=0		.globl __start	
for (i=0;i<100;i++)	__start:	la $2, Array_A	
sum=sum+A[i]		li $6,0	# sum = 0
		li $4, 8	# number of elements
	loop:	lw $5, 0($2)	
		add $6,$6,$5	# sum = sum + Array_A[i]
		addi $2,$2,4	
		addi $4,$4,-1	
		bgt $4,$0, loop	

Fig. 4. Algorithm and corresponding code to study mapping functions.

Block address		offset
tag	index/set	
31 30 ··· 7 6	5 4	3 2 1 0

Fig. 5. Address fields and corresponding bits.

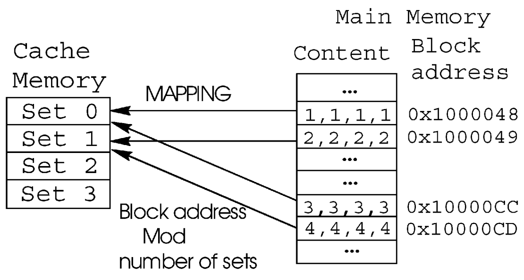


Fig. 6. Algorithm and assembler program example.

		.data 0x10000480	
	Array_A:	.word 1,1,1,1,2,2,2,2	
		.data 0x10000CC0	
	Array_B:	.word 3,3,3,3,4,4,4,4	
		.text	
		.globl __start	
sum=0	__start:	li $8, N	# number of iterations
for (j=0;j<N;j++)	ext_loop:	la $2, Array_A	
for (i=0;i<100;i++)		la $3, Array_B	
sum=sum+A[i]+B[i]		li $6,0	# sum = 0
		li $4, 8	# number of elements
	loop:	lw $5, 0($2)	
		lw $7, 0($3)	
		add $6,$6,$5	# sum=sum+arrayA[i]
		add $6,$6,$7	# sum=sum+arrayB[i]
		addi $2,$2,4	
		addi $3,$3,4	
		addi $4,$4,-1	
		bgt $4,$0, loop	
		addi $8,$8,-1	
		bgt $8, $0, ext_loop	
		.end	

Fig. 7. Algorithm and corresponding code to study temporal locality.

is completed, Spim-cache is used to check the results obtained using *paper and pencil*.

B. Temporal and Spatial Locality

Students work on spatial and temporal localities by analyzing how they impact on the cache performance (i.e., the hit rate). With the aim of analyzing spatial locality, students are asked to run the previously developed program using three different line sizes (16, 8, and 4 B). As in the previous exercise, a 256-B size, four-way set associative cache is considered. For each simulation, students must obtain the *hit rate* and classify the misses. They are encouraged to analyze the obtained results. In this way, they realize that the larger the line size is, the higher the spatial locality is.

All elements in each array are accessed only once; therefore, the program performs 16 memory accesses. If lines are 16-B large, four lines are required to store the 16 elements of an array. The access to the first element in each line results in a miss (4 compulsory misses), while the remaining 12 accesses hit into the cache and achieve a 0.75 hit rate. Analogously, the hit rate is 0.5 and 0 for 8-B and 4-B line sizes, respectively.

With the aim of working on temporal locality, students must implement an external loop in the program. Fig. 7 shows the algorithm and a possible code implementing it. Experiments must be performed for a 256-B size, 16-B line size, and four-way set associative cache. In this exercise aimed at studying temporal locality, students vary the number of iterations ($N = 1$, 5, 10, and 100) of the external loop. As above, for each run they must obtain the hit rate and classify the misses. They are encouraged to interpret the results and draw conclusions about the impact of the temporal locality on the cache performance. The hit rate for 5, 10, and 100 iterations is 0.95, 0.975, and 0.9975, respectively.

C. Replacement Algorithms

To study replacement algorithms, students are supplied with the assembly program in Fig. 8. The cache is assumed to be 256-B size, 16-B line, and direct mapped. The vector size ($512\text{B} = 128 \times 4\text{B}$) is twice as large as the cache storage capacity. Students are asked to identify the content of the set 0 after the program execution.

Only half the vector fits into the cache since the vector is twice as large as the cache size. Elements are accessed in sequential order; thus, when the program finishes its execution, the cache stores the second half of the array (elements ranging from 64 to 127).

An alternative way to study replacement algorithms is to use several small vectors mapping to the same set. For instance, instructors may ask students to provide three vectors mapping to a given set in a two-way set associative cache. With these kinds

```
sum=0
for (j=0;j<N;j++)
    for (i=0;i<128;i=i+stride)
        sum=sum+A[i]
```

```
            .data 0x10000000
Array_A:    .word 0,1,2,3,4,5,6,7,8,9,10,11, ··· ,125,126,127
            .text
            .globl __start
__start:    li $8, 1            # $8 = iterations of external loop
            li $3, 1            # $3 = stride in elements
            li $6, 0            # $6 = sum
            sll $9,$3,2         # $3 = stride in bytes= $3 x 4
ext_loop:   li $5, 128          # $5 = # elements of Array_A
            li $4, 0            # $4 = index
int_loop:   lw $7,Array_A($4)
            add $6,$6,$7
            add $4,$4,$9        # address of next element
            sub $5, $5,$3
            bgt $5,$0,int_loop
            addi $8,$8,-1
            bgt $8,$0,ext_loop
            .end
```

Fig. 8. Algorithm and corresponding code provided to work on replacement algorithms and strides.

TABLE II
SENTENCES AND RESULTS FROM THE QUESTIONNAIRE

Question	Result Distribution (score)					Average	Standard Deviation
	0	25	50	75	100		
1. The simulator helps to understand mapping functions	1%	3%	20%	62%	14%	71	18
2. The simulator helps to understand replacement algorithms	1%	10%	36%	42%	12%	64	22
3. The simulator helps to understand data localities	1%	4%	12%	61%	22%	74	19
4. The simulator solves problems proposed in the classroom	0%	4%	21%	48%	27%	74	20
5. The laboratory work reinforces the understanding of theoretical subjects	0%	7%	8%	49%	36%	74	21
6. The simulator's interface is easy and intuitive	0%	4%	15%	51%	30%	77	19
7. In general, I found the laboratory work interesting	0%	5%	17%	56%	21%	73	19

of exercises, students can follow how the LRU counter values change because of the replacement algorithm.

D. Accessing at Different Strides: Emphasizing Temporal and Spatial Localities

Accessing the elements of an array at different strides helps to acquire a solid understanding on cache memories. At this point, students are asked to run the program shown in Fig. 8 using different stride values (i.e., 1, 2, 4, 8, 16, 32, and 64) to access the elements of the array, and repeating the external loop 100 times. Students must change the content of register $3 with the desired stride value, and modify the content of register $8 with the number of desired iterations of the external loop. Experiments must be run for a 256-B, 16-B line, four-way set associative cache. For each run students must obtain the hit rate and classify the misses. Then, they must draw conclusions about how different stride values and number of iterations of the external loop (i.e., varying spatial and temporal localities) impact on the hit rate.

V. TEACHING CONTEXT AND SPIM-CACHE ASSESSMENT

The computer organization subjects taught at the Polytechnic University of Valencia are organized in two annual courses [19], namely, Computer Fundamentals (first year) and Computer Organization (second year). The Computer Organization course is attended by about three hundred students. This course covers the following topics: the processor, the memory unit (caches, main memories, virtual memory, etc.), the input/output unit, and the arithmetic-logic unit [19]. Each unit covers about 25% of the time of the overall course. Laboratory sessions are weighted at about 20% of the score of the course mark of the student. The remaining mark is obtained from a written examination containing both theoretical questions and practical exercises.

An initial version of Spim-cache has been used in the Computer Organization course to perform the laboratory session described in Section IV. When students used the tool, the learning effectiveness increased compared with the three previous academic courses, since students increased their grades (i.e., those related to cache issues) by about 21%, 26%, and 23%, respectively. During this period of time the remaining aspects of the teaching methodology related to cache memories were not modified. In addition, instructors noticed that they spent less time solving cache-related questions during their office hours. This fact is a result of the wide acceptance of the tool among the students, who extensively used Spim-cache to clarify doubts concerning theoretical concepts and problems studied in the classroom.

Laboratories are organized to be 2 h long. To provide self-assessment of the proposed tool, authors prepared a short questionnaire, which contained seven questions (Table II) related to the tool and the laboratory. The questionnaire was voluntarily completed by about 90% of the students attending the laboratory sessions. Responses were on a 100 point scale, where 100 and 0 indicated strong acceptance or rejection, respectively. Table II shows the results.

The first three questions evaluated how Spim-cache provided support for the study of the basics of caches. Question 4 evaluated how the tool can be used in a self-training way to solve questions previously studied in the classroom. Question 5 dealt with the usefulness of the laboratory for reinforcing theoretical concepts. Question 6 evaluated the goodness of the user interface. Finally, question 7 provided information on how interesting the laboratory was for students. Results are encouraging, since most of the responses had a score of 75.

VI. Conclusion

This paper has presented Spim-cache as a pedagogical tool for dealing with code-based exercises in undergraduate courses. The tool facilitates the understanding of theoretical questions, thus allowing students to feel more confident when studying cache-related issues.

The proposed tool has two main pedagogical strengths. First, the tool permits instructors to cover a wider range of code-based exercises; therefore, the learning process is accelerated. Second, unlike current simulators showing how caches work in an isolated way, Spim-cache displays how the processor and caches interact; therefore, the tool reinforces the learning process. In this context, extra exercises (e.g., write policy-related examples) can be offered to those students wishing to learn more about this subject.

The final aim of this paper is to share the authors' experience with other instructors. For this purpose, a laboratory example has been described. The current version of Spim-cache is available at http://www.disca.upv.es/spetit/spim.htm.

Acknowledgment

The authors would like to thank the anonymous reviewers, instructors, and students of the Computer Organization course for their useful comments to improve the quality of the proposed tool.

References

[1] M. V. Wilkes, "Slave memories and dynamic storage allocation," *IEEE Trans. Electron. Comput.*, vol. EC-14, no. 2, pp. 270–271, Apr. 1965.

[2] C. McNairy and D. Soltis, "Itanium 2 processor microarchitecture," *IEEE Micro*, vol. 23, no. 2, pp. 44–55, Mar.–Apr. 2003.

[3] J. Huynh, "The AMD Athlon MP Processor With 512 KB 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," in *Proc. 17th Annu. Int. Symp. Computer Architecture*, Seattle, WA, Jun. 1990, pp. 364–373.

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

[6] J. Sahuquillo and A. Pont, "Splitting the data cache: A survey," *IEEE Concurrency*, vol. 8, no. 3, pp. 30–35, Jul.-Sep. 2000.

[7] Computer Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering, ACM/IEEE-CS Joint Curriculum Task Force Rep.

[8] D. A. Patterson and J. L. Hennessy, *Computer Organization: The Hardware/Software Interface*, 3rd ed. San Mateo, CA: Morgan Kaufmann, 2005.

[9] W. Stallings, *Computer Organization and Arquitecture: Designing for Performance*, 6th ed. New York: McGraw-Hill, 2002.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. San Mateo, CA: Morgan Kaufmann, 2007.

[11] C. Hamacher, Z. Vranesic, and S. Zaky, *Computer Organization*, 5th ed. New York: McGraw-Hill, 2002.

[12] J. Larus, "SPIM S20: A MIPS R2000 Simulator," Computer Sciences Dept., Univ. of Wisconsin-Madison, Tech. Rep. TR966, Sep. 1990.

[13] 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 Technol. J.*, vol. Q1, 2001.

[14] W. Yurcik, G. S. Wolffe, and M. A. Holiday, "A survey of simulators used in computer organization/architecture courses," in *Proc. Summer Computer Simulation Conf.*, Orlando, FL, Jul. 2001, pp. 524–529.

[15] E. Cordeiro, I. Stefani, T. Soares, and C. Martins, "DCMSim: Didactic cache memory simulator," in *Proc. 33rd Frontiers in Education Conf.*, Nov. 2003, vol. 2, pp. F1C-14–F1C-19.

[16] E. S. Tam, J. A. Rivers, G. S. Tyson, and E. S. Davidson, "mlcache: A flexible multilateral cache simulator," in *Proc. 6th Int. Symp. Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Montreal, QC, Canada, Jul. 1998, pp. 19–26.

[17] J. Edler and M. Hill, Dinero IV Trace-Driven Uniprocessor Cache Simulator [Online]. Available: http://www.cs.wisc.edu/~markhill/DineroIV

[18] D. C. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," *Comput. Architect. News*, vol. 25, no. 3, pp. 13–25, Jun. 1997.

[19] J. Real, J. Sahuquillo, A. Pont, L. Lemus, and A. Robles, "A lab course of computer organization in the Technical University of Valencia," in *Proc. Workshop Computer Architecture Education*, Anchorage, AK, May 2002, pp. 119–125.

Julio Sahuquillo received the B.S., M.S., and Ph.D. degrees in computer engineering from the Polytechnic University of Valencia (UPV), Valencia, Spain.

Since 2002, he has been an Associate Professor in the Computer Engineering Department at the UPV. His current research topics include multithreaded and multicore processors, and memory hierarchy design. His teaching interests include computer organization and computer architecture.

Noel Tomás received the B.S. and M.S. degrees in computer engineering from the Polytechnic University of Valencia (UPV), Valencia, Spain, in 2005 and 2007, respectively.

Currently, he is working towards the Ph.D. degree in the Department of Computer Engineering at the UPV. His Ph.D. focuses on microprocessor architecture.

Salvador Petit received the Ph.D. degree in computer engineering from the Polytechnic University of Valencia (UPV), Valencia, Spain.

Currently, he is an Associate Professor in the Computer Engineering Department at the UPV. His research topics include multithreaded and multicore processors, and memory hierarchy design. His teaching interests include computer organization and computer architecture.

Ana Pont received the B.S., M.S., and Ph.D. degrees in computer engineering from the Polytechnic University of Valencia (UPV), Valencia, Spain.

Since 2003, she has been a Full Professor in the Computer Engineering Department at the UPV. Her research interests include Web and Internet architecture, proxy caching techniques, computer data networks, communication networks, multiprocessor architecture, memory hierarchy design, and performance evaluation. Her teaching interests include computer architecture and organization, networks, and Web infrastructure.