Understanding Cache Hierarchy Interactions with a Program-Driven Simulator *

Leticia Pascual, Alejandro Torrentí
Instituto Tecnológico de Informática
Technical University of Valencia
{lepasmi,altorro}@upvnet.upv.es

Julio Sahuquillo, José Flich
Computer Engineering Department
Technical School of Applied Computer Science
Technical University of Valencia
{jsahuqui, jflich}@disca.upv.es

Abstract

The increasing importance of the cache hierarchy in almost all digital systems governed by a microprocessor is demanding an appropriate set of tools to teach the interactions between caches. Students in computer organization/architecture courses face the problem of understanding how caches interact in a hierarchical organization. As caches grow in size and complexity, designing a tool to explore the different parameter interactions becomes challenging. Also, focusing only on the significant information (accessed lines and events) at different cache levels is required.

In this paper we present SpimVista, a tool developed on top of PC-Spim simulator, that simulates any MIPS code on a system with different cache hierarchy configurations. Each cache memory can be configured independently.

SpimVista is aimed at being used by undergraduate students, and the graphical interface has been designed addressed to ease the learning process. In particular, the tool offers an intuitive and easy interface that allows the student to focus on particular and interesting events as the program instructions are executed step-by-step. Also, students can perform interesting exercises where both the code and the impact of cache organization parameters on performance are analyzed globally.

Categories and Subject Descriptors
B.3.2 [Memory Structures]: Design Styles - cache memories.

Keywords
Cache Organization, Multi-level caches, Write policies.

1. Introduction

Memory speed in current systems is two orders of magnitude lower than the microprocessor core speed. This fact has motivated computer architects to design mechanisms to hide this latency. The classical mechanism implemented from early in computers is the cache memory [1]. Because of the high latency, current microprocessors implement two or even three levels of caches. This fact is more noticeable in multicore processors where memory bandwidth requirements are a major design concern.

Nowadays, cache memories grow in importance as they are not only implemented in high performance systems but also in all those electronic devices governed by a processor. For instance, mobile devices, like PDAs or phone cells, which represent an important segment of the market, or embedded systems, like GPS systems. In other words, the use of caches is almost as spread as the use of microprocessors. Therefore, understanding how caches work is essential to provide a sound understanding about how processors work.

Caches are a core topic in those disciplines that among others, including computer organization and/or computer architecture courses, such as computer engineering, electrical engineering or telecommunication engineering. Computer organization and architecture courses usually include laboratory sessions to reinforce the learning process. Due to the impact of caches on final system performance and the wide variety of concepts, computer organization/architecture courses commonly offer at least one laboratory session dealing with caches.

In this context, many simulators [9] are currently being used by instructors at different universities. Many of these simulators are trace-driven (e.g., Dinero IV [6]) while only a small subset are program or execution-driven (e.g., SimpleScalar Tool Set [7]). Trace-driven simulators are easier to understand while execution-driven are complex but more complete, as they provide a view of the system working as a whole; i.e., they show not only how caches work but how also they interact with the processor. Because of

this fact, execution-driven simulators are extensively used for research purposes, nevertheless, instructors, because of the complexity involving these simulators, usually do not use them for teaching purposes.

Execution-driven simulators handle much more information than trace-driven simulators, e.g., the register file contents, or the main memory contents. Thus, instructors can conveniently use this information, establishing relationships with other computer components. As a consequence, an important challenge for instructors is to simplify execution-driven simulators in order to be used by undergraduates.

An important pedagogical feature of these simulators is that they run code-based exercises, that is, to write a simple program and to study cache behavior when running the corresponding code. An early attempt, SpimCache [8], has been published aimed at covering the existing gap in current simulators. Unfortunately, such tentative represents only one cache level, thus, it is not suitable when studying the cache hierarchy, which is and important and unavoidable topic when referring to current microprocessors. For instance, the interactions of using different policies or line sizes at different cache levels can not be analyzed in SpimCache [8].

Therefore, from our point of view, there is a need for a tool that simplifies the understanding of the different interactions between the different levels of the cache hierarchy. In particular, it is fundamental to appreciate how the information is managed when using different cache configurations. Each cache memory at a given level is independently configured and may behave differently under specific events (e.g. read or write misses). On the other hand, as the cache memories grows in size, the amount of information to be depicted becomes excessive to fit a single screen. All these shortcomings lead to the challenging problem of designing a tool that facilitates the understanding of the interactions between the cache memories while, at the same time, keeping the representation and the use simple. In this paper we take on such challenge with the proposed SpimVista tool.

SpimVista is a program-driven simulator that extends PC-Spim [5] to represent a cache hierarchy, therefore, it allows to visualize how two cache levels interact with each other. In addition, the simulator allows to work with large cache sizes. To this end, we improved the graphical representation by depicting not all the cache contents but only those that are subject of the study; e.g., just a few cache lines.

The remainder of this paper is organized as follows. Section 2 discusses the teaching context of the tool. Section 3 highlights the main SpimVista features. Section 4 describes the proposed pedagogical training tool. Section 5 discusses a simple laboratory program example. Finally, Section 6 presents some concluding remarks and future work.

2 Teaching Context

SpimVista has been developed at the Technical University of Valencia (UPV). It has been successfully applied to the basic Computer Organization course offered in the second year of the Computer Engineering career (five years term career). At this point in the career, the student has received the first-year Computer Fundamentals course, where they have studied the MIPS assembly language. In addition, the PC-Spim simulator has been used to write and test simple programs. As the SpimVista is totally integrated in the PC-Spim simulator, there is no need to learn a new tool or platform.

The Computer Organization course consists of four major blocks. The first block focuses the basic memory organizations of the computer, ranging from the chip level to the memory hierarchy (including different cache levels). In the second block, the student learns the basic I/O systems. The remaining two blocks cover ALU operators and a general pipelining overview. The whole course uses the MIPS R2000 architecture as primary microprocessor model, thus, the MIPS32 instruction set architecture is deeply studied. The MIPS architecture is a well-known architecture and their features, like full functionality and easy understanding are well suited for assembler and computer organization learning purposes. In addition, solid reference books [2, 3] cover in a extensible way the MIPS architecture. The MIPS approach is used both in theoretical classrooms and laboratory sessions. In this context, the PC-Spim simulator is used in three of the four blocks of the Computer Organization course. For this purpose, PC-Spim has been enriched with new features concerning memory hierarchy (the focus of this paper) and I/O.

Regarding memory hierarchy concepts, the course spends half a semester for introducing the main concepts and working with the cache hierarchy. The implications of using different cache organizations on performance are studied while changing the distinct policies used to handle write misses, read misses, and line replacements. Finally, the integration of different cache levels is introduced and analyzed.

3 Main SpimVista Features

SpimVista aims to improve the learning methodology for cache memories and cache hierarchies. In this sense, the tool has three key features discussed below.

First, SpimVista allows a full configuration of relevant characteristics for each cache in the hierarchy, for instance, the number of ways, replacement algorithm, write-hit policy, and write-miss policy. This framework provides students with no restrictions when studying the effect of combining different policies and cache sizes. The tool also re-

ports performance statistics for each cache level, which allows a later analysis and conclusions of the best set of cache parameters for a given workload. These parameters can be also changed in existing simulators, although, usually some policies, like the write-miss policy are not configurable.

Second, the inclusion of a second cache level to the simulator engine differences SpimVista from other similar tools [8] by enabling the simulation of a two-level cache hierarchy. This feature increases the number of possible configurations, approaching the student to manufactured cache hierarchies without losing comprehensiveness. With this characteristic students are able to analyze the performance (e.g., the hit ratio) of the L1 and L2 caches, which helps to understand the goals of the first (small and fast) and second (large and slow) level caches. In addition, students realize that caches work independently and also understand why L2 caches usually implement a write-back and write-allocate policy. In other words, the inclusion of the second level cache is founded and useful providing a worth help in the learning process.

Third, and because of SpimVista is addressed to be used by undergraduates, it is important to work with graphic interfaces, paying special attention to the visualization of cache control information and block content. The tool has been carefully designed to simplify the user interface by using different colors and highlighting techniques. Cache contents are drawn in a matrix, where rows represent the cache sets, and columns the ways in the cache. Colors differentiate the type of access so that users have a faster recognition and identification of the instruction being executed or a given event. As an example, a read miss in L1 that hits L2 will mark in red the block at L1 (meaning a miss) and in green color the block in L2 (meaning a hit). Additionally, users can visualize the data contained in the blocks by moving the mouse over the desired blocks. This way of representation provides a faster and better understanding of cache operation, a natural navigation over the system and lighten information and data visualization.

Finally, SpimVista executes any MIPS assembler code, thus allowing to design interesting laboratory exercises focusing on the cache hierarchy. As an example, when accessing a given array, the size of the data set may be varied to study the impact on performance of the L2 cache size. Also, accessing to the elements of 2D matrix, either by rows or by columns, may lead to different performance (i.e., hit ratio).

4 SpimVista

In this Section we describe the SpimVista tool. First, we detail the configuration procedure and later we describe how to run a program[1].

[1]The tool and an example are available for downloading at http://www.disca.upv.es/jflich/spimvista.zip

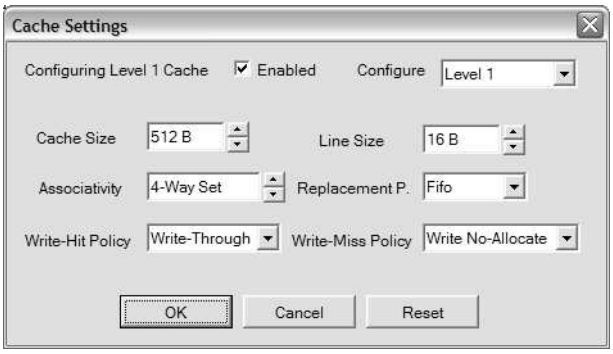

Figure 1. Configuring the cache hierarchy.

Table 1. Cache parameters.

Cache size	512B, 1KB, 2KB, ..., 64KB
Line size	4B, 8B, 16B, ..., 64B
Replacement algorithm	LRU, FIFO, random
Number of ways	1, 2, ..., to fully-set associative
Write-hit policy	Write-through and write-back
Write-miss policy	Write allocate and write no-allocate

4.1 Configuring the memory hierarchy

SpimVista is embedded in the PC-Spim simulator. Thus, all the configuration process has been included in a new menu option of the PC-Spim simulator. The cache simulation process can be enabled or disabled. When enabled, the cache hierarchy can be configured and a new window is shown with all the details of the caches for the running application. Figure 1 shows the configuration dialog box. SpimVista allows a flexible configuration of the parameters. Table 1 summarizes the configurable parameters as well as the range of supported values. Notice that the ranges are quite wide, which allows us to choose between small *toy* caches, aimed at being mainly used for pedagogical purposes, or larger caches as those implemented in commercial processors. Students may choose between simulate a single cache or a cache hierarchy with two caches each one with an independent organization.

Main memory management remain as it was implemented in the original PC-Spim simulator. As for future work we plan to extend the tool to support pagination and segmentation.

4.2 Running a program

To run a program for a given cache hierarchy, users must load the desired program following the PC-Spim usual way. Also, as done by PC-Spim, users may choose between run-

ning the program step-by-step or running the whole program at once.

Both execution types have their advantages and their choice depends on the goals of the study. If the goal is to learn how caches interact between them (e.g., a write miss), then, the program should be run in a step-by-step way. On the other hand, if we are interested on global performance statistics, like the total hit ratio or miss ratio for store instructions, then, the entire program could be executed at once.

When working with undergraduates the visual approach and the step-by-step execution grow in importance. SpimVista design efforts have concentrated on improving this feature. For instance, if a load instruction hits the cache, the corresponding line, set, and word, are drawn in green color. Otherwise, a red color indicates a cache miss. In addition, instruction information (i.e., type and address) and its results (i.e., hit or miss) are displayed at the top of the cache rectangle; if the cache is not accessed in a cycle, then a *No Access* label is displayed. This visual approach helps undergraduates to assimilate and understand cache hierarchy related events. Figure 2 shows an example.

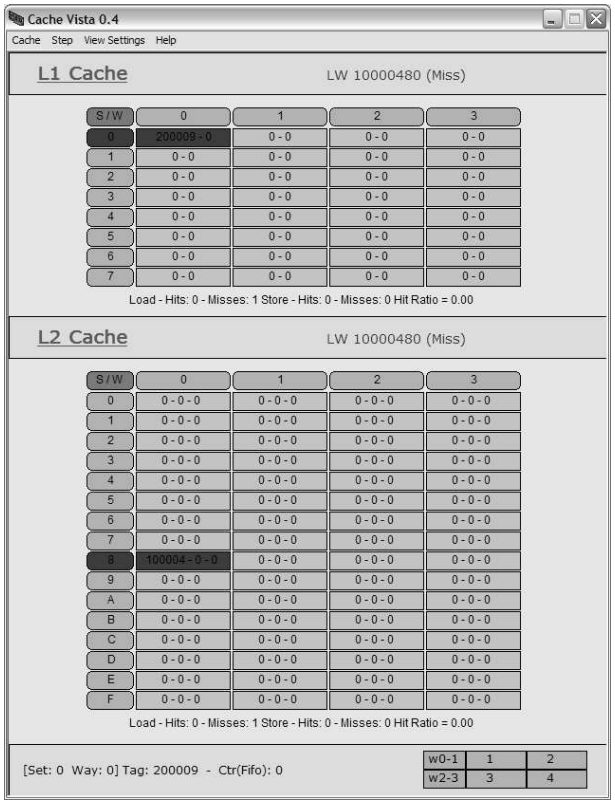

Figure 2. Example of information displayed by SpimVista.

An important feature, to the best of our knowledge not implemented in existing simulators, is the capability of the tool to roll back the execution. For instance, when studying the LRU replacement algorithm, a common situation is that students frequently do not realize which block has been replaced. In that situation, SpimVista allows to roll back the execution one cycle in order to identify the replaced block.

The tool provides detailed information to analyze cache behavior as shown in Figure 2. Among this information, SpimVista shows the line tag, the data values contained in the cache line, and the write policies. This information is displayed when moving the mouse through the desired cache lines. At the same time, a frame in the bottom section shows the state of the current cache line. Separated statistics for load and store instructions are also shown for each cache organization.

When simulating large size caches the visual benefits are lost. To avoid this problem, simulators limit the cache size to small values, thus simulating only small or *toy* caches. However, we believe that working with large caches may encourage students as the caches they use are representative of commercial implementations. To deal with the graphical problem, SpimVista hides the sets of the caches whose lines are not used (i.e., not accessed by the application being executed). For instance, Figure 3 shows an example where the sets ranging from 2 to 7 are not shown as the lines of these sets have not yet been accessed. Notice that a line that is never accessed can be omitted because it is not significant at all for the study. Once a cache line is accessed, SpimVista will show the corresponding set. Also, SpimVista allows to toggle between one or two caches in the window. Thus, the undergraduate has the possibility to focus only in one cache level or the entire cache system at any time.

As a summary, this visual environment with an intuitive and colorful interface, and the possibility to roll back one cycle ensuring an easy identification of the cache events, especially in a step-by-step execution, provides a full-configurable tool that helps students to reinforce the theoretical concepts studied at classroom.

5 A Laboratory Program Example

As mentioned above, SpimVista has been already used in laboratory sessions at the Technical University of Valencia. This Section describes the process to configure and run an assembly program included in those laboratory sessions. Once enabled the cache simulation, the student selects the following properties of the cache system: the L1 cache is set with the write-through and write not-allocate policies and the L2 cache with write-back and write-allocate policies. L1 is a 512-byte 4-way cache whereas L2 is a 512-byte 8-way cache. In both cases, line size is 16-byte wide and the LRU replacement policy is used. Notice that the configured

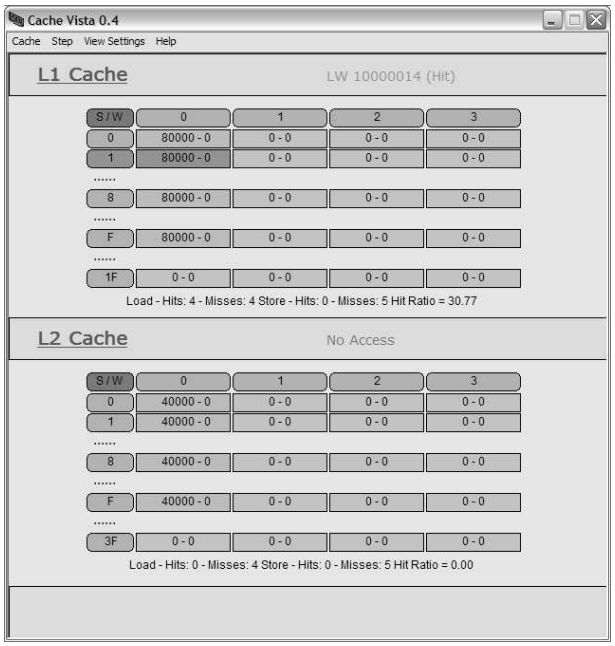

Figure 3. SpimVista hides cache sets that are not accessed.

```
            .data 0x10000000
vectorX:    .word 200, 201, 202, 203    # elements x[0]..x[3]
            .word 204, 205, 206, 207    # elements x[4]..x[7]

            .data 0x10000080
intA:       .word 5                     # A constant

            .data 0x100000B0
vectorZ:    .space 32                   # vector Z (z[0]..z[7])

            .data 0x100000F0
size:       .word 8                     # vector size

            .text 0x400000
            .globl _start

_start:     la $2, vectorX              # vector X address in $2
            la $4, intA                 # intA address in $4
            la $5, size                 # size address in $5
            la $11, vectorZ             # vector Z address in $11
            lw $8, 0($4)                # reads const A in $8
            lw $10, 0($5)               # reads vector size in $10
loop:       lw $6, 0($2)                # reads X[i] in $6
            mult $6, $8                 # multiply X[i] × A
            mflo $7                     # we put mult result in $7
            sw $7, 0($11)               # we write it in z[i]
            addi $2, $2, 4              # X pointer update
            addi $11, $11, 4            # Z pointer update
            addi $10, $10, -1           # loop counter
            bne $0, $10, loop           # loop control
            .end
```

Figure 4. Program code example.

write policies are widely used in current microprocessors.

The program that is used by the students (shown in Figure 4) multiplies the elements of a vector (X) by an integer (A), leaving the result in vector Z, that is $\overrightarrow{Z} = \overrightarrow{X} \times A$. All variables are initially located in main memory, thus relevant memory accesses are the load for the constant, the vector size, and all the elements of the X vector. Also, a store instruction for each element of the Z vector is done at each iteration.

Once the cache hierarchy is configured, the program is loaded and executed step-by-step starting at address `0x00400000`. Figure 5 shows the cache hierarchy state at the end of the execution. This figure illustrates the accesses to the cache memories triggered by the last store instruction. In this case, a miss in the L1 cache and a hit in the L2 cache occur.

Each block in L1 is represented by using two numbers: the block tag and the LRU counter value. As the L1 cache has a write non-allocate policy, the block is not written in that cache. Notice that no block is highlighted (i.e., darker color) at L1. On the other hand, each L2 block is represented by using three numbers: tag, LRU counter value, and dirty bit. Notice that the block depicted in a darker color (green color in the application) in L2 with a `0x400003` tag value represents the hit in L2. Notice also that the LRU counter value is zero meaning that this block is the most recently accessed. Finally, as the L2 uses a write-back policy, the dirty bit is set.

Statistical data is also shown for each level below the contents of the caches. As can be seen, the program exhibits a low hit ratio (33%) in L1 due to the write not-allocate policy used and the relative large number of store instructions. Also, in L2, only cold misses occur, which leads to a higher hit ratio (50%).

6 Conclusions

In this paper we have presented the SpimVista simulation tool aimed at being used by undergraduates. SpimVista is built on top of the PC-Spim simulator and allows the simulation of a two-level cache hierarchy. Cache memories can be configured independently, thus allowing multiple laboratory exercises aimed at understanding the interactions between the different cache structures. The SpimVista tool, implemented by extending PC-Spim, allows to simulate a MIPS assembly program, either in a step-by-step way or to execute the entire program at once.

Additionally, SpimVista provides a carefully designed interface that permits the student to focus on relevant events, from the pedagogical point of view, in order to improve the learning process.

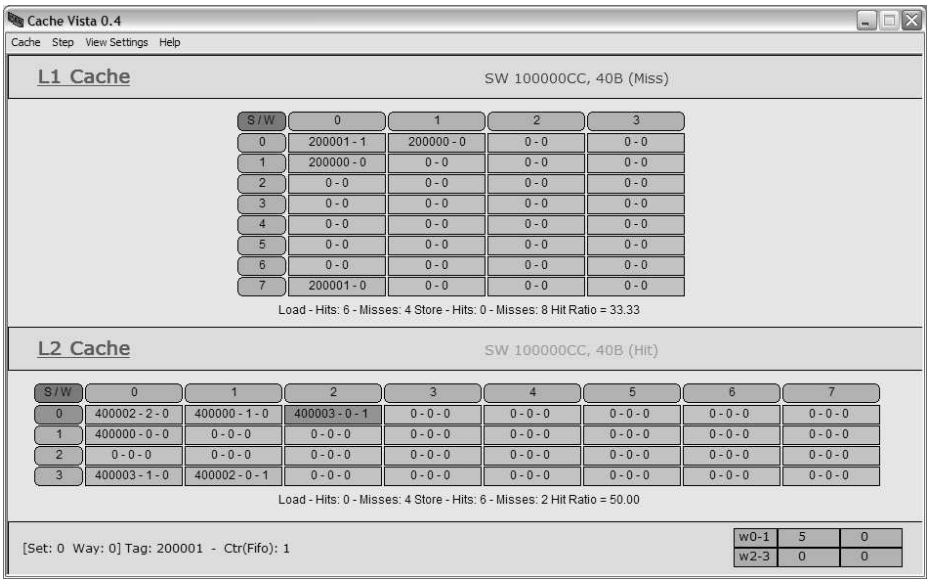

Figure 5. Final state of the cache hierarchy for the provided example.

As for future work we plan to include timing parameters to the different modules in the cache hierarchy (including the memory system). Also, we plan to include pagination and segmentation systems to the main memory module.

Acknowledgments

Authors would like to thank instructors and students of the Computer Organization course for their useful comments to improve the quality of the proposed tool. This work has been partially supported 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.

References

[1] M. V. Wilkes, Slave Memories and Dynamic Storage Allocation, Transactions of the IEEE vol. EC-14 page 270, 1965.

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

[3] J. L. Hennessy, D. A. Patterson, Computer Architecture: a Quantitative Approach, Morgan Kaufmann 2007 (4th edition).

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

[5] J. Larus, SPIM S20: A MIPS R2000 Simulator, Technical Report TR966, Computer Sciences Department, University of Wisconsin- Madison, sep 1990.

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

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

[8] Salvador Petit, Noel Tomás, Julio Sahuquillo, and Ana Pont, An Execution-driven Simulation Tool for Teaching Cache Memories in Introductory Computer Organization Courses, Proceedings of the Workshop on Computer Architecture Education (in conjunction with the 33th International Symposium on Computer Architecture), pp. 119-125, June 2006.

[9] 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.