An Educational Environment for Program Behavior Analysis and Cache Memory Design

Roberto Giorgi, Cosimo Antonio Prete and Gianpaolo Prina
Dipartimento di Ingegneria della Informazione
Università di Pisa, Via Diotisalvi 2, I-56126, Italy

Abstract - *We present an educational software package (Csim) used as a teaching tool to analyze the structure and behavior of a cache memory and to help the student in the design of cache memories for embedded systems. By means of an integrated software development environment, the user can create a program and explore its behavior (locality analysis). The student can observe the cache actions needed for a memory operation and evaluate the cache performance as a function of the configuration parameters. Finally, the parametric-evaluation graphical tools help in the actual design of an embedded system, in order to find the cache and memory configuration which provides the best balance between cost and performance.*

Introduction

In this paper, we present an educational software package (Csim), currently used as a teaching tool in a Computer Architecture course at the University of Pisa. The main goal of Csim is to combine two complementary aspects into a single instrument: on the one side, it supplies the teacher with a tool to be used in practical example sessions while dealing with the main concepts about cache memories [10]; on the other side, it aims to help the student in the actual design activity of embedded application oriented systems [3].

The package includes a parametric cache simulation and performance analysis tool (ChARM) developed by the University of Pisa for VLSI Technology, Inc. [8], and the JumpStart [7] toolkit, a graphical development environment by VLSI Technology, Inc., for the design of ARM-based applications. (ARM [2], [11] is a 32-bit microprocessor designed by ARM Ltd. and largely used in embedded products.)

Design of embedded systems, through new methodologies like co-design approach, *system-in-a-chip* and ASIC solutions, implies the demand for specific architectural knowledge by computer engineers. Actual computer systems and/or commercial design tools are generally not suitable to be used as didactic tools and to present the basic concepts of architecture design in both basic and advanced Computer Engineering courses. Their structure is often too much complex and usually prevents the detection of all the events occurring in the activity of the machine; moreover, the high number and frequency of these events may require a too expensive acquisition system or, also, several events may not be directly observable, since they occur within the chip. As a result, one cannot generally obtain an accurate, step-by-step observation of the internal events occurring in a system. This inadequacy can be particularly true in respect to the emerging co-design technologies for the development of embedded applications.

In embedded systems, low power consumption and/or low cost requirements encourage the adoption of slow memory devices, so that designers often turn to on-chip cache memories to provide both high processing power and large slow main memory. The designer can select the right cache configuration by considering the specific behavior of the program running on the embedded system, and thus optimizing the overall system specification to meet both low power and performance requirements. Tuning such a system, and in particular its cache memory, it's a difficult task. With a minimum of guesswork, the designer must answer a number of basic questions: i) given a system and an application, is it necessary to add a cache memory to obtain the requested performance? ii) if so, which is the optimal cache configuration? iii) given a specific on-chip cache, which is the cheapest main memory satisfying the performance requirements of the application? Without an accurate tool for system configuration and simulation, reliable answers are hard to come by.

All the reasons just exposed stimulated us to develop a new tool which could combine the different needs of students. A relative easiness is guaranteed concerning the practical use of the package, so that the student, when switching from one phase to another, does not need to get familiar with a different – possibly much more complex – tool.

As a mere didactic environment, Csim offers a wide range of opportunities to the student for investigating the structure and the behavior of a cache memory, starting from the basic concepts and definitions [9], up to a relatively complex level of depth. The concept of program locality is particularly emphasized, since it is one of the critical issues in this branch of computer architecture. To this purpose, Csim provides an advanced program-locality analysis and a close evaluation of all the quantities which affect the execution time. The student is actively involved in making authentic choices that affect the target system, such as changing the parameters of a simulation and analyzing the immediate response of that system to the user actions; otherwise, elaborate graphics and simulations may result not effective.

As a design tool, the package allows the user: i) to project his own embedded application within a proper software development environment, and ii) to carry out the performance evaluation, in order to choose the system and cache configuration which can guarantee the best performance for the target application. The designer can perform a parametric simulation to evaluate system performance while varying the timing and architecture features of each component of the system. The final results, shown by means of easy-to-read graphs, help him to find rather quickly an acceptable tradeoff solution.

The Csim environment

In view of a didactic approach, the package has been equipped with a very friendly point-and-click graphical interface, by which the teacher can easily show and discuss, using practical examples, the basic concepts of cache architecture and behavior. The environment consists of five phases shown in Figure 1.

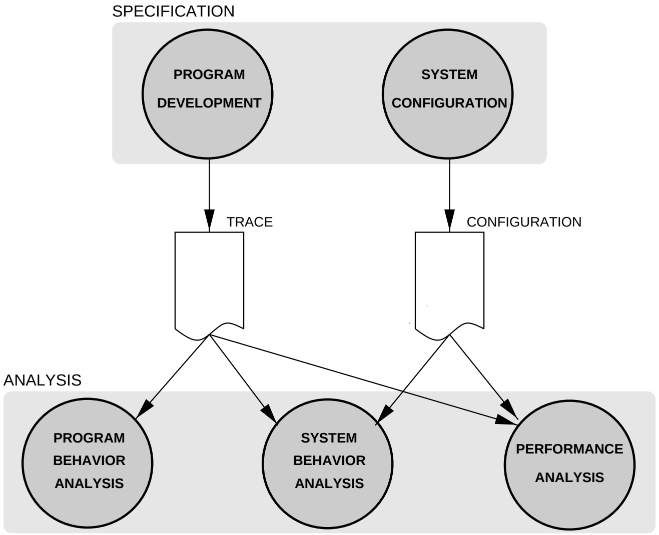


Figure 1. Structure of a Csim session.

In the *Program Development* phase, the user builds an application, debugs it and produces a trace file. Applications can be executed and debugged on a dedicated ARM instruction set simulator or loaded in an ARM CPU-based board for a native evaluation. Once that the application has been developed, the user can generate a trace file by simply pointing-and-clicking while the program is running in emulator mode.

In the *System Configuration* phase, the user defines the system architecture and the features of each component. The system may include the following components: an ARM core, a cache memory, a system bus, memory banks, and a number of I/O devices. For each component, the designer has to specify the timing, the architecture and the management policy.

The *Program Behavior Analysis* phase allows the student to perform two types of trace analysis. The first one uses traditional program statistics such as the percentages of data/code, read/write accesses. The second one regards program locality. An accurate knowledge of locality features plays a crucial role in understanding cache concepts. The locality statistics include: the number of unique blocks, the locality surface [5], and the spatial locality.

For mere didactic intents, the *System Behavior Analysis* phase allows the student to carry out a sort of step-by-step simulation, by executing a single memory operation and examining in detail the sequence of actions necessary to a cache controller to carry out the requested operation, or by executing a specified sequence of operations and examining the cache status and contents after the execution.

The *Performance Analysis* phase allows the user to plan, perform a single simulation or a performance evaluation *experiment*, and finally analyze the results. An experiment is defined by: i) the trace file; ii) the system configuration; and iii) the varying parameters (one or two). Csim may initially simulate an adequate number of memory references without an outcome. This allows the cache to exit its cold state [1] and to reach a steady condition. The results consist of: *global system performance* (execution time, lost time in waiting, and word transfer ratio); *cache behavior* (miss, code miss, data miss, read miss, data read miss and data write miss ratios and cumulative cold misses); and *bus traffic* (occupation rate, number of read-block operations, number of write operations for write-through cache models and number of update-block operations for copy-back cache models).

Use of Csim as a didactic tool

In a didactic approach to computer architecture, one of the key concepts that the student has to deal with is *program locality*. Hence, the first phase of a typical Csim didactic session, concerns the analysis of locality features of a program written directly by the student or chosen within a set of predefined, very simple programs; in the example shown in detail in this Section, the program is the implementation of the median filter algorithm applied to a 34×34 pixel image with a 3×3 pixel window [4].

As shown in the previous Section, during the *Program Development* phase (Figure 1) a trace can be produced to allow a detailed program locality analysis (number of unique blocks, locality surface, spatial locality).

If we define $T[i]$ as the i-th reference of a trace T, for each couple $\{T[i], T[j] \ such \ that \ j > i\}$ we can also define the *distance* (d) as the number $j - i$ of intervening references, and the *stride* (s) as the offset $T[j] - T[i]$ between the two references.

The concept of *spatial locality* refers to the fact that address locations close to the "currently" referenced location $T[i]$ are more likely to occur in the next few references than locations far away. Similarly, the concept of *temporal locality* reflects the fact that the address of the "current" reference $T[i]$ is very likely to occur again in the next few references.

A quantitative approach to the locality analysis was first proposed by Archibald *et al.* [5] by means of the introduction of the *locality surface*. They proposed a 3D-graph where stride and distance are the base axes. The magnitude of locality surface for a specific couple (s, d) is defined as the probability that $T[i] + s = T[i + d]$, where $T[i] + s \notin \{T[i + 1], ..., T[i + d - 1]\}$ and i assumes all the values between 1 and the length of the trace T minus one. From the locality surface, the designer may derive information about locality features like *sequentiality*, *striding*, *temporality* and *loops*. Sequentiality is typically due to the fetching of consecutive instructions. It is visible as a ridge along the diagonal region with $s = d$, in which the length reflects the distribution of sequential run lengths in the reference stream, while the amplitude reflects sequential run frequency. Striding is produced by a series of references with a fixed step and is typical of numerical algorithms, such as matrix operations where the elements are accessed in row order instead of in column order. It is characterized by a ridge in the region with $s > d$. The temporality region, i.e., the region with $s = 0$, shows the distribution of distances between repeated accesses at the same addresses. Finally, loops are characterized by ridges which are parallel to the stride axis, in the region with $-d < s < 0$. Figure 2 gives an example of a locality surface concerning the median filter program. In particular, the temporal locality window shows that, in this case, the latest referenced address has a very high probability (more than 80%) to be referenced again within the next 128 memory accesses.

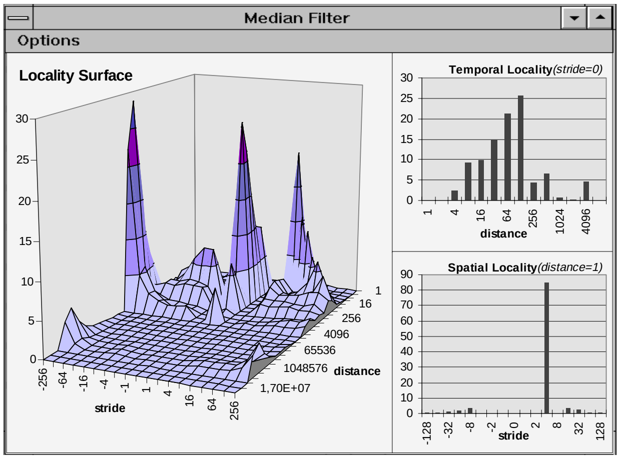


Figure 2. The locality surface for median filter program.

According to the mathematical definition, spatial locality is the distribution of the offset between two consecutive addresses in the trace. It can be obtained from the locality surface in the case of $d = 1$. Csim can show spatial locality in a specific graph as distribution of: i) all accesses and ii) data and code accesses separately. Figure 3 shows the locality of data accesses for our example.

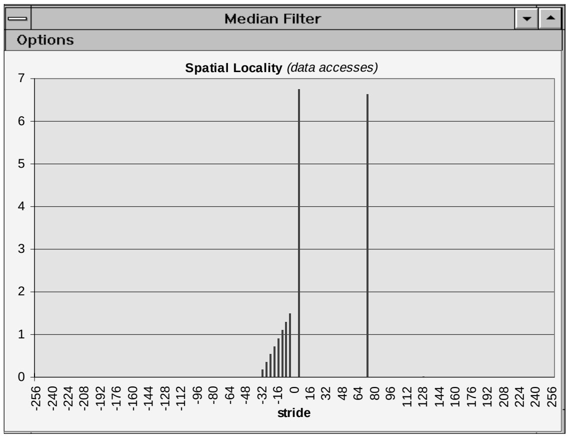


Figure 3. Spatial locality in the data area.

The user can optionally analyze the *unique blocks* graph. The number of unique blocks for a given number i of references is the number of distinct blocks used by a program before the reference i-th. These blocks only cause misses in an infinite cache, the number of unique blocks delineates a lower bound for the miss ratio. Csim shows a family of curves, where the number of unique blocks is given as a function of the number of references and the parameter is the block size (Figure 4).

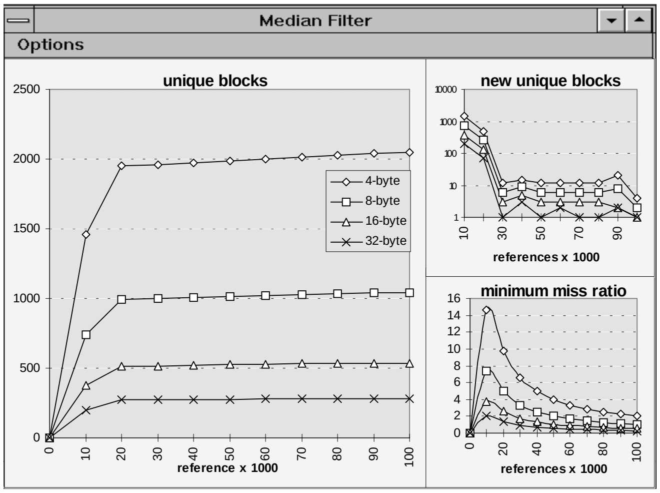


Figure 4. Number of unique blocks (total and incremental) and lower bound of miss ratio for median filter program.

The next step in a didactic path is to show how the presence of a cache memory can exploit program locality in order to improve the system performance. For this purpose, the student has to select the system parameters concerning cache organization. A cache scheme is defined by the following parameters: i) the mapping policy; ii) the replacement algorithm; iii) the update policy; iv) the cache size, v) the block size; vi) the number of blocks for each set (in the case of a set associative cache); and vii) the presence and the length of a write buffer.

Figure 5 shows the miss percentage of the median filter program as a function of the cache size and degree of associativity. For this program, a 2-way set associative cache is recommended since it supplies the best balance between cost and performance. Also, we notice that a 4-way cache produces quite the same result. At a more detailed level of analysis, the student is called to search, for a given cache structure, the cache and block sizes which can provide the best results in terms of global performance. Again a new graph can be produced (like the one in Figures 8 and 9) to find the best value for the cache block size.

Finally, we show how the *Cache Behavior Analysis* environment can be employed to perform the step-by-step simulation. Let us consider the cache configuration just examined (2-KByte, 2-way set associative cache with a 32-Byte block size) and suppose (Figure 6) that the required operation is a read on location with address $(00001FD4)_{16}$; the following events and actions are highlighted: i) the memory block $(00000FE)_{16}$ is

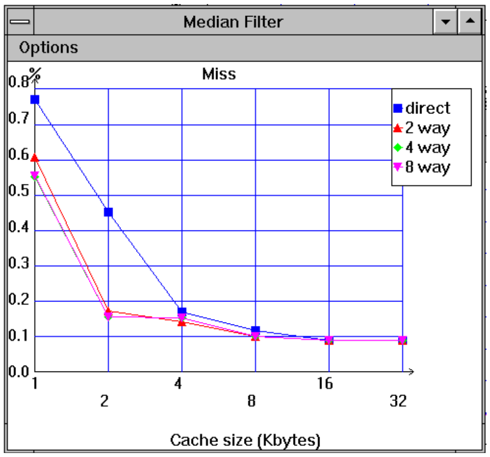
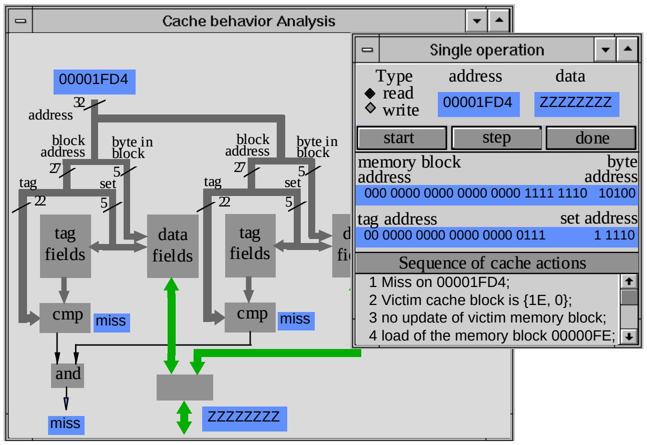


Figure 5. Miss percentage for median filter program.

not present in cache memory (miss condition), ii) the victim cache block to be replaced is the block 0 in the set $(1E)_{16}$; iii) it is not necessary to update the main memory block because the copy is not modified; iv) the cache loads the memory block $(00000FE)_{16}$ and so on.



address: 00001FD4
data: ZZZZZZZZ

memory block address: 000 0000 0000 0000 0000 1111 1110
byte address: 10100

tag address: 00 0000 0000 0000 0000 0111
set address: 1 1110

Sequence of cache actions

1 Miss on 00001FD4;
2 Victim cache block is {1E, 0};
3 no update of victim memory block;
4 load of the memory block 00000FE;

Figure 6. Step-by step operation.

In a window, Csim shows the structure of the selected cache; in a second window, it summarizes the information about the operation being executed and shows how the cache uses the memory address. In our example, the address $(00001FD4)_{16}$ is split in $(000007)_{16}$ used as tag field (compared with the tag field of both blocks of set $(1E)_{16}$); in $(1E)_{16}$ as set field (used as set address), and finally $(14)_{16}$ used as byte offset in the block.

Beyond theory: case studies of actual design

In the design of embedded systems, a key point is the optimization of each component, which needs to meet, as better as possible, the specific application for which the whole system is designed. Also, it should be noticed that often an embedded system runs only one program for all its life. We are going to present an example of design training path, and we will show how Csim can help a student to find out the optimal cache and system configuration for a specific application. We consider the `cjpeg` program, a `jpeg` image compression/decompression tool [12] which is frequently used in commercial embedded systems.

First, the student should wonder about the following question: for a 20 MHz ARM running an application using the `cjpeg` program, is it necessary to add a cache memory in order to achieve the required performance? We suppose that the product requires that the image compression be completed in less than 1s. First, the student traces the execution of the `cjpeg` program while compressing an image stored in, e.g., “ppm” format, using the JumpStart trace facility. The ppm image is a 101-KByte image consisting of 227x149 pixels. (The sample picture is a red rose and the image produced by `cjpeg` occupies 5 KBytes.)

Then, the student defines the system configuration including a 20 MHz ARM core, a system bus, a 1-MByte memory DRAM bank, a 128-KByte memory PROM bank, and a memory-mapped graphical I/O device. The student needs to specify the timing and architecture of each component. In order to obtain a low cost and a low power-consumption solution, a slow system bus and slow memory devices are selected.

In the case of the ARM core, the student provides the timing for both read and write operations. For these operations the simulator requires i) the minimum time necessary for the ARM core to perform the bus operation and ii) the maximum available to a slave to complete an operation without requiring waiting time for the CPU.

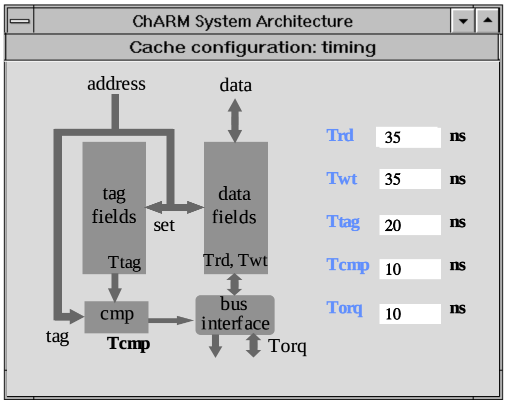
A specific window shows the ARM timing plot, derived from the ARM data book, in order to drive the student to find the proper values. In our example, considering that the ARM processor employs a pipelined bus, the student obtains and sets these values: 50 ns, as minimum time for read/write operations, and 64 ns as the maximum time usable by a slave to complete an operation.

The simulator models a generic bus, which is capable to accommodate the typical memory operations. The student has to specify the data bus width and the time for each type of bus operation. In our example, the data bus width is 32 bits and the time is 200 ns for both read (T_{read}) and write (T_{write}) operations.

Finally, the student specifies the features of memory and I/O devices. For each module, the configuration parameters include the module type, the starting address and the size. In our example, the system includes three modules: a DRAM bank, a ROM bank and a memory-mapped graphical I/O device. The simulator requires to know the delays (additional time with respect to the bus time) introduced by a component to complete each bus operation. Among the three modules considered, only the PROM module needs additional time (200 ns) to complete read operations.

With the system configuration just examined, a simulation shows that, without cache memory, the application takes up 2.898s to execute the program. The addition of a cache memory proves to be necessary, therefore, to meet the time requirement.

As shown in the previous Section, the designer has to define the cache structure in terms of cache size, block size, number of blocks per set, and replacement policy; furthermore, for simulation to be possible, cache timings have to be specified. Figure 7 shows the scheme used to set the cache timings: T_{rd} and T_{wt} are the times for reading data from or writing data to a cache block; T_{tag} is the tag access time; T_{cmp} is the compare time; and T_{orq} is the time needed to initiate an operation involving an attached module, after a miss has been detected.



Trd 35 ns

Twt 35 ns

Ttag 20 ns

Tcmp 10 ns

Torq 10 ns

Figure 7. Setting the cache timings.

Finally, the student has to specify the timing of bus block-operations used by the cache to interact with the main memory. In particular, the cache uses the *read-block* operation to fetch a memory block when a miss condition occurs. An *update-block* operation allows a copy-back cache to update a memory block when its cached copy is dirty and has to be replaced. The time required by a bus block-operation is calculated by considering the bus width, the block size and four timing values: T_{read}, T_{write}, T_{sread} and T_{swrite}. A block operation is described as a single operation followed by fast transfer operations. T_{read} (T_{write}) is the time needed to perform a single read (write) operation, and T_{sread} (T_{swrite}) is the time needed to perform a subsequent sequential read (write) transfer. In this example, the timings are: T_{read} = T_{write} = 200ns, $T_{sread} = T_{swrite}$ = 160ns.

The student can now execute a parametric simulation in order to search the optimal cache configuration. Figure 8 shows the miss ratio and the execution time versus block size (from 8 to 64 Bytes) and cache size (from 2 to 32 KBytes) for two cache configurations. The first cache is a simple write-through, direct access cache without a write buffer; the second one is a more complex copy-back, two-way set associative cache with a two-deep write buffer. The cache uses the LRU technique as replacement policy. In both cases, the timings are: T_{rd} = 35ns, T_{wt} = 35ns, T_{tag} = 20ns, T_{cmp} = 10ns, T_{orq} = 10ns.

The designer can observe that, in both configurations, execution time and miss ratio exhibit different values and behaviors. For cache sizes greater than 16 KBytes, the execution time is constant and independent of the block size. In this way the student can select a configuration that best meets cost-effectiveness and performance requirements (execution time $\leq$ 1s). For example, an optimal choice is a 16-KByte, write-through, direct access cache with 16-Byte block size without a write buffer.

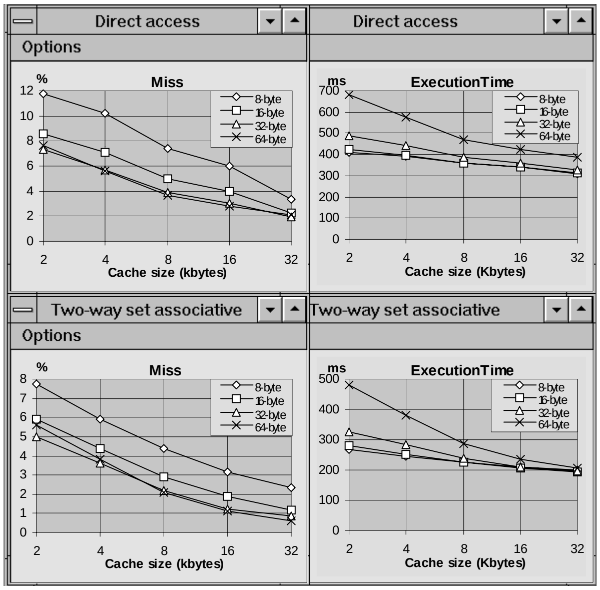


Figure 8. Miss ratio and time execution for `cjpeg` *program.*

Now, the student can also answer the question: for the selected cache configuration, which is the cheapest main memory meeting the time requirement? The designer can find the solution by executing simulations having the RAM bank access time as parameter. The simulation shows that the memory bank delay can be increased by no more than 30ns with respect to the values specified in the configuration.

Now, if the student uses the same design path for a different embedded application, he/she finds a different cache and system configuration. For example, we trace the execution of the `rawcaudio` program [6] while it converts a 6-KByte ADPCM sound sample to a 24-KByte raw 16-bit PCM format. The audio sample is the voice of a man saying “hello, world.” Figure 9 shows the miss ratio and the execution time for the same configurations considered in the first example. The differences between the graphs of Figures 8 and 9 are due to the different locality characteristics of the programs.

We assume that the application requires that the conversion should be completed in less than 90ms, in order to show a configuration tuning session. The system takes up 359ms to execute the program without cache memory, therefore cache is necessary. Table 1 lists some examples of cache configurations that allow the system to satisfy the time requirement. If the designer selects cache configuration 2, the memory bank delays can be increased by 260ns with respect to the values specified above.

These two examples show that meeting time requirements of different applications yields different cache configurations. The student can observe that, in the second example, the write-through cache configurations never guarantee the ful-

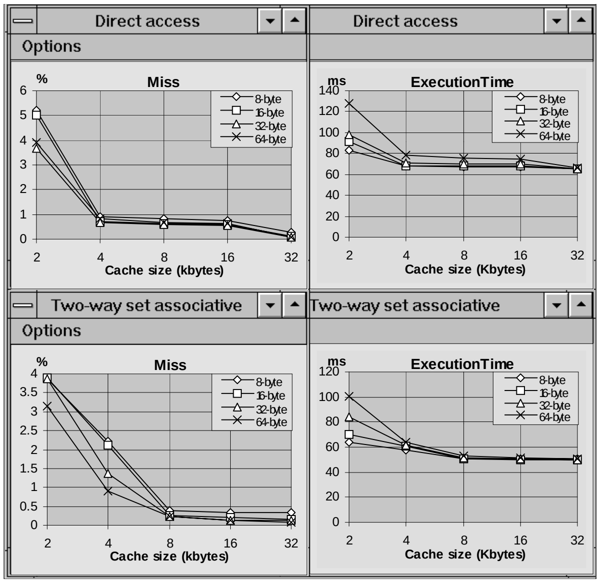


Figure 9. Miss ratio and execution time for `rawcaudio` *program.*

	mapp.	update policy	cache size (KB)	block size (B)	write buffer len.	exec. time (ms)	max delay (ns)
1	direct	copy-back	8	8	0	88.97	40
2	direct	copy-back	16	16	0	86.40	260
3	direct	copy-back	8	8	2	88.75	60
4	direct	copy-back	16	16	2	86.34	260
5	2-way	copy-back	4	16	0	88.96	40
6	2-way	copy-back	8	64	0	88.62	50
7	2-way	copy-back	4	16	2	88.86	50
8	2-way	copy-back	8	64	2	88.61	60

Table 1. Some cache configurations that allow the system to satisfy the time requirements for `rawcaudio` *program.*

fillment of time requirements. The two examples also show that write and miss ratios affect the performance of systems with cache memories in a different way. The `cjpeg` and `rawcaudio` programs have similar write ratios, but exhibit different miss ratios due to different locality features. A lower miss ratio enhances the influence of write operations on global performance. In this case, the choice of an optimal update policy becomes critical.

Conclusions

The growing demand for embedded products requires highly sophisticated computing functions. Designers must select the most efficient cache/system configuration in order to resolve complex – even conflicting – requirements for low-power/high-speed and component cost. This makes accurate and reliable system/cache memory simulation and performance analysis crucial. We have presented an educational environment based on a trace-driven system simulator that can help students in the design activity of cache memory to be employed in ARM-based embedded systems. By means of practical examples, we have shown how the student can successfully use the tool in two typical schemes of a didactic path.

Acknowledgments

This work was supported by the Ministero della Università e della Ricerca Scientifica e Tecnologica (MURST), Italy and by VLSI Technology Inc. We wish to thank Francesco Lazzarini that took part in the development of the cache simulator, Angelo Rappelli that contributed in developing the graphical interface for the Microsoft Windows version, Massimiliano Panico that performed performance evaluations of the case studies.

References

[1] M. Easton. Computation of cold-start miss ratio. *IEEE Transaction on Computers*, C-27(5):404–8, May 1978.

[2] S. B. Furber, P. Day, J. D. Garsidex, N. C. Paver, and J. V. Woods. A micropipelined ARM. In *Proceedings of the IFIP TC 10/WG 10.5 Int'l Conf. on Very Large Scale Integration (VLSI '93)*. Grenoble, France, Ed. Yanagawa, T. And Ivey, P. A. Pub. North Holland., Sept. 1993.

[3] D. D. Gajski and F. Vahid. Specification and design of embedded software-hardware systems. *IEEE Design & Test of Computers*, 12(1), Spring 1995.

[4] N. Gallagher and G. Wise. A theoretical analysis of the properties of median filters. *IEEE Trans. Acoustics Speech and Signal Proc.*, 29:1136–1141, 1981.

[5] K. Grimsrud, J.Archibald, R. Frost, and B. Nelson. Locality as a visualization tool. *IEEE Transaction on Computers*, 45(11):1319–1326, Nov. 1996.

[6] International Telegraph and Telephone Consultative Committee. 32 kbit/s adaptive differential pulse code modulation (adpcm). In *CCITT Recommendation G.721*, 1984.

[7] *JumpStart Reference Manual*. VLSI Technology, Inc., 1994.

[8] F. Lazzarini, C. A. Prete, and M. Graziano. Tuning the configuration of a cache memory for embedded systems. to appear in IEEE Micro.

[9] C. Prete and G. Prina. Experiences in using a cache simulation tool in advanced computer architecture courses. In *Workshop on Undergraduate Computer Architecture Education*. S. Margherita Ligure, Italy, June 1995.

[10] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[11] A. V. Someren and C. Atack. *The ARM RISC Chip, A Programmer's Guide*. Addison-Wesley, 1993.

[12] G. K. Wallace. The jpeg still picture compression standard. *Communications of the ACM*, 34(4):30–44, Apr. 1991.