A New Methodology for Studying Realistic Processors in Computer Science Degrees

C. Gómez
Departamento de Sistemas Informáticos
Universidad de Castilla-La Mancha
Albacete, Spain
Crispin.Gomez@uclm.es

M.E. Gómez, J. Sahuquillo
Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València
Valencia, Spain

***Abstract*—The architecture of current microprocessors has suffered a vertiginous evolution in the last years, leading to the development of multithreaded multicore processors. The inherent complexity makes teaching students how current processors work a complex task and this has led most universities to study superscalar or multithreaded processors in a very simplified manner.**

This situation aggravates even more at lab sessions. Due to this fact students have a faraway view from real working of current multicore processors. To deal with this hard shortcoming, we designed a new methodology to approach students to real processor hardware, that has been successfully used at the Universitat Politècnica de València. The methodology is based on the use of a very detailed and accurate simulator that provides support to simulate all the main components of current microprocessors.

Due to the simulator complexity, it is introduced through several phases in a progressive manner, which leads the students to achieve a solid learning about how current processors work, even including the a detailed model for the on-chip network that connects the cores of the system. Results demonstrate that students are able to use the simulator in a reasonable time period, and understand at the same time the more complex concepts of commercial processors. Furthermore some of them have been able to developed projects with the studied simulation framework that have been published in top international research conferences, such as PACT and IPDPS.

***Keywords*-Teaching, Computer architecture, simulators, Multicore processors.**

I. Introduction

In the last decade, electronic devices have vertiginously evolved and they are being extensively used in our everyday lives. Nowadays, most people use smart-phones, laptops, netbooks, smart-TVs or set-top-boxes, and so on. These advances have been possible thanks to the great computational power exhibited by current processors and to the low power consumption required by them.

This huge computational power has come from an increase in hardware and functional complexity in the processors. Early at 2000's, monolithic processors reached their best performance with superscalar processors and out-of-order execution with complex branch predictors, *smart* cache memories like the trace cache, support for simultaneous multithreading, and so on. Afterwards, power consumption and heating became a major problem, and industry moved to develop processors with several cores, known as multicore processors. In these processors, each core is able to support the execution of multiple independent threads. As user computational requirements keep going up, the number of cores continuously grows, which in turns requires the use of a fast and power-efficient interconnection network to connect efficiently cores among themselves and to the main memory.

This high complexity and constant evolution of processors demand a considerable effort from Computer Science instructors to be up to date with commercial designs. Moreover, it makes difficult and hard the task of teaching and learning this subject. In most universities around the world students start with the study of a simple pipeline of an scalar processor during the first years of their degree. Later, concepts a bit more advanced, like superscalar and sometimes multithreading, are introduced. This fact causes that students have a conception of the actual working of current multicore processors faraway from real hardware.

In this work we present a new methodology consisting of several phases with different difficulty degrees that helps the Computer Science student to acquire a wide and deep knowledge of how commercial products work. For this purpose, we focus on the usage of a detailed cycle-accurate simulator for multithreaded multicore processor, that is widely used among PhD students and researches all over the world, as well as some processor manufacturer companies. The methodology proposes the use of the same simulation environment in various courses of the Computer Science studies to amortize its learning time, and due to its high complexity, its use is planned with increasing difficulty in a progressive manner. Students start, in the first phase, using the tool in a carefully guided way without contact with the source code, and progressively the student implication is increased ranging from minor modifications of the source code to the completion of the Final Degree Project and his participation as co-author in research papers.

The rest of the paper is organized as follows. Section II describes how computer architecture is taught at the Universitat Politècnica de València and presents the simulator framework. Section III presents the proposed methodology using the simulator to get the expected learning experience. Section IV shows a case study. Finally, some conclusions

are drawn in Section V.

II. Computer Science Teaching

A. Computer Architecture at the Universitat Politècnica de València

This section describes teaching of current processors in the Computer Science degree at the Universitat Politècnica de València. Computer architecture is organized and studied in several courses. To focus this paper we will concentrate in processor related topics. At the second year, the *Computer Structure* course presents the pipeline of a simple processor to the students. After that, in the *Computer Architecture and Engineering* course, we study (at the fourth academic year- courses are annual) extensions to the base pipeline to support speculative execution. Nevertheless, in both courses the focus is mainly educational and its aim is to define and clarify the most basic concepts. These courses must provide the pillars to provide a wider and deeper training in subsequent courses. However, this is usually accomplished by means of simplifications that make wider the distance between the studied processor and the commercial ones.

The offered courses that have a closer approach to commercial products are: *Advanced Processor Architectures* studied both in the last year of the degree (Computer Engineering specialization) and in the master degree in Computer Engineering; and *Networks on-chip* also offered in the Computer Engineering master degree. Both courses can be chosen to be studied by students that are interested in having a more accurate view of how real hardware works. These courses are fully complementary and students are encouraged to study both of them, since it is not possible to understand the impact on performance of current commercial processors considering only one of them without considering the other one as well. Keeping this scenario in mind, during the last academic year we have developed and implemented a new methodological approach that includes both courses as if they were only one. The benefits of this approach is twofold. On one hand, students analyze and realize how the different system components interact each other to obtain a single overall system performance. On the other hand, students only need to learn the use of a single tool in an incremental way, thus making easier the learning of complex simulators. Both benefits provide students worth and invaluable skills in future research or professional activities.

B. Problem: the increasing gap between theory and real hardware

The ever evolving processor architectures have forced to continuously update teaching contents to bring them as close as possible to real commercial products, with the aim to provide a relatively updated training to students. This continuous update requires to adapt both the theoretical contents that are studied at classroom and practical contents that are trained at lab.

Concerning practical contents that are trained at laboratory, several simulators are usually used depending on the semester and the aim of the course. Among the typical simulators that are being commonly used, we can find DLXV to study the processor pipeline: the datapath, the instruction flow through the distinct pipeline stages, solving data-hazards, and so on. DLXV is a straightforward simulator quite easy to be learned and used, which comes in handy for learning the understanding of simple scalar processors, which are the base of current superscalar processors. Nevertheless, DLXV does not provide teaching support to study monolithic superscalar processors or multithreading since it cannot model them. This causes a very important gap between theoretical and practical contents, since main theoretical contents are not developed in a practical manner at lab sessions, which as it is widely accepted, helps the reinforcing and understanding of theoretical concepts. Most universities do not cover this gap and more complex and complete simulators, like SimpleScalar [1], Sniper [2] and Multi2Sim [4] that can model those concepts, are exclusively used by PhD students. This is a clear mistake since the support that these tools provide in the laboratory to get a solid understanding of complex concepts is not exploited at all.

The methodology proposed in this paper addresses this problem at its early stages, which also stimulates the interest of students in computer architecture by gradually involving them in more complex scenarios. Next, we describe the main characteristics of the simulator used to develop this methodology.

C. Chosen the Simulator

Our aim was to select a single long-life simulator able to be used in first years courses but also in more advanced courses and even in the development of the PhD Thesis, if students are interested, without changing the experimental framework. To cover all these courses the simulator must be able to model current processors in a detailed way but in the first years the simulator complexity must be hidden. Because of this reason, we have chosen Multi2Sim (M2S) that is capable to model superscalar pipelined processors, multithreaded and multicore architectures, graphics processing units (GPUs). Moreover it supports the more common benchmark suites used in research. M2S belongs to the category of application-only simulators, since it allows to execute one or more applications without having to boot an operative system. In addition, it has to be noticed that M2S is an open source project that can be freely downloaded from [6].

The used programming paradigm in this simulator can be split in two main modules: functional and detailed simulation. Functional simulation consists just in a emulation of the input program that has the same behavior that the program would have when executed in a native

x86 machine. Detailed simulation offers a model of all the structures of the processor: cache memories, CPU functional units, interconnection network, etc. The core is pipelined in six stages (*fetch*, *decode*, *dispatch*, *issue*, *writeback* y *commit*), and it is able to perform speculative execution. The number of threads per core and the number of cores are configurable. The framework allows the modeling of the different multithreading paradigms: coarse grain (CGMT), fine grain (FGMT), and simultaneous (SMT).

Memory hierarchy configuration is highly flexible. The user can define as many cache memory levels in the hierarchy as needed and the number of cache memories in each level is completely configurable (cache coherence is guaranteed by means of a MOESI protocol). Cache memories can be split into data caches and instruction caches or can be unified, they can be private or shared by several cores; threads, or GPUs, and the address range per each cache memory can be also configured.

The interconnection network among the different levels of the memory hierarchy is also widely configurable with a simple model that allows the definition of end nodes (memories or cores), switches, links, the topology and the routing algorithm.

Additionally, the simulator can generate a detailed report of statistics related to the cores pipeline, classifying them per thread and core. This report present generic simulation results (like execution time), and statistics per each hardware structure (ROB, IQ, branch predictor), and pipeline stage.

The simulator generates also a similar report for the memory hierarchy that contains a section per each cache module, main memory module and the interconnection network. For each memory module (either cache or main memory) the report shows the number of accesses, hits, misses, reads, writes, etc. Concerning the interconnection network, the user can obtain the utilization of the links, the used bandwidth, the number of transmitted messages, the average utilization of the buffers, and so on.

The simulation framework also provides user-friendly debugging tools. For instance, the pipeline debugger is a tool integrated in the simulator that can be used to graphically observe the pipeline execution time diagrams of the detailed simulations, in which the user can see a cycle-by-cycle trace of a given stage with a graphical representation of the executed uops.

Furthermore, from version 3.0, M2S has enhanced its simulation capabilities with a model for AMD GPUs, that also has its own pipeline debugger. This debugger is a graphical tool to analyze the execution of the OpenCL code in the corresponding simulator unit.

In the latest versions, M2S uses the INI file format for all its input and output files. An INI file is a plain text file used to store configuration and statistical reports of the simulations.

Finally, M2S has been adapted to provide statistics in the format required by McPAT, which is an integrated modeling environment for power and area consumption that supports thorough design exploration for multicore processors.

III. Proposed Methodology

The methodological approach consists of four phases with an increasing difficulty degree. The objective is to provide to the interested students a solid basis of knowledge about realistic processors architecture by allowing them to have a progressive iteration with the simulator. The defined phases are:

- **Modify simulation parameters:** As aforementioned, as the simulator is very complex, students start using it as a black box. They do not look into the source code and just run different simulations by changing several simulation parameters like branch predictor, issue width in the cores, or link bandwidth in the network. In this way, students can understand how to launch simulations, how to read the statistics reports, how the different parameters affect the performance of the system, and they can see how to apply several of the theoretical concepts that have been taught in the classroom. This first step is done in a very guided way, so we propose to implement it in the practical lessons at lab.
- **Modify simple source code bits:** After the previous phase, the student should have enough background with the simulator to modify small chunks of the source code and implement very specific and well-delimited functionalities. For instance, the student could implement the prefetching mechanism for the cache memories. Notice that this is still a very initial step so this again has to be done with a careful guidance by instructors, for instance, we have given students a very simple prefetching mechanism and they have been asked to study it and implement several new prefetching mechanism from it. This is mainly a phase which aim is that the student loses the fear to modify the source code of a large simulator, but notice that highly modular. In addition to the already implemented functionality, we suggest to provide also a detailed report describing the source code for students. For example, we prepared a report for implementing the prefetcher, which describes how the memory hierarchy is modeled in the simulator, referring to specific functions and files inside the source code. At the end of this phase students are ready to face more important challenges with the simulator. In our experience this phase is best performed as Final Work of the course by students.
- **Implement complete functionalities:** The students that succeed in the second phase would usually have stronger curiosities and would like to face larger challenges that will allow them to go deeper in their knowledge about all the details of the processor architecture.

In this phase, we propose to give students autonomy enough to completely implement a full functionality from scratch. This step can best fit for Final Project degree or Master Project degree. The results obtained by our students during this phase have been astonishing and they have been even published in top-notch conferences like IPDPS and Euro-Par.

- **Complete autonomy:** The students that complete the previous three phases will be in a privileged position to start their PhD studies, since they will have at least one year of experience with a simulator and researching which will nicely benefits their PhD results.

IV. Case Study

This work focuses in the implementation of the first two methodology stages. The methodology has been successfully implemented during the last academic year in two courses at the Universitat Politècnica de València. These courses are *Advanced Processor Architectures* offered in the Computer Science degree and *Networks on-Chip* offered in the Master Degree of the Department of Computer Engineering. In both, M2S has been introduced gradually to the students.

A. Work choices and learning stages

A key point for the success of any methodology is that the work to be done by the students motivates them. With this aim, we have chosen as a baseline system a multicore processor that implements a mesh topology as interconnection network, which is representative of current and next microprocessor generations. With this baseline processor system, the students have to complete several learning stages in a progressive way, and each step forward must motivate them go ahed to the next one. Keeping this idea in mind, we propose the following main four stages:

- Baseline system modeling.
- Execution of multiprogrammed benchmarks.
- Execution of parallel benchmarks.
- Prefetching mechanisms implementation.

B. Baseline system modeling

The student has to obtain only the knowledge about the simulator that is required to reach the desired learning stage. Unlike to what it is typically done in PhD students, it is not advisable to make a thorough tutorial about the simulator before starting the work, as this would discourage the students. To overcome such a problem we opted for a "learn as you work" methodology.

As an initial step to make this approach feasible, instructors of the courses mentioned above have prepared a detailed simulator guide. This is not a typical simulator guide but it combines and makes relations between theoretical

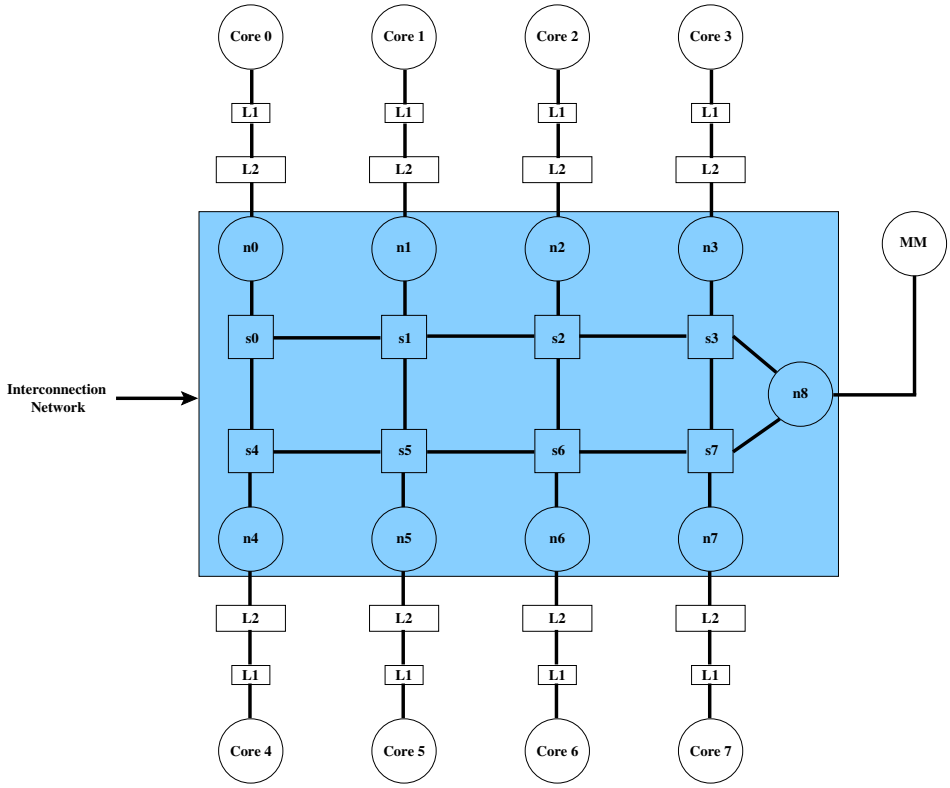

Figure 1. Proposed base system.

concepts presented in the lectures and practical aspects. This guide describes the elements that can be modeled with the simulator: cores, cache memories, and interconnection network. For each of these elements, the guide explains its operation and role in the system, and it provides a brief description of the main configuration parameters for them and the expected consequences when changing them. For instance, in the case of the core the guide shows how to change the branch predictor; in the case of a cache memory it explains how to change the size of the cache block or the access latency; and in the case of the interconnection network it shows how to change the bandwidth of the links or the size of the buffers at the switches. Moreover, when describing the cache memories the guide shows how to model the whole memory hierarchy, interconnecting the cache memories of all the cache levels among them, to the cores and to the main memory. In a similar way, it explains to the students the correct way of defining an interconnection network inside the chip to connect all the different cache memories.

After explaining to the students how to model the distinct system components in the simulator, we present and describe them the baseline system to be modeled. To start with, we have chosen an 8-core CMP with private L1 and L2 caches, interconnected by a mesh topology as depicted in Figure 1. In order to easy the work, we provide the baseline model to the students, with the required configuration files (core, memory hierarchy and interconnection network) to model this system.

In listing Example 1, we show a small fragment of the configuration file for the interconnection network namely *mesh*. In particular, as can be seen in the example this file configures the global network parameters like link bandwidth and buffer size, and the topology and routing algorithm. In the example, the piece of listing shown specifies the connection of node 0 and switch 1, and the routing table associated to switch 0.

Listing Example 2 presents part of the configuration file of the memory hierarchy. It contains the size and access latency for the caches, in this case L1 and L2 caches, the geometry of caches (the piece shown only specifies the ones connected to node 0), their location in the memory hierarchy, the L1 to L2 connection, and the connection of the L2 cache to the main memory through the *mesh* network defined in Example 1.

C. Execution of multiprogrammed and parallel loads

It is important that students experience with standard benchmarks that are used in actual processors and are extensively used by the scientific community, and realize how their model is able to correctly execute them. With this idea in mind, we have chosen the SPLASH2 benchmark suite [5] and the SPEC-CPU 2006 benchmark suite [3].

Example 1. Interconnection network configuration file.

```
[ Network.mesh ]
  DefaultInputBufferSize = 128
  DefaultOutputBufferSize = 128
  DefaultBandwidth = 64

;;;;;;;;;; NODES ;;;;;;;;;;
[ Network.mesh.Node.n0 ]
Type = EndNode
...
;;;;;;;; SWITCHES ;;;;;;;;
[ Network.mesh.Node.s0 ]
Type = Switch
...
;;;;;;;;;; LINKS ;;;;;;;;;;
[ Network.mesh.Link.s0s1 ]
  Source = s0
  Dest = s1
  Type = Bidirectional

[ Network.mesh.Link.s0s4 ]
  Source = s0
  Dest = s4
  Type = Bidirectional

[ Network.mesh.Link.n0s0 ]
  Source = n0
  Dest = s0
  Type = Bidirectional
...
;;;;;;;;;; ROUTING TABLES ;;;;;;;;;;
[ Network.mesh.Routes ]
  ...
  s0.to.n1 = s1
  s0.to.n2 = s1
  s0.to.n3 = s1
  s0.to.n4 = s4
  s0.to.n5 = s1
  s0.to.n6 = s1
  s0.to.n7 = s1
  s0.to.n8 = s1
  ...
```

To relax the amount of work, we provide the students the scripts to launch the applications from these two benchmark suites and students are asked to get an initial performance study of the system by measuring three main performance metrics: execution time, instructions per cycle (ICP), and network latency.

After this first contact with the simulator, students are asked to extend the baseline system to model specific microarchitectural techniques.

Example 2. Memory hierarchy configuration file.

```
[ CacheGeometry l1 ]
Sets = 128
Assoc = 4
BlockSize = 64
Latency = 1

[ CacheGeometry l2 ]
Sets = 1024
Assoc = 32
BlockSize = 64
Latency = 10
...
# Level 1 Caches
[ Module dl1-0 ]
Type = Cache
Geometry = l1
LowNetwork = net-0
LowModules = l2-0

[ Module il1-0 ]
Type = Cache
Geometry = l1
LowNetwork = net-0
LowModules = l2-0
...
# Level 2 Caches
#
[ Module l2-0 ]
Type = Cache
Geometry = l2
HighNetwork = net-0
LowNetwork = mesh
LowNetworkNode = n0
LowModules = mod-mm-1
...
# Main memory
[ Module mod-mm-1 ]
Type = MainMemory
HighNetwork = mesh
HighNetworkNode = n8
Latency = 200
BlockSize = 64

# Nodes
[ Entry core-0 ]
Type = CPU
Core = 0
Thread = 0
InstModule = il1-0
DataModule = dl1-0
...
```

As an example, they can be asked to model the most common prefetching mechanisms. Due to the complexity of this goal, students must accomplish it in three progressive steps.

To start with, students are suggested to vary the block size for the L2 caches of all the cores; but with two limitations, the new block size has to be multiple of the L1 block size and the total L2 cache size must remain the same. In this way, students should notice that as the L2 cache block size is increased, the system shows a similar effect to prefetching when a miss rises at any L1 cache. The effect is similar to prefetching because when a miss rises in both L1 and L2 caches, the L2 will request it to the main memory, but it will request the failing block and the next n blocks, since the L2 cache block is bigger than the level 1 one[1]. Keeping this in mind, students have to execute all the benchmarks varying the L2 block size in between 64B and 2KB, and they have to measure the aforementioned performance metrics. When the simulations finish, students have to study the system performance trend as the block size grows. Results must be delivered in a graphic similar to the one shown in Figure 2. In the figure, all the metrics have been normalized to the value obtained with a 64B block size.

D. Prefetching mechanisms implementation

After this first initial study that was designed to provide autonomy and self-confidence with the experimental framework to the students, they are asked to implement several prefetching mechanisms in the simulator. As already mentioned, the simulator is very complex and students are not still very skilled with it at this point, so we have opted to give them some implementations guides for illustrative purposes. In particular, we provide them an implementation of the most simple prefetching technique. Also, we give them a step by step guide on how to implement a new prefetching mechanism.

In particular, we provide implementation details of the *One Block Look-Ahead* (OBL) prefetching mechanism. In the step by step guide, we show how the core code must be modified. We have integrated a queue for the pending prefetches that is looked up at the pipeline *issue* stage, and if there is i) available enough memory, ii) issue width is not being used by normal memory requests, and iii) the memory can accept new requests, then a new prefetch is triggered. In addition, the memory hierarchy has been updated to add a trigger that jumps when a cache miss rises, the trigger in this case generates a new prefetch according to the OBL policy and inserts it in the corresponding queue for pending prefetches.

Once students become familiar with the implementation of this simple prefetching mechanism, they are in an advantageous position to perform their final course project. The following are example of these works:

[1]The value of n depends on the ratio between level 1 and 2 block sizes.

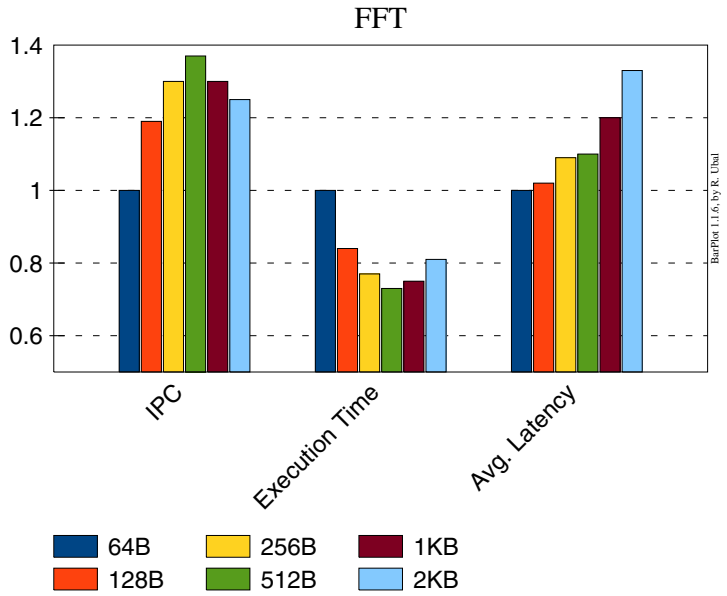

Figure 2. Performance evolution varying the L2 cache block size in the FFT benchmarks. All measurements are relative to the 64B configuration.

- **n-block sequential prefetching:** This work consists in extending the provided prefetching mechanism in order to prefetch not only one block in advance from memory but to bring n consecutive blocks from memory when a cache miss rises. n can be defined in the configuration files. Students must study the performance trends for all the benchmarks, by varying the cache block size for n 1, 2, 4, and 8.
- **n-block stride prefetching:** The difference between this work and the previous one is that instead of working with consecutive blocks, the prefetching technique works with regular strides, which in general provides better results since the previous work is a particular case of this one.
- **Sequential prefetching for different network topologies:** In this work, students must use the prefetching mechanism and change the topology of the network. This work was heavily focused on studying variations on the network traffic.
- **Sequential prefetching. Network evaluation:** In this work, the students change the values of the network parameters to study how they affect the system performance. These parameters are mainly link bandwidth, and buffer size at the switches.

After succeeding in these two courses there are several students that are really motivated in the study and or research on processor architecture. From these students, we select the ones that are really interested to improve their knowledge in a Master Thesis or Final Degree Project, and propose them some ideas to extend their work with the simulator to that end.

For instance, some examples of a Final Degree Project could be:

- **Adaptive prefetching:** In this project the research paper is extended to implement a new adaptive prefetching mechanism that checks the network status before injecting new prefetch requests and adapts its injection to the workload of the network.
- **Load to core scheduling:** In this project the student studies how to assign certain loads to specific cores in the system to maximize the overall system performance.

V. Conclusions

In this paper we have presented a new methodology that has been carried out during the 2011-12 academic year at the Universitat Politècnica de València to update the lab sessions on computer architecture providing a more realistic framework to the students. It is based on using a detailed simulator that allows us to model the main components of current microprocessors. As the simulator is complex, and the learning time is long, instead of being introduced in just one course, we amortize the learning time by introducing it along several courses.

The methodology has been designed in order to provide students a more realistic learning and sound understanding about internal mechanisms of current processors. This is accomplished by training students to achieve good skills with the simulator, which is achieved by introducing concepts in a progressive way through different courses. Contrary to most

instructors could think, results have probed that students are able to use the simulator in a reasonable amount of time and obtain very promising results that have been published in top-notch international conferences, like PACT and IPDPS.

Finally, from the instructors' point of view, now that this methodology has been already developed and the simulator framework has been deployed, we have built up a platform for future editions of the offered courses that will allow us to obtain a high number of works to help students to study processor architectures. These works will be carefully selected to motivate both students and instructors

ACKNOWLEDGMENT

This work has been financed by the Spanish Government under grants CICYT TIN2009-14475-C04 and TIN2012-38341-C04-04, and it has been also supported by UPV under grant PAID-05-12 with reference SP20120748.

REFERENCES

[1] Todd Austin, Eric Larson and Dan Ernst *SimpleScalar: An Infrastructure for Computer System Modeling*. IEEE Computer vol. 35(2), 2002.

[2] Trevor E. Carlson, Wim Heirman and Lieven Eeckhout *Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation*. Conference on High Performance Computing Networking, Storage and Analysis, page 52, 2011.

[3] Cloyce D. Spradling, *SPEC CPU2006 Benchmark Tools*. SIGARCH Computer Architecture News vol. 35(1), 2007.

[4] Rafael Ubal, Julio Sahuquillo, Salvador Petit and Pedro Lopez. *Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors*. 19th International Symposium on Computer Architecture and High Performance Computing, pages 62–68, 2007.

[5] S. Woo, M. Ohara, E. Torrie, J. Singh and A. Gupta. *The Splash-2 programs: Characterization and methodological considerations*. In 22th International Symposium on Computer Architecture (ISCA), pages 24–36, 1995.

[6] The Multi2Sim Simulation Framework Website, http://www.multi2sim.org