High Level Queuing Architecture Model for High-end Processors

Damián Roca Marí

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya - BarcelonaTech

A thesis submitted for the degree of

Master in Innovation and Research in Informatics (MIRI-HPC)

1st of July, 2014

Advisor: Mario Nemirovsky, ICREA Senior Research Professor at BSC-CNS
Tutor: Miquel Moretó Planas, Lecturer at UPC-DAC

To my parents for their support during all this years

To Mario, my advisor, for his patience, his support and his guidance.

Acknowledgements

Finishing this master thesis is the begin of a new stage in my life, my Ph. D. work. I want to thank all people who helped me during these years. I also want to thank to my colleagues and my friends for helping me to arrive here. And of course, I want to thanks Marc Casas and Miquel Moretó for their guidance and support during the development of this work.

This work has been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2012-34557, the HiPEAC Network of Excellence, and by the European Research Council under the European Union's 7th FP, ERC Grant Agreement number 321253. Author is supported for a PhD International Fellowship by BSC-La Caixa.

Abstract

Simulation has become a must in research. Since the real implementation of a processor is very expensive in terms of money and time, it is almost impossible to implement and test the different techniques under investigation. Therefore, simulation techniques are applied to a lot of fields, such as bio-informatics, computer architecture, and physics, among others. However, it is not only used in research. Companies develop simulators to save money and to maximize efficiency. Hence, simulators provide the environment where researchers can test their implementations and obtain a first evaluation of the impact they have before performing the real implementation.

Traditionally in the field of processor architecture, simulators are implemented with Hardware Description Languages (HDL) such as Verilog, or with high-level languages such as C. There are different models implemented, like execution- or trace-driven simulators. However these models are not perfect. Problems appear because these solutions are slow and they have become really complex. Hence, we face the Simulation Wall, where large simulations required for the massive multicore systems or supercomputers can take several days, even weeks. Moreover, if a researcher has to modify the simulator to add functionalities, it is not a trivial work anymore. Additionally, software also implies some difficulties. In the case of trace-driven, the main problem came with the size and complexity of the traces required to perform a good simulation.

In order to solve some of the problems mentioned above, we want to open a new way in the area of simulation. We want to offer a fast and still accurate tool that allows researchers to develop a more precise and quick development of their ideas. This new tool will be based on queuing models and statistical methods. Both techniques offer a great versatility in the development because it will allow much easier changes and tests. To achieve that, the model will emulate the behavior of the real processors. The enhanced scalability and adaptability will be characteristics of this model, plus the modular design. Finally, we will show a model of a specific processor to show the viability of this type of tools applied to processor architecture simulation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context of the Project

In order to preserve Moore's Law [25] architects and designers introduced enhancements to keep the improving performance. Processors have a lot of specific hardware that try to add more performance benefits. Such implementations can be cache memories, and branch predictions among others. Superscalar microarchitecture [32] was introduced to allow more than one instruction per cycle to be executed. Another improvement came with Tomasulo Algorithm [33], which allows non-sequential execution of unrelated instructions. It was a good improvement but with a very high cost in term of complexity, power and area. However, its real performance is determined by the consistency model, the dependencies of the instructions and the available functional units. Some years ago, multicore processors were develop because the single cores arrived to a limit in terms of maximum frequency and performance. Then, we moved from a very strong single core to a combination of smaller cores that can sustain the performance increasing required. These new architectures added some problems like cache coherence, communication, more congestion, etc. A very important issue was, and still it is, the parallel programing required to exploit the main characteristics of these architectures. Another technology developed in order to improve performance was the multithreading architectures [26]. It allows the execution of hardware- or software-defined threads which execute different programs or the same program in a parallel way. These threads can share resources or have it dedicated.

Researchers are working now with manycore systems and they are speaking about hundreds or even thousands of cores in a single processor. These Systems on Chip (SoC) bring more challanges. One of them is the communications. Now the implementation of a Network on Chip (NoC) [4] is mandatory to enable the correct functionality of the processor. There are different kind of NoC's implementations such as electrical, optical or wireless. Problems like bandwidth, network congestion, or multicast become important but it also bring some advantages like the possibility of having a natural broadcast channel among the cores (with wireless solution). Typical network problems like topology or routing come now to the processor domain, affecting its performance and scalability.

All the technologies and different implementations presented above add more performance in most of the cases, but they also make the processor much more complex. Companies and researchers have to deal with all this complexity when they try to develop new solutions and architectures that improve the actual performance. Hence, real implementation of research-based techniques is almost impossible due to the cost in terms of money and time required to achieve a functional implementation. Moreover, there is the doubt about the solution and its real performance once implemented. Since it is research, it can be that the final performance make that is not worthy to implement it. Then, a lot of resources would be expended in an useless solution. To avoid that cost and time penalty, researchers developed simulators that behaves like the desired environment but where it is much more easy to apply a modification and to observe the impact it has. It is true that depend on the accuracy they have, results will be more correct or less, but at least give a first idea of what can be better to implement in real prototypes and which not.

Simulators are used in a lot of fields like bio-informatics, physics, and computer architecture among others. Since they became a must, a lot of different simulations techniques have been developed and used. Depending on the detail level or the accuracy desired, we can find a wide variety of solutions. Traditionally in the field of processor architecture, simulators are implemented with Hardware Description Languages (HDL) such as Verilog, or with high-level languages such as C. Among the HDL area we find options like the Liberty Simulation Environment (LSE)[34], where there are libraries of predifined elements that help us to construct our processor RTL in an easily way. This kind of simulations are very accurate and fast. However, the level of detail implemented is very high too, since the designer has to define even the multiplexers and all the control signals. On the other side, we have the simulators based on high level languages. There are different models implemented, like execution- [6] or trace-driven [30][21] simulators. First kind, execution-driven, are normally cycle accurate executions with a lot of detail in the main parts of a processor like fetch, instructions executed, out of order and memory. Nowadays, they even allow the execution of an operating system and applications run on top of that OS, so final result can be accurate. The other kind, trace-driven, needs a first execution on a real machine to obtain information about the dynamic instructions to be executed or about memory addresses that will be read and write and also its order. Both types offer a quite accurate solution.

Currently, a lot of research articles are based on high-level language simulators. However they are not perfect. Problems appear because these solutions are slow and they have become really complex. Hence, we face the Simulation Wall, where large simulations required for the massive multicore systems or supercomputers can take several days, even weeks. Moreover, if a researcher has to modify the simulator to add functionalities, it is not a trivial work anymore. Additionally, software also implies some difficulties. In the case of trace-driven, the main problem came with the size and complexity of the traces required to perform a good simulation.

Another solution that tries to mitigate the time problem mentioned before are the statistical solutions. Main optimization is to use statistics to just execute a representative sample of the program in order to reduce the time of the simulation [13]. This can potentially reduce the time, but add other problems to the simulation. If we are simulating a multithreaded application, then it is necessary to ensure that the different sections of code of a specific thread will be executed at the same time that all the other sections of the rest of the threads. Reason is clear, try to provoke the potential cases of resource contention to capture the real behavior of the programs in real hardware. Another option is to use statistical simulation to perform the entire simulation, in most of the cases using traces that guide the program [27]. This last option is much more faster than the other but the problem is that you loose accuracy. It also can imply the use of a trace, with the consequent problems.

1.2 Motivation and Goal of this Thesis

The combination of all these techniques in a single simulator makes it a really complex software to deal with. Current solutions have problems with the integration of different existent simulators which were made for a specific part of the processor. As an example, we can see how processor and network are integrated. Or they develop and test a really good network with synthetic traffic or they implement a processor with a very inaccurate network. Integration of really well defined processor and network is not available. Solutions like gem5 [6] try to make it work but it presents a poor scalability (around 64 x86 cores). Trace-based solutions become really complex because now you need a trace from each of the cores that contains the dynamic flow of instructions to be able to reproduce it inside the simulator. Both techniques present another issue, the variability on each execution due to the current state of the machine. On the other side, we find analytic and statistic models. Analytical models are too simple to be used as a valid technique for validation. Statistical models are more accurate, but they also have inconveniences. If we use statical traces to be executed on a more detailed simulator we have the problems of generating the trace and also to use the cycle-accurate model. If we use a complete statistic model can be difficult to capture the behavior of a real chip.

Since most of the papers presented nowadays in the important conferences and journals are based on cycle-accurate models, let's evaluate them a little bit more. Its performance can be accurate, but even which is supposed to be one of its strongest points is not so good. The error achieved with these simulators can be between 10% and 20%. However, main problem came with the complexity and the time they require. If researchers want to a modification, it can become a nightmare. Simulators are not bug free, and since they are a combination of previous solutions, they are written in different languages. Then, you have to identify the part of the code you are interested in, and apply the modifications expecting that the other parts will work fine with your changes. If we suppose this process is completed successfully in a reasonable amount of time, now you have to run the experiments to try the performance of your implementation. Time required for the experiments depends on the simulation detail of the components (core and memory mainly) and the amount of cores you have. Simulations can take weeks where in real machine take minutes or hours. Then the issue is clear, researchers expend more time on programming and debugging the simulators than improving architectures and features for next generations processors.

In order to solve some of the problems mentioned above, we want to open a new way in the area of simulation. We want to offer a fast and still accurate tool that allows researchers to develop a more precise and quick development of their ideas. This new tool will be based on queuing models and statistical methods. Both techniques offer a great versatility in the development because it will allow much easier modifications and tests. To achieve that, the model will emulate the behavior of the real processors. Easy modifications will be possible because our tool is modular-based. Moreover, the enhanced scalability and adaptability are other characteristics of this model. Our tool will be capable of emulate any architecture family (ARM or x86), with a large number of cores and also including the network. Moreover, the achieved accuracy depends on the detail implemented. Users can choose which level of detail they want, even at module level. They will be able to choose a detailed module for cache memory and at the same time a simple module for the fetch unit. Just changing one module by the other the user will have more detail, with a really low impact on the simulation time. As far as we know, there is just a simple model of the SPARC T2 processor implemented in queues [37], done by Ruken Zilan under the supervision of Mario Nemirovsky. However, we will simulate more in detail the entire architecture.

Hence, our main objective is to develop a processor architecture simulator based on queuing models. We will create a detailed model of a specific processor to show the viability of this type of tools applied to processor architecture simulation. This model will emulate the behavior of the real processor and we expect an average error around 20% in the first results. Programs will be emulated in a statistical approach. We will develop a basic instruction flow generator, completely integrated inside the environment of the queue model. Remain as future work to reduce the error of the simulator and also to increment the accuracy of the instruction flow generated.

Chapter 2

State Of the Art

In this chapter we will describe the simulators developed in recent years in Section 2.1 and most important benchmarks in Section 2.2. The main objective of this master thesis is to present a state of the art of the simulators and benchmarks being used by the community in order to be able to put our simulation tool in context.

2.1 Simulators

Current simulators can be divided into a lot of categories. we can find execution-driven, and trace-driven models among others. Other categorization can be functional level or cycle accurate simulators. Functional simulators are able to execute the correct dynamic flow of instructions but they do not provide any correct timing parameter about the execution. They are like interpreters. In that sense there are more like the HDL models, which normally are implemented in a functional way to avoid problems with device timings (and gates or multiplexers for example). Cycle accurate simulators have a complete microprocessor pipeline with all the details and elements present in a real processor. They simulate the instruction flow through this pipeline providing the specific cycles it requires plus a lot of microarchitectural information, like cache misses, number of instructions retired, etc. In the following subsections we will explain in more detail some of the specific solutions implemented nowadays like gem5 [6], SimpleScalar [9], Simics [23], Graphite [24], Sniper [10], ZSim [31], Paraver [28] & Dimemas [21], TaskSim [29], HLS [27], and statistical simulation [13][16].

2.1.1 gem5

The gem5 simulator [6] is a merge of two previous simulators, M5 [7] and GEMS[8]. M5 is a modular-based platform which includes the processor microarchitecture details plus the system-level architecture. Its main characteristics are an object orientation, where major structures like CPUs or caches are represented as objects. Its configuration language helps to describe large systems to allow its simulation. Memory system implemented is event-driven, including caches and buses. Different models of CPUs are interchangeable. It also offer full-system capabilities for some ISAs and multiprocessor and multisystem capabilities. GEMS leverages the power of Simics to simulate a SPARC multiprocessor system. It offloads the correctness requirements of Simics, focusing more on accurate performance than correctness details. It has different modules which allow extra features, like out-of-order execution, branch predictors, execution resources, etc. Memory model is implemented with Ruby module, which allows the simulation of detailed memory hierarchies, including a good variety of cache coherence. They have some features to simulate a network, but not in a very detailed manner.

Combining all these elements, they support a lot of ISAs, like ARM, x86, Alpha, MIPS, Power, and SPARC. It also allow booting Linux on some of these families. Scalability goes up to 64-cores in the case of x86. Joining these features allow the simulation of the entire environment, going from the hardware to the software. However, parallelize the simulations is not possible, which makes that they require a huge amount of time.

2.1.2 SimpleScalar

SimpleScalar [9] is an execution-driven simulator with the possibility of a cycle-accurate simulation. This tool consists of assembler, compiler, linker, simulation, and visualization tool for the SimpleScalar architecture . With this set of tools, users can simulate real programs on some modern processors with a fast execution-driven simulation. It offers from a fast functional simulator to a detailed out-of-order processor with non-blocking caches and speculative execution for example. It is partially derived from GNU software tools and provide researchers with an extensible high-performance platform for processors systems.

2.1.3 Simics

Simics [23] is a full-system simulator used to run unmodified binary codes for the desired hardware at high-performance frequencies. It is fast, extensible, accurate, and scalable. It is really useful because it allows researchers to simulate any kind of digital system, from a CPU or FPGA to a full rack of components. It supports a lot of ISAs like ARM, MIPS, or x86 and also other digital devices like DSP. Scalability is one of its strongest points, allowing mixed architecture systems, hundreds of I/O devices and different network protocols in a single simulation. Hence, very complex and extremely large can be simulated to ensure its usability in real world environments. A remarkable capaiblity is the execution of a program in forward and reverse direction, allowing to solve bugs or mistakes than in another way are really difficult to discover.

The virtual hardware implemented with Simics runs the same binary software as the real target system. This feature allows researchers to create, debug and deploy software first on the real machine and later execute it in the virtual device. Since it is able to run production binaries, developers can use third party software to which they are accustomed.

2.1.4 Graphite

Graphite [24] is an open-source simulator for multicore systems. It allows highly distributed and parallel simulations. These features enhance a great scalability, offering the theoretical possibility to perform simulations with hundreds or even thousands of cores. Its construction is simple than other tools presented before because it was built from the ground. It is a tool for high performance fast design space exploration, including software development. To achieve this fast simulations it uses a relaxed synchronization together with seamless multicore, direct execution and a good distribution among different machines. This last point, distribution among machines, allows the accelerated simulations. This is a huge difference with gem5 for example, where the simulation can just be executed on one processor. It only simulates an application's user space code.

There is no need to modify the original programs, because it has the illusion of being a single process with a single address space. This non-intrusion capability is great because researchers do not have to modify the source code to execute parallel simulations. Moreover, it provides a consistent OS interface, and a threading interface (to ensure fairly distribution of threads over different machines). Network is simulated with a high-level approach, through messaging services between the different tiles. A TCP/IP and a shared memory are used to communicate the different tiles.

2.1.5 Sniper

Sniper [10] is based on the Graphite infrastructure and on the interval core model. Hence, it allows a fast and accurate simulation. Its main objective is to allow a really fast design space exploration in the area of homogeneous and heterogeneous multi-core architectures. Timing simulations are allowed for multi-threaded and shared memory applications. Scalability is one of its main characteristics, performing simulations of hundreds of cores with a really good speedup against other simulators. Speedup is obtained through the parallelization and the core model, based on interval simulation. Basically, interval simulations means that Sniper jumps between miss events to speedup the entire simulation.

It is useful for uncore and system-levels simulations in which the cycle-accurate simulators are too slow and the analytical models are too coarse to be used. Additionally, the interval core allows the visualization of the number of cycles lost for each of the processor's components, like cache memory or branch predictor. Then, researchers can observe which part is better to focus on. They model an OS, with a simple kernel lock contention model, which it is a difference respect to Graphite. To finalize, they remark a very important fact, to obtain an accurate solution only a few key architectural components need to be modeled.

2.1.6 ZSim

ZSim [31] is the last addition to the family of the architectural simulators. It comes with a clear idea, reduce the simulation time at the same time that keeps accuracy in quite enough boundaries. It is based on three techniques. First, the tool uses instruction-driven timing models, using dynamic binary instrumentation. Second, they use a two-phase parallelization that improves the scalability of the simulator. Basically they divide the total execution time in intervals of few thousand cycles. Then they perform a parallel simulation of the cores with zero latency and no contention for these intervals. Later they perform event-driven simulation applying the latencies, with the advantage of the knowledge obtained from first phase(e.g. interaction between memory accesses). Third, they provide a user-level virtualization to support complex workloads without full-system simulations. This is a key point because it allows to skip OSs and ISAs that do not scale until thousands of cores.

They claim to have a speedup over two or three orders of magnitude respect to the most used full-systems cycle-accurate simulators. They still have a very accurate result. However, since the simulator has been released just a few months before, it remains that the community test it and verify this huge improvement.

2.1.7 Paraver and Dimemas

Paraver [28] and Dimemas [21], together with Extrae, are a set of tools developed at the Barcelona Supercomputing Center (BSC-CNS) that allow a fast simulation and an comfortable interface to observe what happens in the architecture. Dimemas is the responsible of the simulation, and it is trace-driven based. Extrae allows the extraction of traces from the real hardware, based on performance counters that the user wants to analyze. Paraver is the visualization tool and it also helps to interpret and to perform operations of the information contained in the trace.

Let's focus on Dimemas, since it is the simulator tool. It takes as a input the trace and it allows to perform re-simulate to obtain the impact of possible changes on the architecture. For example, it allows to change the network pattern, the number of channels between sockets, and number of cores per sockets among other parameters. It is more focused on large cluster systems than individual cores. However, now that researchers are speaking about thousands of cores inside one processor, these collection of cores can be observed as a node of cluster and we can use Dimemas to perform the processor simulations.

2.1.8 TaskSim

TaskSim [29] is a trace-driven simulator. Its main target is the accelerator-based architectures in which the master cores offloads their charge to the available specialized hardware. These systems has a particularity, once the data is charged on the accelerator, there is no interaction between it and the cores. Just when the simulation is finished, there is another transfer of data back to the cores.

Based on this behavior, TaskSim abstracts the different CPU phases as a single delay, without any need of simulating the architectural details. Since they simulate the transfers of data with an accurate detail, there is no error introduced by the abstraction. Precisely, the mentioned abstraction allows the simulation of hundreds of elements in a reasonable amount of time. All the simulator is built with the principles of a discrete event simulator. Moreover, applications are instrumented to obtain the traces from the real execution.

2.1.9 HLS

HLS [27] is a hybrid simulator which combines statistical models with symbolic execution. This combination allows a fast and accurate simulations. It is clear that this tool is designed to perform a quick design space exploration. It takes as input a statistical profile of an application, and it executed this code in a symbolic way on the desired architecture. Use of synthetic code allows the modification of distance between instructions for example, what in real code would be impossible. By using these statical input, it is not executing the entire program. It just needs to execute a representative sample, which drastically reduces the simulation time. This flow is executed on a closely approach to the SimpleScalar tool set.

2.1.10 Statistical Simulation

Statistical simulation [13][16] presented in this papers is very similar to the presented in HLS. Main idea is to execute the program on a real machine, obtain a trace of that execution and at the same time extract the profile of that application (e.g. cache behavior, branch statistics, etc). Once this step is completed, they elaborate a synthetic trace with all these information. This new trace is then executed in a detailed simulator, like M5. Main reduction came with the reduced number of instructions that is simulated now, while if the trace contains enough features accuracy will be still good.

Main objective of these tools is a really fast design space exploration, just at the first stages of the design, both processor and system-level. Its accuracy is not so good as other solutions, but they claim it can be improved adding more details to certain parts of the simulator, while still keeping a reduced simulation time.

2.2 Benchmarks

A critical component in all the simulations are the benchmarks and programs that will test the new architectures and modifications. We are obligated to use synthetic benchmarks because the real applications normally are private and the source code and binaries are not publicly available. These are the main reasons why researchers have developed different set of tools that help to test different parts of processors, like memory intensive benchmarks. Another solution are micro-benchmarks that help to test really specific parts, but they are like toy-tests and are not used to prove the correct execution of architectural features. Then, in the following subsections we will explain the most used benchmarks, like Spec CPU2006 [19], Parsec [5], Splash 2 [36], and CloudSuite [14].

2.2.1 Spec CPU2006

Spec CPU2006 [19] is one of the largest and most significant collection of benchmarks. This benchmark suite comprises of 12 integer and 17 floating point. It is a suite of serial programs and it is not intended to study parallel computers. Its main objectives are to test some of the corners for CPUs, memory subsystem, and compilers. Most of them are writen in C++ and Fortran. It has been very used by the academia and also by the industry.

2.2.2 Parsec

The Princeton Application Repository for Sahred-Memory Computers (PARSEC) [5] is a benchmark suite for studying the multicore processors. They developed a new suite trying to emulate the behavior of new applications that stress different parts of the processors, such as memory hierarchy or dependencies among instructions. Its key points are to simulate multithreaded applications, capture the emerging workloads, present a diverse set of applications, be actualized respect to the current state-of-the-art, and support research. It is not focused on high-performance computing.

2.2.3 Splash-2

Splash-2 [36] is a suite of multithreaded applications but focused more on high performance computing than chips multiprocessors and graphics. Its key points are concurrency and load balancing, size of the working sets, communication to computation ratio, and spatial locality.

2.2.4 Scale-Out

Scale-out [14] are the last addition to the set of available suites. It tries to emulate the behavior of data centers applications and cloud-based solutions. This new suite is used to show the inefficiency of the commodity hardware to execute these set of benchmarks showing that the performance obtained is not equivalent to the area and the power used. Its key points are that this suite suffers from high instruction cache misses, instruction- and memory-level parallelism is really low, size of working sets is huge (much more than the capacity of cache) and bandwidth requirements are low.

Chapter 3

Simulation Tools

In this chapter we will first describe the main network simulators 3.1, which can be potentially used to develop our tool. We will introduce their main features, advantages and disadvantages, and if it is possible discuss future developments. We will then conclude this chapter with a discussion 3.2 about which simulator is the chosen one and main reasons for this election.

3.1 Network Simulators

Since the tool we will develop is based on queue models and statistics, we focus on the network simulators that fulfill all the requirements. Basically we need an environment that allows the simulation of queues, servers, networks, and availability of good mathematic libraries. In the first validation of the models, networks will not be necessary since we will simulate a single core processor. However, in future steps we will establish a network of cores, then it is really important to have this capability from the beginning. To have a great scalability is critical need since the simulations that we want to perform can have thousands of elements. If the simulator allows modular design is a desirable characteristics (it will facilitate its use in the future). We also want a lightweight environment to simulate fast, avoiding large times like cycle-accurate simulators. Additionally, if it is an open source will be better since its main target is research activity. In the following subsections we will explain in more detail the most important simulators under use nowadays, like Opnet [11], NS2 [20], NS3 [18], and Omnet++ [35].

3.1.1 Opnet

Opnet [11] is a registered commercial trademark from riverbed (they acquired the OPNET corporation). It is one of the most popular and famous simulator. It has been used for a long time in industry, so it is a very mature solution. They offer a free license for research in academia. Its software is specialized for network development and research. It offers a great flexibility in the areas of protocols, communication networks, applications, and devices. Since it is a commercial solution its GUI and programing environment are much powerful than other solutions available. It offers a graphical interface, which helps in the construction of the networks. Moreover, it follows an object-oriented programming model to define the functionalities of the modules that can be used and implemented with the graphical interface.

This tool is based on a discrete-event simulation. Then, the system behavior can be simulated through modeling the events generated by the system, always keeping the correct order. Modules can be structured in a hierarchical way, facilitating the simulation of very large systems and networks. Users can define its own protocols and packets, or use the vast libraries they provide with most of the current protocols.

Hence, Opnet is a tool for modeling, simulating, and analysis of the information obtained from the first two uses. Its main features are summarized in the following list.

- Discrete-event simulation
- Analytic simulation
- Excellent graphical interface
- Huge collection of libraries with existing protocols and networks
- Hierarchical approach
- Debugging and analysis tools included in the GUI
- Free license for research

- Object-oriented programming model
- User can define new functions, protocols, and packets
- 32-bit and 64-bit versions
- End-to-end visibility into applications
- Capture and analyze NetFlow data.

3.1.2 NS2

NS2 [20] is the second version of Network Simulator. Its main target is networking research. It is an open source project, reason why it is widely used in academia and research. As the previous simulator, this tool is a discrete-event simulator and uses and object-oriented programming model. Language is C++ and OTcl (Tcl script language). Main reason to use two programming languages is the different characteristics between them. While C++ is efficient with the design, it fails with graphical interface. For OTcl, it is the opposite. Then, they bot complement each other offering a great characteristics to the users.

NS2 separates the data path from the control path. Event scheduler and basic network components are written in C++, while OTcl script is used to start the scheduler, set up the network, and to control the packets generators for sending packets or stopping it. It is also hierarchical and offers the possibility of defining users modules and protocols or use the predefined ones. Remark that the scheduler is the responsible of the simulation time and to release the events in the correct time to be processed.

3.1.3 NS3

NS3 [18] is supposed to substitute NS2, but it is not an updated version of NS2. This is the main reason why we have two different points to explain them. It is also an open-source project. Moreover, it is a discrete-event simulator like all the other network tools. Its main target is research and academia.

NS3 core is now written in C++ and Phyton, instead of OTcl like in NS2. Protocols are now designed to be more realistic because the try to follow real network behavior. They have improved its software integration, facilitating the integration of other open-source projects and the re-use of components. As one of its major improvements, now it supports virtualization (until now just lightweight VMs). Additionally, the tracing and statistics collection will be merged to allow customization of the output without rebuilding core's simulation. Its main features are summarized in the following list.

- Discrete-event simulation
- Object-oriented programming model
- Graphical interface
- Hierarchical approach
- Debugging and analysis tools included in the GUI
- User can define new functions, protocols, and packets
- Modular core
- C++ programs
- Phyton scripting
- Virtualization
- Models updated
- Still under development

3.1.4 Omnet++

Omnet++ [35] is open-source, like NS2 and NS3. It is a discrete-event simulator, like all the other simulators presented in this chapter. Additionally, it offers a component-based network, basically because it is a component-based architecture entirely. Like Opnet, it offers a graphical interface from where users can control and configure their networks in a simple way. One of its key points is its generic and flexible architecture. Then its easy to use it for other areas like hardware architectures, IT systems, and queuing systems.

Module functionality can be defined in C++, which offers a great versatility. From basic modules, developers can define really complex modules and structures in a hierarchical manner by using a high-level language. This language has a similar function to Phyton in NS3. Thanks to the modular structure, re-usability of existent modules is great.

This tool offers a framework approach because it provides the infrastructure to develop different simulators. Integration of other simulators into the OMNET environment is supported, which adds a lot of extra functionalities. It also helps developers to adapt their existing solutions to something more open via OMNET tools. It has a good library of existing protocols and networks, but not so vast as the Opnet simulator. Gather of results and statistics from the simulations is integrated inside the GUI, which facilitate a lot developer's work. Its main features are summarized in the following list.

- Discrete-event simulation
- Object-oriented programming model
- Graphical interface
- Flexible architecture
- Hierarchical approach
- Debugging and analysis tools included in the GUI
- User can define new modules, protocols, and packets
- C++ programs
- High-level language for scripting
- Basic tool to develop other simulators on top of it
- Integration of other open-source projects

- Pre-existent libraries
- Really good documentation

3.2 Discussion

There is not a perfect tool, but it is very clear that Omnet++ fulfill most of our requirements. First, let's describe why we have discarded the other options. Opnet would be an excellent choice, but the the fact that it is a commercial solution its a weak point. Main reason is because we want to use our tool as research, and normally open-source projects have a better penetration. If we use NS2, we will be developing our tool based on a simulator that will be deprecated by the NS3. Moreover, NS3 is still young and lack of developers community and acceptability from the researchers, which is a critical point speaking about simulators.

Omnet++ is our choice because it offers a great flexibility to use it for hardware architectures, queuing models, and statistical models. It is modular-based architecture, which is great to design processor architectures. Users can define a simple module for a memory, or redefine this memory module with much more detail. Thanks to the GUI interface, if we create a good library of processors elements (e.g. caches, fetch units, issue queues, etc), changes can be performed in seconds. Users just need to change one box for another. Moreover, the possibility to define our own modules is perfect because it can improve simulation in both time and programming effort. From the open-source projects, it presents a mature community, which is really important for us to give credibility to our model.

Chapter 4

Queue Model

The main objective of this master thesis is to develop a processor architecture simulator based on queue models and statistics. Omnet++ is the chosen platform, as we have seen in previous chapter. The tool has to be open-source and easy to extend. A current processor is a really complex structure with a lot of details. Since this is the first approach, we will focus on the most important parts of a processor. To determine it, we have studied and observed which main penalties determine the performance of a processor.

In this chapter we will explain how to create a processor simulator based on queue models in Section 4.1. Application generator will be described in Section 4.2. We will then conclude with a brief explanation of main bugs we had to fix in Section 4.3.

4.1 Processor Architecture

In this Section we will explain how to implement a generic processor simulator based on queue models. In Figure 4.1 we show a really high-level approach to a processor block model. Remember we are building our tool from scratch. Then, it is really important to define a solid base. One of our key points is that users will be able to control the complexity, then we have to start from the very basic concepts of computer architecture. Basically, this image gives us the basic idea of what we need to simulate in our first approach.

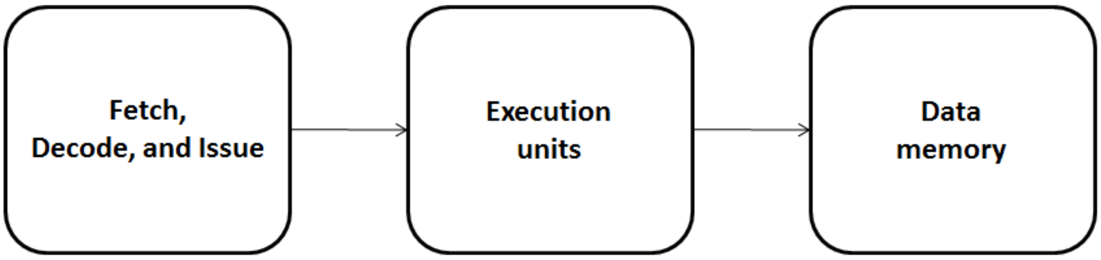


Figure 4.1: High-level processor block diagram

There are a lot of components missing, like branch predictors, load and store buffers, etc. Three boxes presented includes a lot of different elements in reality. For example, the first box is fetch, decode, and issue. These three parts are different basic components of each current processor. However, since this is a high-level simulation and we are emulating the real behavior, we will merge it (at least by now). Depending on the accuracy obtained, these module can be separated into three different ones. Another example observed is the third box, data memory, which includes the entire data memory hierarchy, from level 1 cache to the main memory.

Now we have a first idea of what can be an acceptable first implementation of a processor model, in next subsections we will describe what modules we need from Omnet++ environment and how to use them to simulate the behavior of a real processor.

4.1.1 Omnet++ Basic Blocks

In this subsection we will describe the basic blocks users need to define their processor models.

4.1.1.1 Source

Figure 4.2: Source module in Omnet++

Source module is the responsible to generate the instruction flow. This element is explained in more detail in subsection 4.2.

4.1.1.2 Queue

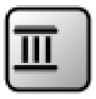


Figure 4.3: Queue module in Omnet++

Queues are present everywhere in modern processors. As an example we have queues in fetch stage, issue queues, retirement queues, etc. This a basic module but it is really necessary. Omnet++ provides this module with basic functions to read & write objects, insert, pop, among other possibilities. An important parameter to control is the length, since we cannot loose instructions and in the real processor most queues have a finite length due limitations of power and area.

There are different types of queues, but for now we will use the First Input First Output (FIFO). Basically we want to keep the arrival order of the instructions.

4.1.1.3 Delay

Figure 4.4: Delay module in Omnet++

Delays are really important because a processor element is in fact a delay applied to an instruction. For example, if we are executing a integer instruction, the functional unit will apply a "delay" of 1 cycle. Another example can be a memory access, where the delay applied depends on the level accessed (moreover it is not a fixed value).

4.1.1.4 Single-server Queue

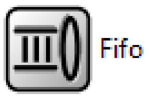


Figure 4.5: Single-server queue module in Omnet++

A single-server queue module is a combination of a queue and a server module. When a new packet arrives, if the server is busy it goes to the queue. If it is free, then the packet can be processed. Same constraints we applied to the single queue apply here, length is really important. On the server side, service time is really important since determines the throughput of this module.

4.1.1.5 Compound Module

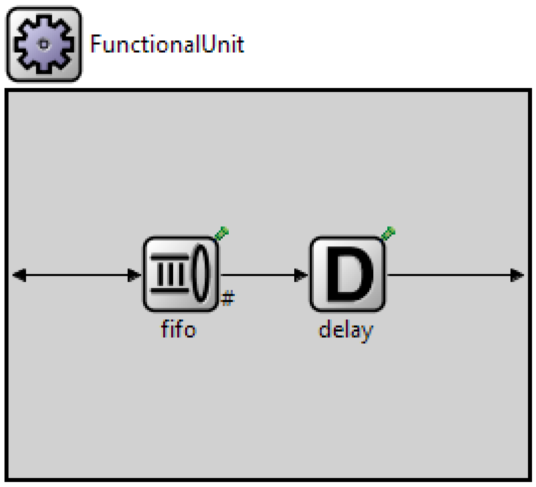


Figure 4.6: Example of compound module in Omnet++

Compound modules can be defined by developers to combine characteristics of different simple modules, like the case of the single-server queue. Using these modules helps the hierarchical and modular design, key aspects of Omnet++. It also helps to encapsulate different functionalities inside a single box from the GUI perspective.

In Figure 4.6 we show a compound module we have defined. This compound module is a key element in the simulation environment. It is composed of a single-server queue and a delay module. Gathering these two modules offers a great advantage, developers have available a module that allows the simulation of pipelined elements in the processor. Moreover, depending on the configuration values they can choose which kind of module it is, without any need of re-programming it. In Table 4.1 we detail the possible configurations for this module. In the first case, one cycle unit, the behavior is exactly like a single-server queue. Same happens for the second case, just changes the value of the service time. In the third case, mixed structure, we have a part of a processor which is pipelined and another part which is not. Then, the pipelined part is configured with the delay and the non-pipeline with the service time . Last case, fully pipeline, is a slightly modification of the previous one. Remember that a pipeline unit is capable of processing a new element each cycle, just as the structures we explained before.

Desired element	**Configuration**
Once cycle unit	serviceTime=1, delay=0
No pipeline	serviceTime=total, delay=0
Mixed	serviceTime=total-y, delay=y
Fully pipeline	serviceTime=1, delay=total-1

Table 4.1: Configurations for a specific compound module

4.1.1.6 User-defined modules

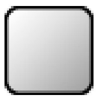


Figure 4.7: User-defined module in Omnet++

In case some element processor functionality cannot be described with one of the previous modules, Omnet++ offers a great solution. Developers can take an empty module and define the desired functionality in C++. This characteristic is very powerful and helps a lot. These modules written in C++ are in function similar to more conventional simulators. However, this modules are just a part of our tool, not all of them. Even if we have a lot of user-defined modules, its functionality can be much simpler than conventional simulators because we are emulating the behavior of the processor (not simulating it with a huge detail).

4.1.1.7 Messages and Packets

In Omnet++ environment messages and packets are going from one module to another, provoking the execution of some predefined functions. In most of the cases, reception or sent of messages are the events that guides the simulation. Remember that Omnet++ is a discrete-event simulator.

There are a default version of messages implemented in Omnet++. However, since our messages will be treated as instructions, we have defined out own message kind to facilitate the entire simulation. In Table 4.2 we can see an easy example of a self-defined instruction message. Some of these values are provided by Omnet++ environment natively, like id and time.

Instruction variables	Description
numReg	Number of registers it needs
dependency	Indicate if it depends on another instruction
type	Describe the instruction type
id	Unique id to identify
time	Lifetime of the instruction

Table 4.2: Variables inside an instruction message

As we have created this instruction type, it is also possible to define more types. For example, in case of an acknowledge message to indicate the retirement of an instruction, we have defined a message with the retired id and instruction type. Future developers can simply modify the types to whatever they believe more convenient with a minimum effort because the entire tool is prepared to work with other message types. Just is necessary to specify which object casting is necessary and that's all.

4.1.1.8 Sink

Figure 4.8: Sink module in Omnet++

In Omnet++ all the messages and packets need to eliminated from the simulation. Main function of sink module is to receive the messages once they are not useful anymore and to eliminate them. This task is critical because if we do not eliminate the messages, system will run out of memory and simulation will break.

Other functionality is to recover statistics. Its critical to recover information about executed instructions, like total lifetime, instructions distribution, etc.

4.1.1.9 Signals

Signals have a lot of functions. For our model, it is the basic tool to recover statistics from any module in the simulator. Developers can configure them so when a user specifies its model he/she just needs to activate it in the configuration. Then, GUI will automatically store the information. Once the simulation has finished, users can process that information with Omnet++ IDE.

4.1.2 Processor Implementation

Now we have the basic tools, we can proceed to explain how to emulate the behavior of a processor. We want to remark we will emulate the behavior, so forget about current cycle-accurate simulators where everything has to be detailed in a perfect manner.

Our main goal is to reproduce the behavior of main elements of a processor as simple as possible. We have chosen the specific parts we want to simulate based on its impact on the final performance. Hence, our main targets will be memory hierarchy (from caches to main memory), dependency model, and fetch, decode & issue. In the next subsections we will explain one by one how to simulate certain key components of a processor.

4.1.2.1 Clock

In modern processors there are different clock areas. In current cycle-accurate simulators it is very important to keep a consistent time. It should be that these tools allow to have different cores operating at different frequencies. In reality there are a lot of problems with the interaction of this different frequencies. Moreover, it is necessary to know exactly what happen in each clock cycle. This fact adds more complexity to the tool.

We have a great advantage over this kind of tools. We are emulating, so notion of cycles is much more relaxed for us. It is true we need to know how many cycles the processor has been working. Since Omnet++ is a discrete-event simulator, we have been using an event as our clock cycle. We do not have problems of synchronization or different domains because Omnet++ scheduler keeps it under control. Events always follows the correct order with respect to the global simulation time.

4.1.2.2 Fetch

Fetch stage is the responsible to interact with the instruction cache and to serve the correct instructions to the decodes. They normally have buffers to receive lines from instruction cache and take the proper instructions into a ready queue, from where instructions are sent to the decoders. Prefetchers between instruction cache and fetch units can play an important role in performance depending on the application.

In our tool we have different options. One of them is simulating the ready queue of instructions. Once the queue is empty you can ask more instructions to the instruction cache or if you do not simulate this cache level, directly to the source. Another possibility, is adding probabilities and statistics about the instruction cache behavior. If there is a miss, then you flush the ready queue and ask for more instructions. This case will emulate the case where you have a cache miss and there are memory latencies involved. The last case also applies to the branch instruction. In case of branch misspredicted, the ready queue should be flushed. In case of normal branches, maybe the desired address is not in the lines and we need to ask memory again. This fact will produce a new penalty that can be considered. As a summary, fetch stage will have a queue and some actions can be performed over it emulating the possible penalties.

4.1.2.3 Decode

Decode stage is the responsible of translating the instructions in binary format so the logic can be able to execute it. Number of decoded instructions per cycle is critical in modern processors. Nowadays, most processors available in the market are superscalar, which means that can execute more than one instruction per cycle. As an example, the ARM A15 has eight functional units and Intel SandyBridge has five. However, to have as many functional units does not mean anything if you do not have instructions to execute. That's why decodes play a critical role. Once instructions are decoded are sent to the issue queues.

In our tool, most easy way to emulate the decode behavior is via controlling the number of instructions can go to the issue queues in the same cycle. If developers want more control over decode stage, you can statistically control how many instructions you send, avoiding to be a constant during the simulation. In case of RISC processors you can keep it with the maximum value, but in CISC machines since there are simple and complex instructions will not be so accurate. Throughput can be controlled by adding more detail to fetch stage too. If fetch just sends one instruction to the decodes, decode stage does not need more control than the maximum value.

4.1.2.4 Registers and Renaming

In this stage processor checks the registers need for each instruction-. It also check dependencies and try to avoid them when it is just a “mistake” of the programmer who always uses the same registers by renaming them. Modern processors have a pool of virtual registers. This pool of registers is not visible to the programmer, so it is not included in the ISA. Renaming registers is also used in the case of out-of-order processors. Since out-of-order can be pure speculation, processor cannot allow the modification of the original register. Then it assigns a virtual register and in case the speculation was correct that value modifies the original register.

In our tool this stage can be substituted with an statistical and probabilistic control over registers. For example, we can keep control of on fly instructions and depending on its type we know the number of registers it uses. This step avoids a lot of complexity to the simulator.

4.1.2.5 Dependency Model

Dependencies are fixed by the program we are executing. Modern processors just apply some optimizations in register renaming. However, out tool is based on statistical information, and until now the registers of each instruction are not emulated. This is the main reason why we need to introduce a dependency model that try to follow as much as possible the real behavior of the processors. To be fair, this dependency model is fixed for a specific processor.

Our tool is configured to facilitate the change of the dependency model. Dependency can be determined as a constant, or with a formula taking into account the number of registers under use in that simulation time. Once you know an instruction will depend on another, we can determine from which one with probabilities. More complex models can be implemented, as much detailed as the developers want. In case the user wants to use a trace from real execution and instructions have the real registers, there is no need to implement that dependency model because it will be included with the instructions.

4.1.2.6 Issue Queues

Issue queues are where instructions waits until they are sent to the functional units. Normally, a processor can take one instruction from every queue. What happens in reality is that same type of functional units share a queue. In that case, as many instructions as functional units of this kind can be extracted.

In our tool each functional has it own queue because they are implemented together with the functional unit. Look next subsection for further details on implementation.

4.1.2.7 Functional Units

Functional units are the responsible of "executing" the instructions. They take instructions from issue queues and execute them as fast as they can. Normally, functional units are made for a specific type of instruction. Normally each instruction type has a fixed processing time. As an example, in the case of an integer instruction is one cycle, but in case of multiply or divide this number can go to tens of cycles. Modern processors have a set of functional units that try to guarantee an acceptable performance. This group of units allows the execution of all instruction types. They are not present in the same number. For example, there are more integer functional units than branch ones. There are a limitation, if codes have a strong dependency, it will not be useful to have a lot of functional units. Hence, it is necessary to achieve a compromise between area and power with modern code needs.

In our model we implement issue queues together with each functional unit to simplify the design. Its reasonable to use an issue queue for each. To implement functionals units we use the basic block described in Figure 4.6. We add by default the possibility of pipeline the units because certain modules (e.g. multiply functional unit) are pipeline. However other units like the ones dedicated to integers do not need this capability. We think to offer a generic module is really comfortable for users since they just need to specify what they need in the configuration file.

4.1.2.8 Load and Store Buffers

These buffers actuate like an intermediary between the load and store units and the first level of cache. They try to avoid the blocking of the pipeline . If you want to do an store you just send this store to the buffer. In some moment, stores in the buffer will go to memory. Advantage is that maybe instead of single values we will write a hole line. They apply some constraints. For example, a load before going to memory has to check the store buffer to see if there is a most recent value. If you find it there, perfect because you will not go to memory. If it is not there, you go to the cache.

In our tool developers can use the queue module to implement it. However, they will need to introduce some probability about the success ratio of finding a value in the store buffer that we want to load. In the first development, we do not consider them critical.

4.1.2.9 Retirement Queue

Since most modern processors have out-of-order issue but in-order retirement this retirement queue is completely necessary. It guarantees the order of retirement. If the instruction two completes before the one, it waits in this queue until the first one finishes. Then, both instructions can retired in order.

Since our simulator works with statistical flows of instructions and data, we do not need a retirement queue. Remember that until now our instructions do not contain any specific registers and that dependencies are modeled with functions. Then, maintain a strict order has no sense.

4.1.2.10 TLB

TLB is the responsible of translating addresses from virtual space to physical and also for security reasons. Depends on the real architecture if the translations are done before accessing the cache (physical addressing caches) or after (virtual addressing caches). It also depends on the page size configured on the real machine. Obviously, its performance also depends on applications.

In our tool we can simulate its impact with a single-server queue or with a functional unit module. Following the current hardware implementations of TLBs, we can implement a hierarchy with different levels of TLB tables. The traductions can be done simultaneosly with the execution of loads or stores. However, if we find a miss in TLB, that instruction will see a bigger penalty and should be executed again after this TLB penalty. If TLB is a hit, then instruction just observes the correspondent memory latency.

4.1.2.11 Memory Hierarchy

For memory hierarchy we are referring from caches to main memory. In most modern processors there are different levels of cache memory, ones private for each core and others shared among them. We will describe them in a very generic way. Important factors to simulate are the number of ports, if they allow read and write at the same time, latencies, accesses in consecutive cycles, among other parameters.

- Cache levels: our tool is not aware of associativities or word size in its simple memory module implementation. It receives as input the cache behavior of the desired application. Then, user will have to introduce the miss ratio for the desired levels. Take into account that these values can be obtained with performance counters in real chip. This cache levels can be implemented with a combination of a functional unit module plus a user-defined one. The first module allows to control how many requests can be accepted in one cycle, if it accepts requests while it is processing a miss, etc. Second module is used to determine if we have a miss or a hit. Hence, it determines if the request will go to the next cache level or to the sink to retire the instruction. To model the number of ports a cache level have, we implement different functional units that goes to the same user-defined cache controller. In Figure 4.9 we show an example of two-port generic cache level. This can be representative if we have a port for read and another for write. Read instructions will be directed to one functional unit while stores will go to the other. Of course developers can configure the ports to obtain a better mapping with the reality.

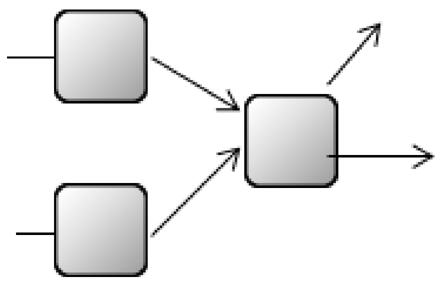


Figure 4.9: Generic two-port cache level

- Main memory: implementation of this level is very similar to the cache level, but with one main difference. The controller now just goes to the sink module since disk accesses are not modeled now.

4.2 Application Generator

We think application generator deserves a special section because it can be a research topic by itself. Main objective of the generator is to be able to reproduce with an acceptable accuracy the original instruction flow. As input it receives the percentage of each kind of instructions, for example 20% loads, 10% stores, 20% branches, and 50% integers. Depending on the detail developers want, more instructions types can be included or maybe less. More parameters can be introduced to produce more realistic outputs.

Main objective behind this simulator is to provide a generic instruction flow that can be used to validate architectures without the need of programming large codes. If users want to reproduce more realistic code, this generator can produce the registers used for each instruction, or the memory addresses for the load & store operations. For example, this addresses can come from an input trace or it has been recollected via pin tools. Then, dependencies will be generated from the source eliminating the need of dependency models inside the processor model.

In the first development, we have a very simple generator. It takes as input the percentage of each instruction type and generates instructions based on that probabilities. Its throughput is determined by the specific implementation. For example, in a system with instruction cache, this generator will be the responsible to fulfill the instruction memory. If we do not have instruction cache, then the source can be connected to the fetch module directly. Number of instructions sent each cycle is determined by the bus width that connects the instruction cache with the fetch unit. In the future we want to improve this implementation being able to reproduce specific patterns or sections of applications. However, this work remain as future work.

4.3 Debugging Issues

Main issues we find implementing these generic modules in Omnet++ was the manual garbage collection required for the IDE to avoid memory problems. If we do not control the number of messages in the simulation, system runs out of memory soon. User-defined instruction messages helped a lot avoiding problems with objects additions due to problems with deprecated methods in the new releases of Omnet++. A small problem we had was the requirements of generic modules which can be used for as many things as it possible. This point complicated the design, but once we had clear how to map the architecture into basic blocks it was not a problem but a great advantage.

Chapter 5

Validation and Results

In this section we present a specific implementation of a real processor with our queue model. In Section 5.1 we describe the architecture of the chosen processor. In Section 5.2 we focus on benchmarks used to validate the model. Then, in Section 5.3 we detail how we implemented the model with the blocks explained in previous chapter. Finally, in Section 5.4 we discuss results obtained and also compare them with previous work.

5.1 Intel Ivy Bridge Processor

We have chosen an Intel x86 processor. More specific, it is the Ivy Bridge. We know that a RISC processor is easier to simulate with statistical information because a lot of parameters for the ISA, like instruction length, are fixed. Moreover, since the instruction set is smaller, control the corner cases is easier. However, we think if we can provide a good validation for a more complex processor the credibility of our model will be higher.

In fact, for our model is independent to simulate an ARM or an Intel. Developers just need to use the basic blocks described previously to follow the real hardware. Remember that notions like instructions, complex structures, etc are vanished because we are emulating the processor, not simulating it with a huge detail. Main reason behind that sentence is that developers are capable of simulate the main parts of a processor in a generic way, the ISA implemented is not so important (for high-level simulation). Since we have to prove its validity, we decided to simulate the Intel Ivy Bridge because we have the real processor. Then, we can experiment and execute benchmarks to know specific details that maybe are not revealed in the microarchitecture manuals.

5.1.1 Overview

We will briefly describe the architecture of the Intel Ivy Bridge. Taking it as our baseline, we will be able to proceed with the queue model implementation.

Intel follows a Tick Tock policy. Basically it means that they change processor architecture keeping the same integration technology (Tock) and they keep the processor architecture when they change the integration technology (Tick). Then, each year there are a single improvement by changing architecture itself or by changing the integration technology. In Figure 5.1 we show the Tick Tock developing model for Intel processors until 2012. The most remarkable thing is that the Ivy Bridge has the Intel Core microarchitecture of Sandy Bridge. There are just a few differences between these two cores, but from the point of view of the model they are the same (just some configuration values can be different). Further information can be found in the Intel Architecture manual [12].

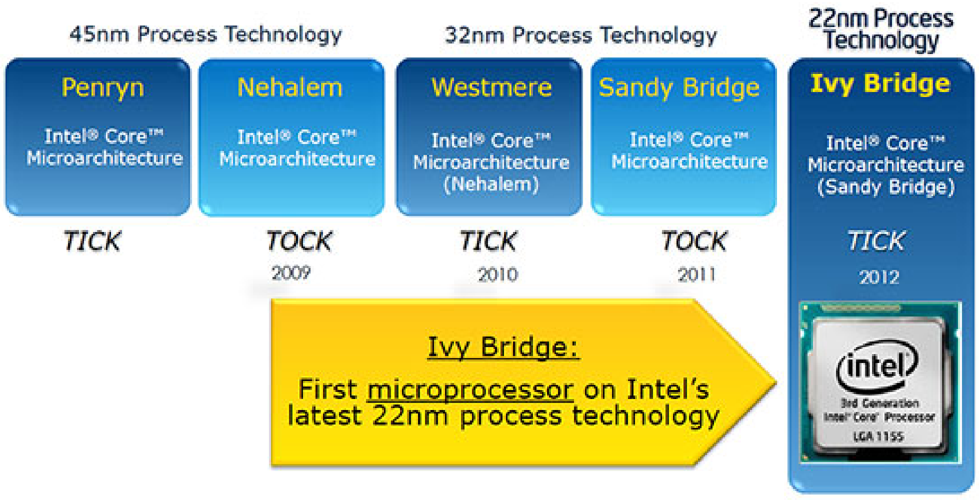
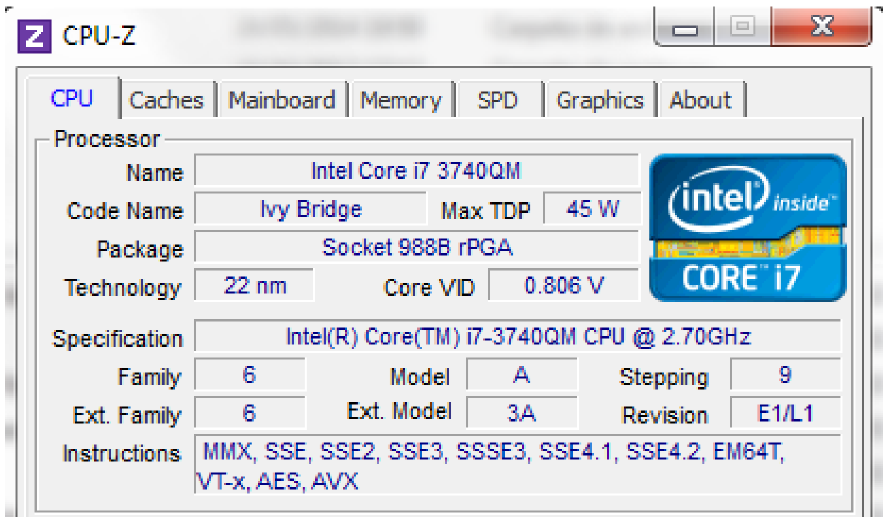

45nm Process Technology
32nm Process Technology
22nm Process Technology
Penryn
Intel® Core™ Microarchitecture
Nehalem
Intel® Core™ Microarchitecture
Westmere
Intel® Core™ Microarchitecture (Nehalem)
Sandy Bridge
Intel® Core™ Microarchitecture
Ivy Bridge
Intel® Core™ Microarchitecture (Sandy Bridge)
TICK
TOCK
2009
TICK
2010
TOCK
2011
TICK
2012
Ivy Bridge:
First microprocessor on Intel's latest 22nm process technology
intel
3rd Generation Intel® Core® Processor
LGA 1155

Figure 5.1: Tick Tock developing model for Intel until 2012 [3]

Now we have chosen the target microarchitecture to study it is time to describe it in more detail as it will determine our model implementation. In Figure 5.2

we show the block diagram for the core microarchitecture. Please note that the values indicated for the reorder buffers, TLB entries, and load & store buffers can be different. However, the main structures are the same. Remarkable details are the presence of three ALUs, where one of them is capable of executing branch or jump instructions. In our first deployment we will not simulate floating point instructions (later we will justify it in more detail). An important parameter is that decoders can send up to four instructions to the issue queues. From memory point of view, this architecture can execute two loads in one cycle, or one load and one store. Note that the Address Generation Units (AGU) work with a store data module in case of stores (imagine as one entity where you can calculate the address and indicate the data you want to write).

In Figure 5.3 we show specific details of the processor we want to simulate, like model or family . To obtain this information we have used CPU-Z benchmark [1].



Name: Intel Core i7 3740QM
Code Name: Ivy Bridge
Max TDP: 45 W
Package: Socket 988B rPGA
Technology: 22 nm
Core VID: 0.806 V
Specification: Intel(R) Core(TM) i7-3740QM CPU @ 2.70GHz
Family: 6
Model: A
Stepping: 9
Ext. Family: 6
Ext. Model: 3A
Revision: E1/L1
Instructions: MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX

Figure 5.3: Detailed information about the Intel Ivy Bridge used

5.1.2 Execution Timings

An important value for our tool is the number of cycles it takes to execute instructions. In case of the Ivy Bridge most of ALU operations can be executed in one cycle. Since in the first implementation floating point and vector instructions

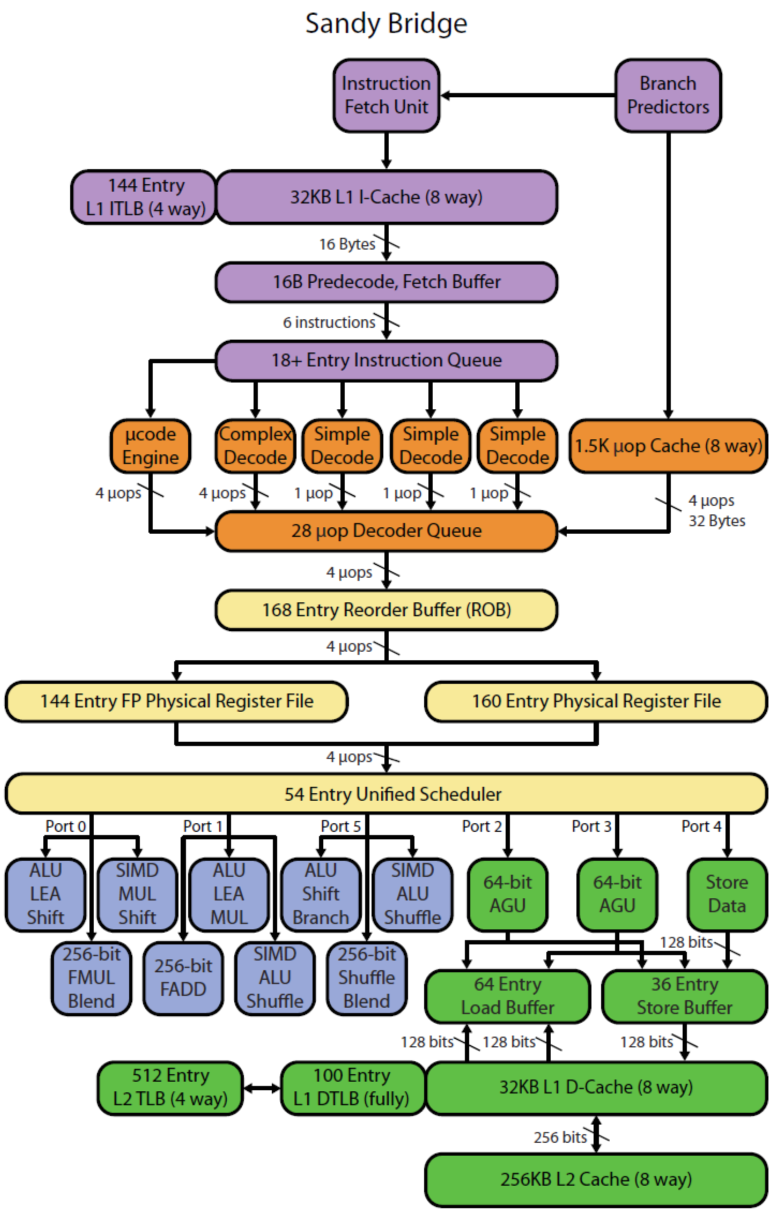


Figure 5.2: Intel Sandy Bridge microarchitecture block diagram [2]

are not implemented, we will not detail how many cycles they need. Instructions that will require more than one cycle are multiplications and divisions. However the probability of having these instructions in the code depends on the application. Hence we will determine if we implement them or not depending on the chosen benchmarks.

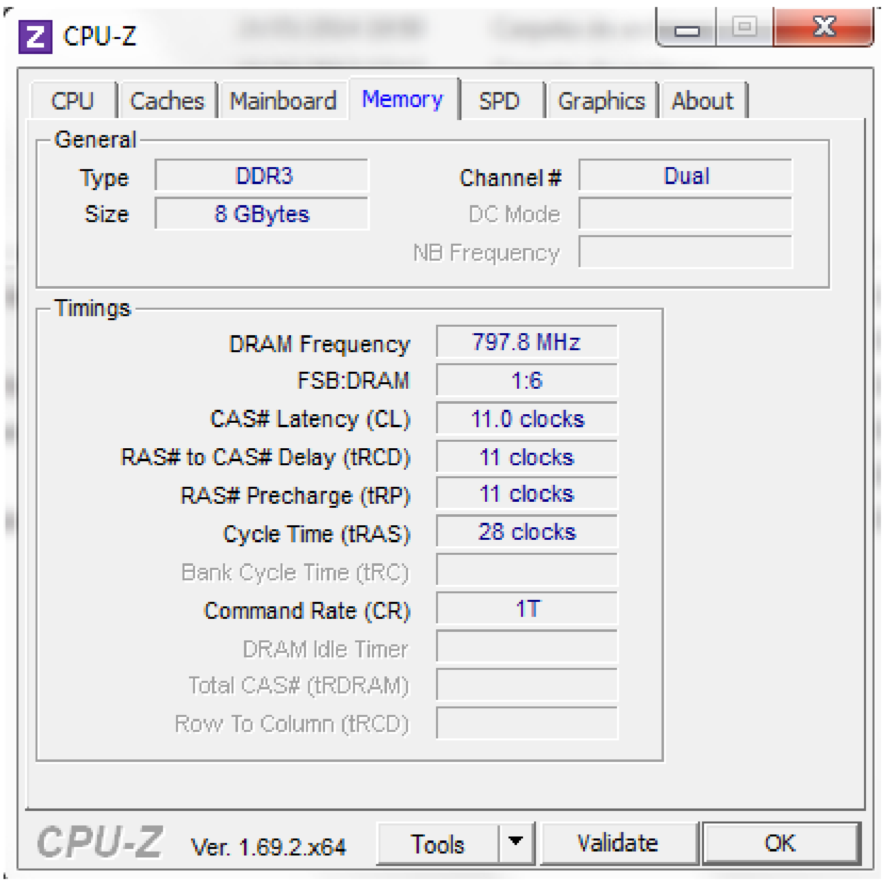
5.1.3 Memory Subsystem

The memory subsystem is one of the main components that determines the performance of real applications. In Table 5.1 we specify the cache hierarchy organization and most important details for each of them. We specify the size, associativity, latency, type, and line size. All that information was obtained and later validated with the Agner Fog microarchitecture manuals [15].

Cache level	Size	Line size	Associativity	Type	Latency
Level 1 - Instructions	32 KB	64 B	8-way	Private	4 cycles
Level 1 - Data	32 KB	64 B	8-way	Private	4 cycles
Level 2	256 KB	64 B	8-way	Private	11 cycles
Level 3	6 MB	64 B	12-way	Shared	28 cycles

Table 5.1: Cache configuration and latencies in Intel Ivy Bridge Processor

Main memory is also important to determine the performance of the processor. In order to show the real characteristics of the processor we want to model, we can observe the measured values for the DRAM most important parameters in Figure 5.4 .


CPU-Z

General

Type	DDR3	Channel #	Dual
Size	8 GBytes	DC Mode	
		NB Frequency	

Timings

DRAM Frequency	797.8 MHz
FSB:DRAM	1:6
CAS# Latency (CL)	11.0 clocks
RAS# to CAS# Delay (tRCD)	11 clocks
RAS# Precharge (tRP)	11 clocks
Cycle Time (tRAS)	28 clocks
Bank Cycle Time (tRC)	
Command Rate (CR)	1T
DRAM Idle Timer	
Total CAS# (tRDRAM)	
Row To Column (tRCD)	

Figure 5.4: DRAM timing parameters

Remember that the first deployment of our simulator just covers until main memory. Then no analysis of disk characteristics is presented here.

5.2 Benchmarks Selection

In Section 2.2 we have described the most used benchmarks nowadays. In Table 5.2 we have a further explanation about the INT benchmarks of the suite. Since we want to validate our model with a real set of benchmarks and not with microbenchmarks, we have to choose an appropriate suite. We have to take into account limitations of our initial deployment of the queue model, like not considering floating point instructions. Our model will not simulate multithreading by now, so some benchmarks can be discarded. PARSEC and Splash-2 benchmarks are designed for multithreading, so we discard them. Additionally, Scale-out benchmarks are really complex programs with object programming and a lot of system calls, which it is really difficult to reproduce for our initial application generator. Then, our natural choice is the integer benchmarks of the SPEC CPU2006 suite. Now we have a set of benchmarks we will obtain the profile information required for the model. In the next subsections, we will describe how we obtained the required data and results.

Table 5.2: SPEC CPU INT 2006 benchmarks description [19]

400.perlbench	Devired from Perl V5.8.7	C
401.bzip2	Julian Seward's bzip2 version 1.0.3, modified to do most work in memory, rather than doing I/O	C
403.gcc	Based on gcc Version 3.2, generates code for Opteron	C
429.mcf	Vehicle scheduling. Uses a network simplex algorithm (which is also used in commercial products) to schedule public transport	C
445.gobmk	Plays the game of Go, a simply described but deeply complex game	C
456.hmmer	Protein sequence analysis using profile hidden Markov models	C
458.sjeng	A highly-ranked chess program that also plays several chess variants	C
462.libquantum	Simulates a quantum computer, running Shor's polynomial-time factorization algorithm	C
464.h264ref	A reference implementation of H.264/AVC, encodes a videostream using 2 parameter sets. The H.264/AVC standard is expected to replace MPEG2	C
471.omnetpp	Uses the OMNet++ discrete event simulator to model a large Ethernet campus network	C++
473.astar	Pathfinding library for 2D maps, including the well known A* algorithm	C++
483.xalancbmk	Transforms XML documents to other docs using a modified Xalan-C++	C++

5.2.1 Evaluation Methodology

Methodology is really important to know exactly how the data was obtained. Since we want to avoid imprecisions with published results, we will collect our own profile information directly on the desired real processor. Then, we will control the environment to know exact characteristics like compiler type, specific configurations, etc. One important fact to take into account is the variation that exists in real executions due to different machine states. To mitigate this effect, we perform three executions and then calculate the average of the values obtained. We also have the standard deviation to control possible errors in some of the executions. To obtain the desired information we will use the hardware counters available in Intel processors. These counters allow us to obtain cache behavior, instruction mix, TLB behavior, branch prediction effectiveness, among other parameters. To collect the values from the hardware counters we have used the *perf* tools, provided with Linux OS [22]. Executions are performed on an OpenSuse distribution. NAtive input size is used. Moreover, executions are fixed to a specific core, since our model just simulates one core now.

5.2.2 Profile Information

In this subsection we will present the detailed information obtained through hardware counters that will be used as input for our model. In Table 5.3 we can find all the probabilities we obtained. In this initial steps of our tool, we have each instruction probability, instructions per cycle, and cache behavior (for each level). In the case of cache information, level 1 and level 2 statistics are just for the data part, while level 3 statistics includes load and stores together. Additionally, if users have cache statistics with differentiation between loads and stores, our simulator can take it as an input too.

Benchmark	**IPC**	**Instruction probability**				**Cache behavior**		
		Branch %	**Load %**	**Store %**	**Int %**	**L1 miss %**	**L2 miss %**	**L3 miss %**
perlbench	1.86	22.14	26.59	12.93	38.34	0.44	0.16	0.03
bzip2	1.43	15.41	29.99	11.32	43.28	1.46	0.66	0.02
gcc	1.11	23.91	23.44	17.42	35.23	3.11	2.16	0.36
mcf	0.35	21.67	32.27	10.01	36.05	14.72	9.25	3.8
gobmk	1.15	18.99	24.5	13.30	43.22	0.57	0.21	0.03
hmmer	2	4.93	44.35	15.87	34.86	0.68	0.03	0.00
sjeng	1.37	23.37	24.39	9.31	42.93	0.28	0.06	0.04
libquantum	1.54	24.28	13.6	4.77	57.36	3.27	2.20	0.89
h264	2.53	9.90	39.09	7.56	43.45	0.4	0.1	0.00
omnetpp	0.56	24.79	28.07	16.58	30.57	3.73	2.93	1.5
astar	0.84	15.9	34.33	12.43	37.34	3.15	2.19	0.75
xalancbmk	1.34	29.31	29.88	6.69	34.11	2.82	0.72	0.27

Table 5.3: SPEC CPU INT 2006 integer benchmarks profile information

5.3 Queue Model Implementation

Once we have clarified the architecture we want to implement, it is time to use the Omnet++ basic blocks described before to develop the Ivy Bridge queue model. In Figure 5.5 we have the overview of what has been implemented via GUI. This model takes as inputs the values of Table 5.3 except the IPC, plus the values determined for the execution units, memory latencies and other parameters described in the previous sections. Between them we can find the number of registers this emulated processor has, if it is an in-order machine or an out-of-order (Ivy Bridge is out-of-order, but we offer this possibility to observe its impact on performance) and the size of the out-of-order window. All this data is introduced in the model via a configuration file. Then users do not need to modify the code. As output, the model will give us the estimated number of instructions per cycle. Hence, we can proceed to compare the real IPC with the simulated one, obtaining our simulation error.

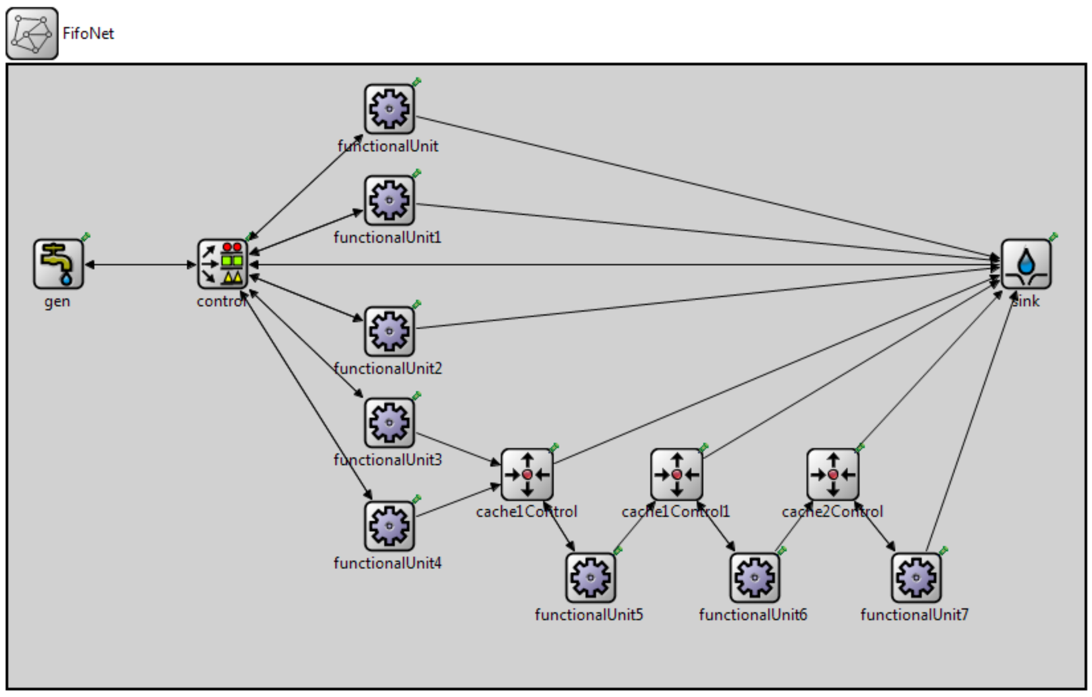


Figure 5.5: Intel Ivy Bridge queue model

Now we will proceed to explain each module in more detail. You will see in our first implementation we have done a high-level simulator of real processors and results will be quite good for the complexity we have. Just have in mind that this tool is designed for a very fast design space exploration, not for a very detailed simulation.

- **Source:** is the application generator module. In the first implementation we keep a really simple generator. It receives the instruction probabilities and produces a flow that keeps that behavior. There is no chain generation or specific patterns. We generate a random number with an exponential distribution and depending on the range it is, we decide one instruction type or another. For example, we can choose between integer, branch, loacd, and store. At the beginning it generates instructions by itself. However, after the first cycle it just sends instructions when receive a request from the control module. Each request is attended with four instructions to control module.
- **Control:** is the fetch, decode and issue elements all together. It is the responsible of having a ready queue of instructions, from where we will send them to the correspondent functional unit. This queue has a maximum number of instructions and it sends a request to source when detects less than minimum value. Moreover, control module is the responsible of keeping control of instructions on the fly, keeping its ids until they are retired. It also keep control of dependencies, since it has this on fly list. Dependency model implemented is a formula that takes into account the number of registers used by on fly instructions. As a result we obtain the probability of dependency. With an exponential distribution we generate a random number, observe if it falls inside the range of dependence and apply the correspondent actions. In case of dependency, we randomly gives the instruction from which is depending. User can define in-order or out-of-order just changing the value of a variable. In case we have in-order and we detect dependency, it will block the control until the acknowledge of the retirement of depending instruction. If it is out-of-order, then control can look into the window defined by the user if it can send more instructions. If not, it will restart on next cycle. Its throughput has a limit, which is

the number of functional units and also the real limitations of the processor (e.g. we cannot sent two stores in one cycle).

- **Functional units:** As you can observe this generic module is used for almost everything. We use them to simulate the execution units and also for the memory levels latency. They have each a configurable execution time, queue length, and delay. Users can define these parameters based on the real architecture ones. Units for integer or branch are configured with a service time of one. For memory, we measured the average access time with lmbench [**?**]. We cannot use the default values because in real machines not all memory requests observe the same latency. Imagine the case of a miss that brings a line, so next miss to consecutive address is a miss too but the penalty is less because the line was on its way to the cache. Remark the fact of having two modules for memory because we have two ports that allow two loads or one store and one load per cycle. Higher levels of cache are configured in pipeline mode, so in consecutive cycles they can accept new request, just like in real hardware.

- **CacheControl:** is a user-defined module that receives as input the cache behavior and it decides if a request is a hit or a miss. Decisions are made with a value obtained from an exponential distribution and a range defined by the probability of miss or hit. This module is prepared to work also with cache behavior specified for loads and stores in a separate way. The module after the last memory level (main memory) is a bit different because all request goes directly to the sink module.

- **Sink:** It is the responsible to "retire" instructions. It also sends and acknowledge message to the control to notify the retired id from the on fly list. It also recover some statistics, like a control of the instructions probabilities, life time of instructions, etc.

- **Missing elements:** We know we omitted a lot of elements from modern processors. As an example we have branch predictors, data prefetchers, micro-ops cache, etc. We did on purpose to show that we do not need to simulate everything to obtain a result that gives an idea of what is happening. Depending on the accuracy achieved we should consider the addition of more elements, always in order of importance for the final performance. For example, our next step can be including a more detailed dependency model or TLBs.

5.4 Results

5.4.1 Validation

Using the processor and the benchmarks explained before with the native input set. We will compare the real IPC with the simulated one. We will also present the percentage error between them, indicating the sign. In case the error is positive means our simulator is optimistic, if it is negative is pessimistic. An important comment about benchmarks is that we will present the plots with the complete integer subset. However there are a few benchmarks that can give a huge error due to their characteristics. We used hardware counters to study which potential benchmarks can be out-layers. We find that mcf and omnetpp have an important percentage of TLB misses and gobmk has a considerable instruction cache miss. Then, these three benchmarks are excluded from the average calculations. In further implementations of our simulator we will add these extra features and then they would be correct.

In Figure 5.6 we have the IPC comparison between the real measured IPC and the result obtained from our model with the in-order processor. In this case we have used an out-of-order window of 5 instructions and the number of registers is 12. Please, take into account that the number of registers in the simulator is used to ensure the correct behavior of our dependency model. It is correlated with the real number of registers of the targeted processor [12] but it is not exactly the same. In the case of the size of the out-of-order window it is not the same number as the real machine because in the simulator these number is used to cover a few aspects of the real machine.

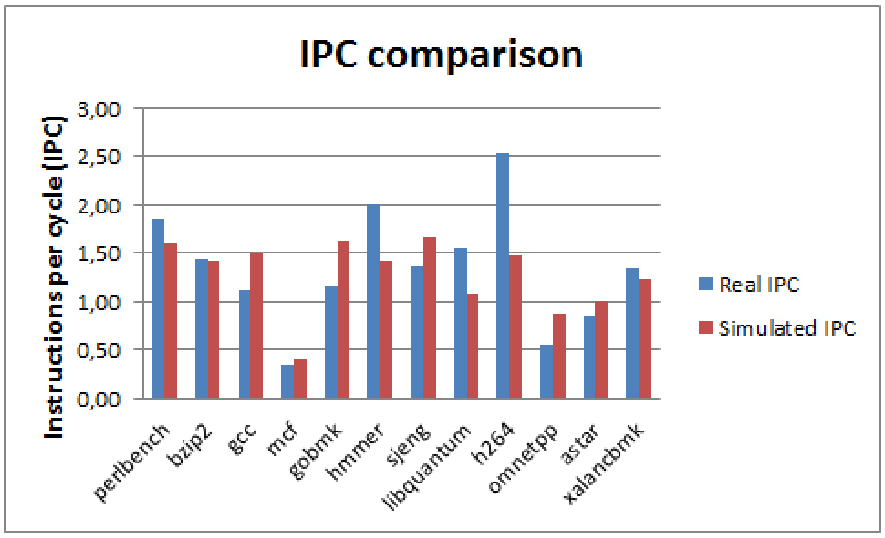


Figure 5.6: IPC comparison with out-of-order processor model (reg=12 & window=5)

In Figure 5.7 we have the same plot but expressing the relative error. Note that also the average of the absolute values is presented. As we can observe, we have an average error around 22%. As we expect, omnetpp and gobmk give us a really optimistic result because we are not simulating the TLB neither the instruction cache. In the case of mcf, result is quite good because we are optimistic in the TLB (we do not apply any penalty) and we are pessimistic with the dependency model. Then, one thing compensates the other. Remarkable cases are hmmer, libquantum and h264. We have studied the codes and we have seen that they are not codes with strong dependencies. Hence, our dependency model applies too much penalty and that's why we obtain such pessimistic results. One of the possible reasons is real compiler optimizations to avoid some load chains for example, while we do not have them yet.

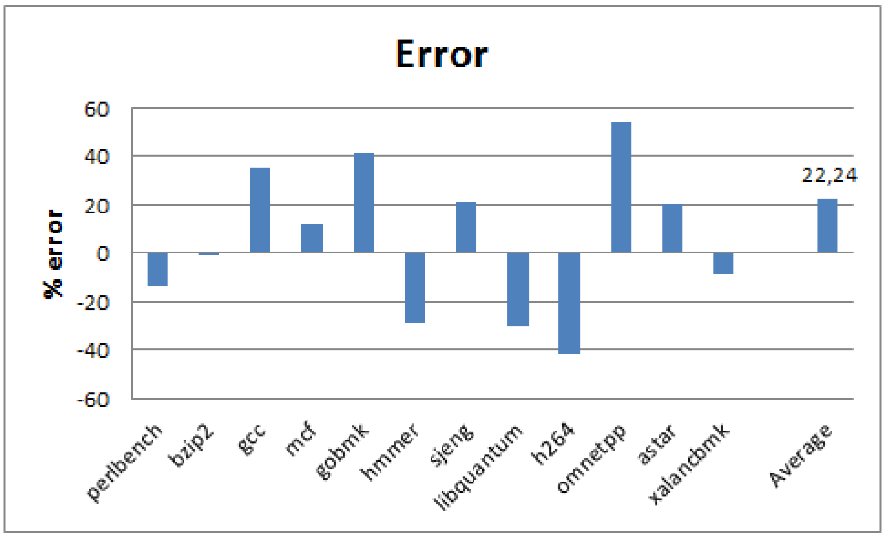


Figure 5.7: Relative error with out-of-order processor model (reg=12 & window=5)

5.4.2 Design Space Exploration

Once we have seen that our simulator is able to reproduce the behavior of a real processor it is time to perform a design space exploration. We will observe the impact of different changes over the performance of the simulator compared to the real machine.

In Figure 5.8 we can see the comparison of IPCs and in Figure 5.9 we have the percentage errors and the average. I. Now, IPCs are all more optimistic due to bigger out-of-order window size. However, average error is really similar because some applications improve and others get worse. Take into account that we have a window of 10 instructions and number of registers is kept to 12.

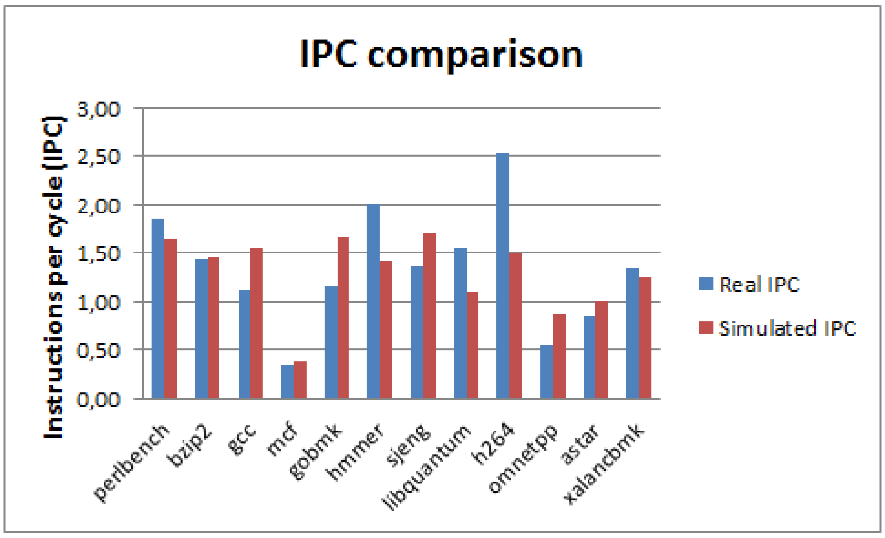


Figure 5.8: IPC comparison with out-of-order processor model (reg=12 & window=10)

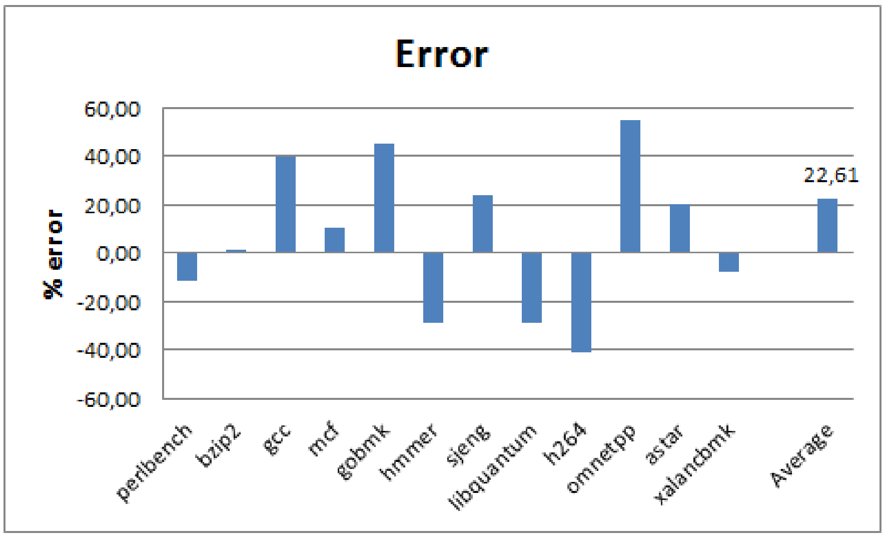


Figure 5.9: Relative error with out-of-order processor model (reg=12 & window=10)

To determine when a simulation is finished in our tool, we need to observe that the desired parameter value is stable in time. For example, here we are simulating the IPC. Then, once the IPC value is stable in time we can stop our simulation. If we keep simulating, result will not be more accurate. This characteristic reduces simulation time a lot.

Through the previous execution we will show how fast is our simulator. In Table 5.4 we show a set of parameters related to the execution of each benchmark in the suite. We have the number of simulated cycles, the total instructions "executed", the execution time required to simulate the correspondent benchmark, the IPC after this execution time, and the final IPC (obtained after a bigger execution time). To make a fair comparison between the different benchmarks we have used the same execution time for all, 3 seconds. After this time we measure the IPC provided by the simulator and we compare it with the IPC obtained from a larger execution. Through the difference between IPCs we can determine if 3 seconds is enough time to obtain our result for the entire benchmark suite. We can observe that in all the benchmarks 3 seconds of execution is enough to obtain an accurate result. In some of them even less execution time provides an accurate result. Additionally, it is remarkable that with the same execution time each benchmark has a different number of simulated cycles and total instructions. Reason is because different instruction mix provokes different latency penalties. As an example, we have mcf application. It has a lot of memory instructions, hence the simulated time is the biggest because memory penalties are the most costly.

Benchmark	Sim. cycles	Total inst.	Time (s)	IPC-3s	Final IPC
perlbench	240210	394299	3	1,6415	1,6431
bzip2	160540	231508	3	1,4421	1,4461
gcc	151858	235032	3	1,5477	1,5472
mcf	723667	277733	3	0,3838	0,3869
gobmk	142106	236224	3	1,6623	1,6666
hmmer	160554	227977	3	1,4199	1,4219
sjeng	192105	325790	3	1,6959	1,7017
libquantum	293476	316287	3	1,0777	1,0902
h264	205730	306136	3	1,4880	1,4866
omnetpp	352350	303733	3	0,8620	0,8653
astar	311515	316024	3	1,0145	1,0096
xalancbmk	264789	326681	3	1,2337	1,2383

Table 5.4: Execution time required by our simulator

Now, we will continue with the design space exploration. Hence, we will simulate the out-of-order processor with a bigger out-of-order window. In Figure 5.10 we have IPC comparison with a window of 15 instructions. In Figure 5.11 we have the percentage error. In both number of registers is kept to 12. We observe that the tendencies are more or less the same, so just a few applications are sensitive to the out-of-order window. Average error is still around the same value, twenty-two.

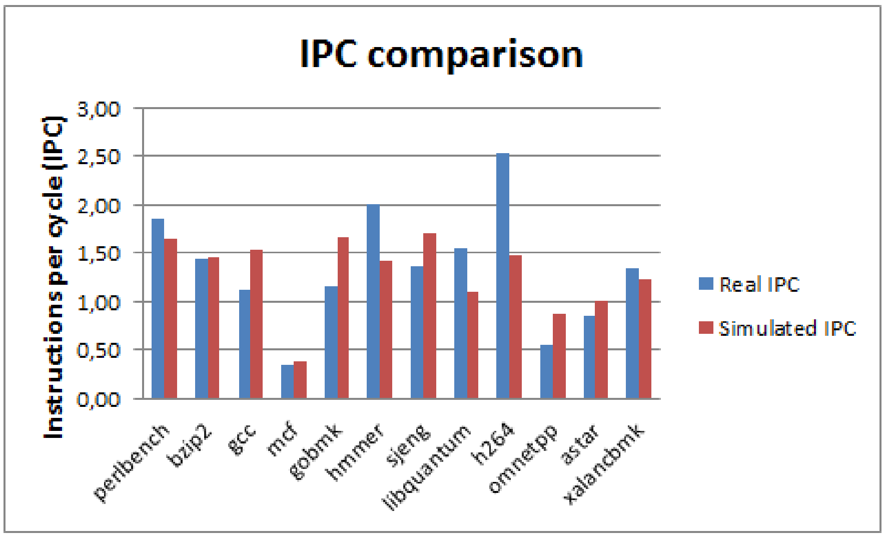


Figure 5.10: IPC comparison with out-of-order processor model (reg=12 & window=15)

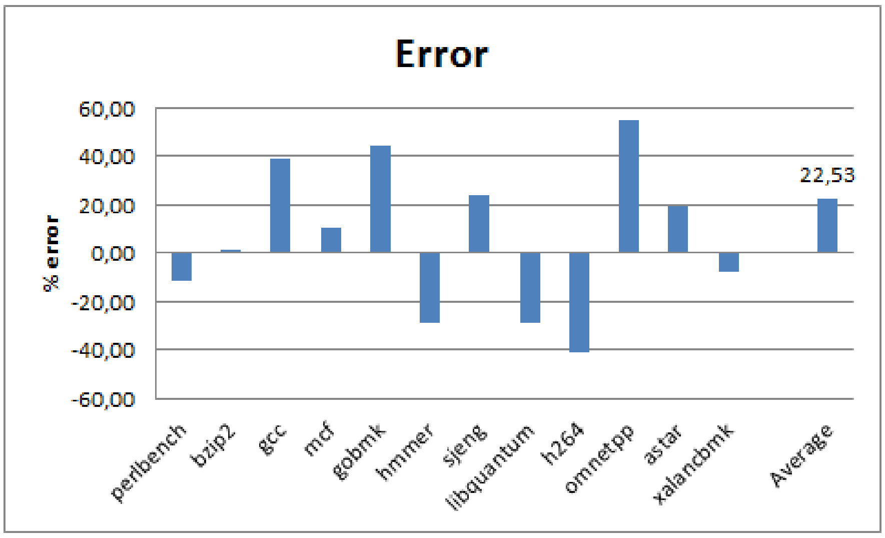


Figure 5.11: Relative error with out-of-order processor model (reg=12 & window=15)

It is time to try other modifications to observe the impact it has on accuracy. It is clear that the dependency model is a key component in the design of our simulation tool. Here we have two sources of errors. One is introduced by our application generator that it is not able to reproduce the instruction flow yet. It just keeps the percentages introduced but not the specific sequences. Secondly our dependency model is a formula which uses the number of registers under use. Then, we will modify the number of register available from 12 to 18. In Figure 5.12 we have the IPC comparison and in Figure 5.13 we have the average error with this new value. As we expect, now most of the errors are optimistic because by changing the number of registers we have relaxed the dependence.

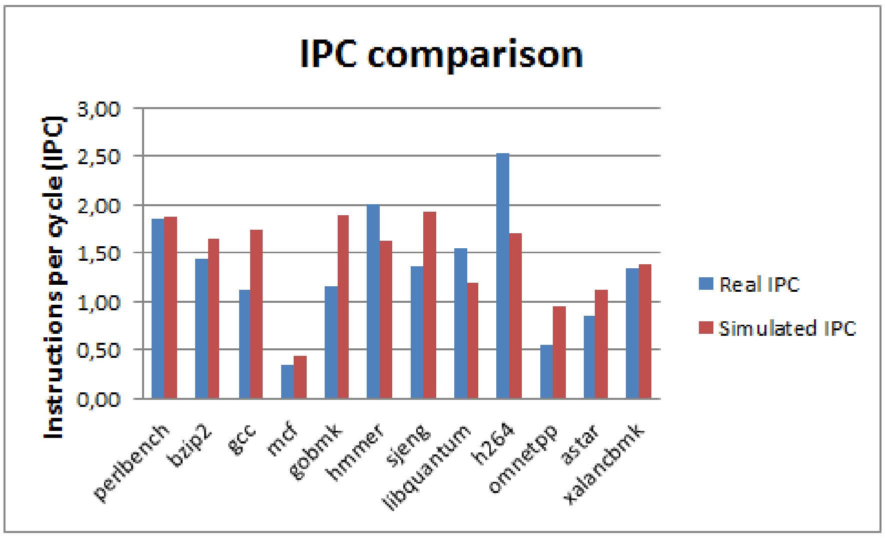


Figure 5.12: IPC comparison with out-of-order processor model (reg=18 & window=10)

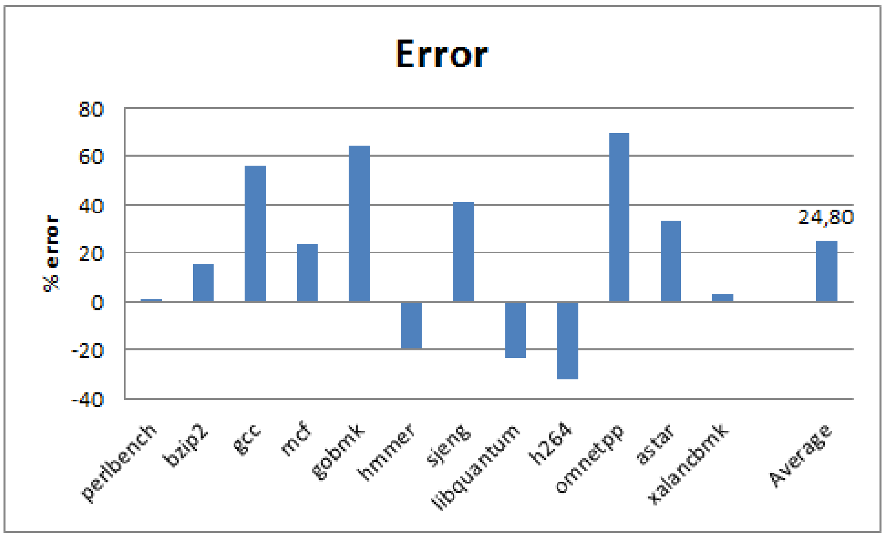


Figure 5.13: Relative error with out-of-order processor model (reg=18 & window=10)

Another option we have is to modify the probability of dependency to observe what happens. It is not a fair simulation but it is really useful for a space design exploration. Now we change the dependency formula and we substitute it for a fixed value of 0.5. Results can be observed in Figure 5.14 and in Figure 5.15. Now most errors are pessimistic, which means that the fixed value is too high for most of them. Then, we can conclude dependency model is critical to achieve a good accuracy.

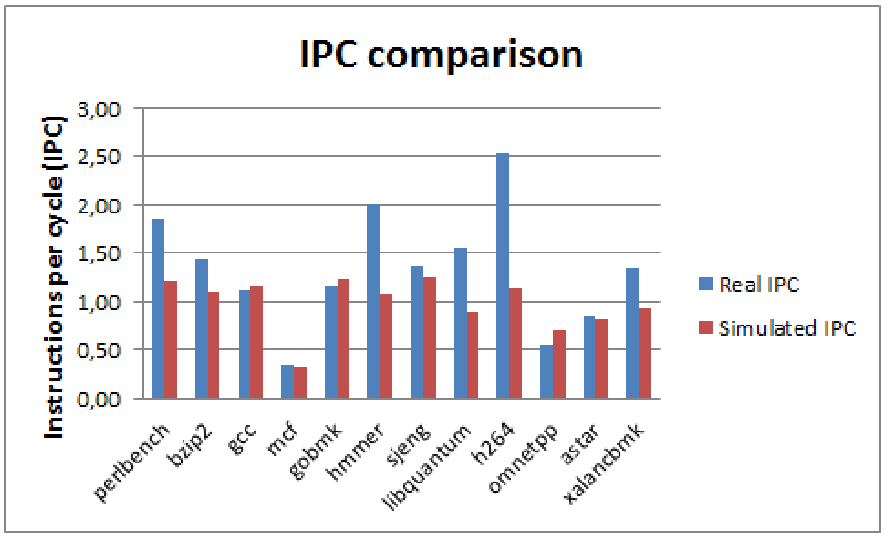


Figure 5.14: IPC comparison with out-of-order processor (reg=12 & window=10) model and fixed dependency probability

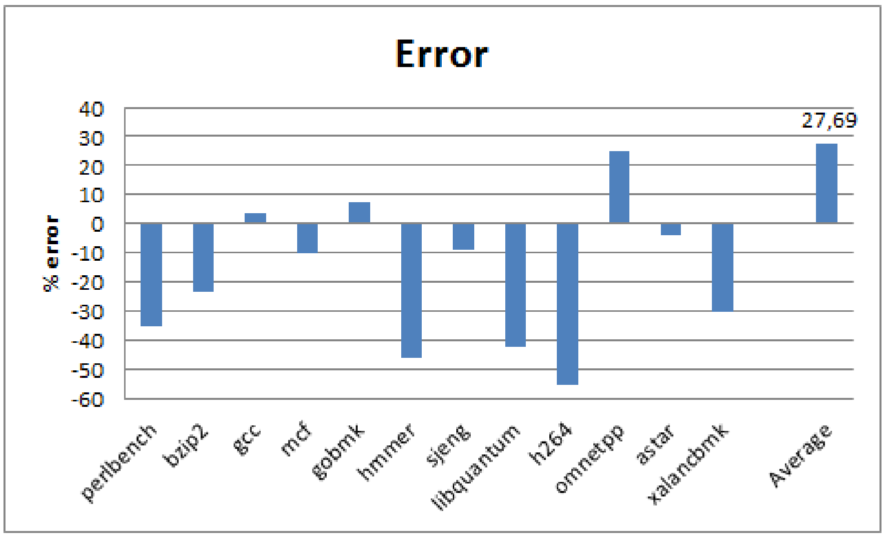


Figure 5.15: Relative error with out-of-order processor (reg=12 & window=10) model and fixed dependency probability

Other experiment we can try is to compare the IPC of a simulated in-order machine with a simulated out-of-order. In Figure 5.16 we can observe the difference between them. Our implementation of in-order is really optimistic because it assumes ALU operations of one cycle, when in a real machine there are multiply and divide, and other instructions that require more than one cycle. In both cases, in-order and out-of-order, just the load and stores are important from the point of view of the dependency model because integer instructions are executed in one cycle. In a real in-order machine the other instructions kinds like divide also plays an important role determining dependencies. Hence, difference between simulated in-order and out-of-order is not so big as researchers can expect. However, when out-of-order will be more tuned, this difference will increase (always depending on the application).

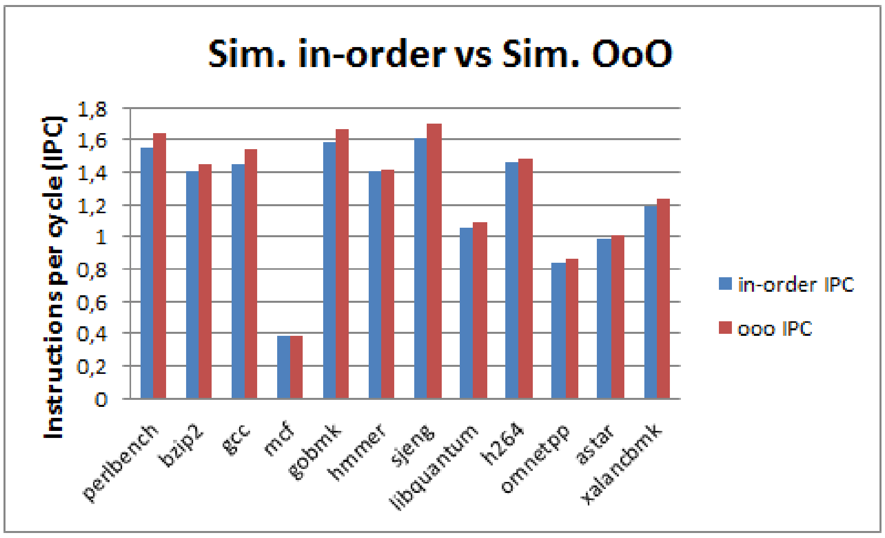


Figure 5.16: Comparison of simulated in-order with simulated out-of-order processor (reg=12 & window=10) model

Our simulator can provide much more information about the behavior of the different elements inside it. A good example is the number of instructions there are in the ready queue of control. In Figure 5.17 and Figure 5.18 we show histograms for the number of instructions present in that queue for the entire execution of the application. Both executions are out-of-order processor with a window of ten and the number of registers is 12.

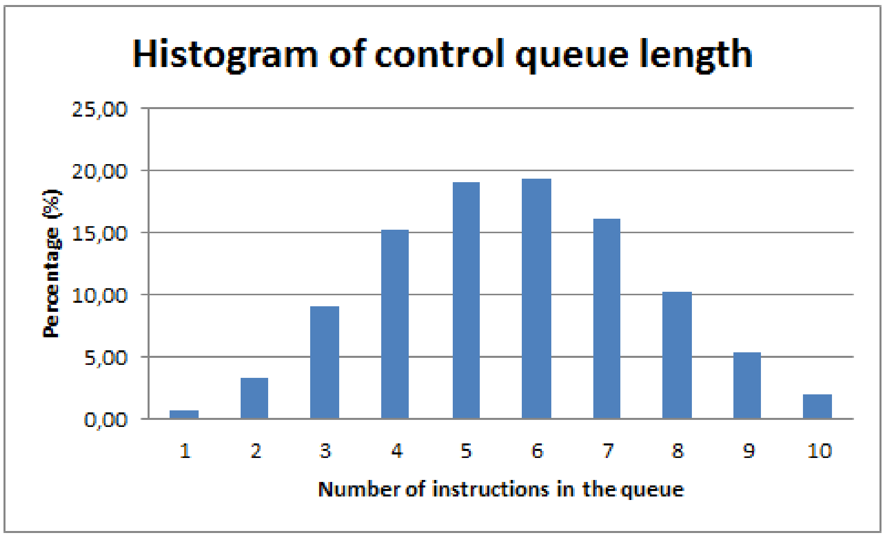


Figure 5.17: Histogram of number of instructions in ready queue of control module for gcc

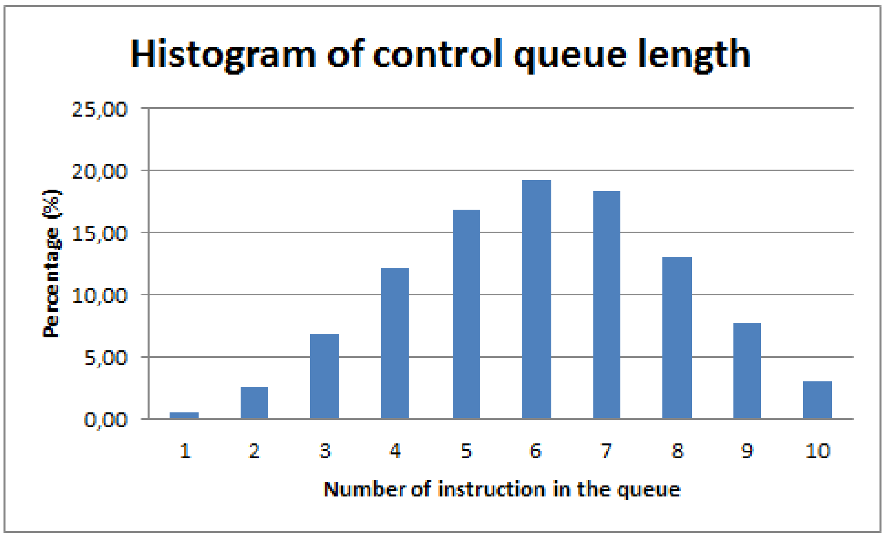


Figure 5.18: Histogram of number of instructions in ready queue of control module for bzip2

We have seen our best result is around an average error of 21%. For the first implementation is a really good result. We have seen previous works presenting a microarchitectural error of 20% with a specific and modified model for gem5 processor [17]. Another case is the results with Sniper simulator executing Splash-2 benchmarks, where they obtain an average error of 25% [**?**]. It is quite surprising the first case with gem5 because they have modified the simulator to follow a real ARM board, and even with that level of detail results are not so accurate. Then, for us the conclusion is clear. It is not necessary to simulate with so much detail because error obtained will be similar without so much complexity. Related to the simulation time required to obtain a result, there is a substantial difference with actual models. We do not need to simulate billions of instructions. We just need to execute enough instructions to obtain an stable IPC (or the desired parameter) from the simulator. Of course the required time can be different for each applications but the required time will be much smaller than traditional full-cycle simulators.

Now, our work is to keep improving the simulator to reduce the average error. We have identified more aspects we need to improve in order to improve our accuracy. These elements are mainly the dependency model, instruction cache, TLB, and branch prediction.

Chapter 6

Conclusions and Future Work

Simulation is a critical process in designing modern processors. Cost and timeare the main reasons to use simulation because real implementation of research techniques is really difficult. In previous work there a lot of possible solutions to optimize simulation. We want a tool that allows quick simulations with as much detail and accuracy as possible. However, there is not any perfect solution because accurate simulators need even weeks to finish and fast simulators lack accuracy. Then, it seems we are moving to faster simulators that maybe loose some accuracy, but reduce simulation time a lot. Following this tendency we have Sniper or ZSim.

In this thesis we present a new methodology to improve simulation and design space exploration. Our tool will help to take quick decisions during design stage and later if it is required a more detailed simulation will be performed, but directly applied to the key elements. It is true our model requires some profile information, but developers can easily obtain it in almost any existent platform. Since our tool is modular, developers will be able to simulate a specific part of the processor with a huge detail while other parts remains really simple. For example, if a user wants to observe modifications in memory subsystem, they can implement a detailed memory model while fetch, issue, etc remains simple. This characteristic offers a powerful tool to developers. We have reduced simulation time by using statistical execution and by just simulating the key components of a processor, which maintains accuracy.

As future work, we need to improve the simulator accuracy by adding more detail in some elements, like dependency model and instruction cache. Development of a library with defined models offering different complexity is important for the release to give more facilities to the users. Another important point is the improvement of the application generator. We think it has a huge potential, even by its own because it will be able to generate specific instruction flows. Once all this is completed, then we will proceed to use this single-core processor as the basic module to develop a full many-core processor model with its correspondent network. When it is finished, we will be able to simulate thousands of cores in a fast and accurate way.

References

[1] Cpu-z application. http://www.cpuid.com/softwares/cpu-z.html. Accessed: 10 June 2014. 38

[2] Intel sandy bridge microarchitecture block diagram. http://www.realworldtech.com/sandy-bridge/10/. Accessed: 10 June 2014. viii, 39

[3] Intel tick tock model scheme. http://www.scan.co.uk/tekspek/cpus/intel-ivy-bridge-processors. Accessed: 10 June 2014. viii, 37

[4] Luca Benini and Giovanni De Micheli. Networks on chips: a new soc paradigm. *Computer*, 35(1):70–78, 2002. 2

[5] Christian Bienia, Sanjeev Kumar, JP Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. *... of the 17th international conference ...*, 2008. 13, 14

[6] Nathan Binkert, Bradford Beckmann, and Gabriel Black. The gem5 simulator. *ACM SIGARCH ...*, 2011. 3, 4, 7, 8

[7] Nathan L Binkert, Ronald G Dreslinski, Lisa R Hsu, Kevin T Lim, Ali G Saidi, and Steven K Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006. 8

[8] Gabriel Black, Nathan Binkert, Steven K Reinhardt, and Ali Saidi. Modular isa-independent full-system simulation. In *Processor and System-on-Chip Simulation*, pages 65–83. Springer, 2010. 8

[9] Doug Burger, Todd M Austin, and Steve Bennett. *Evaluating future microprocessors: The simplescalar tool set.* University of Wisconsin-Madison, Computer Sciences Department, 1996. 7, 8

[10] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 52. ACM, 2011. 8, 10

[11] Xinjie Chang. Network simulations with opnet. In *Proceedings of the 31st conference on Winter simulation: Simulation—a bridge to the future-Volume 1*, pages 307–314. ACM, 1999. 15, 16

[12] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual. (March), 2014. 37, 48

[13] L. Eeckhout, S. Nussbaum, J.E. Smith, and K. De Bosschere. Statistical simulation: adding efficiency to the computer designer's toolbox. *IEEE Micro*, 23, 2003. 3, 8, 13

[14] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 37–48. ACM, 2012. 13, 14

[15] A Fog. The microarchitecture of Intel, AMD and VIA CPUs. *An optimization guide for assembly programmers and ...*, 4, 2011. 40

[16] Davy Genbrugge and Lieven Eeckhout. Chip multiprocessor design space exploration through statistical simulation. *Computers, IEEE Transactions on*, 58(12):1668–1681, 2009. 8, 13

[17] Anthony Gutierrez, Ronald G. Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher D. Emmons, Mitchell Hayenga, and Nigel Paver. Sources of Error in Full-System Simulation. *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2014. 59

[18] Thomas R Henderson, Mathieu Lacage, George F Riley, C Dowell, and JB Kopena. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 2008. 15, 17

[19] JL Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006. x, 13, 42

[20] Teerawat Issariyakul and Ekram Hossain. *Introduction to network simulator NS2*. Springer, 2011. 15, 17

[21] Jesus Labarta, Sergi Girona, and Toni Cortes. Analyzing scheduling policies using dimemas. *Parallel Computing*, 23(1):23–34, 1997. 3, 8, 11

[22] Robert Love, So Here We Are, Along Came Linus, Linux Versus Classic Unix Kernels, and Before We Begin. Linux kernel development second edition. 2004. 43

[23] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002. 8, 9

[24] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture, 2010. HPCA 16. The 16th International Symposium on*, pages 1–12, 2010. 8, 9

[25] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965. 1

[26] Mario Nemirovsky and Dean M. Tullsen. Multithreading architecture. *Synthesis Lectures on Computer Architecture*, 8(1):1–109, 2013. 1

[27] M. Oskin, F.T. Chong, and M. Farrens. HLS: combining statistical and symbolic simulation to guide microprocessor designs. *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, 2000. 3, 8, 12

[28] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. Paraver: A tool to visualize and analyze parallel code. *WoTUG-18*, 44:17–31, 1995. 8, 11

[29] Alejandro Rico, Felipe Cabarcas, Antonio Quesada, Milan Pavlovic, Carlos Villavieja, Yoav Etsion, and Alex Ramirez. Scalable Simulation of Decoupled Accelerator Architectures What is new in TaskSim. 2010. 8, 12

[30] Alejandro Rico, Alejandro Duran, and Felipe Cabarcas. Trace-driven simulation of multithreaded applications. *...Analysis of Systems ...*, 2011. 3

[31] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. *Proceedings of the 40th Annual International Symposium on Computer Architecture - ISCA '13*, 41:475, 2013. 8, 11

[32] James E Smith and Gurindar S Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609–1624, 1995. 1

[33] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967. 1

[34] M Vachharajani and Neil Vachharajani. The liberty simulation environment, version 1.0. *ACM SIGMETRICS ...*, 2004. 2

[35] András Varga. The OMNeT++ discrete event simulation system. *...the European Simulation Multiconference (ESM'2001)*, 2001. 15, 18

[36] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 24–36. ACM, 1995. 13, 14

[37] Ruken Zilan, Javier Verdú, and J García. An abstraction methodology for the evaluation of multi-core multi-threaded architectures. *Modeling, Analysis & ...*, 2011. 5