ECOLE POLYTECHNIQUE
Programme International
DONIEC, Marek

June 30, 2005

RAPPORT DE STAGE D'OPTION SCIENTIFIQUE

A Component Library for the Simulation of Chip Multiprocessors

MicroLib: une approche modulaire de la simulation de processeurs

NON CONFIDENTIEL

Option :	INFORMATIQUE
Champ de l'option :	Architecture des ordinateurs
Directeur de l'option :	Monsieur DOWEK, Gilles
Directeur de stage :	Monsieur TEMAM, Olivier
Dates du stage :	April 11 - Juillet 31
Adresse de l'organisme :	INRIA Futurs Parc Club Orsay Université, ZAC des vignes 4 rue Jacques Monod - Bât G 91893 Orsay Cedex FRANCE

Abstract

Processor simulation is a key tool used in microprocessor research to evaluate the performance and usefulness of new ideas. Currently used simulators such as SimpleScalar are monolithic and optimized for superscalar processors. In the meantime the trend in research and Industry has gone to Chip Multiprocessors (CMP) and especially researchers want to focus on implementations with hundreds of cores. Unfortunately, the monolithic simulators fail to provide the needed compatibility to simulate CMPs. Newer modular solutions provide this compatibility, but they have not yet been used for in this way. Thus the needed CMP modules are missing. The main goal of my intership was the development of such a library. I have researched different existing solutions for CMPs and the Networks on Chip (NoC) connecting the CMP cores. From this information I have decided which modules are needed. Afterwards I have implemented and tested those modules. The second part of my intership was to investigate solutions for the interfacing of modules from different simulation environments. This will further help to facilitate module reuse and simulator construction in general. I have investigated two solutions to this problem. One I have developed and tested myself. The other I have started to work on recently and I have had already possitive results. The library I have developed will help researchers to test their CMP programs and construct a big number of different topologies. Our hopes are that this library will grow due to contributions by others a possibly become a standard toolkit for CMP simulation.

Résumé

La simulation des processeurs est la démarche principale utilisée dans la recherche d'architecture des ordinateurs. Les simulateurs actuellement utilisés, comme SimpleScalar, sont construits d'une manière monolithique et optimisés pour la simulation des processeurs superscalaires. Mais la dernière tendance dans la recherche et l'industrie est d'évoluer vers des multi-processeurs sur chip (chip multiprocessors - CMP). En particulier, les chercheurs veulent se focaliser sur des implémentations avec des centaines de noyaux. Cependant les simulateurs monolithiques ne parviennent pas à offrir la comptabilité nécessaire pour simuler des CMP. Des nouvelles solutions modulaires offrent cette possibilité mais malheureusement elles n'étaient pas encore utiliser de cette façon. Donc les modules nécessaires ne sont pas encore implémentés. Le but général de mon stage était de développer une telle bibliothèque. J'ai recherché des solutions existantes de CMP et de réseaux sur chip (network on chip - NoC) qui associent les noyaux des CMP. A partir de cette information, j'ai choisi les modules qui doivent nécessairement être réalisés. Ensuite, j'ai implémenté et testé ces modules. La seconde partie de mon stage à consisté à explorer les solutions possibles de pour la communication des modules provenant d'environnements de simulation différents. Cela peut faciliter la réutilisation des modules et la construction de simulateurs en général. J'ai envisagé deux solutions pour ce problème. J'ai déjà développé et testé la première. J'ai commencé à travailler sur la deuxième et j'ai déjà eu des résultats positifs. La bibliothèque que j'ai développé va aider d'autres chercheurs à tester leurs programmes pour CMP et à construire une quantité déjà considérable de topologies différentes. Nous espérons que cette bibliothèque va s'agrandir grâce à des contribution faites par d'autres équipes de recherche et peut-être devenir un outil standard pour la simulation CMP.

Acknowledgments

First of all I would like to thank the whole Alchemy research group for the kindness and helpfulness they have offered me. I have really enjoyed my internship and a big part of this is due to the friendly atmosphere here.

I want to thank Prof. Temam for having let me work with his research team and bother them with questions. Also I want to thank him for all the useful comments and advises he gave me regarding this report and for all the other things he has helped me with.

I want to especially thank Daniel Gracia Pérez, Pierre Palatin, Yves Lhuillier, and Gilles Mouchard for having answered all my questions so patiently and for all the usefull comments and remarks they made regarding all aspects of my work.

Another special thanks goes to Sylvain Girbal and Pierre Palatin for having shown me all those linux tricks and having so often fixed some kind of problem with my account.

Orsay, June 29th, time: 31:14[1]

[1] There is no mistake here.

Contents

1 Introduction

The simulation of processor architectures is the core tool used by micro-architecture researchers for the purpose of evaluating the performance and usefulness of new ideas [6]. Most research papers on computer architecture include performance measurements that have been determined by simulation [5]. These simulations range from bit-accurate simulations on the gate level to functional simulations for testing program behavior for example.

The most commonly used simulation tool for this purpose is SimpleScalar [14]. More than 50% of the articles published on computer architecture use it to test their research ideas [39, 45]. SimpleScalar is a tool suit that contains multiple simulation source codes that can be adapted by researchers to suit their needs. It has been developed for the simulation of superscalar processors and is preferably used for out-of-order processor architectures. Research on this type of processors has been highly fruitful. As technology allowed us to fit more and more transistors on a single silicon chip and clock them at higher frequencies the superscalar processors consumed the resources given producing in turn bigger pipelines and wider instruction windows that allowed for ever faster execution and data throughput. Throughout this time SimpleScalar has served as an irreplaceable tool to the developers.

But the situation is currently changing. While today's technology allows us to fit more than a billion transistors on a single silicon chip the huge processors cannot cope anymore with the increasing frequencies due to signal runtimes on the chip. In today's modern processors an electric pulse that runs across the die needs multiple clock-cycles to actually cross it [28]. This effect worsens as frequency increases. Intel has officially announced that it has problems reaching the 4.0 GHz frequency with its processors. And although the new Cell Processor is going to be able to run at 4.6 GHz [26] a current trend towards another solution can be observed. Developers are stopping to increasing the frequency and utilize the transistor budget for area expensive predictions units to accommodate for worsening pipeline effects. Instead the industry has started to invest transistor resources into multiple processor cores on a single chip. Already Intel has produced a dual core Itanium 2 processor with 1.7 billion transistors [24, 25, 21]. IBM also has a dual core processor on his range of products: The Power4 [41]. This new evolution has even led to stripping down the processor cores in order to be able to place more of them on a single die. As an example an eight in-order core solution in the form of the Cell Processor [44, 26] is on its way to the global market. And processor dies with as many as 16 in-order cores have already been implemented by researchers as in the case of the MIT Raw Microprocessor [9]. These new types of processors are called *chip multiprocessors (CMP)*.

Unfortunately SimpleScalar fails to provide adequate service for this new research branch. At the time SimpleScalar was designed the world focused on superscalar processors and multiprocessors were not on debate. Thus the simulator lacks the ability for CMP simulation. A strong handicap that actually prevents SimpleScalar from effectively evolving in this way is the fact that it is based on a monolithic design. The whole simulator is encapsulated as one big program code and thus changes are hard to make. New simulation environments such as SystemC [1] and Liberty [13] have brought a promising solution to the problem. They are based on modules and thus offer a far bigger flexibility than SimpleScalar. Sadly these environments have so far not been used to provide appropriate CMP simulation support. Instead most module libraries published for these environments focus mainly on components needed for the simulation of superscalar processors. An example is MicroLib [5] that contains models like that of the PowerPC 750 processor [15] or a general superscalar processor module called OoOSysC [22]. These modules can be well branched together to form a multiprocessor simulator with a few cores. But they do not resemble the current research trend to use simpler cores in order to be able to have hundreds of them on a single chip.

These issues motivated the development of a new library specially designed for chip multiprocessor simulation. The first goal was to identify the components that would best fit in such a library. This included selecting currently used architectures and the correct granularity of the modules. Afterwards the chosen modules needed to be implemented. Major terms to be fulfilled were a very strong flexibility of the modules and the possibility for expansion. This will allow for better reuse by other research groups. Another very important objective was decent simulation speed of the modules. Reasearch always focuses on future products. Since there already are CMPs with a few cores on the market this means that researchers should start investigating architectures with hundreds of cores and more. Thus every waste of simulation time in a module will be multiplied by the number of hundreds of instantiations of this module. In order to receive significant results for a simulation researchers need to at least simulate a million instructions. If the modules proposed will not result in fast simulators they are of limited usability.

A second goal of this project addressed a related issue. As already mentioned two modular simulation environments are currently evolving as far more flexible tools than SimpleScalar: SystemC and Liberty. Unfortunately until now these environments are incompatible with each other. This means that so far modules written for SystemC cannot be used together with modules written for Liberty. Due to this circumstance reuse is strongly limited to one simulation environment. The Liberty Research Group [38] and the Alchemy Research Group [37, 5] have both expressed the need for a solution to this problem. Thus a second part of this project was to investigate possibilities to connect modules from these two environments.

While these goals might seem very clear it has to be pointed out that they are almost infinite. The library motivated above can easily contain hundreds of components. Since I had only little more than 3 months during this project my goal was to start up this library by creating the absolutely necessary modules for *a working instance of a CMP simulator.* I have tried to make these modules as adjustable as possible to already allow for effective recycling in other simulators. My choices regarding the modules have been guided by current research but also by the needs of the research group I work with, the Alchemy Group at INRIA Futurs. To some extend the fact that a cooperation with the Liberty Research Group at Princeton is planned for the future also influenced my decision.

The rest of the report is organized as follows. Section 2 describes the most commonly used simulation environments, two of which I have been working with. In Section 3 I give an introduction to the domain of Chip Multiprocessors by presenting existing solutions and then describe the design choices I have made. Section 4 talks about the actual implementation details of the modules chosen in Section 3. Interfacing possibilities of different simulation environments, including two possible solutions, are being discussed in Section 5. In Section 6 I demonstrate how to construct a simulator from the modules implemented. Finally Section 7 discusses future work to be done and Section 8 concludes the project experiences.

2 The simulation environments

In this section I will introduce the four simulation environments mentioned above: *SystemC* [1], *MicroLib* [5], *Liberty* [13], and *SimpleScalar* [14]. All three environments have been widely used in past and current research [16].

I have written all modules in SystemC since this way I was able to reuse existing tools developed by the research group which I have joined during this internship. Also SystemC allows for the most flexibility when developing modules. Thus it was the most important simulation environment for this project and I will present it first. Next I will present *MicroLib*, a modular simulation environment that is a continuation of SystemC. After that I will introduce the *Liberty Simulation Environment (LSE)*. It is often just called *Liberty* after the research groups name that has developed it. Liberty has been important in the second part of this project when investigating interface possibilities between SystemC and LSE. At last I will shortly present SimpleScalar. I have not really worked with SimpleScalar and it does not very well suit my purpose since it is not module based. Still I feel that it is worth mentioning due to its widespread use in the computer architecture research community [39].

For a quick overview of all simulation environment please refer to Table 1.

Name	**Properties**
SystemC	+ Highly modular simulation environment. + Implemented as a `c++` library. + Leaves huge freedom to the user in defining modules and communication interfaces. + User has control of the `main` routine. − Communication between modules via ports is slow.
MicroLib	+ Highly modular simulation library for processors. + Common interfacing standards. + Many modules that can be branched together. + Wrappers for other simulation environments.
Liberty	+ Highly modular simulation environment. + Defines its own API and syntax. + Generates `c++` code and compiles it into simulator using `g++`. + Allows for strong optimization due to the fact that the model structure is known at compile time. − Leaves not much freedom to the user concerning communication between modules above the simulation level. − Only a handfull of data types. No possibility to pass classes as data like in SystemC. − Quite a hassle to install.
SimpleScalar	+ Widely used simulator for superscalar processors. + Implemented as parametrized `c++` code. − Impractical for simple processor cores like those used in chip multiprocessors.

Table 1: Different simulation environments

2.1 SystemC

SystemC is a class library implemented in standard C++. It provides hardware-oriented constructs that allow for low-level as well as high-level modeling of hardware architectures. It is supported by more than 50 companies from the System on Chip domain [45]. In the follwing I will only describe the parts of SystemC that have been usefull to me. Please note that a complete and exhaustive description of SystemC would go beyond the scope of this report and only bore the reader. I felt though, that basic knowledge about SystemC should be provided in order to allow to clearly understand this project.

In SystemC a model might consist of *modules* with *ports*, *clocks* and *signals*. In the following I will describe these building parts while explaining how to implement a simple *Linear Feedback Shift Register (LFSR)*. This is an standard example in simulation environment documentations since it is simple to understand and utilizes the most important parts of the simulation environment. I will first demonstrate the definition of a xor-gate and a flip-flop and latter combine those to a working 3-bit LFSR as can be seen in figure 1.

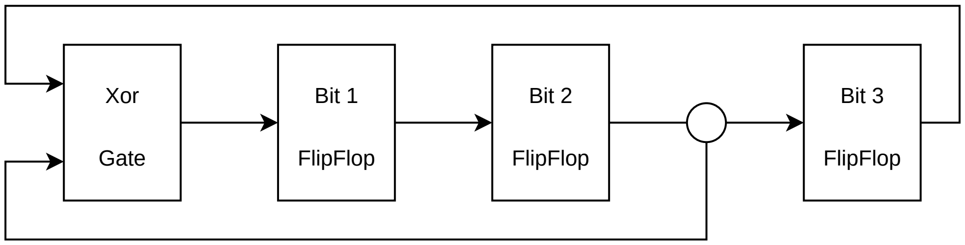


Figure 1: A simple three bit Linear Feedback Shift Register (LFSR).

Modules are the basic building blocks written in C++. A new module is declared like a new C++ class using the construct `sc_module`. They describe basic structures like gate and Flip-Flops for detailed-level simulations or more complex structures like caches, memories and ALUs. In most processor or hardware system simulations more complex structures are implemented by modules. This increases facility of inspection and decreases development time. Also simulation speed is increased since each module is treated as a separate object during simulation and thus generates additional computation overhead. The declaration of our xor-gate will thus look like this:

```
sc_module XorGate {
   ...
}
```

To be able to communicate amongst each other, modules contain *ports*.

There are two groups of ports. Ports declared as `sc_in<type>` serve as inputs to the module and are used to receive data of the type `type`. Ports declared as `sc_out<type>` serve as outputs and are used to send data of the type `type`. Let us assume our xor-gate has two inputs and one output. To add these our declaration has to be changed to:

```
sc_module XorGate {
  sc_in<boolean> op1, op2;
  sc_out<boolean> res;
  ...
}
```

The module can now receive two `boolean` values, one on each of the ports `op1` and `op2`. They can send their result by writting it to the port named `res`. In order to make our gate work we have to register it to the SystemC engine and create a SystemC *processes* that will compute the result. This is done in the constructor of our module:

```
sc_module XorGate {
  sc_in<boolean> op1, op2;
  sc_out<boolean> res;

  XorGate() {
    SC_MODULE();
    HAS_PROCESS(ComputeResult);
    sensitive << op1 << op2;
  }

  void ComputeResult() {
    res = op1 ^ op2;
  }
}
```

First we have registered the module to the SystemC engine by calling `SC_MODULE()`. Second we have registered a new process to the SystemC engine by calling `HAS_PROCESS(ComputeResult)` and making it sensitive to both inputs. This means that the SystemC engine will call the method `ComputeResult` when one or both values of the inputs change. This method than computes the result of the operation and writes it to the outport. In SystemC most ports can be simply read and written like variables. The SystemC engine will automatically register changes to outports and react accordingly.

A process can be made sensitive to any number of inputs but not to outputs. It can also be made sensitive to the falling or rising edge of a clock by writing `sensitive_pos « clk` or `sensitive_neg « clk` where `clk` is the name of the clock inport. A clock inport is declared using `sc_in_clk`. In our flip-flop we will make use of a clock. Additionally we need a variable to store the memorized value. Since SystemC modules are C++ classes they can contain any additional variables and methods as needed. Here the flip-flop code:

```
sc_module FlipFlop {
  sc_in_clk clk;
  sc_in<boolean> opin;
  sc_out<boolean> opout;
  boolean tmp;

  FlipFlop() {
    SC_MODULE();
    HAS_PROCESS(DataIn);
    sensitive << opin;
    HAS_PROCESS(NewCycle);
    sensitive_pos << clk;
  }

  void DataIn() {
    tmp = opin;
  }

  void NewCycle() {
    opout = tmp;
  }
}
```

When a new value arrives at the input `inop` of our flipflop it will be memorised temporarily in the variable `tmp`. In the beginning of the next cycle the process `NewCycle` will write this temporary value to its output `opout`. In order to create our LFSR module we have to instantiate the correct number of flip-flops and a xor-gate and connect them. To create these connections in SystemC *singals* are used. Signal of the type `type` are declared using `sc_signal<type>`. An outport of one module can be connected to an inport of the same or of a different modules by a signal. To do so, both ports as well as the signal need to be of the same data types. To connect `outport` with `inport` we write:

```
sc_signal<type> signalname;
outport(signalname);
inport(signalname);
```

This connection works different with clock inports. A clock inport is directly connected to another clock inport typing `submodule->inClock(inClock)`. In the case of our LFSR its clock inport needs to be connected to the clock inports of the flip-flops. Thus the LFSR code will look like this this:

```
 sc_module LSFR {
   sc_in_clk clk;

   FlipFlop *fp1;
   FlipFlop *fp2;
   FlipFlop *fp3;
   XorGate  *xor;
   Tee<2>   *tee;

   LFSR() {
     ...
     fp1->clk(clk);
     fp2->clk(clk);
     fp3->clk(clk);

     fp1->opin(xor_to_fp1);
     fp1->opout(fp1_to_fp2);

     fp2->opin(fp1_to_fp2);
     fp2->opout(fp2_to_tee);

     tee->opin(fp2_to_tee);
     tee->opout[0](tee_to_fp3)
     tee->opout[1](tee_to_xor)

     fp3->opin(tee_to_fp3);
     fp3->opout(fp3_to_xor);

     xor->op1(tee_to_xor);
     xor->op2(fp3_to_xor);
     xor->res(xor_to_fp1);
     ...
   }

   ...

}
```

The complete code of the LFSR can be seen in Appendix C. Note that there is a new module `tee`. It is needed to communicate the output of `fp2` to both `fp3` and the xor gate `xor` because an outport can only be connected to one inport. The `tee` module simply writes the data it receives on its inport to all its outports. Notice that the class `Tee<outnum>` is templated and the number of outports is passed during declaration.

Finally we want to simulate our LFSR module. For this we have to create a method called `sc_main`. It takes the same arguments as the `main` method of standard C++ programs, namely the command line parameters. When the simulator is compiled the internal `main` method of the SystemC library is used. When executed it calls the userdefined `sc_main` method. This method has to instantiate the modules used for the simulation and signal the simulation engine to start.

Before we can run the simulation we have to declare a clock. This is done using the construct `sc_clock`. We than instantiate the LFSR module and connect the clock to it by simply writing `lfsr->clk(clock)`. The simulation needs then to be initialize and started for `n` clock cycles as follows:

```
sc_initialize();
sc_start(n);
```

When done simulating the first `n` cycles the method `sc_start` returns. It can be re-executed to continue simulation form where it stopped. SystemC is not able to rewind in simulation or simulate backwards due to the complexity it offers.

Since the whole description of the modules is completely done in C++ counting variables, file input/output and so on can be added to the simulator in order to measure and see the effects of the simulation. This is one of the big advantages of SystemC that all powerfull tools given to the user by C++ can be reused in defining simulator modules. The whole text of the LFSR simulator can be found in Appendix C. Output is already contained in this code.

I have just described the general structure by which SystemC modules are defined and combined into more complex modules and finally into an executable simulator. It is of further interest to know a little bit about the simulation engine of SystemC. In fact, simulation in SystemC is event driven. This means that calculation processes in modules are started if certain events arrive. Such an event might be the rising edge of a clock or changed data on one of the input ports. When a clock cycle in simulation starts, all processes are executed that are sensitive to the clock edge. These processes might produce new events by changing the values of their outports who then change the values of the connected inports. These events are stored in a last in, first out queue (LIFO). Once all processes have been called SystemC checks the LIFO for events that would trigger new processes and starts to execute these, memorising again all newly triggered events. These iterations are called *delta cycles* and a clock cycle in simulation is usually split into multiple delta cycles. The simulation for the current clock cycle ends, when the list of new events is empty at the end of a delta cycle [2].

2.2 MicroLib

MicroLib [5] has been introduced as an effort to build a library of processor simulator components. It focuses strongly on a modular approach to simulation. It can be seen as an advancement of the idea provided by the SystemC simulation environment. In fact most modules for MicroLib have been developed using SystemC. But in SystemC the huge freedom has led to a situation where every research group has developed their own modules. These modules contain often completely different interfaces and because of this they are mostly incompatible with each other. One of MicroLibs efforts is to introduce common interface standard such that modules will become compatible even when developed by different groups. The idea is that researchers should be able to easily download these modules in order to reuse them for their purpose. They should be able to connect them at their will and add their own modules. But they should be able to do this without having to further study the behavior of the SystemC ports interfacing these modules. Reuse should become as easy as plugging together a computer workstation: Components that are supposed to fit do so, others not. In fact MicroLib provides already a set of components including processors [22, 15] and a big library of caches [46]. These modules can be arbitrarily branched together as long as their connection makes any sense.

A major difference to past developments is that MicroLib is not trying to compete with other existing modular environments like Liberty. Instead its goal is to introduce wrappers that will allow it to connect modules from different simulation environments. A wrapper for SimpleScalar has already been introduced and as I will present in section 5.2 a Liberty wrapper that is currently being worked on. By this MicroLib is trying to build the largest possible library of component modules for processor simulation.

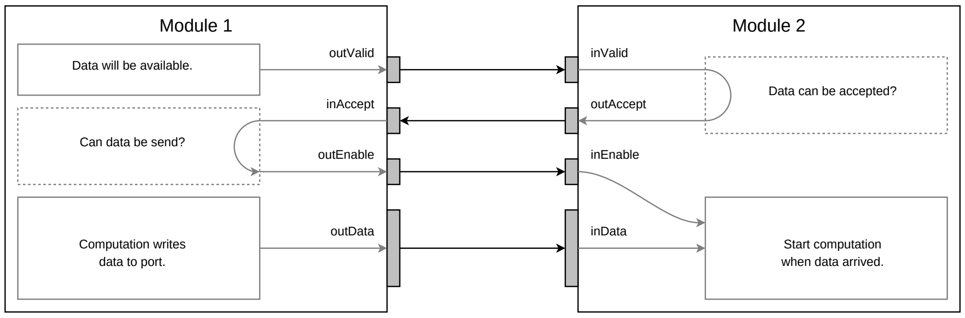


Figure 2: Handshaking protocol introduced by MicroLib.

In order to increase compatibility MicroLib has introduced a handshaking protocol to pass values between modules. This handshaking is always done for groups of ports that are supposed to process coherent data. Such ports might be the data, address and command ports going from the CPU to the cache. It is evident that the cache cannot process the query unless all three values are known, thus they can be seen as a group. Each such group gets a set of three control lines assigned to them, namely *valid*, *acknowledge*, and *enable*. While *valid* and *enable* connect the sending module to the receiving module, *acknowledge* connects the receiving module to the sending module. The functions of the handshaking signals are as follows. *Valid* serves to notify the receiving module that valid data will or will not be available by sending `true` or `false`. The receiver uses *acknowledge* to communicate to the sender if he can process the data. Finally the sender has to write the data to the corresponding ports and set *enable* to `true` to indicate that the data is available. In fact, we will see a very similar handshaking protocol later on in Liberty (Section 2.3). I have implemented all my modules such that they use this handshaking protocol. Figure 2 shows a schema of this protocol. The following code is an example of how the protocol implementation for one group of ports could look like:

```
...

DataTreater(const sc_module_name& name) : sc_module(name) {
  ...

  SC_METHOD(ForwardValid);
  sensitive << inValid;
  SC_METHOD(ForwardEnable);
  sensitive << inEnable;
  SC_METHOD(ForwardAccept);
  sensitive << inAccept;
}

void ForwardValid() {
  if (inEnable) {
    // data arrived, threat it.
    ...
  }

  // forward the signal
  outValid = inEnable;
}

// Forwards and broadcast accept signal.
```

```
void ForwardAccept() {

  // Accept all incoming data.
  outAccept = inValid;
}

// Forwards enable signals.
void ForwardEnable() {

  // Allways send data when accepted
  outEnable = inAccept;
}

...
```

Another tool so far distributed on MicroLib is FastSysC [2], a fast SystemC simulation engine. Since I have used it to speed up my simulations I have added a short description to the end of this report. It is available in appendix A.

2.3 Liberty

When speaking about Liberty most frequently the *Liberty Simulation Environment (LSE)* [13] is meant. While similar to SystemC and MicroLib in its overall structure, LSE does still differ. In LSE models are also constructed modular in a manner parallel to that of real hardware structures. LSE specifications, that are compiled into simulators, are a collection of connected modules that are instantiated from templates. This templates are called modules. There are two types of modules: *leaf* modules and *hierarchical* modules. The behavior of leaf modules is specified by sequential code that uses the LSE API to compute, send and receive data. On the contrary hierarchical modules are composed of a collection of interconnected leaf modules. These hierarchical modules can still contain special build algorithms allowing for an enormous flexibility. Thus, LSE provides low overhead component based reuse [13].

A major difference to other simulation system is, that LSE model specifications can be compiled, since the structure of the model is known at compile time. Thus optimizations like those done by the FastSysC schedule generator[2] can be easily performet by the LSE compiler. In addition the compiler can remove unused signals and optimize code where needed. For example even when FastSysC genereates a static schedule every component still has its own processes which needs to be called during most simulation cycles. Such calls generate computation overhead due to access to the stack. The

[2]Please see appendix A for details.

Liberty compiler can actually generate sequential code. When this code is finally compiled by gcc it can run much more efficient than that written by the user when working with SystemC.

Liberty modules are defined in Liberty Structural Specification (LSS) files. Like in SystemC they contain inports and outports in arbitrary numbers. Unlike in SystemC where each port can be of any type, even a class type, LSE supports only very basic types like integers, characters, strings, arrays and structs. This is due to an otherwise very usefull feature in Liberty. This feature is the support for polymorphism that is not provided in this manner by SystemC. In Liberty data types for ports and other entities can be declared to be polymorphic (i.e. to have many types). Utilization is very convenient since LSE tries to determine the appropriate type for polymorphic ports and variables automatically when possible [13]. This futher facilitates module reuse.

To specify component behavior or allow for later parametrization *userpoints* are used. These are variables that contain algorithms called on special events. There are userpoints for module initialization, userpoints for when data arrives on a port or simply user defined userpoints that are called according to module specification. During compilation LSE simply inserts the code contained in userpoints in the appropriate places.

The distribution of LSE contains already a component library of standard components like queues, routers, arbiters, control modules, and other utility modules. These components can be easily reconfigured using userpoints to create most common modules. To use these modules the library has to be included using the command `using corelib`. For example in order to create a xor gate a `combiner` module can be used. The `combiner` module is a flexible module whose instances can be parametrized to compute an arbitrary number of outputs from an arbitrary number of inputs. The *userpoint* `combine` contains the code that computes the result. The LSS code of the xor gate looks like this:

```
module xor_gate {
  using corelib;

  inport in0:boolean;
  inport in1:boolean;
  outport out:boolean;

  instance gate:combiner;

  gate.inputs={"in0","in1"};
  gate.outputs={"out"};

  gate.combine = <<< *out_id=in0_id;
```

```
                    *out_data = (*in0_data) ^ (*in1_data);
                    *out_status = LSE_signal_something; >>>;

  LSS_connect_bus(in0,gate.in0,in0.width);
  LSS_connect_bus(in1,gate.in1,in1.width);
  LSS_connect_bus(gate.out,out,out.width);
};
```

To connect two ports in LSE no special signal object is required. Instead the user simply writes `module1.outport ->[type] module2.inport`. The `type` is optional and can server for type resolution during compilation when there is no other mean of determining a connection type. This can happen when data is circularly passed between polymorphic modules.

Another difference in regard to SystemC is, that LSE handles timing quite different. Instead of providing clocks, LSE propagates stall signals. A special control signal structure is used for that, similar to that described in Section 2.2 regarding MicroLib. In addition to the data a port contains three control lines, namely if the data is *valid*, an *acknowledge* signal, and an *enable* signal. At the beginning of a cycle all control lines are set to *unknown*. In the same manner as described above for FastSysC the singal lines are used in LSE. *Valid* signals to the receiver whether or not data is going to be available. The receiver signals the sender via *acknowledge* if he can accept the data. Final the sender writes the data to the port and signals the receiver using the *enable* signal.

LSE also offers a far more advanced way of data monitoring and logging than SystemC. In SystemC the user monitored actions by placing if-statements and printf calls all over the module code. LSE provides special collector objects for this. They are defined using the keyword `collector`. The eventname to be monitored and the name of the monitored module are given as parameters. After that actions to be taken when the monitored event takes place can be defined. Again during compilation LSE can choose the best possible placement for the code in the final simulator.

With these tools the LFSR from figure 1 can already be constructed. Parts of the code follow. Please not that we only needed to redefine the xor gate. The other modules like flipflop (`delay`) and the `tee` are taken directly from the Liberty `corelib`:

```
using corelib;
include "xor_gate.lss";

instance bit0 : delay;
instance bit1 : delay;
instance bit2 : delay;
instance xor : xor_gate;
instance bit1_tee : tee;

bit0.initial_state = <<< *init_id = LSE_dynid_create();
                         *init_value = TRUE;
                         return TRUE; >>>;
...

bit2.out -> bit1.in;
bit1.out -> bit1_tee.in;
bit1_tee.out[0] -> xor.in0;
bit1_tee.out[1] -> bit0.in;
bit0.out -> xor.in1;
xor.out -> bit2.in;

...

collector STORED_DATA on "bit2" {

  ...

  record=<<<
      printf(LSE_time_print_args(LSE_time_now));
      printf(": bit2=%d\n", *datap);
  >>>;
};

...
```

Again the full code of the LFSR can be found in Appendix D.

2.4 SimpleScalar

SimpleScalar [14] is a suit of simulation tools that offer low detail and high detail simulation of microprocessors. These levels range from a high-performance functional simulator to detailed models of superscalar processors including caches, speculative execution and branch prediction. In difference to SystemC and Liberty SimpleScalar is *not* modular. It is designed to simulate processors and a part of their environment, not more. Because of this SimpleScalar is well optimized and has been widely used in processor simulation [39, 45]. SimpleScalar is also available as source code. It has been initially developed to support a MIPS derived instruction set architecture (ISA) called PISA. Later versions have added support for architectures such as Alpha, ARM, PowerPC, and x86. Thus it really is not one simulator but contains multiple simulation and emulation programs. The most used of these simulators is *sim-outorder* that lets the user simulate out-of-order[3] processors.

3 Design choices

The goal was to create a library that would allow to construct a running CMP simulator. In order to do so I needed to implement a set of components that every CMP system needs. These components are clearly at least a processor core, caches and memories, and the substantial interconnect. Especially the interconnect needs to be chosen with care since it dominates the overall architecture of the CMP. In the following I will present different existing CMPs and their interconnection networks as well as some stand-alone interconnect solutions. I will evaluate these approaches and excerpt from them different possibilities for the desired modules. Finally I will present and explain my choices.

3.1 Existing CMP architectures and interconnects

I will start by presenting three existing Chip Multiprocessors and their different interconnection architectures. The first two, the Stanford Hydra CMP and the MIT RAW Microprocessor, are research projects. The Stanford Hyrda CMP implements a shared bus architecture. A completely different approach is presented by the MIT RAW Processor that implements multiple packet-switched networks. The third solution presented, the Cell Processor, is a commercial product that is currently in development. Its interconnect is a mergence of the first two solutions. After having presented the CMP solutions and their networks I will describe two further interconnection architectures. One is an interconnect called crossbar and the other is an all-round implementation of a packet-switched network on chip called SPIN.

3.1.1 A shared bus solution: Stanford Hydra CMP

The Stanford Hydra Chip Multiprocessor [12] contains four MIPS [4] based cores. Each of the cores contains its own primary instruction and data caches. The Hydra cores are in-order processors. They are connected to a single shared second level cache via two *shared buses*: one 256 bits wide bus

[3]Out-of-order means that the processor rearranges instructions in regard to the constraints given by dependencies. This allows to execute further instructions when only a part of the processor is stalled by the current instruction. Out-of-order execution is widely used in today's superscalar processors. In contrast in-order processors execute instructions in the same order as they arrive from memory. These processors are slower but usually consume much less die area.

[4]MIPS stand for Machine without Interlocked PipeStages. It is a Reduced Instruction Set Computer (RISC) architecture developed 1981 at Stanford University that is being used in processors worldwide until today.

for read operations and one 64 bits wide bus for write-through operations. Both buses are pipelined in order to maintain a high clock frequency. Thanks to this data can enter and leave each bus on every clock cycle.

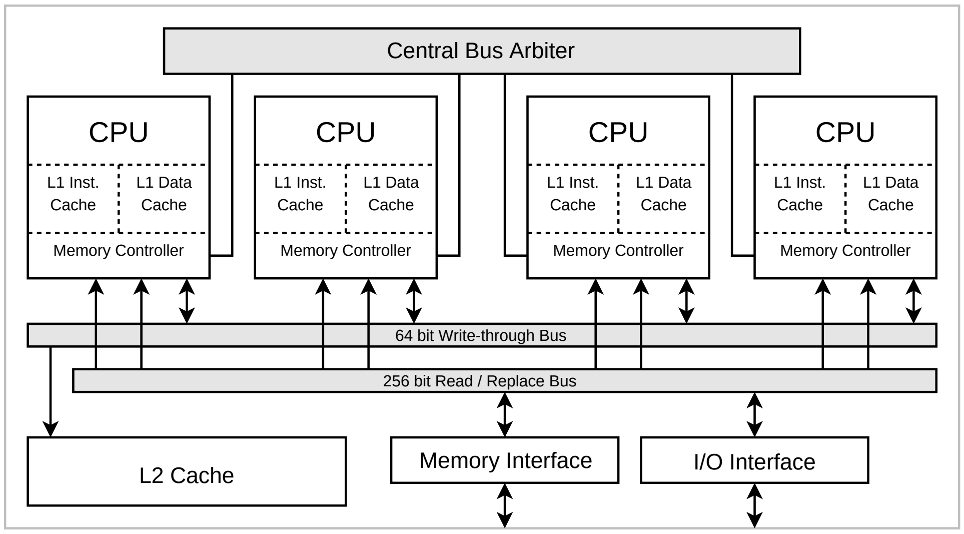


Figure 3: Architecture of the Hydra CMP without the speculative support.

The read bus was designed to handle an entire cache line in one clock cycle. It thus allows for a very fast communcation between processors, secondary cache and off-chip memory. The write-through bus communicates all writes made by the four cores directly to the second level cache. It also broadcasts the write to other processors, eventually invalidating corresponding cache lines in their primary caches. Thus it helps to maintain coherent level one caches. The general Hydra architecture can be seen in figure 3.

Hydras shared bus solution allows for a latency of 10 to 20 cycles for interprocessor communication. Unfortunately it does not scale well beyond 8 processors as its creators have admitted themselves [12]. It is clear that a single bus cannot cope with an arbitrarily increasing number of processors. Even if we increase the bandwidth of the bus it has no ability to respond to multiple requests at a time. Thus with the number of processors the latency for memory requests grows. The authors of the Hydra CMP propose to use either more buses, a crossbar interconnection or a hierarchy of connections to account for this problem. Some of these suggestions I will discus in the following sections.

The Hydra CMP implements what is called *thread-level speculation.* This method tries to exploit *instruction level parallelism (ILP)* in sequential uniprocessor programs. To do so the Hydra CMP arbitrary splits a program code sequence into a sequenced group of threads that can be executed in parallel on different cores. The Hydra CMP contains special hardware that ensures that no data dependencies [5] between threads are violated. In the case that a thread from the sequence causes a true dependency violation the Hydra will re-execute this sequence with the correct data once this data is available.

The performance of the Hydra CMP is given as speedup compared to the execution time of the same code on one Hydra core. While in fact the Hydra CMP with its four cores does extract quite high speed up this needs to be seen with careful consideration since a single Hydra Core is far less powerful than a superscalar processor taking the same area of the die chip as four Hydra cores.

3.1.2 Packet-switched networks: MIT RAW Microprocessor

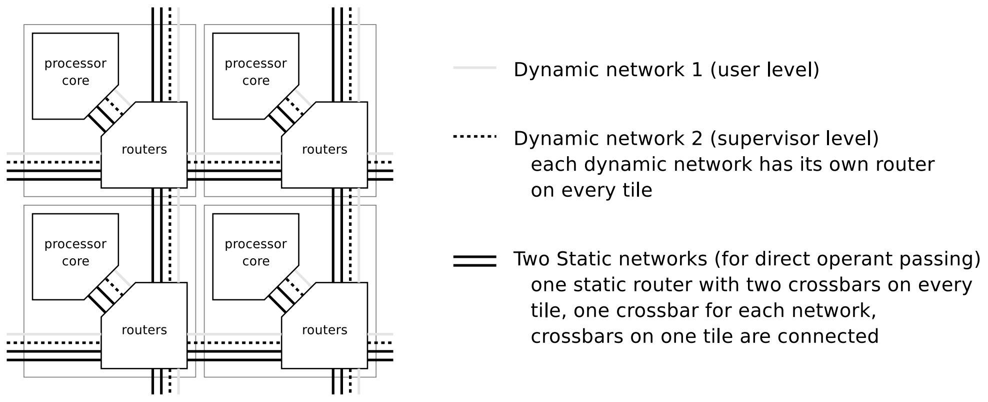

processor core
processor core
processor core
processor core
routers
routers
routers
routers
Dynamic network 1 (user level)
Dynamic network 2 (supervisor level) each dynamic network has its own router on every tile
Two Static networks (for direct operant passing) one static router with two crossbars on every tile, one crossbar for each network, crossbars on one tile are connected

Figure 4: A schematic 2x2 grid of RAW tiles.

The MIT RAW Microprocessor [9] is another example of a chip multiprocessor. It is also implemented using MIPS based cores. In contrast to the Hydra CMP that used a shared bus the RAW microprocessor implements several networks on its chip to connect its cores. The chip is segmented into quadratic tiles that each contain a processor, a level one data cache, a software managed level one program cache, and programmable routers. The routers implement nodes of the network that control the data flow. They are interconnected via links and form three *global on chip networks*. The schema of this network can be seen in figure 4. Each Tile contains three routers: two dynamic routers and one static. Routers of one network are connected to routers of the same network on neighboring tiles via a full duplex 32 bit link. In the case of the static routers two such links are used between two neighboring tiles.

[5]The possible dependencies are: *read after write (RAW)*, *write after read (WAR)*, and *write after write (WAW)*. The exact algorithm by which Hydra handles these dependencies is not of importance to this project and is thus not further explained. It can be seen in detail in [12].

The two dynamic networks are *packet-switched* and implement *wormhole-routing.* Packet-switched means that data is divided into several smaller chunks called packets that are send over the network. This is done instead of sending the data in a continuous stream. While a continuous stream guaranties bandwidth it completely blocks a part of the network for other users. This is only reasonable if high-bandwidth applications are used. Packet-switched networks on the contrary do not guaranty bandwidth but allow multiple users to take advantage of one connection simultaneously.

Wormhole-routing means that instead of waiting until the whole packet arrives the router only waits for the packet header and then forwards it already to the next node in the network. This way packets *worm* their way through the network. Unfortunately in a wormhole-routed network congestion might occur. This can happen when to many messages are fed into the network and the buffers of the routers fill up. In this case a network will simply stop to accept new messages and not deliver the ones pending. Thus this phenomenon is also called *deadlock.*

There are two solutions for this problem. The first one is called *deadlock prevention.* It allows every network user to send only a certain amount of messages at a time based on a credit system. While this solution prevents congestion it limits network performance. The second solution is called *deadlock recovery.* In this case there is special hardware to detect network congestion. When this happens the network contents are partially flushed to a temporary buffer in order to allow for recovery. While this solution does not limit performance it necessitates an additional framework to move the data out of the network.

The RAW Microprocessor implements both types of networks. One of the dynamic networks is dedicated to trusted users, like the operating system, data caches and DMA controllers. This network uses the *deadlock prevention* technique. The other dynamic network can be used by programs to pass messages. It implements *deadlock recovery.* When congestion occurs the contents of the network are flushed to external buffers using the first dynamic network.

The third and static network has a quite different purpose. It is used directly by the hardware to pass operands between the compute pipelines of different cores. In order to be efficient this network is directly connected to the pipeline. In fact when the program writes to register 25 through 28 the value is not stores in the register file but send to the static network. When reading from these registers the processors reads from the incoming static network buffers. If these buffers are empty the processor stalls and waits until data arrives. The output is buffered as well and when buffers are full the pipeline will stall as well.

In order to control the network each tile contains two dynamic routers and static router. Each router is connect to one of the three networks. While the dynamic routers switch the packets according to the packet header the static router is run by a program. It fetches its instructions from the corresponding level one instruction cache. These instructions contain routes for both of the static routers crossbars. Crossbars are the hardware structures that create connections between the incoming and the outgoing network links. They are further explained in section 3.1.4. The crossbars are connected to the four links of the four neighboring tiles, the processor, the router pipeline and to each other. A static router instruction sets the connections of the crossbars and instructs the router pipeline in order to implement simple loops and conditions for the static router program. This architecture allows the RAW Microprocessor to pass values between two computing cores with a latency of only $2 + X + 1 + Y + 2$ cycles where X and Y are the number of tiles the signal has to travel horizontally and vertically. Two cycles are needed to introduce the signal into the network, X cycles for the horizontal tiles, one cycle for the turn, Y cycles for the vertical tiles, and finally two cycles for the singal to reenter the destination pipeline. The one cycle latency for the turn is due to optimizations to the wires that exploit the fact that they mainly run straight between the tiles.

Figure 4 displays a grid of only 2x2 tiles. It is evident that this structure is scalable in the number of tiles. The RAW Chip is implemented with 16 tiles but multiple chips are branched together to form larger structures. Performance of the RAW Chip is also measured in speedup compared to a single RAW tile. The Results show also an impressive speedup in certain applications. Like in Hydra this speedup is very dependant on the ILP present in the program executed.

A critical point not mentioned in any RAW paper I have found is actual network saturation when the grid grows bigger. Since memories are connected to the outside of the raw grid (as on all processor dies) the inner tiles of the grid are further away from them. The RAW team has failed to investigate the effect of larger grids then just 16 processors. Most likely in the case of a huge grid the inner cores will experience very fast rising memory access latency in applications that require high bandwidth. This is a very important point that should be yet considered.

3.1.3 A ring bus: Cell Processor

The Cell Microprocessor [44, 42, 43, 26] is being developed in cooperation by Sony, Toshiba, and IBM. Unlike the first two processors described this is a commercial product that is intended to be used in Sony's Playstation 3. Due to this fact there is only little detailed information publicly available. Thus I was not able to find more reliable sources than the three mentioned above except for magazine articles. Due to this secrecy the Cell processor is *not* suited for serving as a main model for our modules. It would be almost impossible to implement it realistically enough due to a lot of missing information on architectural details. Since it implements a different approach and confirms some general trends in CMPs I felt that the Cell Processor was still worth mentioning.

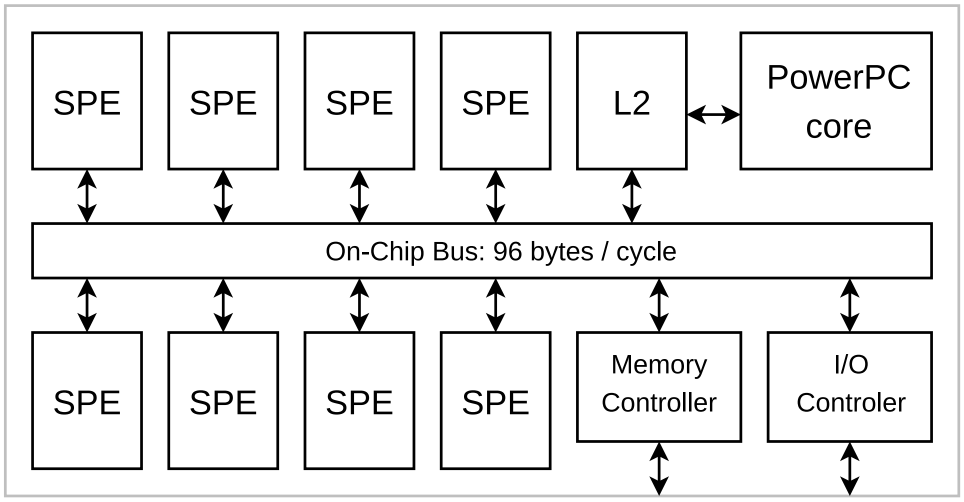


Figure 5: High-level organization of the Cell Processor.

The Cell processor implements a non homogeneous CMP. Figure 5 displays its general structure. It contains one master CPU and eight slave cores. The master CPU is a 64 bit dual threaded *Simultaneous Multithreading (SMT)* PowerPC core. SMT means that multiple threads are executed in parallel on a machine in order to exploit ILP. In the case of the Cell Processors master CPU two in-order pipelines allow for the execution of two threads in parallel. This core is equipped with primary 32 KB instruction and data caches and a 512 KB level two cache.

The eight smaller cores are called *Synergistic Processor Elements (SPE)*. These cores are dual issue *SIMD*[6] processors. This is why they are sometimes also called *SIMD Processor Elements (SPE)*. A new ISA has been defined in order to allow for addressing of a 128 entry register file. Each entry is 128 bits wide. Thus each of the two threads of a SPE is capable of operating on four 32 bit values in parallel. Each SPE has a local 256 KB static RAM. The SPEs abstain from the use of a cache hierarchy. Instead the master CPU can instruct them to copy a certain location of main memory via *DMA*[7] into its local SRAM. In the same manner the master CPU can later instruct the SPEs to copy the data back again.

All SPEs as well as the master CPU are connected to memory and I/O

[6]Single Instruction Multiple Data: This means that one instruction of the processor operates on multiple values of data in parallel.

[7]Direct Memory Access: A method were not software but a special hardware resource manages the transfer of data blocks from one memory address to another or to a completely different memory component as in the case of the Cell Processor.

via a high performance bus called Element Interface Bus (EIB). It consists of 4 unidirectional data rings, two in one direction and two in the other. The rings run at half the processor frequency and transport a total of up to 96 bytes per cycle. Each ring supports up to three simultaneously transfers when these are done between neighboring nodes. Thus the EIB hardware does more resemble a network than a real bus.

3.1.4 A crossbar interconnect

A crossbar can be seen in Figure 6. In general a crossbar connects n components on one side with m components on the other side. The particularity of a crossbar is, that it allows for multiple parallel connections if these are independent. Let us take for example a crossbar that connects eight CPU cores to four secondary cache banks. In this case any four of the processors can access the level two cache in parallel as long as they each access a different cache bank. In order to allow for this the crossbar needs four internal buses. Each cache bank is connected with exactly on of these buses (and each with a different). Opposite to this each CPU core is connected to all four buses. When any of the cores makes a request to access one of the cache banks it will be connected to the appropriate bus given that this bus is not currently in use. An internal arbiter controls accesses and eventually puts a CPU core on wait if two or more cores try to access the same cache bank in parallel. Crossbars are not only used for huge buses. In each of the routers as used by the RAW Microprocessor a crossbar is used. The static router even uses two.

However it is evident that a crossbar scales by the square of the number

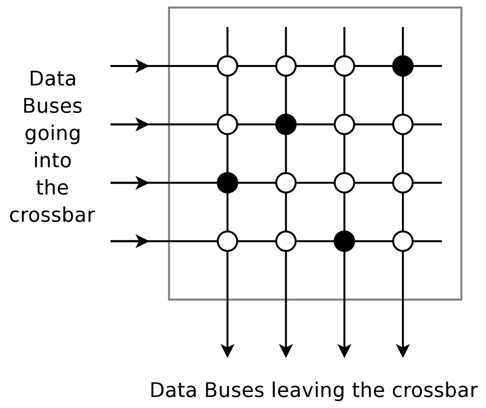

Data Buses going into the crossbar
Data Buses leaving the crossbar

Figure 6: A 4x4 crossbar. Data arriving on the inports of the crossbar (left) can be routed in an arbitrary manner to the outports (bottom) by setting the appropriate connections (dark dots).

of components connected to it. In the example above we needed four buses and 32 connections to these. Thus I have found no implementations using a crossbar as a chip interconnect with more than eight processors attached. A recent research effort [36] modeled such a crossbar connecting eight cores with a shared level two cache. It confirms the huge area overhead generated by a crossbar interconnect. It has to be mentioned though that in this paper area overhead was also very dependant on the mechanism of cache sharing. Thus the results are hard to interpret regarding a simple crossbar that connects for example only processors.

3.1.5 SPIN

SPIN stands for Scalable Programmable Integrated Network. While the RAW networks are adapted to the processors needs, SPIN has been designed to serve as an interconnect for systems on chip (SoC) in general [8]. SPIN is a packet switching network that consists of routers as the network nodes and bidirectional point to point links as connections. Additional wrappers server as an interface to the network that allows clients to send and receive messages. These wrappers translate address space requests into packets with the according node address and inject them into the network. When the packets arrives at its destination a wrapper re-translates it into an address space request. The packets send over the network are split into 32bit pieces that are called *flits* (*fl*ow control un*its*). Each packet contains a header flit with the destination address and packet type and size. The packets are terminated by a control flit that contains a check sum (CRC). The packets are wormholed through the network by the routers.

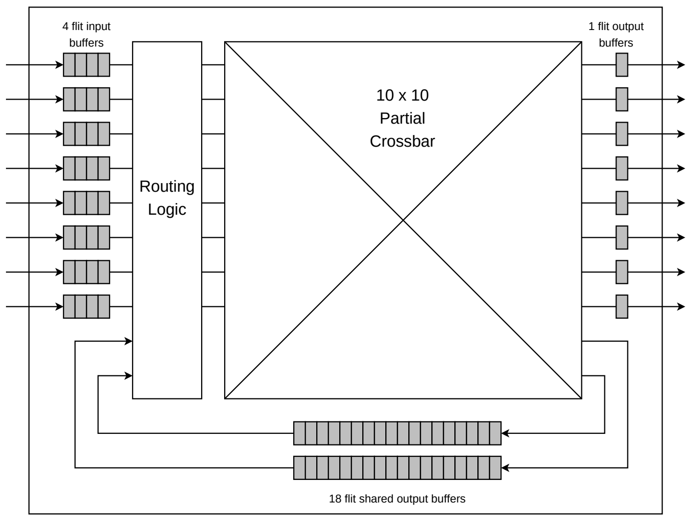


Figure 7: RSPIN: The router for the SPIN network.

A router for SPIN, called RSPIN, can be seen in figure 7. It has eight ingoing and eight outgoing connections (network links). The ingoing connections contain buffers for up to four flits each and the outgoing connections contain a buffer for one flit each. In addition each RSPIN contains two 18 flit output buffers that are shared by all incoming connections. These buffers have greater priority when competing with the input buffers for an output link. This allows to reduce contention [7]. The central component of the RSPIN is a 10 x 10 partial crossbar. This crossbar is only partial since it is designed for a special network topology and only certain connections between links are possible.

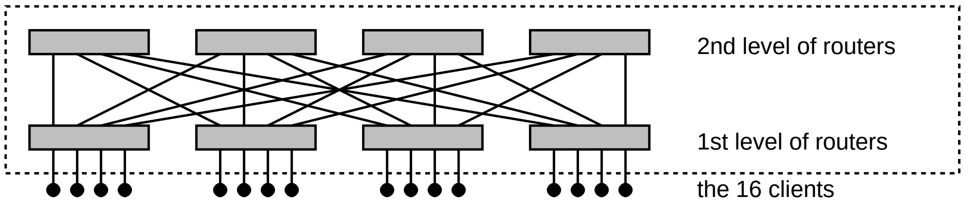


Figure 8: The FAT tree topology of SPIN.

The topology that has been chosen by the SPIN designers is called a fat tree. This is a tree structure with routers on the nodes and terminals on the leaves of the network graph. In a fat tree each node can contain multiple fathers and guarantees enough bandwidth to the higher level for all of its children. An example of a fat-tree topology with 16 clients (leaves) can be seen in figure 8. Another fat-tree can be seen in figure 9. It is evident that fat-tree topologies are scalable.

For evaluation the network has been compared to a PI[8] bus. In the test between four and 32 cores were used and delivery time was measured to deliver all messages in a *pooling*. Pooling means that each client sends messages to all other clients. Another measurement was the latency of messages when varying overall network load. Results showed that SPIN beats the PI bus in latency time when at least 16 clients were used. It is important to note that this result was only received when SPIN used a special *split* protocol allowing it to send a second request (message) before the first one was answered by the target. If this protocol was not used SPIN produced higher latencies than the PI bus even with 32 clients. In the saturation test SPIN offered a lot more possible load by saturating for an offered load of 28% of the maximum possible load. The PI bus saturated already with an offered load of 4%.

[8]Peripheral Interface

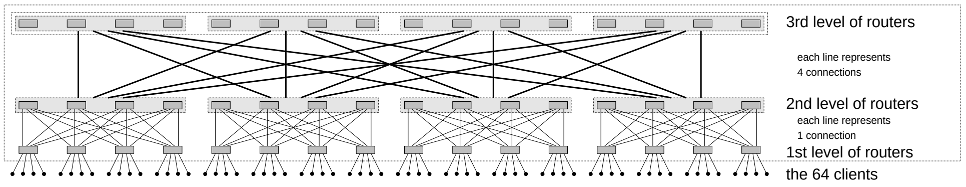


Figure 9: A 3 level fat-tree topology. The thick connection lines connect each router from a group of four with the correspondint router from a second group of four routers.

3.2 Selecting a Network on Chip for CMP

When selecting an interconnection network I was in particular looking for a solution that would scale well beyond hundreds of processors. As possible candidates I have considered the interconnects presented above. They were:

- A shared bus system like in the Hydra CMP.
- A virtual shared bus system that is implemented using a small network (four rings) as in the Cell processor.
- Statically or dynamically routed networks, like in the RAW Microprocessor or SPIN.
- A crossbar interconnect.

As I have mentioned in section 3.1.1 when talking about Hydra the shared bus system suffers from lack of scalability. A recent research effort has further examined this problem by connecting 4, 8, and 16 cores via a shared bus. The percentual performance decrease due to connection overhead grew with the number of processors [36]. The same was true for a crossbar and when looking at current solutions it is clear that a crossbar for a hundred processors is virtually impossible.

There is still another problem encountered with the interconnection architectures just mentioned. As well the shared bus as the crossbar suffer from a common disease: they rely on a global control of the information traffic. But this method is limited in scalability due to the fact that the interconnect needs to know each components state [28]. Thus neither a shared bus solution nor a crossbar seemed to suit the requirements for a highly scalable interconnection component.

The Cell Processor has solved the connection issue by implementing a virtual bus that consists really of a small network on chip. This solution is similar to SPIN where a complete network on chip is virtually hidden to the clients by introducing wrappers that translate address space requests into packets. RAWs solution differs in the fact that the clients actually *see* the network. They are responsible themselves to correctly address packages and for their coherency.

While all these network based approaches have not really been tested beyond a handfull of processors they seems to be much more promising than a simple hardware bus or crossbar. The RAW processor actually succeeded in connecting the most cores. And even though I doubt that RAWs architecture will flawlessly allow to connect hundreds of corres with its current topology further research might find new network topologies that solve this problem. Considering that three of the solutions proposed were based on a network (RAW, Cell, SPIN) I have decided to implement a packet-switched network module. I have tried to keep the module as general as possible only hardcoding the most necessary structures for packet-switching. This way I have left the freedom to the user of the module to actually implement the details by setting parameters and adding code.

3.3 Selecting a core for CMP

Above I have presented three existing Chip Multiprocessors. The first two, the Stanford Hydra CMP and the MIT RAW Microprocessor, are research projects. The third processor presented, the Cell Processor, is a commercial product that is currently in development. While there are other CMPs I believe these processors cover a wide range of possibilities. In fact the Hydra CMP and the RAW Microprocessor are the most referenced CMPs found in research papers. The Cell Processor seems to be the first high-performance general-purpose commercial product that implements as many as 9 separate cores on a single chip. Since I was looking for structures that would scale to huge numbers of processors I have not investigated further CMPs like the IBM Power 4 which implements two superscalar cores on a single chip [31, 41].

When looking at the three cores presented several similarities stand out:

- All cores have some type of local instruction and data storage. These are either level one caches or simple SRAM.
- Each of the CMPs uses in-order execution in their cores. Even the master CPU of the Cell Processor is in-order.
- The cores are all held simple executing RISC code. In fact the Cell Processor is even the only one using SMT. All other cores are strict single-issue processors.
- The Cell Processor is the only one even having a special master CPU. The other architectures are completely homogeneous and the master is chosen in software by the operating system for example.

The first point is quite logical and inherited from the single processor. In order to allow for fast execution the instructions and data worked on have to be as close to the compute elements as possible. This is required to reduce latency due to wire delay. Thus the implementation of level one caches for instructions and data is essential even in the most simple cores in order to have an efficient execution.

The second and third point results from the fact that the consumend area of the processor increases drastically with its complexity. As noticed in [40] the gain from more cores due to saved space is higher than the loss due to in-order execution. Thus I have decided to implement a simple core that was designed for resuse. Because of this the use of existing models like PowerPC750, OoOSysC or even SimpleScalar [15, 22, 14] as cores was not an option.

The forth point did not change anything in the choice of the core. It is worth mentioning that my implementation does well support such a master. One can easily branch more complex processor modules like the PowerPC750 or OoOSysC to a constructed simulation network with lots of basic cores.

Further points to consider were the *instruction set architecture (ISA)* and that the processor be not out of date. I considered the *MIPS* and the *PowerPC* ISA since they have been used in the existing solutions. Also both are very widespread and supported by *gcc*, a very important point when choosing the ISA since researchers need to be able to compile test programs.

I finally decided for the *PowerPC* version since our team had already experience with it and since we had a valid *PowerPC* emulator to verify my results. As a concrete processor I chose the *PowerPC 405*. My choice was motivated by the fact that it is a PowerPC core specially designed for the use in systems on chip [18]. It implements a 5 state in-order pipeline and contains separate instruction and data primary caches. This suited perfectly the requirements for local storage in each core. The PowerPC 405 is commonly used on the market segment an still being supported [31, 30]. Because of that it is relatively[9] well documented [17, 18]. Thus the PowerPC 405 suited all the requirements.

3.4 Selecting a memory

A last decision I made and which needs no further discussion, is to implement a memory module. This module could serve as the central memory or be used locally for each processor core. Again I have tried to make it as flexible as possible by allowing the memory to have as many ports as desired and to adjust timing.

[9] Architectural details of commercial processors are often kept secret by the companies. With relatively I mean that human-readable information is available that is far more detailed than that available on some other processors. Please refer to section 4.1.1.

4 Implementation of the modules

This section describes architectural details about the three modules implemented. These modules are a PowerPC 405 module, a network on chip (NoC) module, and a memory module as decided in the previous section.

4.1 The PowerPC 405 processor module

Developement and coding of the PowerPC 405 processor module required by far the most time. Thus I will start by naming some common problems encountered when working on the PowerPC 405 module. Next I will give an introducing the hardware architecture of the PowerPC 405 chip followed by some key facts about the PowerPC 405 ISA. The final part concerns the actual implementation of the module for SystemC. It talks also about how I tried to improve simulation speed.

4.1.1 Problems encountered

The following list names only a few difficulties encountered when implementing a processor:

- The manuals and white papers often contain only sketchy information that mostly creates more new questions than it answers.
- Processors often split incoming instructions into multiple microinstructions. This behavior is left completely undocumented and the developer has to make guesses based on bits of information gathered from the most unlikely places of the manual or from the web community.
- Choices made regarding implementation techniques can later on create new problems. Using the host system to execute system calls for example creates the need to have direct accesse into simulated memory instead of using the simulated datapaths.
- Debugging is very time consuming due to the complexity of the structured model. It can be shortened by generating a lot of output during execution. But even then the error has to be found in this output.
- Finally in order to test and debug certain parts of the module each time new testing code has to be written and compiled. In special cases this includes time-killing coding in assembler.

I felt that these problems are worth mentioning since they are specific to this domain and do not just result from typical problems encountered during code development. The research team I have been working with has confirmed to me that the mentioned issues are mostly unavoidable. Some are also mentioned in [5]. There is no real solution to these matters. I have tried to make my code as readable and documented as possible in order to facilitate debugging and reuse by others. When a *verbose* variable is set the simulator will print almost every of its actions on the screen in detail allowing again for better debugging. This has helped me save a lot of hours during development I believe. Unfortunately the problem mentioned concerning the documentation has cost me a lot of time. It happend to me multiple times that I had implemented a feature into the processor and have later found more detailed information that made me change the implementation. The reason for this was not that I have not searched exhaustively enough, but that this information is given on completely unexpected placed.

4.1.2 Hardware architecture

Figure 10 shows the PowerPC 405 block diagram. The PowerPC 405 consists of three parts: the *CPU*, the *MMU* and the *caches*. There are two separate caches, one for instructions and one for data.

Please note that the MMU can be simply left away. Since I am only executing user code at the moment this is what I did in order to save development time and increase simulation speed. This is also why I am not going to further investigate its functions in this report. For future implementations the MMU can be added relatively fast since the code is structured in a very clear manner in the area where CPU and caches interact. The same is true

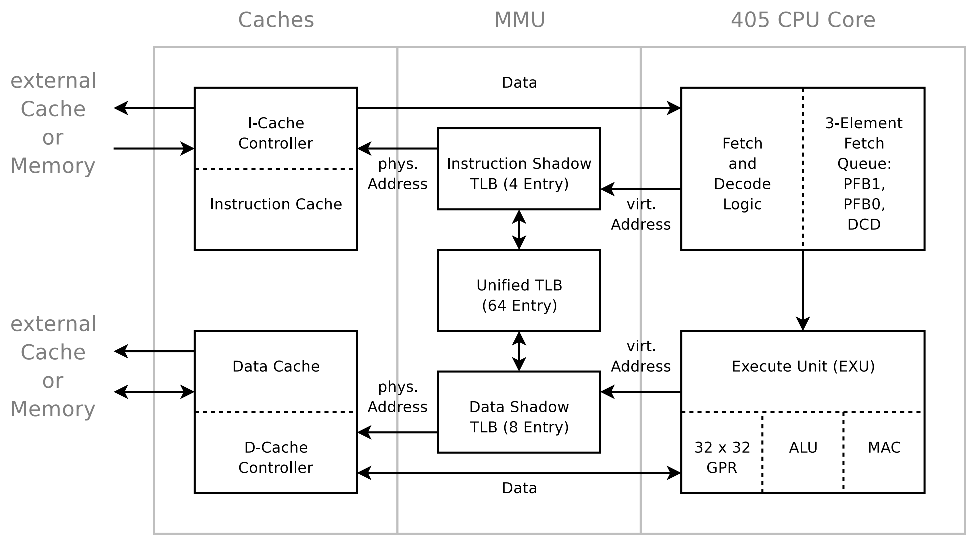


Figure 10: PowerPC 405 CPU core block diagram. Note that the MMU can be switched of in the PowerPC 405. Since we are momentarily only executing user code we simply left out the MMU in our first implementation.

for the processors handling of exceptions. Since I run test programs only in user mode and suppose them to be correct no exceptions need to be handled. I have thus not implemented the exception handling in the simulator.

The CPU contains *five pipeline stages*. They are: *instruction fetch (IF)*, *instruction decode (ID)*, *execute (EX)*, *register write-back (WB)*, and *load write-back (LWB)*. These stages are implemented by two units: the *instruction fetch unit* and the *execute unit*.

The *instruction fetch unit* containts the *instruction fetcher*. It implements a three stage *instruction queue* with its elements in the following order: *prefetch buffer 1 (PFB1)*, *prefetch buffer 0 (PFB0)*, *decode (DCD)*. Instructions are fetched two at a time along the predicted path. *Branch prediction* is done in DCD and PFB0. If for example a branch instruction in PFB0 is predicted taken than the predicted path is fetched into PFB1. PFB1 is flushed before if it already contained the next instruction. When the instruction queue is empty instructions are fetched directly into the DCD buffer. The Instruction contained in DCD is decoded during the ID stage. It can then pass into the Execute stage next cycle. All instructions must pass through DCD before entering the execute unit.

The *execute unit (EXU)* contains the *arithmetic logic unit (ALU)*, the *multiply-accumulate unit (MAC)*, and the *register file (RF)*. It is a *singe issue* unit, meaning that it can only start executing one instruction at a time. Instructions are executed in-order with an exception being load instructions. These can wait for the data to arrive from memory while following instructions continue execution given that they are independent of the load result. All results except for load results are written into the register file during the write-back stage. The results of load instructions are written back during the load write-back stage in witch those instructions also wait for the data to arrive from memory. Thus all but load instructions retire in the write-back stage and load instructions retire in the load write-back stage. This has an important effect when using an emulator for simulator verification. Please refer to Section 4.1.5 for details.

Internally the PowerPC 405 core architecture is *big endian* but it supports both *little endian* and *big endian* data format. This further increases compatibility with other hardware blocks being placed on the same chip or externally. In order to provide this feature the instruction set contains *reverse order load/store instructions*. In addition the instruction cache rearranges instructions stored in *little endian format* before storing them in the cache array.

The instruction cache is read only. It is connected by an unidirectional eight byte bus to the fetcher. The data cache is connected to the execute unit using a bidirectional eight byte bus. The bus allows the cache to simultaneously accept a new request and answer an outstanding one. The widths of the buses allow for fetching two instructions at a time and writing and reading two registers at a time for certain multiple load/store instructions.

Both caches are two-way-associative with a line width of 32 bytes. The total number of cache lines depends on the size of the cache[10]. Since the caches are independent they can be of different size. Both caches have 32 byte line fill buffers. These buffers allow for further cache requests to be served if they are cache hits during an on-going line fill. For non-cacheable line accesses both caches utilize the line fill buffer even thought the line is not going to be written into the cache array. The data read stays in the buffers until it is overwritten by a subsequent memory request. This way subsequent non-cacheable request can be served a lot faster since they do not need to load the corresponding line multiple times.

In addition the the properties mentioned above the data cache contains a two line flush queue and a flush buffer. Data lines to be flushed are first placed in the flush queue and then moved into the flush buffer. From there they are written to the memory. This way a single ongoing flush does not block the data cache pipeline. The data cache supports both the *write-through* and the *write-back* strategies.

To control the behavior of the caches special purpose registers are being used by the PowerPC 405. These 32 bit registers are the *Data Cache Write-through Register*, the *Data Cache Cachability Register*, the *Instruction Cache Cachability Register*, and the *Storage Little-Endian Register*. When the PowerPC 405 operates in real mode each bit of these registers controls one 128 MB region of the 4 GB address space. Bit 0 sets the properties for the lowest-order region and ascending bits control ascending memory regions.

The caches are supposed to be connected to an 8 byte wide system bus. This is rather unusual since most processors have buses that support a cache line read in on access [12].

4.1.3 Instruction Set Architecture

The ISA of the PowerPC 405 is actually the PowerPC ISA with a small set of new instructions. These new instruction can be seen in table 2. They are very implementation specific and a big part of them needed not to be implemented in the module since they are for operating system use only. The module I wrote does decode all PowerPC 405 instructions correctly. Thus instructions whose *functionality* I have not implemented are skipped and simulation continues. This allows future users to faster add the functionality if desired.

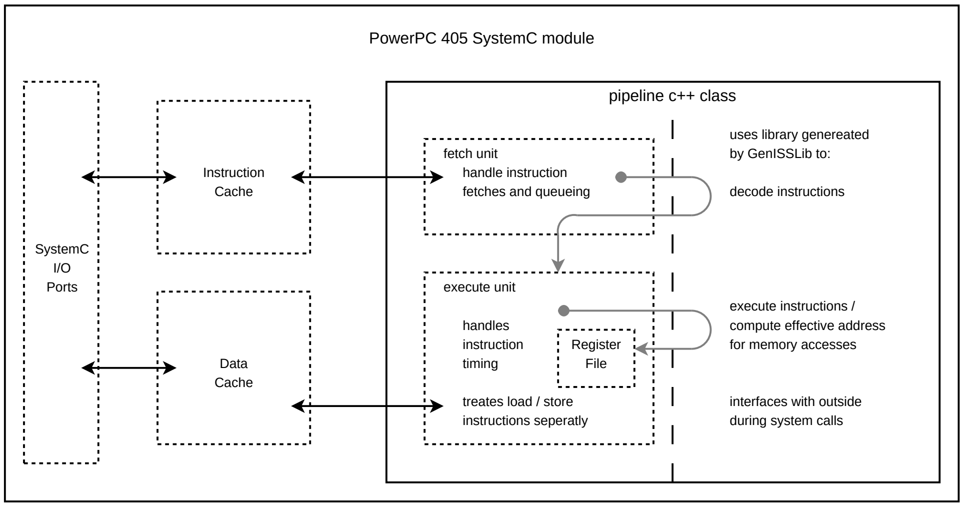
4.1.4 Implementation

The PowerPC 405 module is implemented with two port groups to the outside. One is for instruction cache memory accesses (read-only). The other

[10]Commercially available sizes are (per cache): 0 KB (no cache), 4 KB (64 lines), 8 KB (128 lines), 16 KB (256 lines), 32 KB (256 lines), and 64 KB (512 lines) [18].

Name	Description
Cache invalidation and debug instructions	These instructions serve to invalidate the whole cache on power up and to debug programs. They are for operating system use and thus not implemented in the module.
MAC instructions	The MAC instructions are used for fast multiplications and access the additional MAC unit of the PowerPC 405 core. They are implemented in the module.
TLB instructions	These instructions are used for the management of the translation lookaside buffers (TLB). Since the MMU is not implemented in the module these instructions are ignored.
Data control register read/write instructions	The data control registers (DCR) serve for interfacing external components in embedded systems. These instructions are ignored in the implemented module.
External and critical Interrupt control instructions	These instruction serve to enable and disable external interrupts and to return from critical interrupt handling routines. They are for operating system use only and thus not implemented in the module.

Table 2: Implementation-specific instructions of the PowerPC 405 compared to the standard PowerPC ISA.



PowerPC 405 SystemC module
SystemC I/O Ports
Instruction Cache
Data Cache
pipeline c++ class
fetch unit
handle instruction fetches and queueing
uses library genereated by GenISSLib to:
decode instructions
execute unit
handles instruction timing
Register File
treates load / store instructions seperatly
execute instructions / compute effective address for memory accesses
interfaces with outside during system calls

Figure 11: The internal structure of the PowerPC 405 module.

is used by the data cache for memory read and write accesses. The module is implemented in SystemC. This allowed me to use GenISSLib [4] for the creation of the ISA interface. GenISSLib takes input files similar to VHDL that can contain C++ code to be executed for each instruction. In addition a CPU model has to be defined on which the instructions operate. GenISSLib is ideal for creating emulators[11]. But with only small changes to the ISA description files used as its input the resulting library is suitable for decoding all instructions for the simulator. GenISSLib is perfectly suited for this task since it lets the user define what information to extract from the binary instructions and how to present it. Also, when decoding, libraries generated with GenISSLib use a hash based on the instruction address. This creates an enormous speedup for instructions in a loop for example. The possible drawback is, that it assumes constant instruction memory. Should this not be the case it is of course possible to turn off the hashing.

GenISSLib can also even handle the execution of simple instructions, like arithmetic ones. In fact, since I wanted the best possible speed I have decided to leave the biggest possible part of the internal execution to the emulator library generated by GenISSLib. This led to an internal structure of my module as depicted in figure 11. The state of the processor is in fact mainly kept in the emulator, namely the register file, status registers and the instruction counter. The emulator is encapsulated by a C++ class called `pipeline` that maintains the state of the `fetch unit` and the `execute unit`. The `fetch unit` simulates the branch prediction and the instruction prefetching. The `execute unit` is responsible for filtering out instructions that cannot be correctly simulated by the emulator alone. These are *load and store* instructions as well as *system calls*. In fact system calls are executed by the emulator. But they need special attention regarding memory access. For this each component that implements some part of the memory needs a backdoor for read and write operations. This includes as well the memory module as the caches.

There is also a special case regarding system calls, namely when a second emulator is run in parallel for verification of the simulator. In this case the `execute unit` has to make sure that only the emulator actually executes the system call. After that the result needs to be copied into simulation memory and caches.

When handling load and store instruction the GenISSLib library used for the simulator does actually nothing except for decoding the instruction and calculating the effective address. The `execute unit` finally simulates the memory access which is done internal (in the PowerPC 405 module) to the data cache.

The instruction cache and the data cache are both implemented in the outermost part, the SystemC module-class definition. Both offer access

[11]In fact this is how I generated the emulator used for verification (see Section 4.1.5).

methods to the pipeline class. This way they can be easily shut out by simply changing the code in these methods. These methods enqueue the new requests if possible and signal this to the pipeline. At the beginning of each cycle methods are called that treat the dataflow of the cache pipelines and do the necessary port communication with the outside buses.

In the case of an exception the simulation will simply stop.

4.1.5 Verification with an emulator

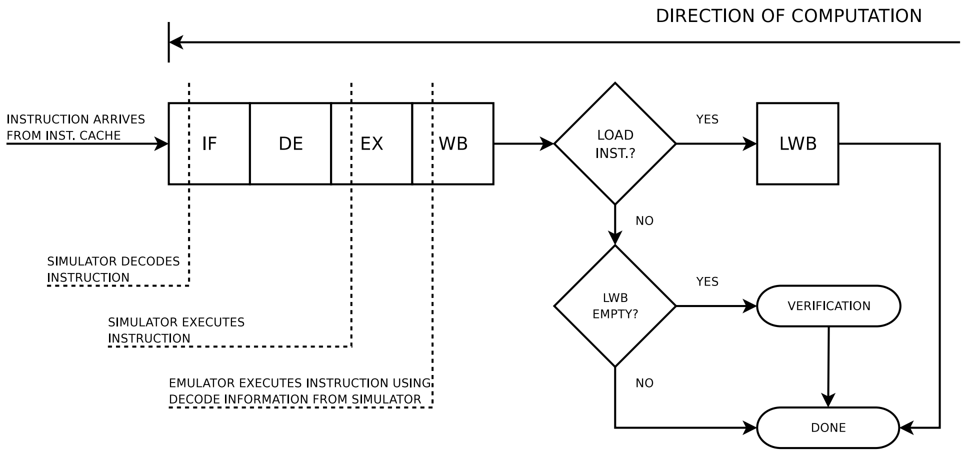

DIRECTION OF COMPUTATION
INSTRUCTION ARRIVES FROM INST. CACHE
IF
DE
EX
WB
LOAD INST.?
YES
LWB
NO
LWB EMPTY?
YES
VERIFICATION
NO
DONE
SIMULATOR DECODES INSTRUCTION
SIMULATOR EXECUTES INSTRUCTION
EMULATOR EXECUTES INSTRUCTION USING DECODE INFORMATION FROM SIMULATOR

Figure 12: The pipeline compution process of the PowerPC 405 module.

As already mentioned the simulator can be run in parallel with an emulator. In fact, when instantiating the simulator module a pointer to an emulator and an emulator CPU instance (to guard the state of the emulator CPU) can be passed. This way, if I simulate a shared-memory model with multiple processors, I simply create one emulator with the desired number of processors. Then I pass the pointer to the same emulator to each processor simulation module and a pointer to its corresponding emulation CPU.

The emulation feature of course slows down overall performance when used. But there are two reasons for its use. First of all, it serves to verify the simulator correctness. When a new module is branched to the processor the emulator can be used to see if everything works as it is supposed to do. When changing the used memory module for example I was able to check that my program still behaved correct by running the emulator in parallel. But mainly the emulator becomes irreplaceable when debugging. When using the emulator the simulator will stop on the first caught [12] occurrence of a

[12]In some cases the result of instructions cannot be compared directly and the difference in results will be noticed in the next instruction. This special case occurs in combination with load instructions. For details please continue reading Section 4.1.5.

difference between simulator processor state and emulator processor state. It dumps information that is crucial in debugging, namely when and where the difference was found and what the processor state is (register file and pipeline).

Verification is done when instructions commit in the *write-back* stage of the pipeline, i.e. when the result of an operation is written into the register file. Following this the same instruction is executed in the emulator. Afterwards the register files and the PCs of simulator and emulator are compared. Unfortunately when a load instruction is waiting in the *load write-back* stage such a comparison is not possible. This is due to the fact, that newer instructions might retire that are independent of the load instruction waiting for its result. Thus in this special case the simulator processor behaves not completely in order and its state differs from that of the emulator processor who executes all instructions atomic and in order. Due to this constraint verification takes only place for cycles in which an instruction retires in the *write-back* stage and the *load write-back* stage contains no waiting load instruction. Independent of this every instruction that passes through the *write-back* stage is executed in the emulator, even load instructions.

4.2 The Network on Chip module

I will start by describing the structure of the network I implemented and how I tried to make it as adjustable as possible. After that I will speak about the actual implementation which focuses also on simulation speed.

4.2.1 Architecture

As described in section 3 I have tried to implement a NoC module that was generic enough to allow for the simulation of various kinds of packet-switched interconnection networks. SPIN served me as a model since it was the most versatile network of all those mentioned in section 3. But the module I have implemented is not a clone of SPIN. It can be configured to model SPIN but it can in general model any topology and allows for far bigger flexibility than SPIN does with its modules. With only little extensions the NoC module can be also turned into the EIB of the Cell Processor or a dynamic network as used in the RAW Microprocessor. In theory it can be also transformed to model the static network of the RAW Microprocessor. But since this network is programable and its routers fetch instructions from memory this would require larger changes to the module.

In order to allow for high flexibility I have made the following design decisions for the NoC module:

- The connection links are unidirectional. In order to simulate a bidirectional link simply two unidirectional links are used. This way the user can implement more network topologies. It allows for example to implement the network rings used in the EIB of the Cell Processor.

- Routers can contain any number of incoming and outgoing links and almost arbitrary routing rules can be defined. This way the user can implement different topologies and different arbitration techniques. Also he should be able to change the router to implement packet-switching instead of worm-hole routing by applying only small changes.
- There are wrappers that serve in the same way SPIN wrappers served.
- The network should be implemented in a way that it runs as fast as possible.

These decision are quite general and were confined a little more during implementation.

4.2.2 Implementation

First of all it needs to be said that I have implement one big SystemC module that hides the whole network inside. The alternative would be to implement every part of the network as a separate SystemC module. The advantage would be that the user is already familiar with SystemC and could easily use the modules or even add new ones if needed. Unfortunately the SystemC overhead due to module communication would ruin simulation speed since there would be simply to many modules. Alone in the Simple fat-tree network modeled by SPIN (figure 8) there are 32 full duplex links and eight routers, a total of at least 40 components that would have to be connected. The number of SystemC ports to connect those would be still higher. This is why I have implemented the network topology as simple C++ structs that hold the state of all components. In addition the NoC module contains code that changes the state of these components accordingly to what happens on the SystemC ports that connect the NoC module with the clients. This code is split into three parts so that changes can be easily applied. Each part is responsible for one type of internal components. As decided above they are: a wrapper, a router, and a link. Each component can be instantiated inside the NoC module as many times a needed.

For a quick overview of the role and parameters of each component please refer to Table 3.

The NoC module is connected with other modules through an adjustable number of SystemC ports. The treatment of the dataflow on these ports is left to the code responsible for the wrappers. Each wrapper is assigned to one group of inports and one group of outports. In the current implementation these port groups allow for address space requests. This behavior can be changed by altering the part of the code that manipulates the wrapper

Name	**Description**	**Parameters**
Link	Unidirectional pipelined connection.	number of pipeline stages, cycles per pipeline stage
Switch	Routes incoming flits or packets to outgoing links according to internal route-table. Buffers data when desired.	number of connections, route-table, buffer sizes
Wrapper	Manages the data transfer between the SystemC ports and the internal network links. Buffers data when desired. Designed to be completely reprogrammed to suit needs.	buffer sizes

Table 3: The three internal construction elements of the NoC module.

state. The wrapper is also responsible for internal buffering of incoming and outgoing messages and requests. Each wrapper is internally connected to two links: one to send data into the network, one to receive data from the network.

The simplest structure implemented in the NoC module are the *links*. Links represent pipelined unidirectional wires that serve to transport flits across the chip. Each link is represented by an array that contains an entry for each of the pipeline stages of the link. The data width of the links is parametrized on instantiation by assigning it one of the integer datatypes `uint8_t` through `uint64_t`. Since hardware implementations with higher data widths than 64 bits are strongly unrealistic for a NoC due to area consumption I felt that the implemented possibilities are sufficient. Using integers to hold the data instead of further arrays also speeds up the computation. In addition the data values in each pipeline stage can be marked valid or invalid by a boolean flag. Instead of defining array cell number 0 as pipeline stage 0 and moving data from one stage to another, there is an additional integer per link that points to the array cell representing the first pipeline stage of the link. When the data is supposed to advance one pipeline stage really only the pointer to the first stage changes. The disadvantage of this is that the link can be stalled only as a whole, not in separate stages. But this actually represents most hardware implementations. Also again I save a lot of computation time.

The *switches* are a little more complicated. Each switch can contain an arbitrary number of incoming and outgoing links. Incoming links contain at least one buffer to read in the incoming data. Additional buffers can be easily added to the code. The routing is actually done using a routing table that has to be defined during initialization of the simulation. For each possible destination address the routing table contains an arbitrary number of pointers to outgoing links that can be used for this address. They are checked in that order, i.e. when the first link pointed to is occupied already by a different connection the next one is examined. If all possible outgoing links are occupied than the flit stays in the buffer and the incoming link is stalled for the next cycle. Right now the switches implement wormhole routing, so the route table is only examined on the arrival of a header flit and an connection is made between the corresponding in link and out link lasting for as many flits as are designated in the header flit. Nevertheless the switches can be easily changed to implement a different type of packet treatment than wormhole routing since the code is very structured and only small portions need to be changed. The solution with the routing tables allows for very flexible configuration possibilities.

The only network module parameter that cannot be changed at runtime is its number of SystemC ports. Otherwise it can be completely reconfigured during simulation if this should be desired. Parameters such as the number of switches or links, as well as their details like buffer sizes or pipe stages can be adjusted on the fly. In order to initialize the network there are several *topology methods* that initiate the arrays containing the component structs and basic parameters. These topology methods can be called by the user after the NoC module has been instantiated in order to configure it to simulate a chosen topology. Current implemented topologies include but are not limited to: a unidirectional as well as a bidirectional torus, the fat-tree topology of the SPIN network, or a binary tree topology. If a desired network topology is not supported the user can simply add his own.

4.2.3 Possible improvements

In order to further facilitate the configuration and adaption of the NoC module to the users purposes it would be of use to make it far more templated than it already is. Right now code needs to be changed inside the module for certain changes in behavior. Mixins [33] seem to be a promising solution to this problem, since they allow to specify a superclass containing code during module instantiation. It would help the user of the module enormously if he could define the behavior of wrappers by "passing code as an argument" during instantiation instead of changing it inside the C++ class. This is exactly what mixins would provide.

4.3 The memory module

Implementing the memory module was rather a simple task. I have used an C++ class called *MemoryContainer* as the foundation. It was written some time ago by Daniel Gracia Pérez. Externally the component represents a huge memory that can be written to and read from by two interfacing methods. Its theoretical size is defined by the type of the address passing variable that is templated inside the module. For a 32bit unsigned integer (`uint32_t`) as used by my PowerPC 405 module this module allows for 4 GB of storage if enough memory is provided by the system on witch the simulation runs. Internally the MemoryContainer class splits the storage space into pages that are allocated from main memory only when written to or read from. They are organized as concatenated lists in a hash table. Compared to an array based solutions this keep the memory consumption very low, since only really used memory has to be allocated. Nevertheless an adequate speed is guaranteed due to the organization in a hash table. Another advantage when compared to an array based solution is, that the module does not have to know beforehand on which address space the software running on the simulated processors is going to operate.

The SystemC memory module I wrote simply wraps around the MemoryContainer by introducing different access latencies for write and read operations and offering an adjustable number of ports. Special write and read methods are offered for simulator bypass in the case of *system calls*. For details regarding this behavior please refer to section 4.1.4.

With the memory module and the PowerPC 405 module alone a working simulator can already be constructed. Figure 13 shows an implementation containing four CPUs and an eight port memory. In order to be more realistic the memory should be customized to contain an arbitration between those ports. For example a *round-robin* implementation can be implemented that allows one port at a time to access the memory. This is equivalent to a shared bus solution.

5 Communication between different simulation environments

As mentioned in the introduction the second goal of this internship was the investigation of interfacing methods with other simulation environments. This is an important issue in order to futher facilitate reuse of the library components proposed. But the motivation for this research comes not only from this project. The simulation research community in general has expressed the wish for easy reuse of components across simulation environments [38, 5]. So far reuse has been limited since in order to use a component from one simulation environment in another a great part of it has to be rewritten. This is coupled with an enormous overhead in work for each component. Because of this overhead reuse based on recoding is impractical. Thus different solutions needed to be found. In this section I am going to present the two possible solutions I have studied. Both I have successfully used to connect SystemC modules with Liberty modules since these were the two environments I worked with.

The first one is the introducing of a common communication channel between the two simulation environments. In this case simulators from both environments run in parallel and exchange information in-between each other. This communication channel can be either made available by the simulation environments themselves or by newly introduced special modules. The first solution implies changes to both simulation engines. Since these engines are tweaked to reach a maximum possible performance it is not a good idea to temper with them. In addition changing the behavior of a simulation engine like in the case of *LSE* is more than just a simple hack. On the contrary introducing a new module encapsulates the whole communication mechanism. The communication module can be used when needed and only then impacts simulation performance. I have implemented two such modules, one for SystemC and one for Liberty. This modules are described in section 5.1.

The other solution I have investigated is the introduction of *wrappers*. A *wrapper* is a module written in one simulation environment that *wraps* around a module from another simulation environment. The wrapper interfaces the input and output ports of the wrapped module and makes them available to other modules from its own simulation environment. It achieves this by providing the same input and output ports as the wrapped module and copying data between both. A schematic wrapper can be seen in figure 14. In order to be able to make a module from another simulation environment work a wrapper often has to run the corresponding simulation engine. This behavior is also depicted in figure 14. Another wrapper technique is to have a parental simulation environment wrap around the two simulation

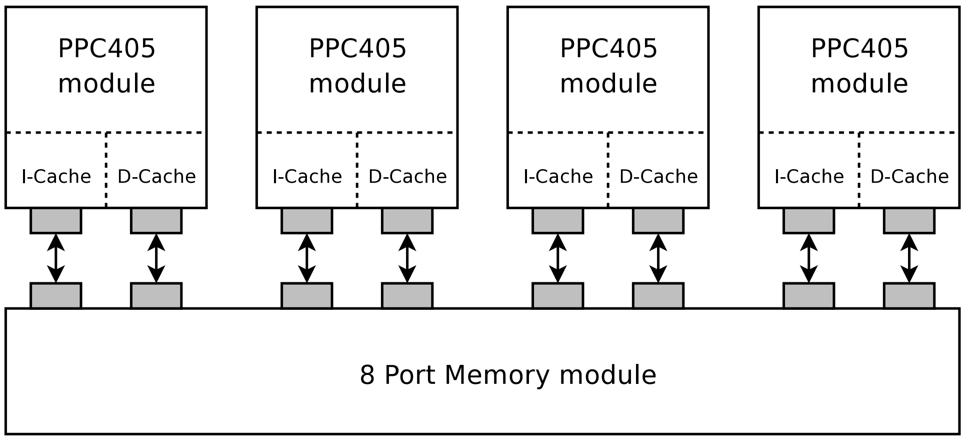


Figure 13: A possible configuration of the memory module and four PowerPC modules to form a working simulator. The gray boxes represent the port groups responsible for memory accesses on each module. This is a shared memory, shared bus implementation. The bus is actually simulated within the memory module by arbiting the ports to allow only one access at a time.

environments to be interfaced. In fact this solution has been proposed by the both, the Alchemy and the Liberty research groups. Both are currently working together on a LSS[13] based wrapper for SystemC and Liberty, i.e. a solution that allows to use SystemC modules together with Liberty modules. I have so far tested this wrapper and am going to further work on it within the next weeks. A short description is given in section 5.2.

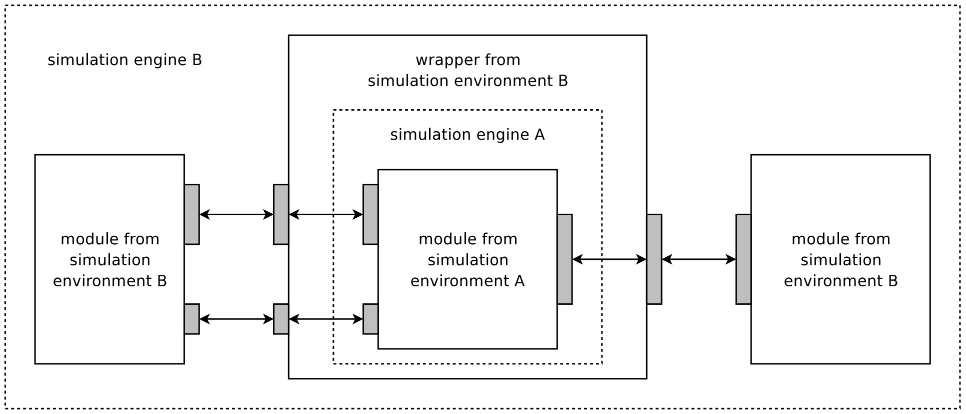

simulation engine B
wrapper from simulation environment B
simulation engine A
module from simulation environment B
module from simulation environment A
module from simulation environment B

Figure 14: Schematic of a wrapper.

5.1 A common communication channel

In order to make two modules from different simulation environments communicate a connection channel between both needs to be established. The most rapid solution would be to have interfacing methods or even a system of shared memory. But this solution necessitates that both modules are created from the same parent process and thus that both simulation environments already have a way of communicating. This is a contradiction to the reason why I wanted to use special modules, namely in order to not have to change the simulation engines.

5.1.1 Socket-based communication

My solution is quite different. I have used sockets provided by the operating system to make two or more modules communicate with each other. The idea for this arose when I was designing the network on chip module (see section 4.2). In this module an actual network was modeled. The concept is to extend this modeled network onto the actual network provided by the operating system in order to make communication with other simulators possible. A virtue of this solution is that it allows to spread the compution of

[13]Liberty Structural Specification language

a simulation on multiple machines. This approach is described more detailed further down in this section.

The downside of the socket based solution is that it introduces additional latency into the computation process of the simulation. Simulation data exchanged between the modules is strictly timed. Thus the receiving module has to wait until the requested data arrives before it can continue the simulation. If such data exchange takes place multiple times during a simulated clock cycle then the simulation speed will become too slow. Thus, a use of sockets in the scope of simulation needs to be well planed in order to achieve any gain.

To keep latency as low as possible a module should thus send data only once in a *simulation cycle* (not a delta cycle!). This is exactly what happens in the NoC module. Data is moved exactly once at the beginning of a new simulation cycle. I decided to exploit this behavior by allowing the user to split the simulated network and connect it with sockets. To do so I have introduced a forth component inside the network on chip module and named it *socket-port*. The *socket-port* can be connected with *links* to other components. This happens in the same way as *links* are used to connect two *switches* or a *switch* and a *port*. The *socket-port* has an adjustable number of incoming and outgoing links. Thus the user can arbitrarily cut the network represented by a network on chip module by splitting the links. An example of this is given in figure 15. A NoC module can now be connected to any other NoC module whether written in SystemC or Liberty.

The *socket-port* allows also for the distribution of the simulation over multiple machines. Already at the beginning of this project I have had the idea to spread the simulation over multiple workstations to allow for faster

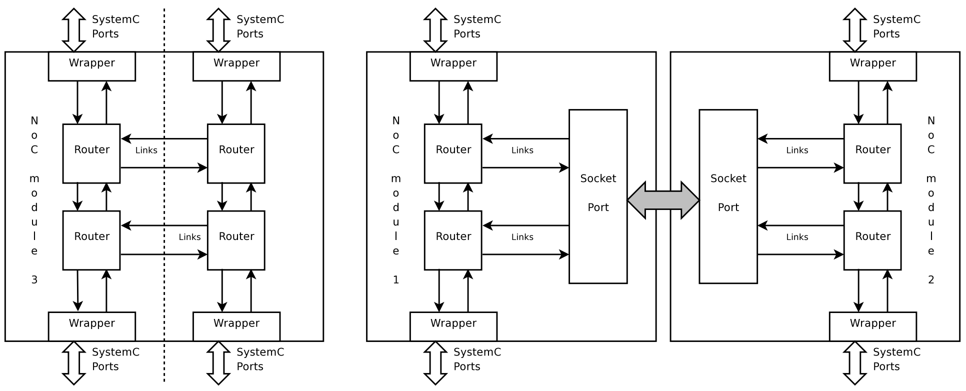


Figure 15: The network to the left (NoC module 3) is equivalent to that on the right (Noc modules 1 & 2) when simulated. The computation of NoC modules 1 & 2 and the components connected to each can be spread over two different workstations.

simulation speeds. This has been also proposed by other researchers [45]. Unfortunately SystemC and Liberty, as well as SimpleScalar for that part, do not provide any way of mapping the hardware description to different workstations. When developing and later coding the NoC module which is heavily packet-oriented I came up with the idea to not only use the operating system sockets to simply communicate between two modules on one machine but to route these packets in-between different workstations. In fact my idea was to run the computions for a few processors on each workstation and have the local part of the NoC implemented on each workstation. The connections in-between these workstations could than be mapped to *tcp*. Figure 16 shows how the NoC modules and the server can be used to distribute computation on multiple machines and at the same time interface SystemC with Liberty.

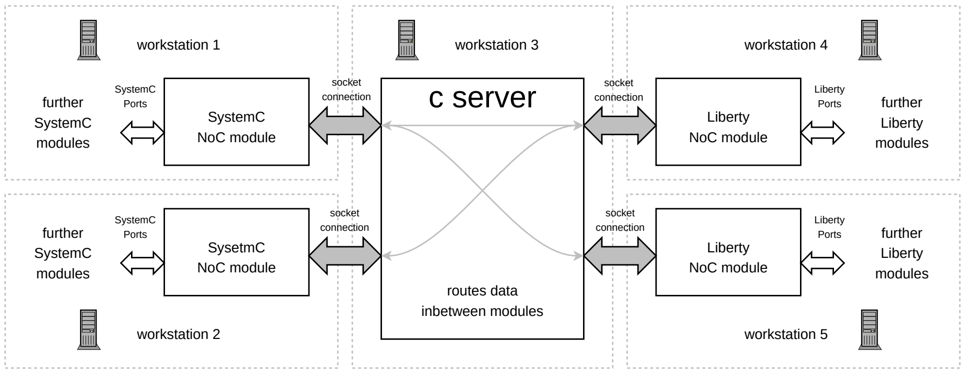


Figure 16: Distribution of computation on different machines with using NoC modules and a server written in c.

5.1.2 Implementation

Internally the socketport works similar to a switch. During initialization of the NoC module a socketport will connect to a server using a system socket. I have used a server instead of connecting directly with the Liberty module since it helped debugging. The server can log the data that passes and allows further control over it. The final product should use the server only in the beginning to create peer-to-peer links and later send the data directly to its destination module. Each socketport identifies itself by a small header during connection and it is the servers responsibility to route the data correctly in-between the socketports. For that the network topology has to be known beforehand and programmed into the server, a task that takes only a few minutes for a simple network as I found.

In order to minimize network traffic I have coded the socketports such that they only send the data that really arrives on the links. For this a header word is used as a 32 bit vector. Each bit describes whether or not data was actually sent for the corresponding link. If more than 32 links are used the header word can be expanded to 64 bits. This allows to connect 64 outgoing and 64 incoming links on each socket. Futher expansion is not foreseen since already connecting 64 links over a socket will generate up to 264 bytes of traffic in one direction per cycle. Supposing a (slow) simulation speed of 10.000 cycles per second and bidirectional link connections this will generate about 5 MB/sec of traffic.

5.1.3 Performance

When testing[14] the new module I found that simulation performance on one system sinks by at least 20% when using the socketport module. This is due to the latency of the sockets. Also in order to maintain a synchronous and cycle correct simulation the socketports have to wait each cycle for the server to tell them if data will or will not be available on each link connected to them. This process blocks the simulation and introduces further speed loss. Thus for connecting two simulation environments on *one computer* sockets are not the ideal solution. Wrappers as presented in section 5.2 seem to have a brighter future for this particular task.

There is another problem presented by the socket based solution. As described in section 4.1.4 I have used an emulator to execute system calls that needed direct access to the memory simulated. The solution I have implemented does not yet support this. Since I have used a shared memory model when simulating the distribution of simulated processors on multiple machines was not possible. In order to be able to do so a second communication channel needs to be introduced that serves for direct atomic memory accesses. This is possible but needs to be implemented yet, for example in the form of a second server. Meanwhile a solution could be to have each processor have a local memory and let system calls access only this memory. The processors could communicate via message passing and thus no further atomic accesses would be necessary across socket-boundaries. I am still working on an simulator implementation that allows to do this.

I hope to be able to examine this behavior further during the last weeks of my internship. Although sockets have proven to be of little use as a connection between simulation environments they might still stand the challenge as a tool to spread simulation over multiple workstations and thus increase overall performance. Once I have finished the above mentioned simulator, speed measurements should lead to a final decision regarding the socket issue.

5.2 An LSS wrapper

The general idea of the wrapper has been already described above. Since it is far from ready yet I will just give a short description of the current

[14]Please see section 6.2 for details on testing and simulation.

implementation. In fact we have been able to simulate the LFSR from section 2 using a xor-gate module written in SystemC as well as a flip-flop module and a tee module taken from Liberty's `corelib`.

The idea behind wrapping a SystemC module is the following. The Liberty Structural Specification (LSS) language is used to describe the general structure of the LFSR. Even when only using `corelib` modules the LSS files serve only as the structural specification. This means that the placement of each module and the interconnections with other modules are all described using LSS. When the LSS engine is run it generates code according to the specifications inside the LSS input files. For the LSS engine it does not matter wheather the module it wraps into the final source files is from Liberty, SystemC, MicroLib or just written in standard C++. This is exactly what we do in order to connect our SystemC xor-gate with the other modules taken from Liberty's `corelib`. Each module gets is own LSS file describing what connections it will have and how it is generated inside the wrapper. The tee and the delay modules LSS files will result in the generation of LSE API calls. The xor-gate LSS file in turn will generate code that runs the SytemC engine when compiled and instanciates the SystemC xor-gate inside it. To facilitate this it encapsulates a special LSS file called `microlib` that automates the wrapping code generation. The xor-gate LSS module needs only to set special parameters of the `microlib` LSS module and the rest is done by this module.

In fact the `microlib` module uses the parameters set by its parent to specify a the detailed code structure that executes the SystemC environment with the SystemC module that is being encapsulated. Additional code is being generated by the LSS engine when processing the `microlib` module that connects SystemC signals to the ports. Further code generated reads and writes these signals to interface the SystemC module.

In general this is what LSS was made for from the beginning. It is a structural specification language: it defines the structure of the LFSR. In addition it defines the structure of each module and what simulation engine is going to be used for that module. In particular in our example we have defined that Liberty's engine should be used for the delay and the tee modules. For the xor-gate we have defined that SystemC should be used. When the LSS engine processes this specification it will generate the appropriate code by following the LSS file guidelines. After further processing by Liberty and finally the compilation with gcc, the simulator can be run.

As an example of how the `microlib` module works internally this is the part of the code:

```
for (i = 0; i < inputs.size; i++) {
  ...

  copy_inputs += <<< *signal_${inputs[i]}_${localname} =
        *${inputs[i]}_data[0];>>>;

    check_inputs_known +=
        <<<LSE_signal_data_known(${inputs[i]}_status[0]) && >>>;
    check_inputs_present +=
        <<<LSE_signal_data_present(${inputs[i]}_status[0]) && >>>;

    ...
}
```

This is the code that creates the Liberty c code that is responsible for sending data towards the SystemC module. The variables used are as follows: `inputs` is an array with the names of the SystemC ports to interface. A unique identifier is represented by `localname`. While `localname` is determined automatically `inputs` is set by the parent module. In the LFSR case this would be the xor-gate module.

The code is encapsulated in a loop over all the inputs set. The code generate in the line that starts with `copy_inputs` is actually responsible for copying the data send on the LSS ports towards the SystemC signals. `copy_inputs` itself only holds the c code to be executed when necessary. There is another LSS userpoint defined that contains this code:

```
    if (${check_inputs_known} !${cycle_finished}) {
      if (${check_inputs_present} 1) {
        /* copy input values */
        ${copy_inputs}
        ...
```

Here the two other variables defined are first used to check if data is actually available on the incoming LSS ports of the `microlib` module. If so the code contained in `copy_inputs` is called.

The other way round works similar. In fact there is code generated for every task necessary: the initialization of the SystemC environment, the execution of each cycle, and as demonstrated the writting and reading of the ports. The usefulness of this module will be enormous once it is ready. Right now there are still things that it fails to achieve: It is not possible to connect arrays of port for example. Further no parameters can be passed to the SystemC module except for template parameters. The possibility to add user defined code is a must in order to allow for a better flexibility. Also there should be interfacing methods introduced that allow for example for bypass accesses to memory as in the case of system calls on a processor. These issues we will be working on in the upcoming weeks in cooperation with the Liberty Research Group.

6 Simulation

In this section I will describe some simulators I have build using the constructed modules. For each simulator constructed I will give some performance data. The configuration of the machines I used to run simulations is given in table 4.

Parameter	**Value**
CPU type	Intel Xeon CPU 2.80 GHz
CPUs	2
Level 1 Cache	512 KB
Memory	2.5 GB
Operating system	Linux version 2.6.8.1-24mdksmp
Bogomips	5505.02

Table 4: Configuration of the workstations on which simulations were run.

6.1 A 16 core network on chip simulator

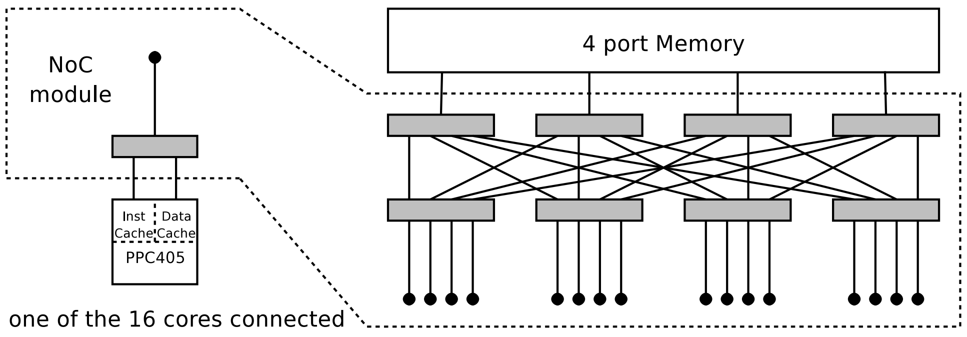


Figure 17: The network structure for the shared bus simulator.

This simulator implements a simple tree topology as shown in Figure 17. 16 CPUs are plugged into the network and connected to a shared 4 port memory. Each CPU is assigned a *rank* ranging from 1 to 16. The CPU with rank 1 is the (*master*), all other CPUs are *slaves*. To allow for each CPU to find out it's rank local memory address $0x00000000$ was set to read only and its value contained the rank number for each CPU. This was hardcoded into the memory module with one simple line of c code. This same program was launched on each CPU and different behavior depended only on the *rank* of each CPU and resulting branching within the code. The simulator stops when any of the processors make an *exit system call*.

The program used is a simple one: the master CPU prints a hello message and signals slave number one. After that the master waits to be signaled. All slaves wait to be signaled and when so, print their hello message and signal the next slave. The last slave signals the master once he is done and the master exists program execution. The program code can be seen in Appendix B. When executed the simulator presents this output:

```
Network On Chip: Building fat tree network topology
    for 4 memory ports and 16 CPUs.
Initializing SystemC
****************
MASTER: I'm the master CPU!
SLAVE(2): Hello World!
SLAVE(3): Hello World!
SLAVE(4): Hello World!
SLAVE(5): Hello World!
SLAVE(6): Hello World!
SLAVE(7): Hello World!
SLAVE(8): Hello World!
SLAVE(9): Hello World!
SLAVE(10): Hello World!
SLAVE(11): Hello World!
SLAVE(12): Hello World!
SLAVE(13): Hello World!
SLAVE(14): Hello World!
SLAVE(15): Hello World!
SLAVE(16): Hello World!
MASTER: That's all, folks!
PowerPC 405 CPU (#0) halted.
sim_total_time 8 # seconds
sim_cycle 217590 # clock cycles
sim_cycle_rate 26215.7 # cycles/sec
PowerPC 405 CPU (#0): Instructions Executed: 24651 #
PowerPC 405 CPU (#1): Instructions Executed: 163168 #
PowerPC 405 CPU (#2): Instructions Executed: 154342 #
PowerPC 405 CPU (#3): Instructions Executed: 146627 #
PowerPC 405 CPU (#4): Instructions Executed: 137975 #
PowerPC 405 CPU (#5): Instructions Executed: 130132 #
PowerPC 405 CPU (#6): Instructions Executed: 121917 #
PowerPC 405 CPU (#7): Instructions Executed: 112983 #
PowerPC 405 CPU (#8): Instructions Executed: 104608 #
PowerPC 405 CPU (#9): Instructions Executed: 95849 #
PowerPC 405 CPU (#10): Instructions Executed: 87138 #
PowerPC 405 CPU (#11): Instructions Executed: 78039 #
PowerPC 405 CPU (#12): Instructions Executed: 69460 #
PowerPC 405 CPU (#13): Instructions Executed: 60481 #
PowerPC 405 CPU (#14): Instructions Executed: 51714 #
PowerPC 405 CPU (#15): Instructions Executed: 43191 #
```

As expected the CPUs have printed their messages in the order of their rank. Following that the master CPU has halted (`PowerPC 405 CPU (#0) halted.`) and simulation a stopped. The simulator prints the time in seconds it took for the simulation to exit, the number of clock cycles simulated, and the simulation speed in simulated cycles per second. The average of ten successive measurements on my test machine was 27031 cycles per second. Multiplied by the number of CPUs this is an execution speed of 432500 cycles per second per machine including the otherhead for network and memory.

6.2 Simulation using sockets

As described in section 5.1.3 as of now it is impossible to simulate a shared memory model with the NoC module across different workstations. I still needed some type of simulation that would realistically let me evaluate the socket performance. I have thus taken the simulator from section 6.1 and made only a small change to the layout from figure 17. I have introduced the socketport in-between the four top level switches and the memory routing the eight unidirectional links connecting those two components across the socketport. The server was configured to just reflect the incoming data. Thus all memory requests and responses had to pass across the socket. This was I was able to keep the CPUs and the memory inside one simulator and thus maintain direct memory access by system calls.

I have measured timing of this simulator in three different setups. In the first setup the simulator and the server were executed on the same simulation workstation. This allowed to measure the performance loss due to the use of the socket API calls. Also the measured results allowed for evaluation of the sockets as an interfacing method between different simulation environments. During the second the simulator and the server were running on two different systems that were both in the same rack. The network optimizations behind this allow for a ping of 0.05 msec between both machines. This allowed measurement of simulation speeds when using two machines on on rack. Since this setup is often available in research laboratories its results are of interest. Finally I executed the simulator and the server on two different machines connected by a normal ethernet network. Again I measured the simulation speed in simulated cycles per sec. The ping of the network measured before the simulations was 0.3 msec on average. This is six times the ping on the rack machines and allows for evaluation of the socket solution for distribution over normal networks.

The results of the measurements can be seen in table 5. As can be seen the simulation on one machine and on two machines from the same rack

	one machine	**two machines on one rack**	**two machine on network**
network ping	0.02 msec	0.05 msec	0.3 msec
average	21,453	21,793	11,058
best	24,506	23,740	16,585
worst	17,484	18,977	7,158

Table 5: Measurement results in simulated cycles/sec. For each configuration 10 measurements were made.

yield the same simulation speed. This is surprising at first since the pings are different but in fact there is a simple explanation. The simulator only sends and receives messages at the beginning of each simulation cycle and then computes what actually happens in the cycle. The pings for the first two simulations are so small that the messages arrive before the next simulation cycle starts. In fact when an average of around 20,000 cycles per second is calculated this is exactly one cycle every 0.05 msec. Thus both systems yield the same result. When compared to the simulation without sockets this method introduces a performance loss of 21%. This result has already been interpretted in section 5.1.3.

When running the socket based simulation on two different machines on a network the situation is far worse. The introduced loss is almost 60%. This value makes the method completely useless. Thus if the simulation with sockets has any future then only when run on machines that have a special network connection like in the case of server racks.

7 Future work

What I have done is implement the basis elements for the library motivated in the introduction and tested these modules. But the work is far from over. More researchers will need to contribute to the library in order to make it versatile enough that it will allow for all possible configurations of CMPs. New modules need to be developed: new cores and eventually different interconnects. An implementation for message passing with standard libraries like MPI would also be of use.

Another thing to do is to add operating system support to the PowerPC 405 core I implemented. This means implementing the functionality of not yet done instructions and adding components such as bios, harddrive and input/output device simulation modules. Operating system support is useful in order to for example make measurements with realistic workloads for multithreaded servers.

The second issue is the discussed wrapper. We will continue to work on it and our hopes are that by the end of the month we will be able to use the PowerPC 405 core in combination with a Liberty memory module. For this the emulator interface needs to be added to the wrapper.

In order to distribute the created library needs to be published on MicroLib. This will allow for other researchers to access it and use it in their designs.

8 Personal conclusion

The internship is not over yet so it might seem a little early to draw a conclusion. But in fact I feel already that I am very satisfied with my choice and that I like the work here very much. I was able to strengthen the knowledge in computer architecture that I gained from classes before. The FPGA course from my first trimester and the connected project have proven to be of immense value and benefit to my work here. In the FPGA project we had implemented a MIPS32 including a low resolution graphics card and programmed it with our own code. I believe the experience from this class has helped as lot during the internship when looking at other architectures and considering changes to my simulation layout.

But in the course of this work I was able to learn far more than about further architectures and optimizations regarding those. Due to the emphasis on networks on chip I have not only learned about connecting cores on a chip. I was able to gain a lot of knowledge about networks in general since there are strong relations between both. Passing information seems to remain a bottleneck for overall performance in many applications and I feel that it is crucial to have some fundamental knowledge in this domain. Due to the large amount of papers I have read I believe that I have understood more than just the basic principles.

In addition I have acquired new skills in code development and how to effectively work in a linux environment. While this is not hard to gain it takes some time and is best done learning from others. Since I have worked with a lot of code written by other researchers I was able to draw my conclusions from its performance and learn new tricks.

But I feel that the very most important part for me is that I have worked for some time in a research group. Due to the intense contact I learned to work and cooperate with other researchers. Also Seminars and meetings where we talked about other colleagues' projects have often sparked my interest. It was refreshing to talk about ideas with others and listen to their suggestions. Still I had a lot of chances of making my own design decisions. I feel that I was able to integrate my ideas well with the team and that I have profited best possible from these opportunities. I see this internship as a big success and the the experience gather as something that will help me in the future.

For quite some time now I have been working and studying towards being a researcher. This intership has further backed up this decision. In addition I feel that computer architecture and simulation will stay one of my favorite research domains.

A FastSysC

FastSysC is also a standard C++ library. It implements a SystemC engine whose main goal is to allow for a faster simulation speed than the orinial SystemC engine. To reach this goal FastSysC implements only the subset of SystemC constructs mentioned in section 2.1. While in SystemC a signal is in fact implemented by finer grained objects FastSysC implements them as one class. In order to further improve simulation speed FastSysC allows for only one clock. In processor simulation this is usually sufficient since processor circuits on the die are synchronous. Because SystemC was implemented for hardware system simulation in general it allows for multiple asynchronous clocks which it needs to manage during simulation. Even when only one clock is finally used this creates computation overhead. FastSysC is implemented to use only one clock and thus saves again time.

The possibly most important feature of FastSysC is its possibility to generate static schedules for the processes in a simulation. This removes the computation overhead needed to decide during simulation which process has to be called next (the LIFO queue in SystemC). This is done by letting the user specify signal dependencies inside a module and combining this with the knowledge about signal dependencies between modules due to the connection of ports. Thus in the case of the xor-gate the folowing code would have to be added to the constructor in order to tell the schedule generator that changes to the output depend on changes to the input:

```
res(op1);
res(op2);
```

The FastSysC schedule generator creates a graph from these dependencies and first checks if it is circular. In this case a static schedule is not possible. But a circular dependency means that there is an oscillation in the simulation. Since this should not be the case in processor simulation there should also be no circular dependencies in the dependency graph for such simulators. After this check the schedule generator searches for an optimal schedule. Once found the schedule generator will create a C++ file that contains a static simulation engine and can be compiled. By this the SystemC engine reaches speedups of up to 3.56 in comparison to the standard SystemC engine [2].

FastSysC implements the subset of the most important SystemC constructs I have mentioned above (Section 2.1). Because of that, all Simulations written in FastSysC can be run in SystemC. The author of FastSysC has in fact included an extension that is not available in SystemC. This extension I will describe in the following since I have made broad use of it. While it is not implemented in the SystemC library it can be easily added to it. Thus code using the additional constructs described below is still SystemC compliant, it just necessitates an extra layer between itself and the SystemC library.

In SystemC it is possible that a calculation unnecessarily takes place multiple times. For example the process of a four input and gate module might be called 4 times during a cycle. Each time it will compute the new value of the output and write it to the port, eventually causing more processes to be called multiple times. In some cases this behavior is unwanted. If the circuit following the and gate is pure combinatorial and we are only interested in the value at the end of the cycle, than costly simulation time is wasted in this case. As a solution to this problem FastSysC implements a new type of ports: `sc_in2<type>` and `sc_out2<type>`. The difference to the standard ports is, that with the enhanced version the user can check if the value of a certain port is already know for this cycle. To check this, the `sc_in2` module provides two methods `known()` and `unknown()`. In the beginning of a clock cycle all inports are set to *unknown*. When an outport is being written to, the corresponding inports value is marked as *known*. With this technique the above mentioned and gate can simply check if all four value on the inports are known before computing the result and writting to its outport. All modules that I have written in SystemC are in fact written for FastSysC using the `sc_in2<type>` and `sc_out2<type>` ports.

B A simple test program

```
#include <stdio.h>
#include <string.h>

// this constant returns the CPU rank
#define cpu_n (*((char*)0x00000000))
#define flag  (*((char*)0x00000010))

int main(int argc, char **argv, char **envp)
{
    if (cpu_n == 1) {
        printf("MASTER: I'm the master cpu!\n", argc);
        flag = 2;

        // flush the freshly written data cache entry into memory!
        asm("dcbf 0, 0x10");

        while (flag != 17) {
            // invalidate the data cache entry so
            // that we do a fresh read from memory!
            asm("dcbi 0, 0x10");
        }
    } else {

        // slaves wait until signaled...
        while (flag != cpu_n) {
            // invalidate the data cache entry
            asm("dcbi 0, 0x10");
        }

        // ...then print hello message and signal next CPU
        printf("SLAVE(%d): Hello World!\n", cpu_n);
        flag = cpu_n + 1;

        // flush the freshly written data cache entry into memory!
        asm("dcbf 0, 0x10");

        // slaves loop forever when done:
        while (1) {}
    }

    return 0;
}
```

C A LFSR implementation in SystemC

C.1 xor.h

```
sc_module XorGate {
  sc_in<boolean> op1, op2;
  sc_out<boolean> res;

  XorGate() {
    SC_MODULE();
    HAS_PROCESS(ComputeResult);
    sensitive << op1 << op2;
  }

  void ComputeResult() {
    res = op1 ^ op2;
  }
}
```

C.2 flipflop.h

```
sc_module FlipFlop {
  sc_in_clk clk;
  sc_in<boolean> opin;
  sc_out<boolean> opout;
  boolean tmp;

  FlipFlop() {
    SC_MODULE();
    HAS_PROCESS(DataIn);
    sensitive << opin;
    HAS_PROCESS(NewCycle);
    sensitive_pos << clk;
  }

  void DataIn() {
    tmp = opin;
    printf("Flipflop %s in: %d", name(), tmp);
  }

  void NewCycle() {
    opout = tmp;
  }
}
```

C.3 tee.h

```
template<int nPorts>;
sc_module Tee {
  sc_in<boolean> op;
  sc_out<boolean> res[nPorts];

  Tee() {
    SC_MODULE();
    HAS_PROCESS(ComputeResult);
    sensitive << op;
  }

  void ComputeResult() {
    int i;
    for (i = 0; i < nPorts; i++)
      res[i] = op;
  }
}
```

C.4 lfsr.h

```
sc_module LSFR {
  sc_in_clk clk;

  FlipFlop *fp1;
  FlipFlop *fp2;
  FlipFlop *fp3;
  XorGate  *xor;
  Tee<2>   *tee;

  LFSR() {
    fp1 = new FlipFlop("Flipflop 1");
    fp2 = new FlipFlop("Flipflop 2");
    fp3 = new FlipFlop("Flipflop 3");

    xor = new XorGate("Xor");
    tee = new Tee<2>("Tee");

    sc_signal<boolean> xor_to_fp1;
    sc_signal<boolean> fp1_to_fp2;
    sc_signal<boolean> fp2_to_tee;
    sc_signal<boolean> tee_to_fp3;
    sc_signal<boolean> tee_to_xor;
    sc_signal<boolean> fp3_to_xor;

    fp1->clk(clk);
    fp2->clk(clk);
    fp3->clk(clk);

    fp1->opin(xor_to_fp1);
    fp1->opout(fp1_to_fp2);

    fp2->opin(fp1_to_fp2);
    fp2->opout(fp2_to_tee);

    tee->opin(fp2_to_tee);
    tee->opout[0](tee_to_fp3)
    tee->opout[1](tee_to_xor)

    fp3->opin(tee_to_fp3);
    fp3->opout(fp3_to_xor);

    xor->op1(tee_to_xor);
    xor->op2(fp3_to_xor);
    xor->res(xor_to_fp1);

    SC_MODULE();
  }

  virtual ~LFSR() {
    delete fp1;
    delete fp2;
    delete fp3;

    delete xor;
    delete tee;
  }
}
```

C.5 main.cpp

```
#include <stdlib.h>
#include <systemc.h>
#include <lfsr.h>

int sc_main(int argc, char *argv[]) {
  sc_clock clock;
  LFSR *lfsr;

  duration = 0;

  lfsr = new LFSR("LFSRTEST");
  lfsr->inClock(clock);

  for(i = 1; i < (unsigned long long int)argc; i++) {
    if(strcmp(argv[i], "-t") == 0) {
      if(++i >= (unsigned long long int)argc) break;
      duration = atoll(argv[i]);
    }
  }

  if(duration == 0) duration = 1000;

  cerr << "Running simulation during " << duration << " cycles" << endl;

  cerr << "Initializing SystemC" << endl;
  sc_initialize();
  lfsr->Reset();
  lfsr->Init();


  cerr << "Starting simulation" << endl;
  cerr << endl;

  sc_start(duration);

  cerr << endl;

  return 0;
}
```

D A LFSR implementation in LSE

D.1 xor_gate.lss

```
module xor_gate {
  using corelib;

  inport in0:boolean;
  inport in1:boolean;
  outport out:boolean;

  instance gate:combiner;

  gate.inputs={"in0","in1"};
  gate.outputs={"out"};

  gate.combine = <<< *out_id=in0_id;
                     *out_data = (*in0_data) ^ (*in1_data);
                     *out_status = LSE_signal_something; >>>;

  if(in0.width != in1.width) {
    punt(<<<in0.width (${in0.width}) must equal in1.width (${in1.width}).>>>);
  }

  if(in0.width != out.width) {
    punt(<<<in0.width (${in.width}) must equal out.width (${out.width}).>>>);
  }


  LSS_connect_bus(in0,gate.in0,in0.width);
  LSS_connect_bus(in1,gate.in1,in1.width);
  LSS_connect_bus(gate.out,out,out.width);
};
```

D.2 lfsr.lss

```
using corelib;
include "xor_gate.lss";

instance bit0 : delay;
instance bit1 : delay;
instance bit2 : delay;
instance xor : xor_gate;
instance bit1_tee : tee;
```

```
bit0.initial_state = <<< *init_id = LSE_dynid_create();
                          *init_value = TRUE;
                          return TRUE; >>>;
bit1.initial_state = <<< *init_id = LSE_dynid_create();
                          *init_value = TRUE;
                          return TRUE; >>>;
bit2.initial_state = <<< *init_id = LSE_dynid_create();
                          *init_value = TRUE;
                          return TRUE; >>>;

bit2.out -> bit1.in;
bit1.out -> bit1_tee.in;
bit1_tee.out[0] -> xor.in0;
bit1_tee.out[1] -> bit0.in;
bit0.out -> xor.in1;
xor.out -> bit2.in;

bit2.in.control = <<< return LSE_signal_extract_data(istatus) |
                             LSE_signal_extract_enable(istatus) |
                             LSE_signal_ack; >>>;

collector STORED_DATA on "bit2" {
  decl=<<<
#include <stdio.h>
  >>>;

  record=<<<
      printf(LSE_time_print_args(LSE_time_now));
      printf(": bit2=%d\n", *datap);
  >>>;
};

collector STORED_DATA on "bit1" {
  decl=<<<
#include <stdio.h>
  >>>;

  record=<<<
      printf(LSE_time_print_args(LSE_time_now));
      printf(": bit1=%d\n", *datap);
  >>>;
};
```

```
collector STORED_DATA on "bit0" {
  decl=<<<
#include <stdio.h>
  >>>;

  record=<<<
      printf(LSE_time_print_args(LSE_time_now));
      printf(": bit0=%d\n", *datap);
  >>>;
};
```

References

[1] OSCI, SystemC, *http://www.systemc.org*, 2000-2005.

[2] Daniel Gracia Pérez, Gilles Mouchard, Olivier Temam, "A Fast SystemC Engine", In *DATE '04*, Paris, France, March 2004.

[3] Olivier Temam, Daniel Gracia Pérez, Gilles Mouchard, FastSysC, *MicroLib, http://www.microlib.org*, 2003-2005.

[4] Olivier Temam, Daniel Gracia Pérez, Gilles Mouchard, GenISSLib - Instruction Set Simulator Library Generator, *MicroLib, http://www.microlib.org*, 2004-2005.

[5] Daniel Gracia Pérez, Gilles Mouchard, Olivier Temam, "MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms", In *WDDD '04*, Munich, Germany, June 2004.

[6] Daniel Gracia Pérez, Gilles Mouchard, Olivier Temam, "MicroLib: A Modular Simulation Library", *http://www.microlib.org*, 2005.

[7] Adrijean Andriahantenaina, Hervé Charlery, Alain Greiner, Laurent Mortiez, Cesar Albenes Zeferino, "SPIN: a Scalable, Packet Switched, On-Chip Microßnetwork", In *DATE '03*, Munich, Germany, March 2003.

[8] LIP6, "SPIN", *http://asim.lip6.fr/ adrijean/*, 2004.

[9] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, J. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, V. Frank, S. Amarasinghe, A. Agarwal, "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs", In *IEEE Micro*, vol 22, Issue 2, 2002.

[10] Anant Agarwal, "Raw Computation", In *Scientific American*, Vol. 281, No. 2, August 1999.

[11] Michael Bedford Taylor, Walter Lee, Saman Amarasinghe, Anant Agarwal, "Scalar Operand Networks", In *IEEE Transactions on Parallel and Distributed Systems (Special Issue on On-chip Networks)*, February 2005.

[12] Lance Hammond, Ben Hubbert, Michael Siu, Manohar Prabhu, Mike Chen, Kunle Olukotun, "The Stanford Hydra CMP", In *IEEE MICRO Magazine*, March-April 2000.

[13] Manish Vachharajani, Neil Vachharajani, David A. Penry, Jason Blome, David I. August, "The Liberty Simulation Environment, Version 1.0", In *Performance Evaluation Review: Special Issue on Tools for Architecture Research*, Volume 31, Number 4, March 2004.

[14] Doug Burger, Todd Austin, "The SimpleScalar Tool Set, Version 2.0", In *Technical Report CS-TR-1997-1342*, Department of Computer Sciences, University of Wisconsin, June 1997.

[15] Gilles Mouchard, "PowerPC G3 simulator", *http://www.microlib.org/G3/PowerPC750.php*, 2002.

[16] Manish Vachharajani, Neil Vachharajani, David A. Penry, Jason A. Blome, David I. August, "Microarchitectural exploration with Liberty", In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, Istanbul, Turkey, November 2002.

[17] IBM, "PPC405Fx Embedded Processor Core User's Manual - Preliminary", *http://www-306.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_405_Embedded_Cores*, 2005.

[18] IBM, "PowerPC 405 CPU Core White Paper", *http://www-306.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_405_Embedded_Cores*, 2005.

[19] GCC, "GNU Compiler Collection", *http://gcc.gnu.org*, 1987-2005.

[20] Dan Kegel, "Crosstool", *http://kegel.com/crosstool*, 2003-2005.

[21] Kevin Krewell, "Best Servers of 2004", In *Microprocessor Report, http://www.mpronline.com*, Jan 2005.

[22] Olivier Temam, Daniel Gracia Pérez, Gilles Mouchard, "OoOSysC", *MicroLib, http://microlib.org/projects/ooosysc/*, 2004-2005.

[23] Olivier Temam, Daniel Gracia Pérez, Gilles Mouchard, "AlphaISS", *MicroLib, http://microlib.org/projects/alphaiss/*, 2004-2005.

[24] Intel, "Intel Itanium 2 Processor", *http://www.intel.com/business/bss/products/server/itanium2/*, 2005.

[25] Paul Otellini, "Intel Developer Forum", In *Intel Keynote Transcript, http://www.intel.com/pressroom/archive/speeches/otellini20040907.htm*, Fall 2004.

[26] Kevin Krewell, "Cell Moves Into the Limelight", In *Microprocessor Report, http://www.mpronline.com*, Feb 2005.

[27] INTEL, "Intel Researchers Build World's Fastest Silicon Transistors", In *Intel Press Release, http://www.intel.com/pressroom/archive/releases/20010611tech.htm*, June 2001.

[28] LACA BENINI, GIOVANNI DE MICHELI, "Networks on Chip: A New Paradigm for Systems on Chip in Design", In *DATE '02*, March 2002.

[29] PIERRE GUERRIER, ALAIN GREINER, "A Generic Architecture for On-Chip Packet-Switched Interconnections", In *DATE '00 Proceedings*, pp. 250-256, March 2000.

[30] XILINX, *http://www.xilinx.com.*

[31] IBM, *http://www.ibm.com.*

[32] INTERNATIONAL TECHNOLOGY ROADMAP FOR SEMICONDUCTORS, *http://public.itrs.net.*

[33] YANNIS SMARAGDAKIS, DON BATORY, "Mixin-Based Programming in C++", In *Second International Symposium on Generative and Component-Based Software Engineering (GCSE'2000)*, Erfurt, Germanz, October 9-12, 2000.

[34] ANTOINE JALABERT, SRINIVASAN MURALI, LUCA BENINI, GIOVANNI DE MICHELI, "xpipesCompiler: A Tool for Instantiating Application Specific Networks on Chip", In *DATE '04*, vol. 02, no. 2, p. 20884, Design, 2004.

[35] PAUL WILLMANN, MICHAEL BROGIOLI, VIJAY S. PAI, "Spinach: A Liberty-based Simulator for Programmable Network Interface Architectures", In *ACM SIGPLAN Notices*, Volume 39, Issue 7, July 2004.

[36] RAKESH KUMAR, VICTOR ZYUBAN, DEAN M. TULLSEN, "Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling", In *ISCA 2005*, Madison, Wisconsin, USA, June 2005.

[37] THE ALCHEMY RESEARCH GROUP, *http://www.inria.fr/recherche/equipes_ ur/alchemy.fr.html.*

[38] THE LIBERTY RESEARCH GROUP, *http://liberty.cs.princeton.edu.*

[39] SIMPLESCALAR LLC, "SimpleScalar Overview", *http://www.simplescalar.com/overview.html.*

[40] ALLISON HOLLOWAY, MATTHEW ALLEN, "Exploring Core Designs for Chip Multiprocessors", *http://www.cs.wisc.edu/ david/courses/cs838/projects/ahollowa.pdf.*

[41] Kevin Krewell, "IBM's Power4 Unveiling Continues", In *Microprocessor Report, http://www.mpronline.com*, Nov 2003.

[42] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, K. Yazawa, "The Design and Implementation of a First-Generation CELL Processor", In *ISSCC 2005*, Seccion 10, Microprocessors and signal processing, 10.2, 2005.

[43] H. Peter Hofstee, "Power Efficient Processor Architecture and The Cell Processor", In *Proceedings of the 11th Int'l Symposium on High-Performance Computer Architecture (HPCA-11)*, 1530-0897/05, 2005.

[44] IBM, "The CELL project at IBM Research", *http://www.research.ibm.com/cell/*, 2001-2005.

[45] Gilles Mouchard, "Modelisation de Processeurs et de Systemes", *PhD Thesis, Universite Paris XI Orsay*, In French, 2004.

[46] Olivier Temam, Pierre Palatin, Daniel Gracia Pérez, Gilles Mouchard, "Generic Cache Library", *MicroLib, http://www.microlib.org/genericcaches/*, 2004-2005.