Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Christopher James Barnes

Entitled A Dynamically Configurable Discrete Event Simulation Framework for Many-Core System-on-Chips

For the degree of Master of Science in Electrical and Computer Engineering

Is approved by the final examining committee:

Jaehwan John Lee	
Chair	
Brian S. King	
Yung Ping Stanley Chien	

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Jaehwan John Lee

Jaehwan John Lee

Approved by: Yaobin Chen — March, 10, 2010
Head of the Graduate Program — Date

Graduate School Form 20
(Revised 1/10)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation:
A Dynamically Configurable Discrete Event Simulation Framework for Many-Core System-on-Chips

For the degree of Master of Science in Electrical and Computer Engineering

I certify that in the preparation of this thesis, I have observed the provisions of *Purdue University Teaching, Research, and Outreach Policy on Research Misconduct (VIII.3.1)*, October 1, 2008.*

Further, I certify that this work is free of plagiarism and all materials appearing in this thesis/dissertation have been properly quoted and attributed.

I certify that all copyrighted material incorporated into this thesis/dissertation is in compliance with the United States' copyright law and that I have received written permission from the copyright owners for my use of their work, which is beyond the scope of the law. I agree to indemnify and save harmless Purdue University from any and all claims that may be asserted or that may arise from any copyright violation.

Christopher James Barnes
Printed Name and Signature of Candidate

March 10, 2010
Date (month/day/year)

*Located at http://www.purdue.edu/policies/pages/teach_res_outreach/viii_3_1.html

A DYNAMICALLY CONFIGURABLE DISCRETE EVENT SIMULATION FRAMEWORK FOR MANY-CORE SYSTEM-ON-CHIPS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Christopher J. Barnes

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

August 2010

Purdue University

Indianapolis, Indiana

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Jaehwan John Lee for all of his support and encouragement throughout my pursuit of a Master's degree. His knowledge and professionalism have given me high standards that I will always strive to achieve.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure Page

GLOSSARY

Module	A simulated representation of a functional unit within a resource, such as a processor stage, branch predictor, or forwarding unit.
NoC	Network-on-Chip, an emerging approach to communication between resources, or subsystems, on a chip that is similar to large scale computer networks (i.e., a LAN, or the Internet). Network-on-Chips are also referred to as an on-chip interconnect.
Resource	A simulated representation of any high level component in a simulated system such as a processor core, I/O, or memory. Resources can perform any sort of behavior that the simulation designer wishes.
Simulation Configuration	The collection of settings and source code that is used to produce a processor simulator.
Simulation Designer	A person that is using the simulation framework for the purpose of producing or revising a processor simulator.
Simulation Framework	Software used to gather simulation configuration information and dynamically compile processor simulators. The software described in this thesis.
Simulation Host	The machine used to execute a simulation.

ABSTRACT

Barnes, Christopher J. M.S.E.C.E., Purdue University, August 2010. A Dynamically Configurable Discrete Event Simulation Framework for Many-Core System-on-Chips. Major Professor: Jaehwan J. Lee.

Industry trends indicate that many-core heterogeneous processors will be the next-generation answer to Moore's law and reduced power consumption. Thus, both academia and industry are focused on the challenges presented by many-core heterogeneous processor designs. In many cases, researchers use discrete event simulators to research and validate new computer architecture innovations. However, there is a lack of dynamically configurable discrete event simulation environments for the testing and development of many-core heterogeneous processors. To fulfill this need we present Mhetero, a retargetable framework for cycle-accurate simulation of heterogeneous many-core processors along with the cycle-accurate simulation of their associated network-on-chip communication infrastructure. Mhetero is the result of research into dynamically configurable and highly flexible simulation tools with which users are free to produce custom instruction sets and communication methods in a highly modular design environment. In this thesis we will discuss our approach to dynamically configurable discrete event simulation and present several experiments performed using the framework to exemplify how Mhetero, and similarly constructed simulators, may be used for future innovations.

1. INTRODUCTION

1.1 Background

Many-core heterogeneous processor designs have proven to improve performance and power efficiency in a variety of applications [1, 2] resulting in the wide spread adoption of heterogeneous designs in emerging processors [3, 4]. As the industry moves toward merging many different, highly specialized processor resources on one physical chip [5], there is a need for a highly configurable discrete event simulation environment for the study of heterogeneous processor designs. Moreover, as many-core designs become more prevalant, there is an urgent need for an environment that integrates both the study of heterogeneous designs and an intra-core communications infrastructure. Introduced in this thesis is Mhetero, a novel simulation framework that enables users to construct and perform heterogeneous discrete event simulations with a network-on-chip that meets these needs.

Computer architecture simulation is often a cornerstone in the research of new processor concepts and in the education of computer architecture students. Simulators are used by researchers to validate architecture designs and explore new concepts. Educators use simulators to elucidate concepts in computer architecture through hands-on exercises and demonstrations. To be useful for both researchers and educators, simulators must be flexible, easy to use and understand, and fast.

Simulation speed and configurability are two important aspects in the design of computer architecture simulators. In the past, fast simulations typically had a monolithic design and were written to simulate a particular architecture. However, this approach required a complete understanding of the source code before the user could deviate from the simulator's original design. To overcome this drawback, some simulators embraced a more modular design, while others attempted to provide some customizability in the simulator by integrating and using Architecture Description Languages (ADLs) to describe the resulting simulator's functionality. This approach provides some configurability, but still requires the user to undergo a lengthy learning process to begin generating useful results. Additionally, many ADL implementations depend on several third party programs, which can make the process of producing and executing simulators disjointed and error prone.

Our simulation framework addresses the need for fast as well as configurable simulations by taking advantage of the dynamic compilation capabilities of the Microsoft's .NET development libraries in several ways. First, we use dynamic compilation to produce simulations based on configuration information gathered through an easy-to-use GUI. The entire process is seamless and user-friendly, meaning that the user does not need to leave the framework to execute external compilers, write source code, edit configuration files, or execute simulations. Second, the simulations produced by the framework are compiled to an intermediate language [6], resulting in quick compilation time as well as execution speeds matching that of other compiled .NET programs.

Our framework also supports single threaded and multithreaded execution, allowing the user to optimize their performance based on the available hardware and needs of their simulation. Moreover, the framework's design is open and modular, allowing simulation designers to produce any sort of simulator that they may desire, even simulations extending beyond the typical tasks associated with a processor simulator.

Mhetero's simulation infrastructure is similar to other discrete event/time simulators, but with a few notable differences that facilitate processor simulation. First, instead of using a single, global event queue, Mhetero maintains several event queues each modeling a communication channel between two entities/modules of simulation. Second, instead of activating modules after certain events occur, entities/modules are activated during every cycle and these modules can then choose to process corresponding events immediately or after a specified number of cycles ensuring causality and synchronism between events in the simulation [7]. Hence, Mhetero's simulation infrastructure can be categorized as a cycle-accurate discrete time simulation infrastructure which by definition itself is a discrete event simulation infrastructure [7]. As a result, the framework is not only an interesting and powerful alternative to other discrete event simulators but also a useful tool for computer architecture researchers and educators.

1.2 Objective and Purpose

The objective of this research was to develop a framework that enables students and researchers to design, configure, generate, and execute processor simulators, through a user-friendly interface. Furthermore, the framework should be able to produce simulators that are not only relevant for the research topics of today, but those of tomorrow as well. To this end, Mhetero was developed to be highly modular, pervasively object oriented, and support many heterogeneous processor components, from the ground up.

The purpose of this research was to allow researchers to rapidly explore a large variety of architectures and IPs, as well as investigate new computer architecture innovations. Additionally, this framework should provide a platform for teaching topics related to computer architecture.

1.3 Organization

In this thesis, we will discuss the design and construction of Mhetero. We will begin by reviewing some of the previous work in the area of computer architecture simulation (Chapter 2). Next we will discuss our configuration interface (Chapters 3), dynamic compilation technique (Chapter 4), and network-on-chip implementation (Chapter 5). We will then introduce our approach to multithreaded simulation and address some multithreaded performance issues (Chapter 6). Finally, we will discuss several experiments that were conducted to validate the framework's design (Chapter 7).

2. PREVIOUS WORK

2.1 Overview

Over the past several decades a considerable amount of research has been performed in the area of computer architecture simulation. Computer architecture simulators vary greatly in purpose, scope, and detail level. These simulators can be broadly divided into several categories: full-system simulators, Instruction Set Architecture (ISA) simulators, and retargetable simulators. Each category serves an entirely different purpose, but all have been used for the advancement of computer architecture research.

The purpose of full-system simulators is to model an entire computer system, including the processor, memory system, and any I/O. These simulators are typically capable of running real (i.e., not for testing purposes) software completely unmodified, similar to a virtual machine. There are many simulation suites that take this approach, including PTLSim [8], M5 [9], Bochs [10], ASIM [11], GxEmul [12], and Simics [13] (Simics is a commercial product). Simics has several extensions that constitute their own full-system simulators such as VASA [14] and GEMS [15]

ISA simulators are less comprehensive than full-system simulators as their purpose is typically limited to modeling the processor alone. ISA simulators serve several purposes: to simulate and debug machine instructions of a processor type that differs from the simulation host, to investigate how the various instructions (or a series of instructions) affect the simulated processor, and for developing and testing experimental ISAs. To this end, modeling of the full computer system is unnecessary and would impose additional complexity and delay. Examples of this type of simulator include SimpleScalar [16], WWT-II [17], and RSIM [18].

Retargetable simulators are similar to ISA simulators, but are more focused on the research and development of new computer architecture concepts, ISAs, and design space exploration of Chip Multiprocessors (CMPs). The research and development focus results from the capability to reconfigure the simulator to produce a new simulation target through some mechanism that is easier and/or faster than writing a new simulator. Typically this process is done through the use of an Architecture Description Language (ADL). Thus the user of the simulator can model experimental processor designs by adjusting parameters in an ADL script instead of writing or rewriting a new processor simulator. Examples of this type of simulator are Expression [19], LISA [20], nML [21], and RCPN [22]. The author of this thesis has also produced a simulator of this type called CADL [23].

In the case of retargetable simulators, it is important to note the distinction between the simulator and the program that creates the simulator. In many articles this distinction is unclear, which may lead to confusion. In our case, Mhetero is a framework for creating simulators, and not a simulator itself. During the development of Mhetero, a simulation configuration for a MIPS64 architecture was created (Mhetero's simulation capabilities are not limited to MIPS64 however). The framework coupled with the simulation configuration is used to generate an executable simulation.

Mhetero can be defined as a retargetable simulation framework. However, the framework attempts to cross the bridge between retargetable simulators and ISA simulators. To this end, the framework is very modular and open to allow simulation designers to produce virtually any sort of ISA imaginable. However, once produced, the simulation configuration can be easily distributed and used similarly to an ISA simulator. Mhetero is also similar to ISA simulators in that it yields simulators that execute virtually as fast as other simulators targeted towards a particular ISA due to its compilation technique (discussed in Chapter 4).

In the remainder of this chapter we will review the previous work completed in the area of retargetable simulators, and briefly discuss full-system and ISA simulators.

We will then review the previous work in the area of network-on-chip simulators, and briefly discuss some of the previous work in the area of multithreaded simulation. We will conclude this chapter with a discussion of how these previous works influenced the design of Mhetero.

2.2 Previous Works In The Area of Retargetable Simulators

2.2.1 Anahita Processor Description Language (APDL)

APDL [24] is one of the more recent contributions in the area of retargetable simulation. The language was introduced in 2007 by N. Honarmand et al. from the Shahid Beheshti University, IRAN. The primary difference between APDL and other ADLs is the addition of the Timed Register Transfer Level (T-RTL), which enables the simulation designer to define the latencies and hardware requirements of processor operations. This separation of configuration data enables APDL to better integrate with external software for analysis as the T-RTL data is organized separately from the remainder of the processor description. Moreover, APDL can describe both instruction and structure descriptions of a target processor.

The Pascal-like syntax of APDL is clearly more intuitive than many other ADLs such as LISA and EXPRESSION. A short example of APDL's syntax is shown in Figure 2.1. While this language is easier to read and understand, the researchers have not yet implemented a compiler to produce simulations. Furthermore, despite APDL's relative ease, users are still faced with the task of learning the details of the syntax.

```
operation add ( s0: reg_src, s1: reg_src, d: reg_dst ) is
    s0'rf_read_port_number := 0;
    s1'rf_read_port_number := 1;
    action := { d'val := int32(s0'val) +|ALU,EX,1| int32(s1'val); }
end operation;

operation sub ( s0: reg_src, s1: reg_src, d: reg_dst ) is
    s0'rf_read_port_number := 0;
    s1'rf_read_port_number := 1;
    action := { d'val := int32(s0'val) -|ALU,EX,1| int32(s1'val); }
end operation;
```

Fig. 2.1. An example of the APDL ADL describing the add and subtract instructions (Courtesy of N. Honarmand et al. [24])

2.2.2 EXPRESSION and ReXSim

EXPRESSION [19] was introduced in 1999 by the Department of Information and Computer Science, University of California, Irvine. EXPRESSION was developed for the investigation of System-on-Chip (SoC) architectures through the use of an ADL that mixes both structural and behavioral descriptions, with a particular emphasis on describing memory systems. Additionally, EXPRESSION supports cycle-accurate simulation with pipelines and cache hierarchies. One interesting feature of EXPRESSION is its ability to create retargetable compilers based on simulator descriptions.

The EXPRESSION ADL uses a LISP-like syntax for describing processor configurations. An example cache description is shown in Figure 2.2, and several instructions are defined in Figure 2.3. While this approach is capable in several respects, the ADL has a significant learning curve which makes the process of generating simulators difficult to learn and use. Additionally, EXPRESSION is primarily useful for the design space exploration (i.e., exploring the effect of removing or adding functional units) rather than producing fast simulations. Finally, EXPRESSION does not support many-core simulations, and thus also does not support heterogeneous simulations or network-on-chip.

```
(STORAGE PARAMETERS
(L1 cache (TYPE cache) (SIZE 1024) (LINE 32)
(ASSOCIATIVITY 3) (ADDRESS RANGE 0 511)
(ACCESS TIMES 1)))
```

Fig. 2.2. An example description of a cache subsystem using EXPRESSION ADL (Courtesy of A. Halambi et al. [19])

ReXSim [25] was introduced in 2003 by a computer architecture research team also at University of California, Irvine (the ReXSim team has several researchers in common with EXPRESSION). ReXSim is an extension of the EXPRESSION language which sought to improve simulation speed by integrating a novel method of decoding instructions of the simulated program before execution of the simulation. As a result, the instruction decoding process was removed from the execution loop of the simulator, and thus improved the simulation speed significantly. Using this method, the team was able to produce retargetable simulations that showed performance in excess of major simulators such as SimpleScalar [16], which is widely considered to be a simulation performance benchmark.

```
(OP_GROUP alu_ops
    (OPCODE add
      (OP TYPE DATA_OP)
      (OPERANDS (SRC1 g1) (SRC2 g1) (DST g2))
      (BEHAVIOR DST = SRC1 + SRC2)
     )
    (OPCODE sub
      (OP TYPE DATA_OP)
      (OPERANDS (SRC1 g1) (SRC2 g1) (DST g2))
      (BEHAVIOR DST = SRC1 - SRC2)
     )
)
(OP_GROUP mult_ops
    (OPCODE mult
      (OP TYPE DATA_OP)
      (OPERANDS (SRC1 g1) (SRC2 g1) (DST g3))
      (BEHAVIOR DST = SRC1 * SRC2)
     )
)
```

Fig. 2.3. An example description of add, sub, and mult instructions using EXPRESSION ADL (Courtesy of A. Halambi et al. [19])

2.2.3 Instruction Set Description Language (ISDL)

ISDL [26] was introduced in 1997 by G. Hadjiyiannis, S. Hanono, and S. Devadas from the Massachusetts Institute of Technology. The purpose of ISDL was to provide a language for describing instruction sets along with a limited amount of details of a processor structure for the automatic construction of compilers, assemblers, and simulators. The language is not similar to any particular programming language;

however, ISDL does make use of C syntax to describe instruction behavior. In particular, ISDL was designed to describe VLIW machines such as digital signal processors (DSPs).

ISDL enables users to define their target processors in several ways. First, users can define operations, their format, and the associated assembly language instruction. Second, users can define the storage resources available to the processor, including the register file and memory. Third, users can define constraints in the processor, such as instructions requesting the same data path, or restrictions regarding assembly syntax.

Alone, ISDL does not provide any sort of mechanism for actually producing either compilers or simulators. In the literature describing the ADL, the authors mention ongoing research into the development of tools and simulators utilizing ISDL. However, we were unable to find any such tools. Additionally, very little detail is provided as to how to actually use the language to describe a processor. We discuss this previous work because it is frequently cited in many works in the area of retargetable simulation.

2.2.4 LISA

LISA [20] was introduced in 1996 by Zivojnovic et al. from Aachen University of Technology, Germany. The main characteristic of LISA is its ability to model pipelines and simulate instruction sets that are described through the LISA ADL. LISA is more simple than others because its functionality is much more restrictive. While EXPRESSION is able to describe both the instruction set and structure of the modeled processor (e.g., memory and cache systems) and the flow of data between functional units, LISA is only capable of describing the instruction set and pipeline.

The LISA ADL arranges small pieces of C/C++ source code in a format that groups the pieces together by instruction type. An example of this arrangement is shown in Figure 2.4. LISA's compilation process outputs C/C++ source code, which must be compiled separately by the user afterwards. Thus the simulator compilation

```
<insn> LD
{
   <decode>
      {
      %ID: {0x7121, 0x1004}
      %src: { mem[%OPCODE1 & 0x7F] }
      %dest: { accu[(%OPCODE1 >> 8) & 1] }
      }
   <schedule>
   {
      LD1(PF, w:ebus_addr, w:pc) | LD2(IF) |
      LD3(DE) | LD4(AC, w:dbus_addr) |
      LD5(RE, !r:treg, r:dbus_addr) | LD6(EX, r:treg)
   }
   <operate>
   {
      LD1: { ebus_addr = pc; }
      LD1.control { pc++ }
      LD2: { ir = mem[ebus_addr]; }
      LD4: { dbus_addr = %src; }
      LD5: { treg = mem[dbus_addr]; }
      LD6: { %dest = treg; }
   }
}
```

Fig. 2.4. An example of the LISA ADL, which describes a load (LD) instruction (Courtesy of V. Zivojnovic, S. Pees, and H. Meyr [20])

process requires the simulation designer to configure and use several programs every time a simulator is produced. This multi-step process is likely problematic; for example, if a syntax error exists in the behavioral description of an instruction, it will be difficult to determine where the error is located in the ADL. In other words, when a compilation error occurs, there is no way to trace back specifically to the origin of the error in the ADL since the C/C++ compiler is unrelated to the LISA ADL compiler. Moreover, the configuration of the external compiler and its associated project (i.e., any external libraries and source code files that must be included into the resulting simulator executable) is left up to the user.

2.2.5 nML

nML [21] is also an ADL-based language, which was developed at TUBerlin in 1991 by M. Freericks. The nML ADL language was one of the earliest ADLs, and as such it is considerably more simple than both LISA and EXPRESSION in terms of functionality. The ADL does not support cycle-accurate simulation, or complex pipelining. The primary purpose of nML, like LISA, is to describe the instruction set of a target processor, and thus it does not describe the structure of the processor.

Similar to LISA and EXPRESSION, nML users must overcome a large learning curve to begin producing simulators. In fact, the nML ADL has its own unique grammar, which is more akin to an assembly language than any programming language (i.e., the language has very specific formatting and rules that are not intuitive, and thus it is not useful without a complete understanding of the language). An example of the language syntax is shown in Figure 2.5.

nML was later extended in Sim-nML [27] which was introduced in 1999 by V. Rajesh and R. Moona from the Indian Institue of Technology Kanpur. This extension enabled cycle-accurate simulations; however the resulting simulations are slow, and the language's simplicity (relative to other ADLs) limits the user's ability to describe complex pipelines.

```
resource fetch_unit,execution_unit,
retire_unit
reg AC [ 1, card( 8 ) ]
reg PC [ 1, card( 32 ) ]
reg temp [ 1, card ( 8 ) ]
op plus ( )
syntax = "add"
image = "000000"
action = { AC = AC + tmp; }
uses = execution_unit #1
preact = { PC = PC + 1; }
uses = fetch_unit : preact&#1,x.uses,
retire_unit #1 : action
```

Fig. 2.5. An example of the nML ADL used to describe a simple processor (Courtesy of M. Freericks [21])

2.2.6 Reduced Colored Petri Net (RCPN)

RCPN [22] was introduced in 2005 by M. Reshadi and N. Dutt (who also contributed to EXPRESSION), from University of California, Irvine. RCPN takes a vastly different approach to retargetable simulation, in which pipelines are modeled using a simplified version of Colored Petri Nets (CPN). Petri Nets are a graph-based mathematical method of describing a process. The nodes of the graph represent particular discrete events, states, or functions, and the graph edges represent the transitions of data between nodes. The transitions can be enabled or disabled based on conditions specified at the nodes. Colored Petri Nets (CPNs) extend the concept of Petri Nets by adding color to the nodes to symbolize the data type that passes through it.

The purpose of RCPN is to provide retargetable simulations for modeling of pipelined processors. RCPN reduces the functionality of a regular CPN by limiting the capabilities of the nodes in the graph for the purpose of increasing simulation speed and usability. Additionally, RCPN takes advantage of some of the natural properties of CPNs to prevent structural and control hazards as well as modeling latencies. Using this method, RCPN was able to produce very fast simulation, well in excess of SimpleScalar [16].

Simulations are modeled as a series of nodes and transitions which can resemble a processor block diagram. While this may be more intuitive from a user's perspective, this process may be difficult to implement as the user must have a complete understanding of how CPNs work in order to determine the conditions that enable transitions between nodes.

2.2.7 A Note About Full-System Simulators and Instruction Set Architecture Simulators

Full-system simulators are typically very capable and complex simulation suites. For example, a full-system simulator provides all of the essential 'virtual hardware' necessary to boot a simulated computer. While these types of simulators have frequently been used for academic research, this complexity makes retargeting the simulator very difficult. To exemplify this, most of the currently available full-system simulators only simulate one type of system (i.e., x86, Alpha, etc.). As far as we know, Simics [13] supports the most processor types of all the full-system simulators with ten different supported processor types.

ISA simulators can also make the exploration of new architectures difficult. These types of simulators are typically faster than retargetable simulators as they can avoid any computational delay associated with retargetability. However, ISA simulators are usually monolithic and difficult to understand and modify.

Direct execution of simulations is another restrictive approach employed by several simulators [17, 28]. This simulation execution methodology allows the simulator to directly execute the simulated program on the host processor to speed up the simulation. Thus, the simulator is only capable of executing programs that could only be executed on the simulation host. As a result, simulators using this methodology are not capable of supporting heterogeneous simulation, or exploring architecture types beyond that of the simulation host.

2.2.8 Previous Work Performed By The Author

The author of this thesis introduced an ADL in 2009 called Computer Architecture Description Language (CADL) [23] based on work performed at the IUPUI Computer Architecture Lab. The research defined an XML-based ADL which enabled simulation designers to describe the structure, instructions, and behavior of a target processor. An example of CADL is shown in Figure 2.6. The CADL files were used to produce executable simulators through the use of a CADL compiler coupled with an external C++ compiler. Simulators generated with the CADL compiler were multithreaded as well as cycle-accurate and performed competitively to SimpleScalar [16].

CADL quickly highlighted many of the drawbacks associated with retargetable simulators. First, despite the fact that the simulator descriptions used the industry standard XML format, it was difficult to use for those not familiar with the data format. Additionally, this method allowed users to input parameters that would generate invalid simulations. While error checking of the XML was performed during compilation, it occurred as a separate process so the origin of the error was unclear. Moreover, configuring the CADL compiler to automatically initiate the C++ compiler was a lengthy process that was error prone for those not familiar with the usage of command-line compilers. Thus, the CADL research provided valuable insight that inspired our approach to Mhetero.

```
<ADL>
...
<Channels>
   <Channel name="ID2EX">
    <Output>ID</Output>
    <Input>EX</Input>
    <Data>
       <uint32>rs_data</uint32>
       <uint32>rt_data</uint32>
       <uint32>base</uint32>
       <uint32>originPC</uint32>
       <bool>nop</bool>
    </Data>
  </Channel>
...

<Memory>
   <Instruction size="0x8000" type="uint32" bytesPerFetch="4"/>
   <Data size="0x8000" type="uint64" bytesPerFetch="8"/>
   <Cache mapping="DIRECT" replacement="LRU" blocks="64" wordBits="2"
                              tagBits="60" setBits="0" blockBits="0"/>
</Memory>
...
</ADL>
```

Fig. 2.6. An example of the CADL ADL describing the data channel between stages as well as the memory and cache subsystems (Courtesy of C. Barnes et al. [23])

2.3 Previous Works In The Area of Network-on-Chip (NoC) Simulators

2.3.1 Garnet

Garnet [29] was introduced in 2008 by N. Agarwal et al. from Princeton University. The purpose of Garnet was to extend the GEMS [15] full-system simulator with a detailed NoC simulator extension. Since Garnet's introduction, it has been entirely integrated into the GEMS simulator. Garnet is an interesting NoC simulator because it is able to provide very detailed and cycle-accurate information about network power consumption, network timing, and packet ordering. Additionally, the simulation is suitable for the investigation of network topology, bandwidth usage, and routing algorithms.

Configuration of the simulated NoC is performed through the adjustment of a small number of parameters, such as flit size, buffer sizes, and routing tables. Since Garnet has a fixed amount of adjustable parameters, the simulation can only operate with predictable configurations. This characteristic enables the simulation to provide useful statistical information; however, this restriction does place a limitation on the creativity of simulation designers. Therefore, the Garnet simulator is more useful for the design space exploration of today's network-on-chip designs rather than investigation of hypothetical and innovative network topologies and routing algorithms.

2.3.2 Noxim

Noxim [30] is an open source project that was developed by F. Fazzino, M. Palesi, and D. Patti in 2005. The purpose of Noxim is limited to network simulation, and it does not integrate with any processor simulator. Thus, this simulator is limited in scope and usefulness. The simulator allows users to adjust a series of parameters for the purpose of analyzing the resulting changes, which include the energy consumption, communication delays, and the amount of packets/flits received at a particular destination.

Despite the fact that this simulator is cited in many NoC studies [31,32], there is surprisingly little information available about it.

2.3.3 SICOSYS

SICOSYS [33] was introduced in 2002 by V. Puente, J.A. Gregorio, and R. Beivide from the University of Cantabria (Spain). This NoC simulator is similar to Noxim in that it does not provide a means to simulate any other processor resources beyond the network itself. However, SICOSYS does provide the capability of co-simulation with RSIM [18] (an ISA simulator) which is facilitated by YACSIM [34].

SICOSYS was designed to ease the process of exploring new NoC designs. To achieve this, simulation designers can configure their simulations through description files in an XML-like language called SGML [35]. An example of a router defined using SGML is shown in Figure 2.7. This language is used to specify the various hardware connected to the network, router architecture, network type, and other simulation parameters such as message size and delays.

While SICOSYS offers more flexibility than either Garnet or Noxim, the simulation environment requires the setup and integration of three different programs (YACSIM, RSIM, and SICOSYS), which is a difficult and error prone task. Additionally, SICOSYS only supports a limited number of network topologies and routing functions. Moreover, SGML network configuration format is not intuitive for novice users.

2.3.4 Integrated Network-on-chip Simulators in RSIM and ASIM

Both RSIM [18] (an ISA simulator) and ASIM [11] (a full-system simulator) include functional NoC simulation capabilities. These implementations have a very limited amount of configurability relative to the external NoC simulators covered here. For example, the ASIM NoC simulator only supports ring interconnects, and RSIM's NoC simulator only supports 2D mesh networks. Additionally, the primary

```
<Router id="DOR2D-BU" inputs=4 outputs=4 bufferSize=64 bufferControl=CT
               routingControl="DOR-BU">
  <Injector id="INJ">
  <Consumer id="CONS">
  <Buffer id="BUF1" type="X+" dataDelay=2>
  . . . . . . . . . . .
  <Buffer id="BUF5" type="Node" dataDelay=2>

  <Routing id="RTG1" type="X+" headerDelay=1 dataDelay=0>
  . . . . . . . . . . .
  <Routing id="RTG5" type="Node" headerDelay=1 dataDelay=0>

  <Crossbar id="CROSSBAR" inputs="5" outputs="5" type="CT" headerDelay=2
                 dataDelay=1>
    <Input id=1 type="X+">
    <Input id=2 type="X-">
    . . . . . . . . . . .
    <Output id=5 type="Node">
  </Crossbar>

  <Connection id="C01" source="INJ" destination="BUF5">
  <Connection id="C12" source="RTG5" destination="CROSSBAR.5">
  <Input id="1" type="X+" wrapper="BUF1">
  <Output id="1" type="X+" wrapper="CROSSBAR.1">
  <Output id="4" type="Y-" wrapper="CROSSBAR.4">
</Router>
```

Fig. 2.7. An example of the SGML-based lanuage used in SICOSYS (Courtesy of E. Gamma et al. [33])

motivation of these NoC simulators are to facilitate many-core simulation, and therefore, their detail level is low and they are considered to be inaccurate [29].

2.4 Previous Work in the Area of Multithreaded Simulators

Several of the processor simulators mentioned in this chapter support multithreaded execution of simulations. Simics [13], a full-system simulator, is multithreaded capable and has several variants, such as GEMS [15] and VASA [14]. Wisconsin Wind Tunnel II [17] supports parallel direct-execution simulation which can be executed across multiple computers in a cluster. However, all of these simulators are not fully suitable for design-space exploration for reasons that were previously noted. As far as we are aware, there are no other retargetable simulators designed for CMP simulation that support multithreaded execution.

In addition to these simulators, there have been several parallel processing techniques presented to speed up processor simulation. Chidester et al. [36] presented a distributed simulation methodology for CMPs based on Message Passing Interface (MPI), which shows simulation speed-up of 5%. Several authors [37–39] also presented methods of using program traces (execution paths of simulated programs based on previous observed executions) to improve simulation speeds on a multiprocessing platform.

2.5 Our Design Decisions Based On Previous Work

Several of Mhetero's design decisions were made clear based on the previous work studied here. First, our simulation framework must be built to minimize the difficulty of describing the target processor by providing an easy-to-use GUI intended to offer a minimal learning curve. Second, users should be able to load the framework and immediately begin producing simulators, and performing experiments. Third, simulators generated by our framework should also be compiled using a technique that is completely concealed from the user, avoiding any compiler configuration concerns.

Fourth, a flexible and integrated network-on-chip simulation infrastructure is necessary for the exploration of many-core designs. Finally, simulators generated by our framework must be capable of performing competitively with other major simulators in terms of instructions per second.

The remainder of this thesis will focus on the design and operation of Mhetero. We will also descuss some experiments performed with simulators constructed with the Mhetero simulation framework. The next chapter (Chapter 3) will begin by looking at Mhetero's user interface.

3. RESOURCE CONFIGURATION INTERFACE

3.1 Overview

Option-based or text-based configuration of processor simulators can often be a confusing and difficult task for novice and expert users. This process typically requires the user to learn a new programming language or data format, and may require external, third party tools. To improve the configuration process, our framework allows users to completely configure their simulator in a Microsoft Windows GUI, making the learning curve minimal to non-existent. Discussed in this section are the various editors that can be used by the simulation designer to configure their simulations. Figure 3.1 depicts the organization of the editors for the design and configuration of simulations.

3.2 Simulation Editor

The *Simulation Editor*, the first editor that users encounter, acts as a gateway to the *Resource* and *Network-on-Chip (NoC) Configuration Editors.* Simulation configurations are composed of multiple types of resources and networks; therefore, this stage is necessary to allow users to choose either editing existing resources and networks, or defining new ones. Once the user selects a resource or network, its respective editor is initiated for the user to modify its functionality. A screen shot of the *Simulation Editor* interface is shown in Figure 3.2. The remainder of Chapter 3 details the *Resource Configuration Editor*, and the *Network-on-Chip Configuration Editor* is discussed in Chapter 5.

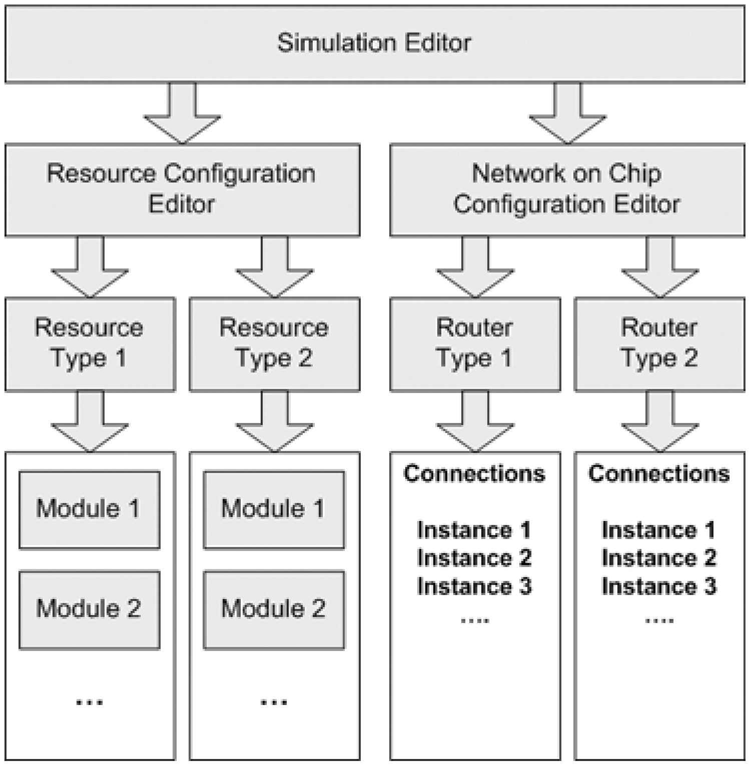
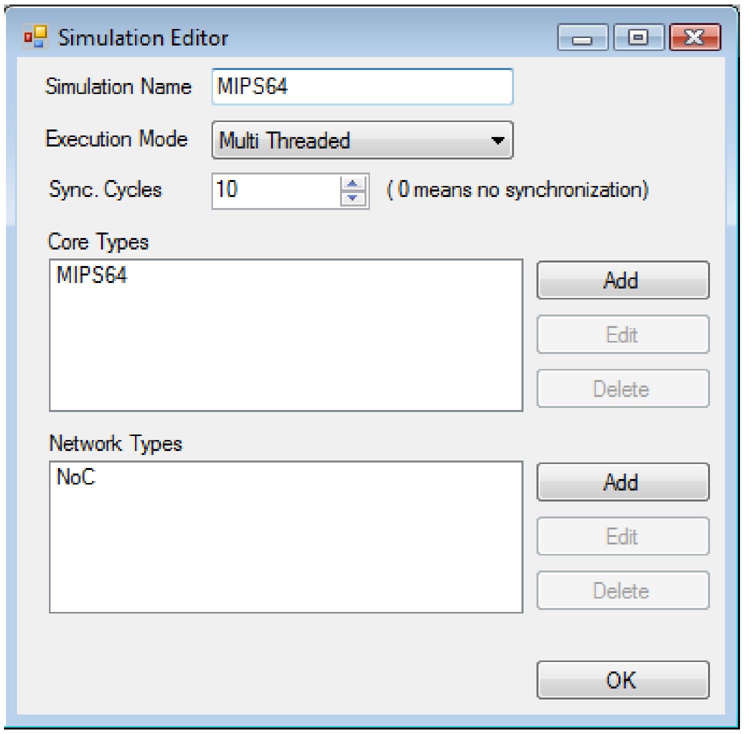


Fig. 3.1. Organization of editors within the simulation framework


Simulation Editor

Simulation Name: MIPS64

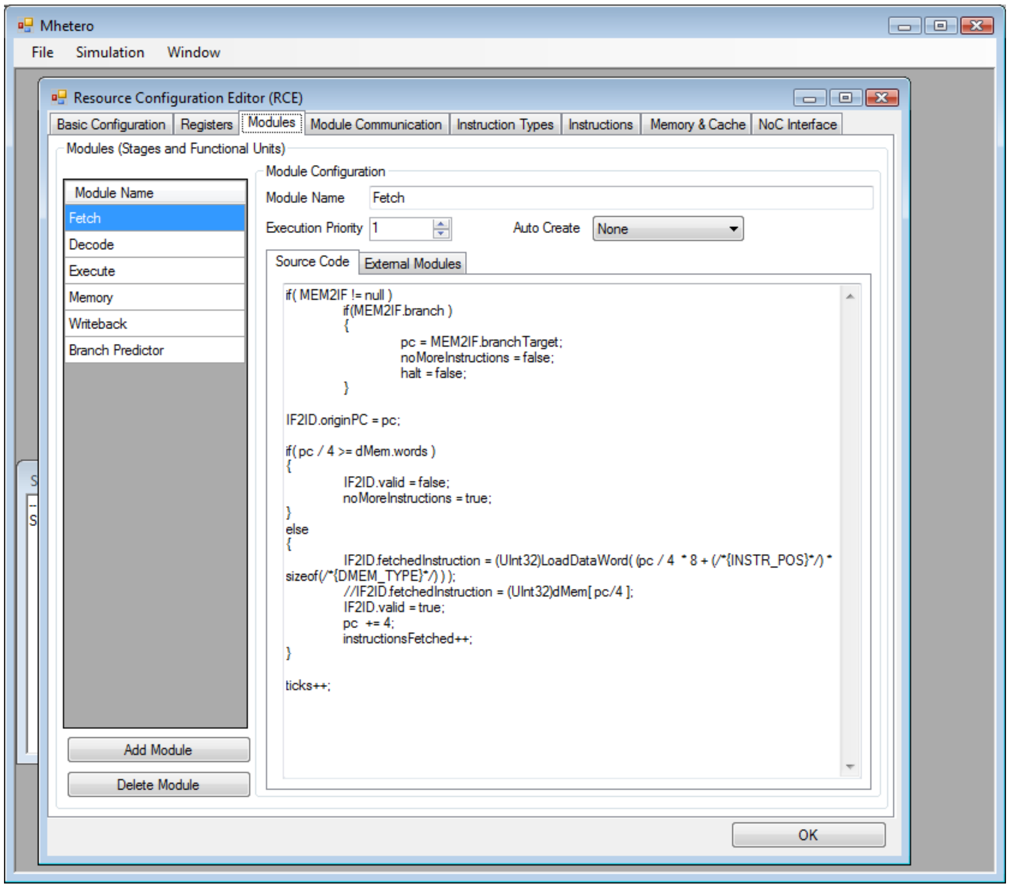
Execution Mode: Multi Threaded

Sync. Cycles: 10 (0 means no synchronization)

Core Types: MIPS64

Network Types: NoC

Fig. 3.2. A screenshot of the *Simulation Editor*



Module Name: Fetch
Execution Priority: 1
Auto Create: None

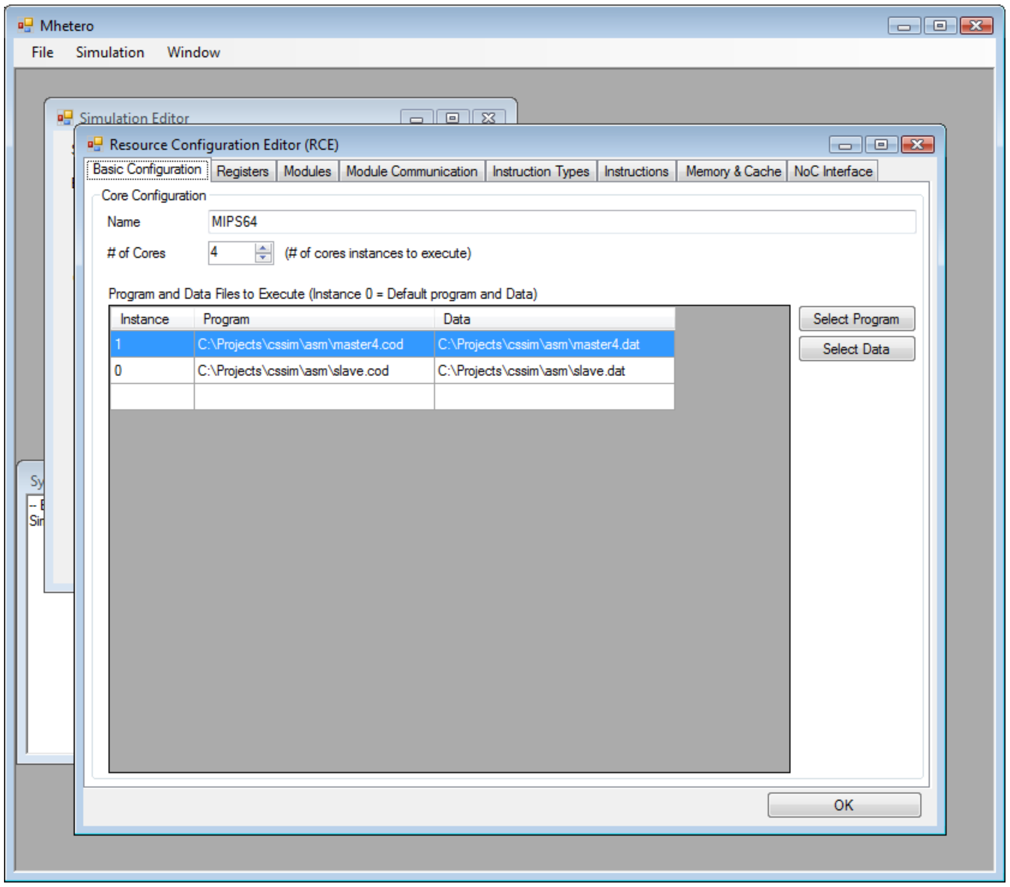
```
if( MEM2IF != null )
        if(MEM2IF.branch )
        {
                pc = MEM2IF.branchTarget;
                noMoreInstructions = false;
                halt = false;
        }

IF2ID.originPC = pc;

if( pc / 4 >= dMem.words )
{
        IF2ID.valid = false;
        noMoreInstructions = true;
}
else
{
        IF2ID.fetchedInstruction = (UInt32)LoadDataWord( (pc / 4  * 8 + (/*{INSTR_POS}*/) *
sizeof(/*{DMEM_TYPE}*/) ) );
        //IF2ID.fetchedInstruction = (UInt32)dMem[ pc/4 ];
        IF2ID.valid = true;
        pc  += 4;
        instructionsFetched++;
}

ticks++;
```

Fig. 3.3. A screenshot of the *Resource Configuration Editor* with the *Module Editor* selected

3.3 Resource Configuration Editor (RCE)

The *Resource Configuration Editor (RCE)* is the central location for editing the settings of a resource type. Within the RCE, there are many tabs that enable users to modify every aspect of the resource type, including instructions, registers, memory, cache, data flow, and behavioral logic. Figure 3.3 shows the RCE interface. Several of the more simple tabs are discussed in this subsection, and the remaining tabs are described in Sections 3.4 - 3.7.

The *Basic Configuration* tab (shown in Figure 3.4) contains fields for the name of the resource type, number of instances, and the applications to execute on each instance of the resource. Users are able to choose a default program that will run on



Name: MIPS64

# of Cores: 4 (# of cores instances to execute)

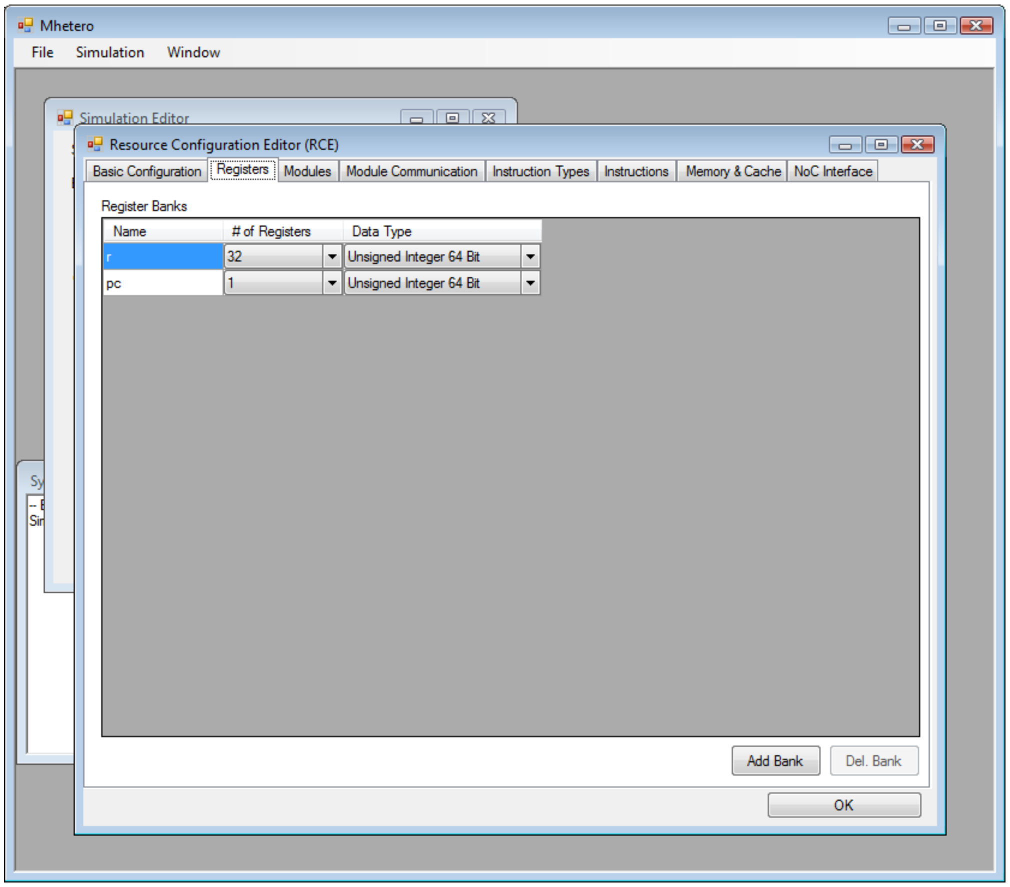
Program and Data Files to Execute (Instance 0 = Default program and Data)

Instance	Program	Data
1	C:\Projects\cssim\asm\master4.cod	C:\Projects\cssim\asm\master4.dat
0	C:\Projects\cssim\asm\slave.cod	C:\Projects\cssim\asm\slave.dat

Fig. 3.4. A screenshot of the *Basic Configuration* tab in the *Resource Configuration Interface*

all instances, and/or choose particular programs to run on specific instances. For example, to implement a master/slave distributed processing application, two programs could be used. The master program, executing on one resource instance, would be used to aggregate the results of the slave resources, running a different program.

The *Registers* tab (shown in Figure 3.5) provides an interface for the user to specify register names, number of registers, and data types. The *Instruction Types* tab allows the user to specify the instruction format which is additionally used to disassemble the resource's program for debugging purposes. The *NoC Interface* allows the user to specify the input and output queues, queue size, and data type for the resource's network interface.



Name	# of Registers	Data Type
r	32	Unsigned Integer 64 Bit
pc	1	Unsigned Integer 64 Bit

Fig. 3.5. A screenshot of the *Registers* tab in the *Resource Configuration Interface*

3.4 Module Editor

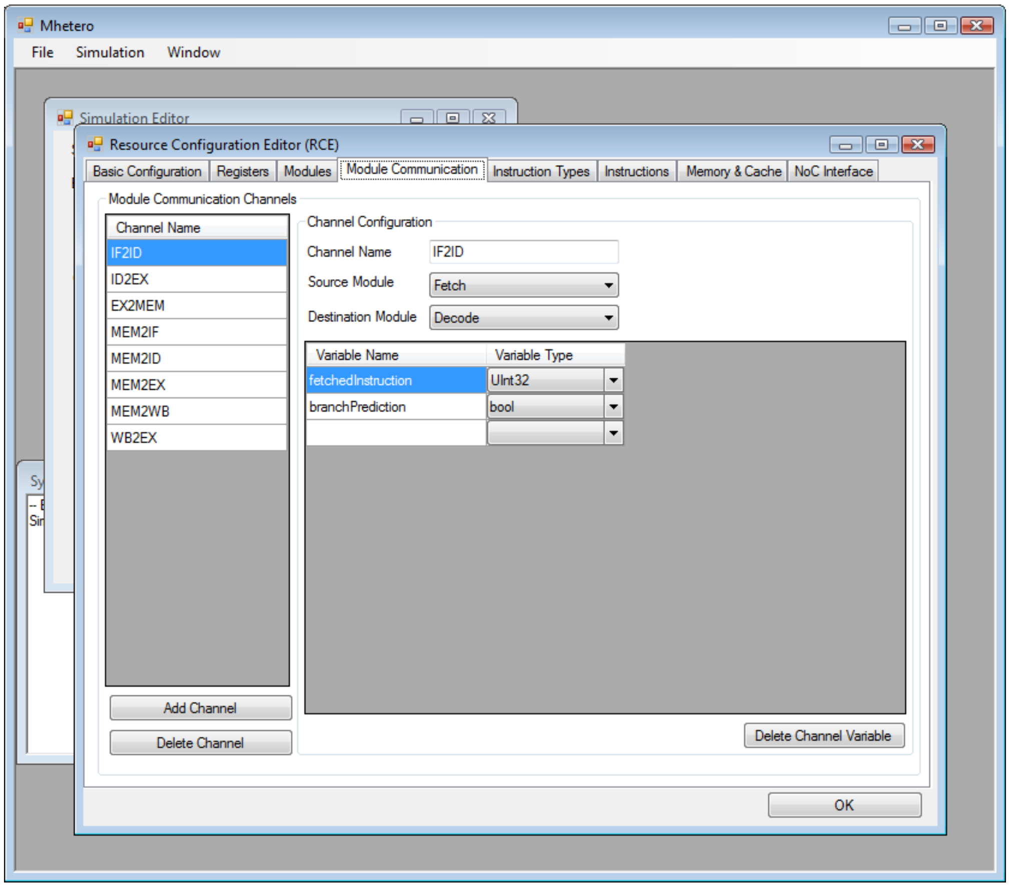
Modules are a core concept to the extendibility and configurability of the framework. Modules can represent stages or components such as branch predictors, data forwarding units, hazard detection units, or any sort of experimental unit. The modularity of the framework provides completely configurable simulations, enabling users to conceive of any sort of chip resource. Moreover, modules allow the user to easily extend the functionality of their simulations by defining a new module and assigning it a position in the resource's execution loop. The newly defined module will become a part of the simulation in its next execution.

The *Module Editor* (shown in Figure 3.3) allows the user to input the module's name, execution precedence, and a section of C# source code describing the module's behavior into the framework. The module's behavioral source code has access to all of the inputs and outputs to the module, as well as the resource's memory and registers.

External modules can also be linked to the resource in this tab. The user can choose a precompiled Dynamic-Link-Library (DLL) file, the name of the class to instantiate, and the variable name of the instantiated class (which may be referenced by other behavioral source code). External modules give the user complete control over the modules' implementation, including the ability to define additional functions, classes, and variables that will be available to other modules. Details on how external modules can be linked to the resource are given in Section 4.5.

3.5 Module Communication

In the *Module Communication* tab (shown in Figure 3.6), the user can describe data channels that connect one module to another as the resource is executed. The user must specify the source and destination modules, channel name, and variables to be included in the data channel. During the compilation process, these channels become data structures that are available as variables to the module's behavioral source code. A module should read its available inputs and act upon them, as well



Module Communication Channels

Channel Name
IF2ID
ID2EX
EX2MEM
MEM2IF
MEM2ID
MEM2EX
MEM2WB
WB2EX

Channel Configuration

Channel Name: IF2ID
Source Module: Fetch
Destination Module: Decode

Variable Name	Variable Type
fetchedInstruction	UInt32
branchPrediction	bool

Fig. 3.6. A screenshot of the *Module Communication* tab in the *Resource Configuration Interface*

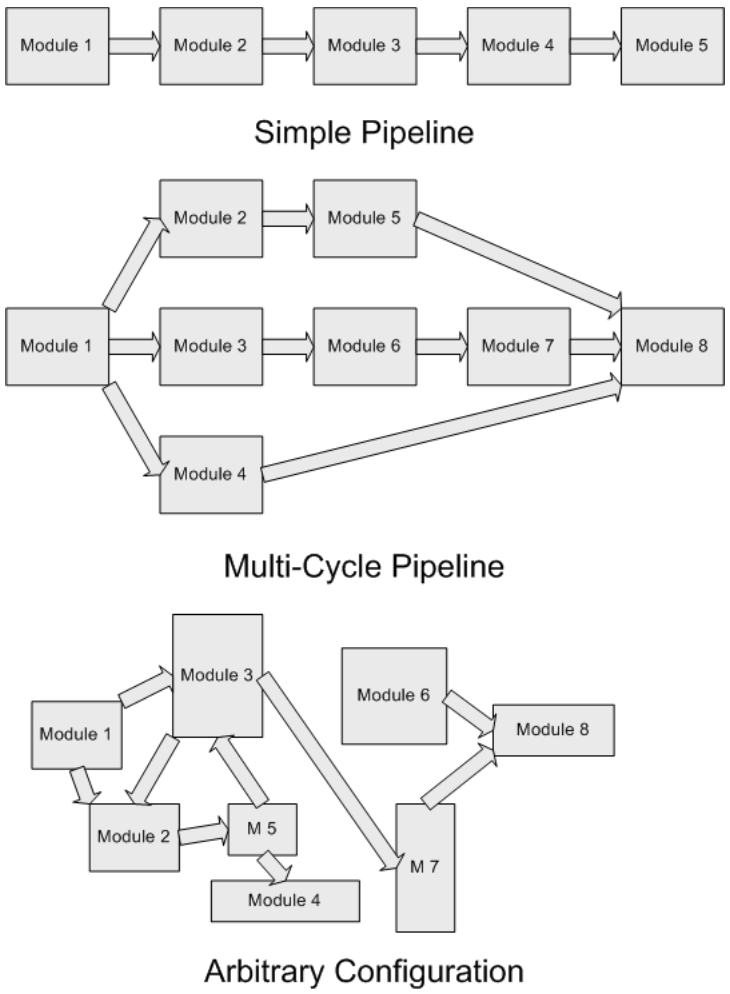


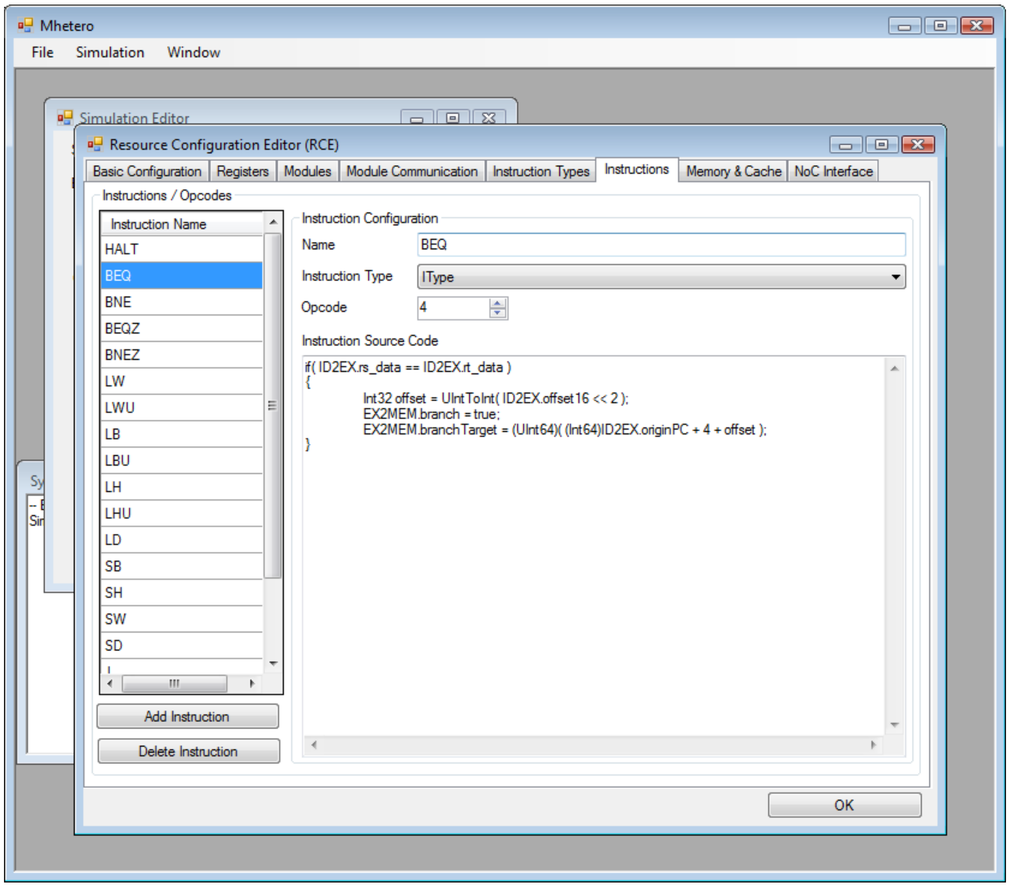
Fig. 3.7. Potential pipeline configurations

as produce valid outputs, if necessary. The transmission of data between modules is handled automatically by the framework.

Module communication combined with the module's execution precedence allows the user to design versatile resources such as a pipelined execution unit. The open architecture of our framework allows users to specify arbitrary pipeline designs as shown in Figure 3.7.

3.6 Instructions

The *Instructions* tab (shown in Figure 3.8) provides access to the instructions that are implemented in the resource. Here, users can add, delete, or edit instructions. Instructions have an associated name, op code, and instruction format type (which are specified in the *Instruction Types* tab). The C# source code that describes the behavior of the instruction is also entered here.



Name: BEQ
Instruction Type: IType
Opcode: 4

```
if( ID2EX.rs_data == ID2EX.rt_data )
{
        Int32 offset = UIntToInt( ID2EX.offset16 << 2 );
        EX2MEM.branch = true;
        EX2MEM.branchTarget = (UInt64)( (Int64)ID2EX.originPC + 4 + offset );
}
```

Fig. 3.8. A screenshot of the *Instructions* tab in the *Resource Configuration Interface*

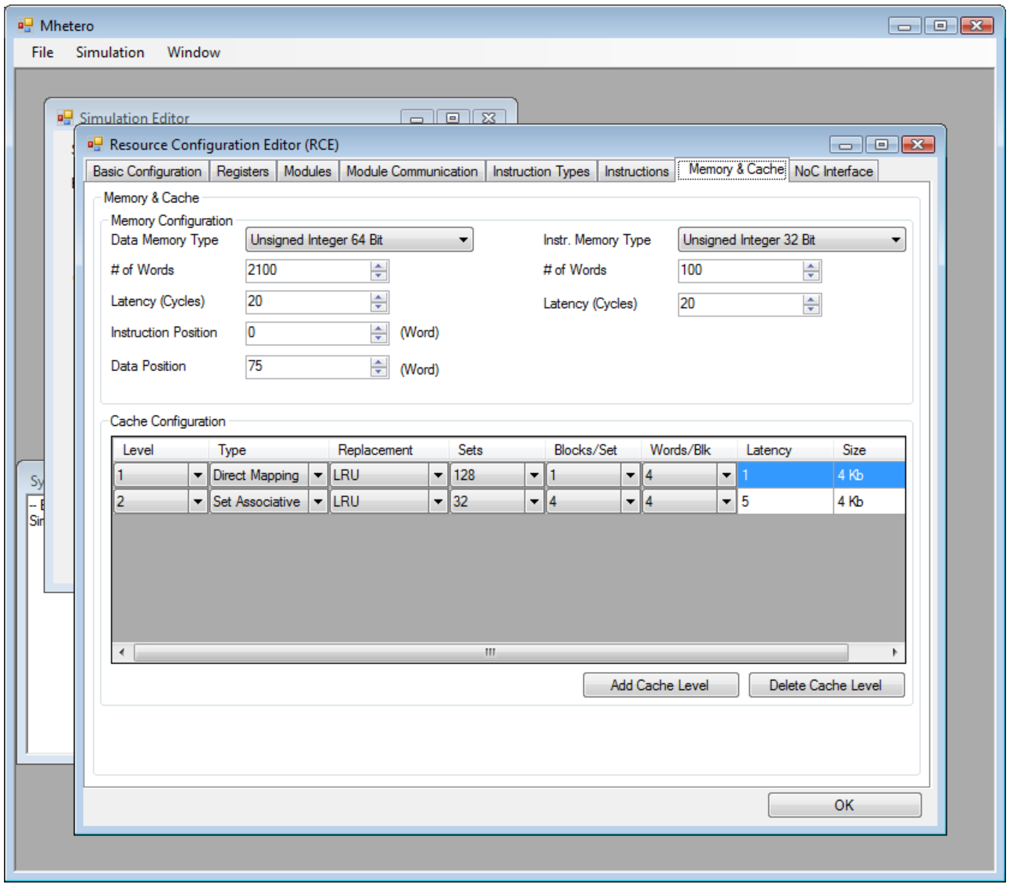
3.7 Memory and Cache

The *Memory & Cache* tab (shown in Figure 3.9) enables the user to specify the size and type of the data and instruction memory. The user may specify single or multi-level cache systems with various configurations. The framework supports Direct Mapped, Set Associative, and Fully Associative cache types, as well as Least Recently Used (LRU), and random replacement schemes. The user may also specify the cache size and latencies of each cache level.

The cache and memory systems are built into the framework and are optional for the simulation designer to use. If the cache system is used, information regarding the cache's performance is reported at the end of the simulation. Each core has direct access to the memory system; however, it may be desirable for memory to be accessed over an intra-core network. In this case, the simulation designer must implement a network and network protocol to access memory from a resource modeling a memory module. Details about intra-core networks are explained in Section 5.

3.8 Simulation Configuration Data File

Information regarding the simulator's configuration is saved in an XML format utilizing the .NET Document Object Model (DOM) XML classes: *XmlDocument* [40], *XmlNode* [41], and *XmlTextWriter* [42]. The process of saving a configuration starts with creating an empty XML configuration file. Another class, *ResourceConfig*, was implemented to store resource settings and handle the saving and loading of configuration data for resource types. Similarly, a *NetworkConfig* class was created that performs the same functions for networks. Once the output file has been created, the *Simulator* class loops through each resource and network (stored in a list as *ResourceConfig* and *NetworkConfig* classes, respectively) and invokes their individual *SaveConfiguration()* functions. The *SaveConfiguration()* function creates a new node in the new XML document, and inserts its settings.



Memory & Cache

Memory Configuration

Data Memory Type	Unsigned Integer 64 Bit	Instr. Memory Type	Unsigned Integer 32 Bit
# of Words	2100	# of Words	100
Latency (Cycles)	20	Latency (Cycles)	20
Instruction Position	0 (Word)		
Data Position	75 (Word)		

Cache Configuration

Level	Type	Replacement	Sets	Blocks/Set	Words/Blk	Latency	Size
1	Direct Mapping	LRU	128	1	4	1	4 Kb
2	Set Associative	LRU	32	4	4	5	4 Kb

Fig. 3.9. A screenshot of the *Memory & Cache* tab in the *Resource Configuration Interface*

To load a configuration, the *Simulator* class must load the XML file, and examine the XML tree to determine the number of types of resources and networks to prepare for loading. *Simulator* instantiates the appropriate number of resources and networks, and then invokes their *LoadConfiguration()* function. Each *LoadConfiguration()* function is sent a reference to the appropriate portion of the XML tree to load settings from.

The behavioral source code of modules, instructions, and routers entered by the user through their respective editors, is also stored in the configuration XML file. The behavioral source code must be encoded so that characters such as greater-than and less-than signs do not interfere with the XML format. We solve this problem by using another Microsoft .NET class, *HttpUtility* [43] typically used for Internet communication. This class contains two functions which encode and decode text to and from a format that will not interfere with the XML file's formatting. This arrangement of configuration data allows the framework to store and load entire simulation configurations, including multiple heterogeneous cores and networks, into a single file.

A brief example of a file format with one resource is shown in Figure 3.10. The file format places no restriction on the number of resources, modules, instructions, and networks defined in the framework. When a new resource/module/instruction/network is defined in the framework, the new data is stored in memory until the data is saved. The new data overwrites the old data, resulting in the addition of a new resource/module/instruction/network tag into the appropriate location of the XML tree.

```
<Simulator Name="MIPS64">
  <Resource Name="MIPS64" Num="1">
    <Instructions OpcodeLoBit="26" OpcodeHiBit="31">
      <InstructionType Name="RType">...</InstructionType>
      ....
      <Instruction Name="HALT" Type="RType" OpCode="0x1">
        (Encoded C# Source Code)
      </Instruction>
    </Instructions>
    <Registers>
      <Register Name="r" Size="32" Type="UInt64" />
      ...
    </Registers>
    <Modules>
      <Module Name="Fetch" ExecutionOrder="1" Type="None"
                      InstanceName="" ClassName="" ExternalDLL="">
        (Encoded C# Source Code)
      </Module>
      ...
    </Modules>
 ...
</Simulator>
```

Fig. 3.10. An example of the Mhetero XML file format

4. THE FRAMEWORK'S STRUCTURE AND DYNAMIC COMPILATION

4.1 Overview

One of the primary benefits of our framework is its ability to dynamically compile source code into executable code quickly and seamlessly. Dynamic compilation refers to the framework's ability to take configuration and behavioral data, and produce an executable library at run-time. Without leaving the framework's interface, the user can make large and small modifications to a simulator's configuration and test those modifications immediately. The framework does not generate any external executable files that the user would need to run as a separate process. Instead, the framework takes the simulation configuration that is entered into the framework's GUI and assembles a complete simulator, which is loaded into memory and executed as part of the main framework.

Simulator compilation generally takes less than a second as the source code is compiled to an intermediate language called Microsoft Intermediate Language (MSIL) [6]. The behavioral source code of a module, instruction, or router can be modified through their respective editors. If there are any errors present in the behavioral source code, the framework provides detailed error reports similar to those provided in Microsoft Visual Studio. Errors can be quickly and easily corrected inside the framework's GUI, and a new simulator can be built. Since the .NET framework includes all of the necessary functionality, the entire process has no external dependencies that are required for the user to download and install.

Integrating the C# compiler into the framework provides users with a very convenient and excellent development experience specialized for computer architecture simulation without any of the pitfalls associated with relying on third party compilers or development tools. This technique also enables the framework to compile and link processor simulators to memory leaving no left-over files in the file system for cleanup.

In this section, we discuss how we structured the simulator to support this behavior, how the dynamic compilation is implemented, and how the framework communicates with the newly generated simulator executed inside the framework.

4.2 Framework Structure

Two classes, *Simulator* and *Network*, make up the core of the dynamic compilation implementation. Figure 4.1 shows the organization of these two classes within the framework. The *Simulator* class was constructed to interface the simulation framework to the chip's resources and networks. *Simulator*, once invoked, handles the compilation, initialization, and instantiation of the various resources within the simulator. The *Network* class provides an interface from the *Simulator* class to the individual routers and is treated similar to other resources. The primary difference between the *Network* class and other resources is the compilation process. After *Simulator* has compiled all of the resources, *Network* is invoked to compile routers. Resources and routers are represented by the *Resource* and *Router* classes respectively, which are detailed in Section 4.3.

The framework allows the user to create multiple types of resources and routers in the simulated system. Each resource and network type can be instantiated an arbitrary number of times, according to the simulation's configuration. Creating multiple types of resources thus leads to a heterogeneous simulation. In addition, multiple types of networks are desirable for transferring different types of information. For example, one network may transmit data streams, while another may transmit small packets. Moreover, some NoC implementations may include a memory system modeled as a chip resource, so networks for accessing memory may also be necessary.

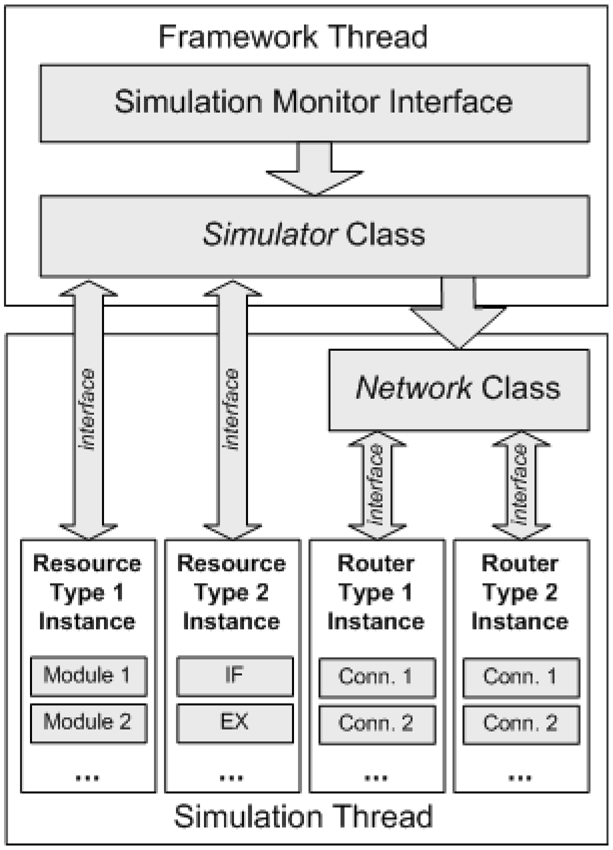


Fig. 4.1. Organization of the framework structure and communication between resources and routers

4.3 Implementation of Dynamic Compilation

Before the compilation process can begin, the source code of the simulator must be gathered by the framework. Figure 4.2 shows the flow of how the source code is combined to produce an executable simulator. A generalized parent class, *Resource*, is included in the framework that contains only the basic structure and functionality needed to interface with the framework. The configuration data gathered in the RCE for each resource is combined into the *Resource* class to construct new classes that implement the behavior of the simulator. The source code of the modules within each resource is gathered and inserted into the *Resource* class at the appropriate locations based on each module's execution precedence (defined in the RCE). The network routers undergo a similar process as the resources; their configuration data is combined with a generalized *Router* class and then instantiated and managed by the *Network* class. The remaining resource configuration and simulation settings are also analyzed and interpreted by the framework to generate the remainder of the source code.

In addition, the framework can automatically generate source code for an execution stage during compilation. This is necessary to make use of the instruction source code that is entered by the simulation designer in the *Instructions* tab of the *RCE*. In the *Module Editor*, the user can specify a module for the framework to insert the automatically generated execution stage source code. If this option is chosen, the framework will assemble the every instruction's source code into a *switch* [44] statement during the compilation process. The *case* statements in the *switch* correspond to the instructions entered by the user. The instruction's behavioral source code is then inserted into the body of the *case*. During the simulation, a decoded instruction's op code is used to select the appropriate *case* to execute.

Once the simulator source code has been pieced together, the program is compiled. The compilation utilizes the C# Compiler (CSC.exe) [45] included in the .NET framework distribution, assuring wide availability with no additional configuration or installation. The C# compiler produces the same error messages along with their line numbers as the Microsoft Visual Studio development environment does. If errors are found, they are displayed in a status window for users to examine and make corrections to their module, instruction, or router source code.

The execution of the C# compiler is managed by the *CSharpCodeProvider* class [46]. We use the *CompileAssemblyFromSource()* [47] function included in the *CSharpCodeProvider* class to produce an *Assembly* [48] which represents the compiled code. *CompileAssemblyFromSource()* takes two parameters, the source code and *CompilerParameters* [49]. *CompilerParameters* contains many of the compiler settings available to developers in the Microsoft Visual Studio, such as setting warning levels, and including debug information. We make use of the *ReferencedAssemblies* property to include external modules, as well as System.dll. *CompileAssemblyFromSource()* re-

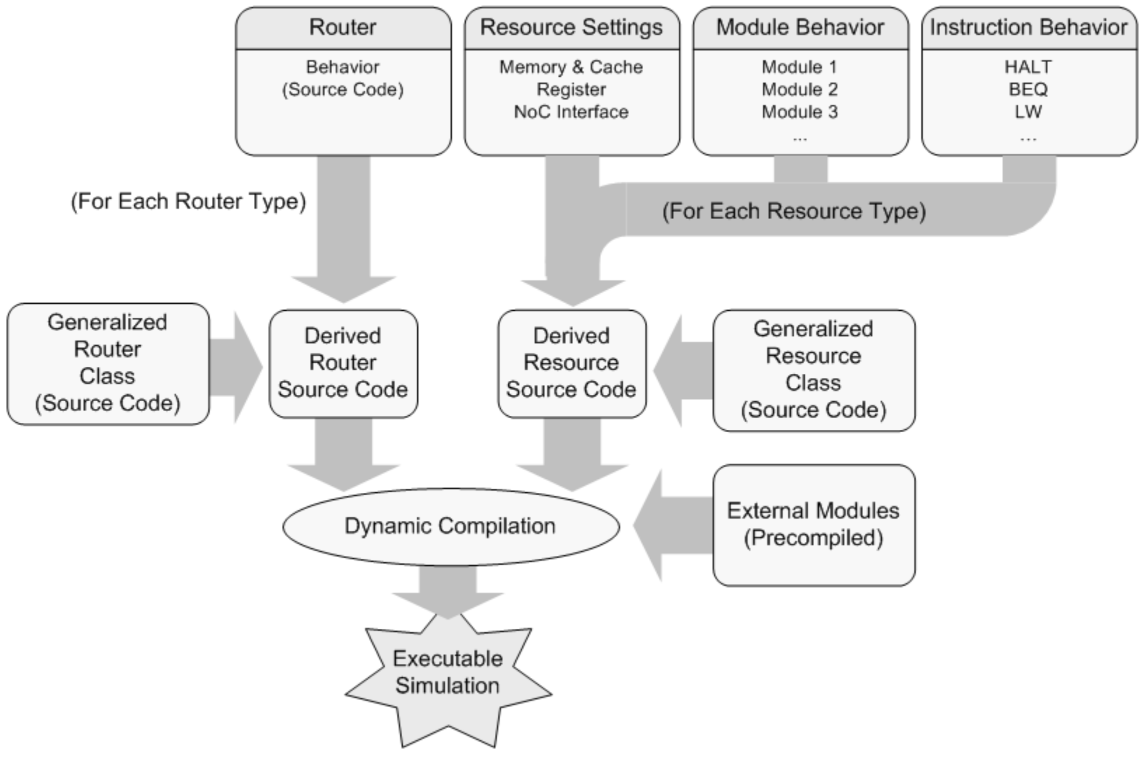


Fig. 4.2. Flow of configuration data, source code, and external modules in the compilation process

turns compilation results which provide error messages or a reference to a compiled *Asssembly*. The compiled Assembly data structures are stored in a *List* [50] and used for instantiating the new resource and router classes.

4.4 Communication Between the Framework and Simulator Components

Communication between the resources, modules, and routers are facilitated by the *interface* capability [51] which is provided in C#, as well as other .NET languages. *interface* enables developers to generalize the signature of function calls which should be included into the dynamically linked library or program. That is, *interface* provides a method to initiate function calls between the framework and the classes of the dynamically compiled simulator. The generalized resource and router classes (shown in Figure 4.2) implement standard function calls that allow the framework to communicate with the compiled and instantiated code. The communication is primarily used for transmitting statistical information, as well as starting and stopping the simulation.

4.5 External Modules

External modules are precompiled Dynamic-Link Library (DLL) [52] files containing a class that implements the functionality of a module. During the compilation process (described in Section 4.3), any external modules that are included in a resource are loaded and linked into the compiled code. This is accomplished by adding the external module to the *ReferencedAssemblies* variable [53] in the *CompilerParameters* class that is prepared before compilation is initiated. When the resource is instantiated, the external module is available to the resource and executed as if it were an internal module.

External modules provide several benefits that may make them desirable to some users. First, enabling external modules make it easier to swap modules into and out of the framework, and transmit them with other users. Second, external modules give users complete control over the programming of the module, as long as it implements the *Init()* and *Run()* functions. For example, the user can declare new classes, additional global variables, or additional functions in an external module, which regular modules do not provide since they must only implement the behavioral source code. Third, the functions declared in external modules are available for other modules to call, which may be desirable in some circumstances. For example, if the user chose to implement a power consumption external module, the module could implement a function that would be called from other modules to tally power consumption. Finally, the module can be implemented using any .NET-compatible language whereas internal modules must be written in C#.

An external module must be compiled with a reference to the framework's executable (i.e., in the Visual Studio project settings). The reference enables the external module class to implement the appropriate *interface*, *IModule*, which ensures that the DLL file will be compatible with the framework. Additional functions may also be implemented to be used within the module, or to be called from other modules.

Figure 4.3 illustrates how two external modules could be integrated into a resource's execution loop.

4.5.1 Debugging External Modules

Simulation designers have severals options for debugging their external modules. The first method is to send a message (as a *string* [54], similar to the C function *printf*) from the external module to the framework to be displayed to the user. When the *Init()* function of the external module is called, an *interface* to its parent resource (i.e., the resource to which the external module belongs) is sent as a paramter which provides access to debugging function *WriteToUI()*. *WriteToUI()* enables the external module to write to the *Simulation Monitor's Simulation Status* panel. *WriteToUI()* also includes a priority level parameter to specify if the message is a warning or error. Warnings can be disabled by the user to improve performance.

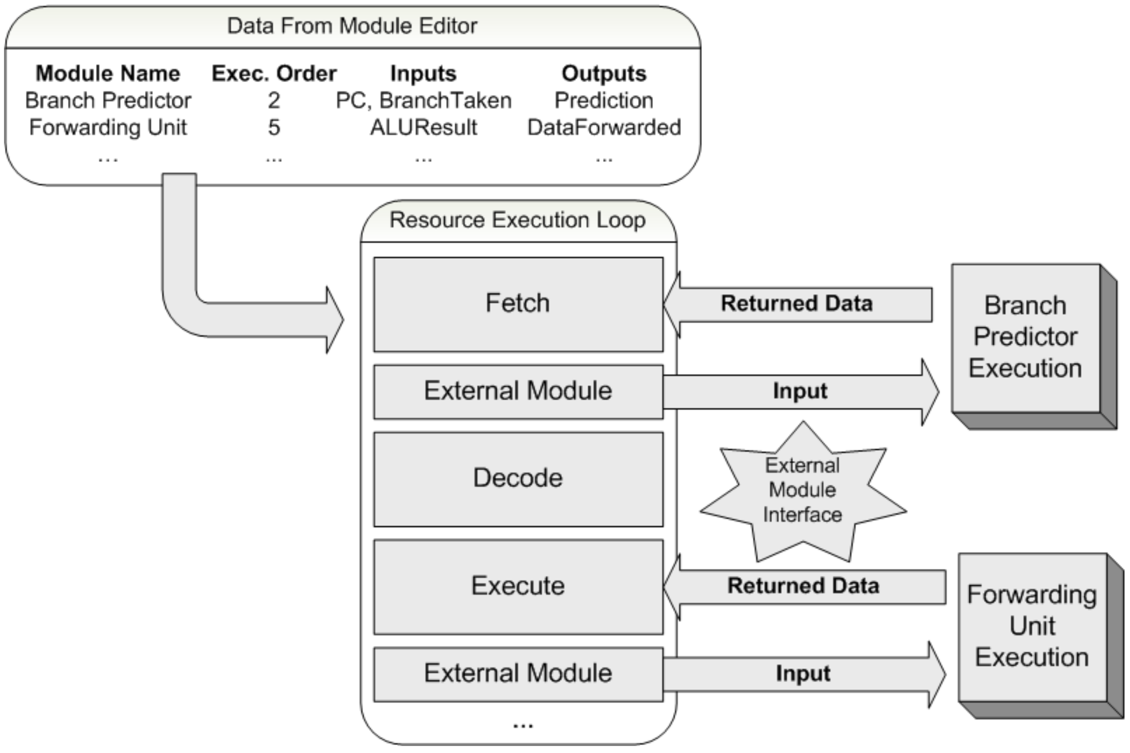


Fig. 4.3. Diagram of two external modules integrated into a resource's execution loop

Another method of debugging external modules is through Microsoft Visual Studio [55] (MSVS). This method enables the user to debug and breakpoint their external modules as though they were developing a regular application. To accomplish this, the simulation designer must alter the *Configuration Properties* of their project (which is a configuration window within MSVS) by setting the executable file used for debugging to the simulation framework's executable (specified in the *Start external program* field of the *Debug* tab, located in the *Configuration Properties* window). This will execute the framework when the external module's debugging process is started. Once the framework has started and a simulation is loaded, the framework will load the exernal module's DLL file that is monitored by MSVS. This will allow the simulation designer to analyze the execution of the external module by setting a break point, similar to how a regular application would be debugged.

The final method is to debug the simulation framework project and the DLL project at the same time. In this case, the simulation designer would need to setup both projects in MSVS (multiple projects can be added through the *Solution Explorer*). The simulation configuration should then be adjusted (in the framework) to load the DLL file produced when MSVS compiles the external module (i.e., the newly compiled external module will be loaded by the framework when the simulation configuration is loaded). When a debugging session is started, the simulation designer will then be able to choose a breakpoint in the external module's source code, and debugging may proceed as normal. Additional information regarding debugging DLL files can be found in the Microsoft Developer Network website [56].

5. NETWORK-ON-CHIP

5.1 Overview

Network-on-Chip (NoC) has become one of the leading methods for intra-core communication in current and emerging processor designs. NoCs are widely viewed as fast, power efficient, and scalable to hundreds of cores. Furthermore, NoCs can support multiple voltage domains, clock frequencies, and heterogeneous designs. Thus, NoC support is a critical part of our support for heterogeneous many-core simulations. In this section, we will introduce NoCs, followed by a discussion of our NoC implementation and the NoC Configuration Editor.

5.2 Introduction to Network-on-Chips

One of the first publications that introduced NoCs was written by W. Dally and B. Towles in 2001, titled *Route Packets, Not Wires: On-Chip Interconnection Networks* [57]. The publication introduced the concept of connecting network clients (i.e., processor resources such as cores and memories) together through a network instead of dedicated lines and busses. This concept enables network clients to communicate with each other through a series of network routers that connect to other clients, and thus all clients are able to send or receive packets from any other client. Dally's publication also described higher level protocols for network communication and discussed some of the challenges that would occur in NoC designs.

NoCs have several benefits that have lead to their adoption among current [58–60] and future concept [3] processor designs. First, NoCs inherently enable parallel processing since each resource is free to execute independently from either the network or other resources. Second, the NoC circuitry is small relative to the size of the resources on the processor (the estimated size of a NoC is 6.6% of a chip's silicon [57]) due to the high duty-cycle of wires since they are to be utilized for many purposes. Third, NoCs enable the chip to have reduced power disipation and fast signal propagation between network clients [61] due, in part, to the shorter wire length between routers. Fourth, NoCs are scalable to large numbers of cores (such as the 100 core Tilera *TILE-GX* processor [60]) due to the large amount of paralellism that is achieved from the concurrent operation of routers. Finally, NoCs are considered to be very reliable relative to their larger-scale counterparts since the network's environment (i.e., integrated into the circuitry on the chip) is predictable and known during the design phase.

NoCs can also help avoid problems caused by dedicated wiring from resource to resource. The routing of NoC lines are generally defined at the beginning of the design phase to be isolated from internal resource wiring. This reduces unwanted parasitic capacitance or cross-talk between lines. Additionally, as processors become larger and more complex, the dedicated connections between resources become longer; eventually requiring repeaters. Placing repeaters at their proper location on the chip is difficult in later stages of the design phase. Repeaters also increase power consumption. NoCs eliminate this problem since routers are placed at regular intervals throughout the network. The topology of the network is generally known at the beginning of the design phase, and as a result, NoCs will have predictable levels of resistance and capacitance. The network topology can later be used to calculate a worst-case transmission scenario to specify the network's speed (frequency) [57].

Some considerations for NoC designers (and simulation designers) include the network topology, routing function, as well as the number of networks and their purpose. Network topology refers to the organization or layout of resources on the chip and the arrangement of network connections and routers that connect them. The routing function describes how data will flow between routers and the direction that the data is redirected by the router (thus, the routing function is heavily influenced by the network topology). Seperate networks may be implemented on the chip to seperate tasks for the purpose of maximizing bandwidth and minimizing traffic.

Data is usually transmitted over a NoC as either a packet or a stream. Data packets can be split up into smaller pieces called flits. Flits are sections of a data packet that can be transmitted over the network's wires at once, and thus their size is limited to the number of wires. Some NoC implementations allow for streaming data transmission, which is a function of the routers. Data streams are implemented as a series of network connections that are reserved, for some period of time, between one resource and another. This process is usually implemented by routers that relay data along the reserved path as soon as it's received.

In real-world NoC implementations, routers and resources usually interface to the network through some kind of buffer. Buffering allows resources to reassemble the flits into packets before processing. Routers have some flexibility on how flits are handled, depending on the routing function. For example, flits can be immediately relayed to its destination upon arrival, or it can wait for the entire packet to arrive before beginning to relay. Buffers allow resources to continue working on other tasks until they are ready to receive network data. Furthermore, buffers also serve to minimize transmission errors.

5.3 Network-on-Chip Implementation

In the simulation framework, networks are a collection of connections and routers that facilitate communication between resource instances. Routers and resources interface with the network through inputs and outputs, which are implemented using the FIFO queue .NET class, *Queue* [62]. Connections (described in more detail later) in the network simulate the wires of a physical network which make the connection from an output to an input. Routers are responsible for managing the flow of data from its inputs to the appropriate output, which occurs within the routing function. The *Network* class (described in Section 4.2) manages the flow of data through the

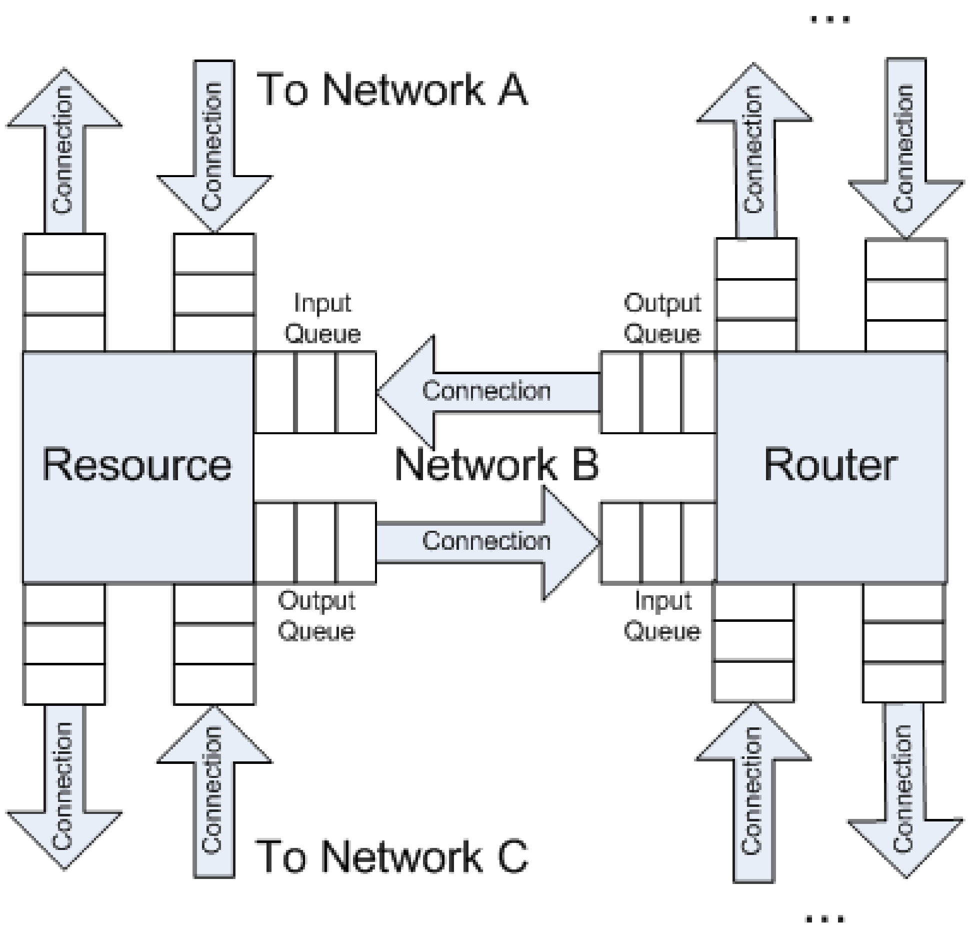


Fig. 5.1. A detail of how resources, routers, and connections form a network

connections and executes the routing functions for each Router instance. Figures 5.1 and 5.2 show how all of these components can connect to form a network.

The simulation designer may choose to implement multiple networks. This is common in modern NoC designs, as each network is used for a specific purpose such as memory requests, cache synchronization, or streaming data. Each network type can define multiple router types, as well as multiple instances of each router type. Since the network interfaces of routers and resources are standardized, connections can span between different router types; even router types existing in different networks.

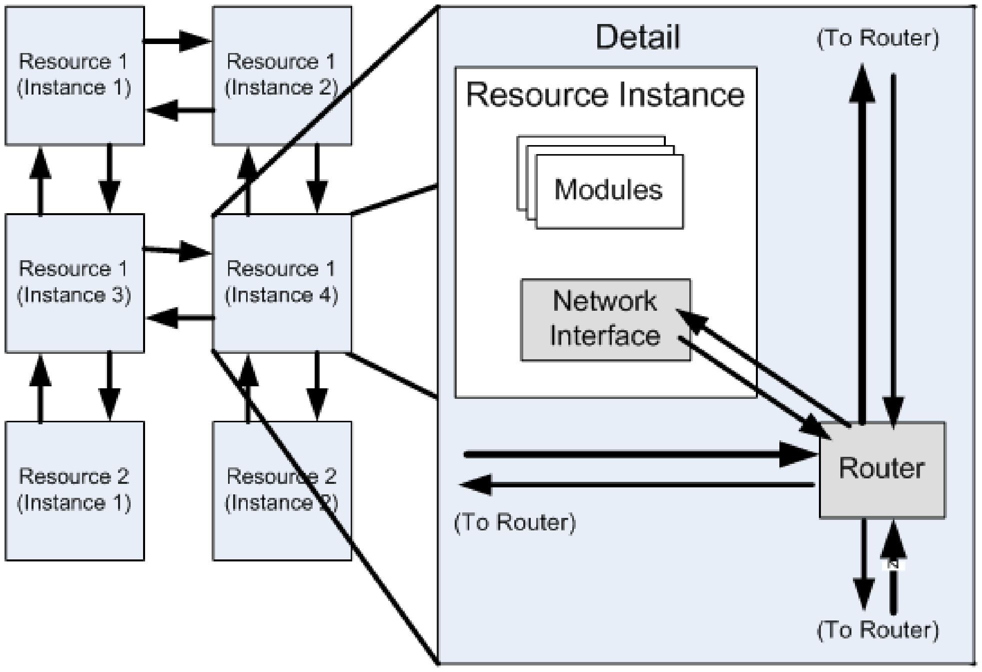


Fig. 5.2. An example of a small 2D mesh network (on a larger scale than Figure 5.1)

This results in an extremely flexible NoC implementation that can simulate arbitrary network topologies.

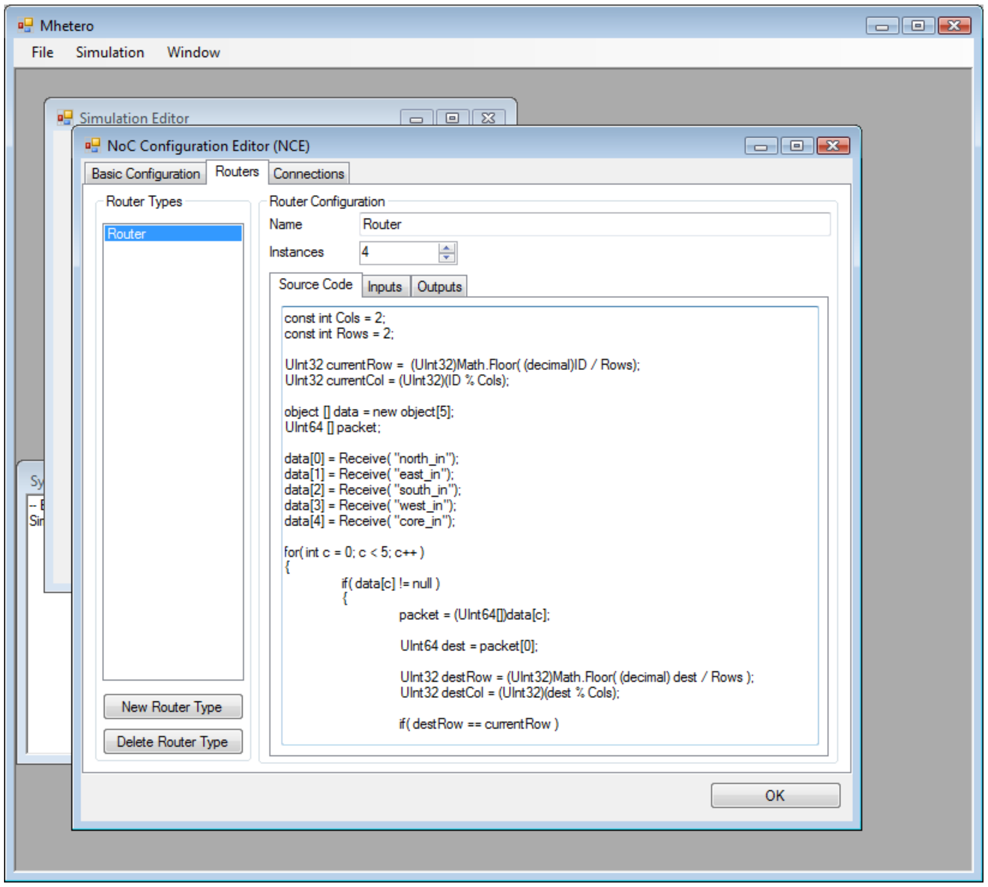
During each cycle in which the simulator is executing, each network will process all of its connections and initiate the routing functions of each router instance. The *Network* class stores the connection configuration data in a list through which it iterates to move data packets from outputs to their corresponding inputs assigned to the other end. The size and data type of the data packet depend on the output and input types, specified by the network interface in either the *NoC Configuration Editor*, or the RCE. Packets can also be represented by arrays, enabling simulation designers to transmit large amounts of data per cycle (this functionality is provided to maximize configurability, and may not be realistic in a physical implementation).

Resources and routers communicate through the network by manipulating their input and output queues. These queues are available through the behavioral source code of resources and routers that are connected to the network. Resources can expose the network interface to the simulated program any number of ways, and it is left up to the simulation designer to specify how this should work. For example, network transmissions can be implemented by either mapping a register to an input/output, memory mapping, or by executing an instruction.

This NoC implementation is extremely open, allowing the simulation designer to produce virtually any kind of network topology imaginable. Moreover, the simulation configuration is also not limited to any particular routing function or router placement.

5.4 Network-on-Chip Configuration Editor

The *NoC Configuration Editor* (shown in Figure 5.3) allows users to define the router types and connections between the routers and resources. Router types have a name, number of instances, source code, and input and output queues. The source code describes the routing function of the router, i.e., which inputs connect to which



Name: Router

Instances: 4

```
const int Cols = 2;
const int Rows = 2;

UInt32 currentRow =  (UInt32)Math.Floor( (decimal)ID / Rows);
UInt32 currentCol = (UInt32)(ID % Cols);

object [] data = new object[5];
UInt64 [] packet;

data[0] = Receive( "north_in");
data[1] = Receive( "east_in");
data[2] = Receive( "south_in");
data[3] = Receive( "west_in");
data[4] = Receive( "core_in");

for( int c = 0; c < 5; c++ )
{
        if( data[c] != null )
        {
                packet = (UInt64[])data[c];

                UInt64 dest = packet[0];

                UInt32 destRow = (UInt32)Math.Floor( (decimal) dest / Rows );
                UInt32 destCol = (UInt32)(dest % Cols);

                if( destRow == currentRow )
```

Fig. 5.3. A screenshot of the NoC Configuration Editor interface

outputs. The input and output queues are assigned a name, size, and data type. The queues are accessible by the routing function, along with the router's instance number (ID). The instance number can be used to determine its location within the network.

Connections must specify which type of resource or router it is connecting to, and which input and output queues to read from or write to. The user must also specify which instance number that the connection is operating on. Connections can also have a delay (in cycles), which enables users to simulate the transmission of a packet of data over the connection in multiple flits.

5.5 A Note About Mhetero's Network-on-Chip Infrastructure

Mhetero's NoC implementation was designed to facilitate many-core simulation and is not designed to be a detailed network simulation. Thus, our implementation should be considered to be a functional NoC and should not be considered for analysis of power consumption or network timing. However, the simulation designer is free to add such functionalities either through external modules or by adding the functionality into the routing functions.

6. SIMULATION EXECUTION

6.1 Overview

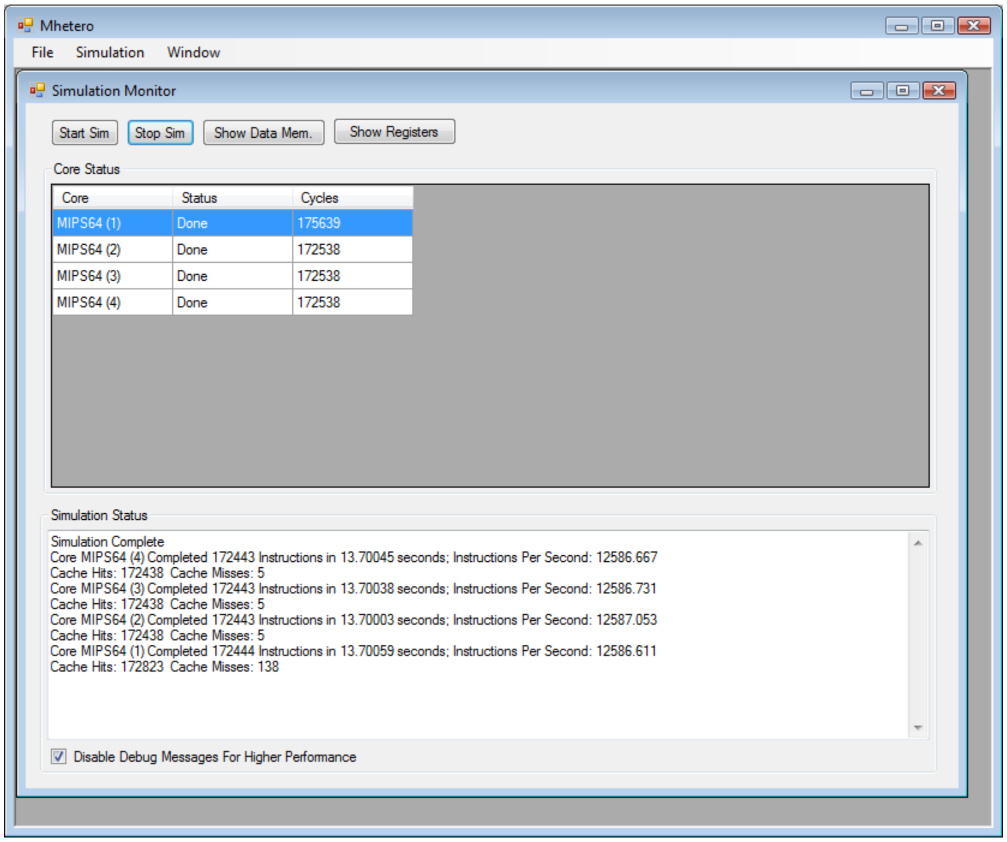
This chapter will focus on our approaches to simulation execution within the framework. We begin by looking at how single threaded execution was implemented. Then we will discuss several concerns with multithreaded execution, and how we implemented two different multithreaded approaches into the framework.

6.2 Single Threaded Execution

The simulation executes in a different thread (referred to as “Simulation Thread” in Figure 4.1) from a thread of the framework and its GUI (referred to as “Framework Thread”). The compilation of the resources and routers is initiated when the user builds the simulator, which is a process that must be completed before the user can initiate the *Simulation Monitor*. The *Simulation Monitor* is an interface that monitors the execution of the simulation. A screen shot of the Simulation Monitor is shown in Figure 6.1. Executing the *Simulation Monitor* instantiates the *Resource* and *Network* classes, required by the simulation, and prepares the execution of the simulation thread. The user must press the “Start Sim” button to begin the simulation.

Once the simulation has been started, the simulation thread is initiated and every instance of the resources and networks is executed. They are executed sequentially (i.e., each resource and network executes one cycle, then proceeds to the next cycle) until each resource has completed their programs. During a resource’s cycle, all of its modules are executed within a try-catch [63] block which protects the framework thread from exceptions. During a network’s cycle, each connection is examined for data waiting to be transmitted and then each router's routing function is executed to process the data.

The *Simulation Monitor* periodically checks on the status of each resource to see if execution has completed. Once the resource has completed its execution, its status is changed to "Done", and performance and statistical information regarding the resource's performance are presented to the user. Runtime exceptions are also reported to the user in an information text box that is located in the *Simulation Monitor* window.



Core	Status	Cycles
MIPS64 (1)	Done	175639
MIPS64 (2)	Done	172538
MIPS64 (3)	Done	172538
MIPS64 (4)	Done	172538

```
Simulation Complete
Core MIPS64 (4) Completed 172443 Instructions in 13.70045 seconds; Instructions Per Second: 12586.667
Cache Hits: 172438  Cache Misses: 5
Core MIPS64 (3) Completed 172443 Instructions in 13.70038 seconds; Instructions Per Second: 12586.731
Cache Hits: 172438  Cache Misses: 5
Core MIPS64 (2) Completed 172443 Instructions in 13.70003 seconds; Instructions Per Second: 12587.053
Cache Hits: 172438  Cache Misses: 5
Core MIPS64 (1) Completed 172444 Instructions in 13.70059 seconds; Instructions Per Second: 12586.611
Cache Hits: 172823  Cache Misses: 138
```

☑ Disable Debug Messages For Higher Perfomance

Fig. 6.1. A screenshot of the *Simulation Monitor* interface

6.3 Overview of Multithreaded Execution

Unlike single threaded execution, where one simulation thread executes all resources and networks, multithreaded execution enables resources and networks to execute concurrently over many threads. Multithreaded execution enables the framework to fully utilize all of the physical cores available in the simulation host (i.e., the machine that is executing the simulation) by allowing the operating system to schedule the simulation's threads on them, thus enabling concurrent execution of the simulator. Theoretically, as the number of physical cores increases in the simulation host, multithreading becomes more advantageous. While multithreading may seem like a large advantage, there are several challenges that must be overcome to realize improved performance.

By definition, discrete event simulations must be deterministic and repeatable. However, multithreading may introduce some uncertainty into the timing of the simulation for two reasons. First, the simulator cannot guarantee that each thread receives equal processor time since the operating system manages the assignment of a thread to a physical core. Second, the framework supports heterogeneous resources and networks, and therefore the execution time for each thread is likely to be different, producing synchronization problems. Thus, simply assigning different resources to different threads will produce simulations that are neither deterministic nor repeatable, which would invalidate the simulation's results. Our approach to thread synchronization is discussed in Section 6.6.

Executing resources and networks each in its own thread does not guarantee improved performance because the overhead associated with allocating and initiating the threads causes a large reduction in the simulation speed as the number of threads becomes large (i.e., greater than 100 resources and networks). Moreover, depending upon the granularity (i.e., the modeling detail of a resource/network, and thus, the processing time required) of the simulation, the overhead of executing resources in different threads may be greater than the speed benefit that comes from multithreading. If the simulation is not detailed enough (e.g., if each resource runs only hundreds of cycles), the resulting simulation would be slower than if it were executed in a single thread.

Overhead in multithreaded execution is caused by context switching and thread synchronization. The framework must use kernel-level threads to take advantage of multiple cores in the simulation host as only the operating system can handle the scheduling of such threads on multiple processor cores. However, there is some delay caused by context switching from one thread to another. Context switching refers to interrupting the execution of one thread, saving its registers, loading another thread's registers, and resuming execution of the restored thread. In addition to the context switching delay, the operating system itself causes some delay as it must also manage the scheduling of the threads. Thread synchronization, which is necessary for cycle accuracy in multithreaded implementations, also imposes some overhead as some threads may stop working while waiting for other threads to complete their tasks. In a case where the amount of work performed by each thread is very small relative to the multithreading overhead, the resulting program would run slower than if it ran in a single thread.

Therefore, multithreaded execution may or may not be desirable for any particular simulation configuration and simulation host. For example, a simulation configuration that is very detailed would likely be able to take advantage of multithreading, while a simple functional simulation would likely run faster in a single thread. Fortunately, our framework is designed to leave this decision up to the user, who is free to choose an execution method based upon trial and error, or experience.

In our framework, we have implemented two different multithreaded execution approaches. The first approach is to initiate one thread for each resource and network. This method was our first attempt to implement multithreaded execution because the modular nature of the framework's resource and network classes create a natural division for which to assign threads. Additionally, this approach allowed for an easier transition from the single threaded implementation. We will refer to this approach as One Thread per Resource and Network (OTRN) approach to be concise. Later, a second approach was implemented that utilizes a thread pool in which a fixed number of threads iterates through resources and networks for execution. The primary motivation for the second approach was to support simulations with very high amounts of resources and networks where dynamically managing allocation of hundreds of threads is not practical. All three methods (single-threaded and the two multithreaded approaches) coexist within the framework, and the user is free to choose which execution method is preferred in the *Simulation Editor*.

The organization of the simulation framework is similar in both the single-threaded and multithreaded implementations. The main difference between the two is related to the execution of the resources and networks in the *Simulator* class. As described in Section 6.2, the single-threaded implementation instantiates a single thread (separate from the framework's GUI) where all of the resources and networks are executed serially in a loop, one cycle at a time. Sections 6.4 and 6.5 will discuss how each multithreaded approach is implemented.

6.4 Approach 1: One Thread Per Resource and Network (OTRN)

The OTRN approach allocates one thread for each resource and network instance, which is then free to execute in a loop independently from others. The process of initiating multithreaded execution using the OTRN approach begins by counting the number of threads necessary to execute every instance of every type of resource and network. Next, the threads are allocated and assigned to execute the *Run()* function of its assigned resource or network. A parameter is set in each resource and network instance indicating that the *Run()* function should loop until complete or asked to stop (by user intervention, or due to a completed simulation). When the "Start Sim" button is pressed in the *Simulation Monitor*, each thread is started and the simulation proceeds.

We use the .NET class *Thread* [64] to implement the multithreading in our framework. The operation of the *Thread* class is very simple. A reference to the function to be executed in the new thread is passed to the constructor when it is declared. After a thread is instantiated, it can be started by simply calling the *Start()* method of the new *Thread* instance.

When the simulation is first started (by pressing "Start Sim" in the *Simulation Monitor*), all of the threads are allocated at once. As such, for large amounts of resources and networks (i.e., more than 100), there is a considerable delay caused by initializing all the threads. In the case of a simulation containing over 1000 resources, this execution method may take over 20 minutes on regular desktop PCs before the simulation actually begins to execute.

6.5 Approach 2: Thread Pool

There are several disadvantages with the OTRN approach which became apparent after it was tested with a large number of threads. First, the overhead of context switching is significant. Second, the length of time it took to instantiate the threads was so large that single-threaded execution was the only practical option. Third, thread synchronization (discussed later) caused many of the threads to idle while few threads were actually performing work. To avoid these problems we chose to implement the thread pool approach, which is a balance between single-threaded and the OTRN multithreaded approach.

The thread pool approach uses a fixed number of threads to execute all the resources and networks. With this approach, before the simulation is executed, the thread pool is created and initialized with a fixed number of threads, which is determined by the user in the *Simulation Editor*. The number of threads in the thread pool is typically set to the number of threads that the simulation host can ideally execute in parallel (i.e., the number of available cores, or ideal number of threads in the case of Hyper-Threading technology). Thread initialization is similar to the

OTRN approach (described in Section 6.4) except that all *Thread* classes are set to execute the *RunThreadPool()* function. This new function manages the allocation of tasks to each different thread.

Once the simulation is executed, all of the threads enter the *RunThreadPool()* function concurrently. At the beginning of the function, a critical section iterates through an index indicating which resource or network is ready to be worked on. Each thread executes a resource or network by calling *RunOnce()*, a special version of *Run()* which only executes one cycle. As each thread completes a task, it proceeds to work on the next resource/nextwork until all resources and networks have completed execution in the current cycle. As soon as all of the resources and networks have been completed their one cycle, each thread must stop and synchronize to ensure that all processing in the current cycle is complete before proceeding to the next cycle. Thread synchronization is described in Section 6.6.

Using the thread pool approach, when one resource or network has completed its processing, the current thread can proceed to work on another resource or network within the cycle. There are typically fewer kernel-level threads (in a pool) performing work concurrently, and thus the operating system is not tasked with scheduling as many threads as the OTRN approach; thereby task management time is reduced. Additionally, the simulation framework does not depend on the operating system's thread scheduling to determine which resources or networks require additional processor time since this is managed within the framework.

6.6 Thread Synchronization Using Barriers

Keeping the resources and networks synchronized for enforcing cycle-accuracy (and signal exchange between components) is a primary concern with either multi-threaded implementation. To address this concern, we implemented a barrier after each cycle, which forces the threads to operate in lock-step. Barriers prevent each thread from continuing to execute until all threads have completed their cycle. Figure

6.2 shows a visual representation of four resources and one NoC executing over five threads with barriers (slack, denoted in the figure, will be introduced in Section 6.7).

The barrier was implemented using the .NET class *Semaphore* [65], which was used to create a critical section to count the number of threads that must arrive at the barrier (after completing their cycle). Threads that arrive at the barrier earlier (due to simple operation) are forced to wait until the number of waiting threads equals the number of executing threads, indicating that all threads have completed their cycle. The last thread to arrive at the barrier releases all of the threads, which then go on to process another cycle. This synchronization method ensures that the simulation maintains its cycle-accuracy.

6.7 Slack

Slack is a concept that can be used to trade off simulation accuracy for simulation speed [66]. The amount of slack indicates the number of cycles between synchronizations. A large amount of slack provides a large increase in simulation speed at the cost of introducing a large amount of simulation errors (inaccurate timing of events). However, a small amount of slack can still yield significant speed improvements, while only introducing a small amount of simulation errors. Unfortunately, any simulation

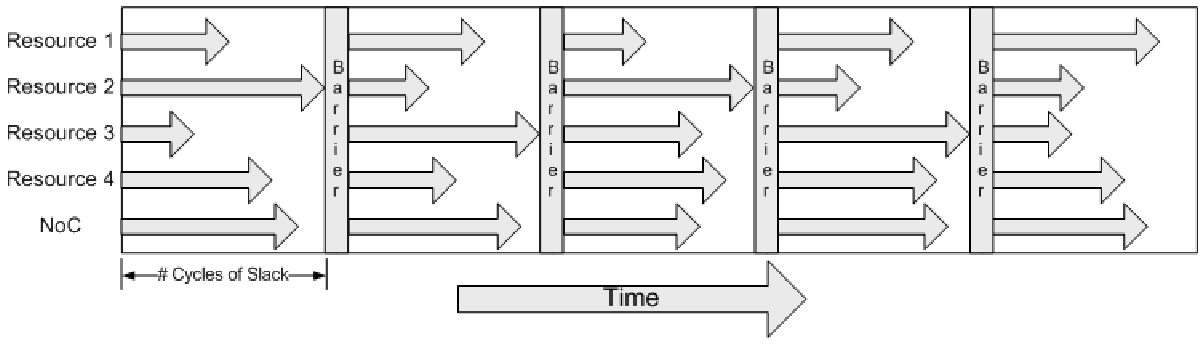


Fig. 6.2. A representation of barriers synchronizing the execution of four resources and a network

error breaks the claim of cycle-accuracy; however, some users may tolerate these timing errors for the benefit of increasing simulation performance. In our multiprocessor implementation, we allow the user to tune their simulation performance by choosing the amount of slack to be used in the simulation.

Slack has the effect of increasing the granularity of resources and networks. For example, if a resource is waiting for a memory operation to complete, it may have several cycles of no operation followed by intensive calculation. If the amount of slack allowed is large enough, the resource could group the waiting period with the calculation before it needs to synchronize again. In this example, the simulation will be able to make much better use of multiple processing cores available on the simulation host.

Slack was implemented in the OTRN approach by including a counter in the generalized *Resource* and *Network* class. When the resource/network is instantiated, the amount of slack is passed to its constructor and stored. A counter is incremented after each execution cycle (of the resource or network) and compared to the desired amount of slack. When the counter reaches the desired amount of slack, the synchronization function, *Sync()*, is called, which is implemented in the *Simulator* class. The *Sync()* prevents the thread from proceeding until all of the other threads have arrived at the barrier.

Slack was implemented in the thread pool approach by executing the *RunOnce()* function in a loop until the number of slack cycles is reached. This has the result of executing resources and networks in groups of cycles, each equal to the amount of slack in the simulator.

Before the simulation starts, the *Simulator* class counts the amount of resources and networks that must be synchronized. As resources complete their execution (or are prematurely stopped by the user), the number of resources and networks that must be synchronized is reduced to ensure that the simulation does not deadlock. In the thread pool approach, the number of synchronizing threads is constant (as specified by the user in the *Simulation Editor*).

6.8 Performance Concerns

Due to the nature of the framework, the performance of the resulting simulation can vary greatly depending upon the simulation configuration. During the development of the framework, every effort was made to keep the simulation overhead to a minimum. In Section 7.6, we show that simulations generated using our framework can be competitive with other major simulators. As the complexity of the user's simulation increases, the performance (simulated instructions executed per second) of the simulation is likely to degrade.

7. EXPERIMENTATION

7.1 Overview

The goal of these experiments is to prove the validity of our framework in producing cycle-accurate discrete event simulators. Five experiments were conducted, each exploring different areas of the framework's functionality. In each experiment, several different simulators were constructed by varying settings within the framework. Then each simulation was executed, and the results of the new settings were observed. Each experiment used a MIPS64 configuration as the basis for the simulation, which is described in Section 7.2.

The experiments were conducted on a computer equipped with a 2.4 GHz Intel Core 2 Quad CPU and 4GB of RAM, running the 64-bit version of Windows Vista. Similar experiments have been conducted on different machines, and the results of the experiments are reproducible across various hardware platforms.

7.2 Description of the MIPS64 Simulator

A MIPS64 simulation configuration was developed during the design of the simulation framework. The configuration was designed to model a five-stage MIPS64 implementation similar to the one shown in Figure 7.1. The configuration has five stages (instruction fetch, instruction decode, execute, memory, and writeback), and includes an additional branch prediction external module. In the NoC experiment (Section 7.5), a network interface module was created that handles the register mapping functionality, which is required to queue and dequeue packets from the network. Communication between stages is also modeled similarly to Figure 7.1. For example,

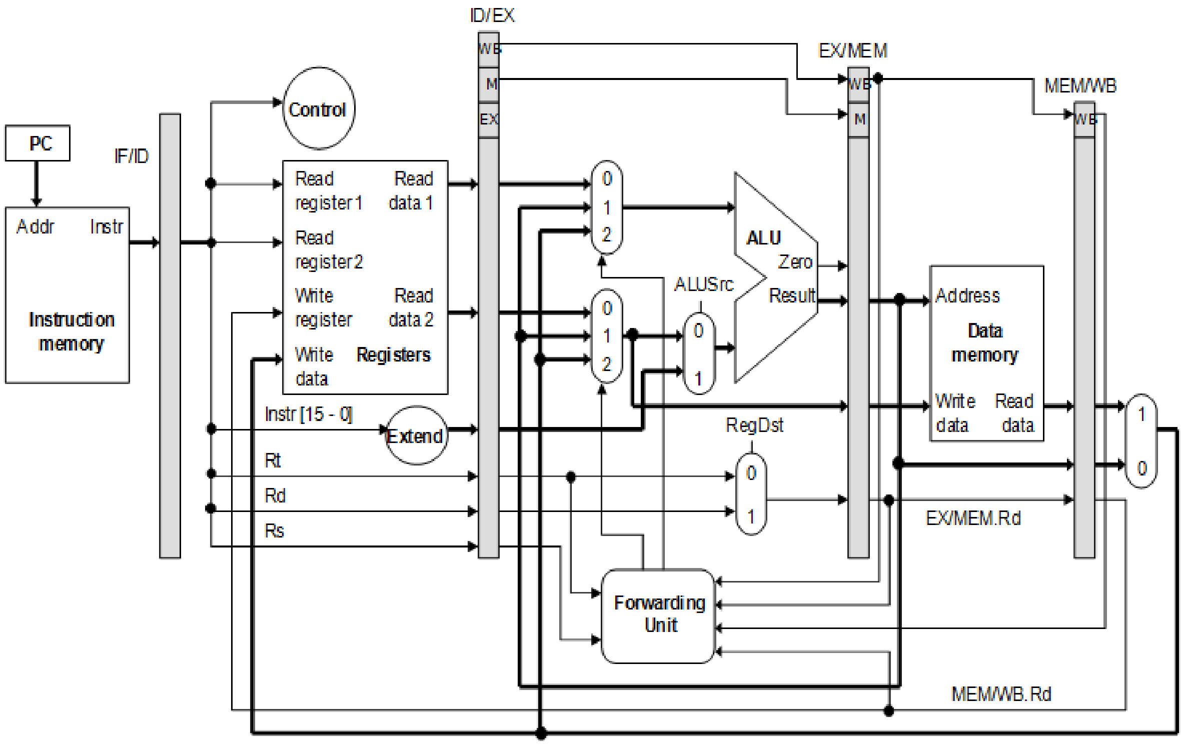


Fig. 7.1. Diagram of a typical MIPS five stage pipeline (Courtesy of D. Patterson and J. Hennessy [67])

register data is read during the instruction decode stage and communicated to the next stage.

The MIPS64 configuration implements all of the control, ALU, and memory functions of the MIPS64 instruction set architecture. The simulator was verified by testing a series of MIPS64 programs and comparing the results to other MIPS64 simulators. The configuration also properly handles data and control hazards, as well as data forwarding.

7.3 Cache Simulation Experiment

The purpose of this experiment was to demonstrate and validate the framework's cache system. One level of 1KB cache was used with three different mapping schemes,

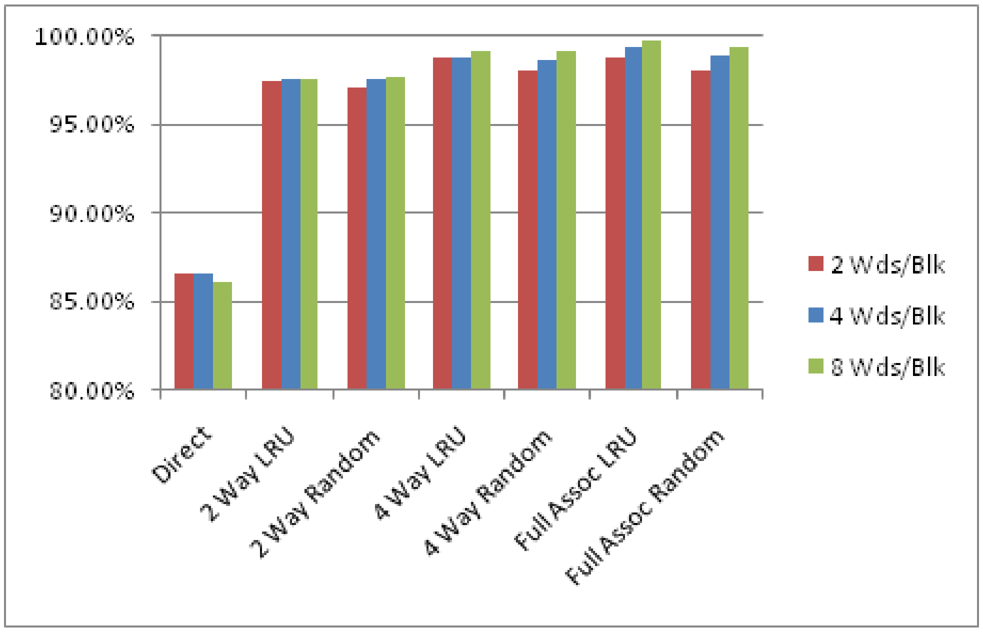


Fig. 7.2. Cache hit rate using different cache schemes and block sizes

direct, set associative, and fully associative. Three different block sizes were used for each test, 2, 4, and 8 words per block. Set associative and fully associative mapping schemes also tested with the Least Recently Used (LRU) and random replacement methods. The small cache size is used because we used a micro-benchmark for this experiment.

This experiment was conducted using a single-processor configuration based on the MIPS64 instruction set architecture. An insertion sorting was performed on 1600 64-bit values, which executed 106,740 instructions that took between 118,620 and 255,060 cycles to complete. The cache accuracy results (shown in Figure 7.2) demonstrate that the cache's performance varies with different configurations and the accuracy responds in a manner that is in line with expectations.

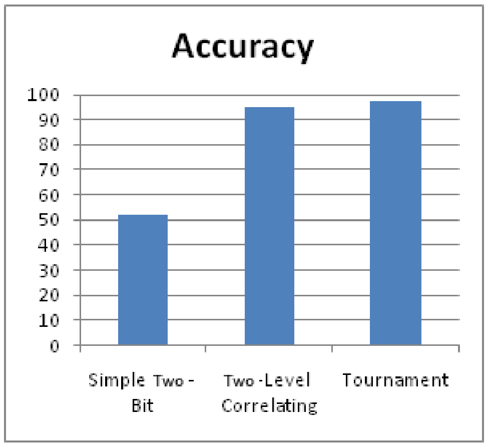
7.4 Branch Prediction Algorithm Comparison Experiment

This experiment was conducted to demonstrate the capability of external modules, and further validate the framework. The framework along with a preconfigured MIPS64 simulation was given to a group of graduate computer architecture students to produce external branch predictor modules. Each student was provided with the source code for a simple two-bit branch predictor and was tasked with creating a two-level correlating predictor and a tournament predictor. The students produced DLL files which were loaded by the framework as the simulator was constructed as described in Section 4.5. The program that tested the branch prediction modules was comprised of many loops and conditional statements in an attempt to emulate program flow that is commonly observed in a typical program, but does not perform any specific function. It was observed that a typical program consists of about 15-20% of branches [67].

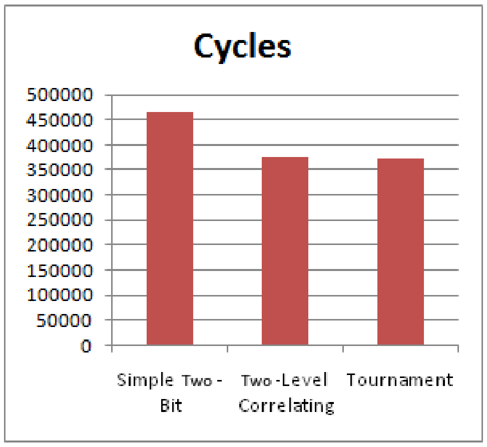
Results across all of the students were similar and in line with expectations. The branch prediction results from one project are shown in Figure 7.3(a) and Figure 7.3(b). Figure 7.3(a) shows the branch prediction accuracy across each branch prediction scheme. As the branch prediction accuracy improves, the number of cycles used to complete the program is reduced, as shown in Figure 7.3(b). The results demonstrate that the external modules are a viable method of integrating functional units into a simulation. Additionally, the nature of the external modules allowed the students to focus only on their portion of modeling and simulation, which provided for an easy-to-use and standardized environment for testing and comparison.

7.5 Network-on-Chip Experiment

This experiment is a brief demonstration of the NoC infrastructure in the framework. The simulation has one master core (resource) that is used to distribute data and aggregate the results of calculations performed on a varying amount of slave cores. At the beginning of the simulation, when ready, each slave core sends a request for data to perform calculations with. The master core responds by sending a packet of data to the slave core, and the master core moves on to the next portion of data. Once the slave core receives the packet, the calculations are performed and the results are transmitted back to the master core, and then the process repeats until all of the calculations have been completed. This is similar to how MPI [68] or PVM [69] processes.



(a) Accuracy results by algorithm.



(b) Number of cycles required per algorithm.

Fig. 7.3. Branch prediction algorithm results

To demonstrate the capabilities of the NoC, we implemented a 2D mesh network topology (detailed in Figures 7.4 and 7.5) and then varied the number of slave cores (from 3 to 511) performing the calculations, and observed the number of cycles needed to aggregate all of the results. In this experiment, each 600 pairs of 64-bit values was transmitted to the slave cores to perform a multiplication calculation. The results are returned to the master core, which aggregates them. For each experiment, the amount of calculations is fixed. The cores interacted with the network through registers mapped to network inputs and outputs.

The first resource instance was assigned to execute the master program and the remaining cores were assigned to execute the slave program. The network grew to the south and east (right and bottom) direction as additional cores were added.

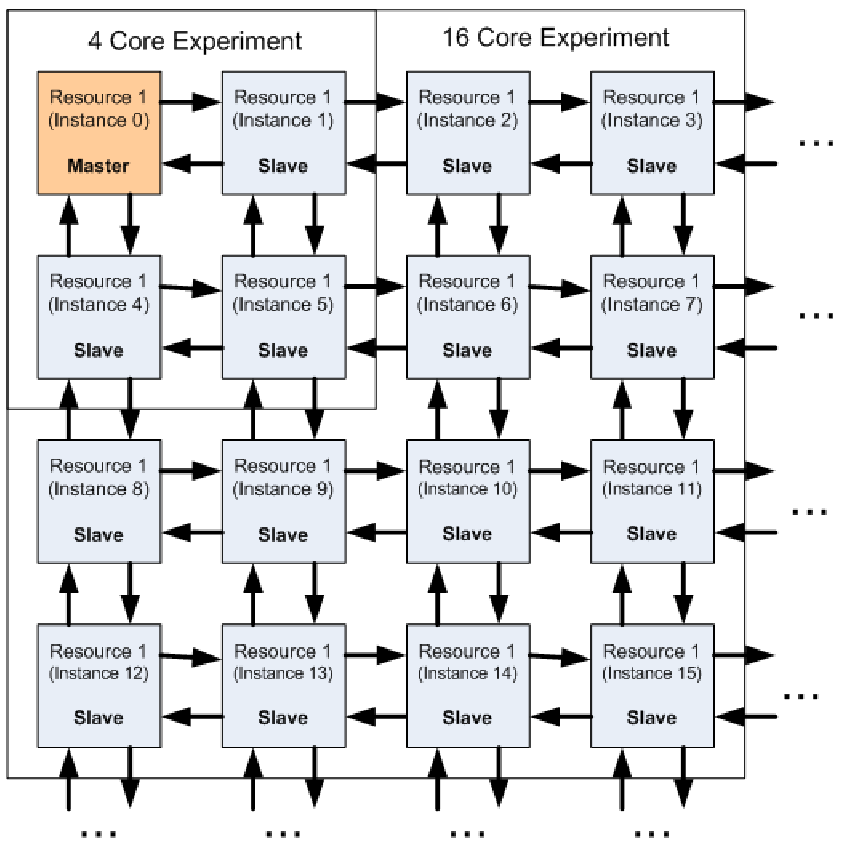

4 Core Experiment
16 Core Experiment
Resource 1 (Instance 0) Master
Resource 1 (Instance 1) Slave
Resource 1 (Instance 2) Slave
Resource 1 (Instance 3) Slave
Resource 1 (Instance 4) Slave
Resource 1 (Instance 5) Slave
Resource 1 (Instance 6) Slave
Resource 1 (Instance 7) Slave
Resource 1 (Instance 8) Slave
Resource 1 (Instance 9) Slave
Resource 1 (Instance 10) Slave
Resource 1 (Instance 11) Slave
Resource 1 (Instance 12) Slave
Resource 1 (Instance 13) Slave
Resource 1 (Instance 14) Slave
Resource 1 (Instance 15) Slave
...

Fig. 7.4. The 2D mesh network topology used in the Network-on-Chip Experiment

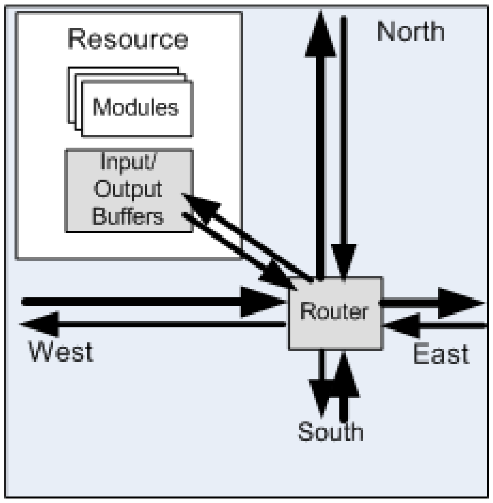


Fig. 7.5. Detail of the connections between routers and resources in the 2D mesh network

The master core stayed in the same position for all experiments for consistency and simplicity. (Positioning the master core in a central location in the network would likely yield improved performance, though this experiment's purpose was to validate the NoC infrastructure, not to demonstrate the NoC's performance.)

Figure 7.6 describes the routing function that was used in the network's routers using pseudo code (for clarity). In this experiment, all of the routers used the same routing function (though this is not a limitation of the framework).

In addition, the number of cycles required to perform the calculation was varied to produce large and small workloads. The large workload required twice the number of cycles to complete the calculation as the small workload. The purpose of collecting the two different sets of results was to observe how the total number of cycles required to produce a result was affected by increasing the runtime of the slave application.

The results (shown in Figure 7.7) demonstrate that as additional slave cores are added, the number of cycles required by the application to complete the calculation is reduced. However, in both data sets, the speedup is diminished as the number of cores increases, due to the network overhead approaching the workload required to perform the calculation. In other words, as the number of cores increases, the number

```
Columns <= 2
Rows <= 2

//ID is Given to The Routing Function
CurrentRow <= Floor( ID / Rows )
CurrentColumn <= ID mod Columns

//Retreive Data From Input Queues
Inputs[0] <= Receive(``north_in'')
Inputs[1] <= Receive(``east_in'')
Inputs[2] <= Receive(``south_in'')
Inputs[3] <= Receive(``west_in'')
Inputs[4] <= Receive(``core_in'')

for i <= 0 to 4
   if Inputs[i] != null then  //null Input means no data in Queue
     Packet <= (UInt64[])Input[i]  //Cast the packet to an array of
                                   // 64 bit unsigned integers
     //First 64 bits of Packet specifies the packet destination
     DestinationRow <= Floor( Packet[0] / Rows )
     DestinationColumn <= Packet[0] mod Columns

     if DestinationRow = CurrentRow then
        if DestinationColumn = CurrentColumn then
          Send(``core_out'', Packet )  //The packet is at the correct
                                       //router, send packet to resource
        else if DestinationColumn > CurrentColumn then
          Send(``east_out'', Packet )  //Send packet east
        else
          Send(``west_out'', Packet )   //Send packet west
     else
        if DestinationRow > CurrentRow then
          Send(``south_out'', Packet ) //Send packet south
        else
          Send(``north_out'', Packet )//Send packet north
```

Fig. 7.6. Pseudocode explaining the routing function used in the 2D mesh network

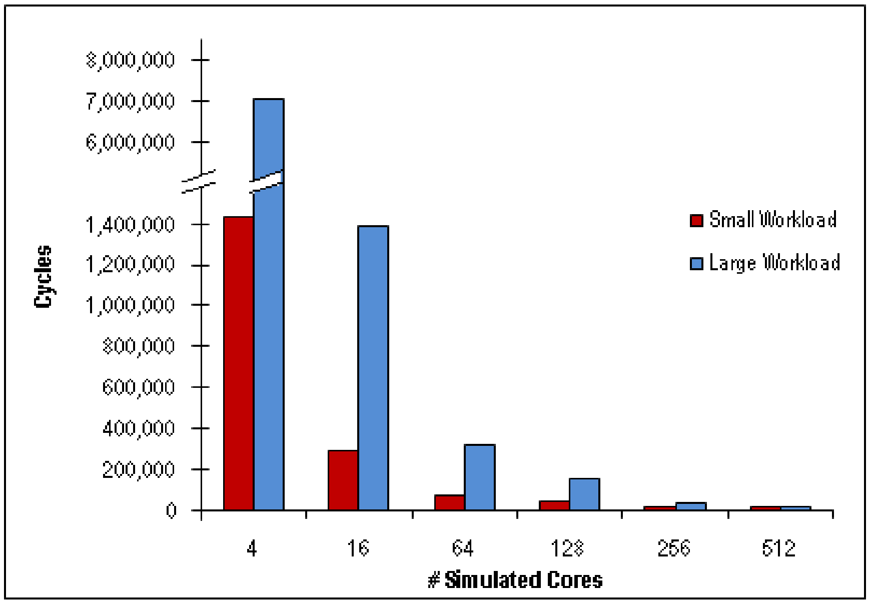


Fig. 7.7. The number of cycles required as the number of cores is varied

of routers that each packet must traverse increases, reducing the benefit of additional cores. With 512 cores, the total execution times for the small and large workloads became nearly identical.

7.6 Single Thread Simulation Performance Experiment

The purpose of this experiment was to examine the performance of a simulator generated by the framework and reveal how a simulator's performance responds when executing many resources in a single thread. The single threaded implementation executes the resources and networks serially, one cycle at a time, and thus the simulation will always be in sync as each resource and network will have executed the same number of cycles. Therefore, the combined performance degradation of increasing simulated cores should be small (in terms of total IPS), and performance per core should degrade proportionally to the number of simulated cores.

Again, we chose the MIPS64 configuration to perform this experiment. The same configuration was utilized for every experiment, with the exception of the number of cores to be simulated. Each core executed an insertion sort application independently; there was no network simulated during this experiment.

The results of the experiment (shown in Figure 7.8) confirmed our hypothesis. As the number of cores increases, the total Instructions-Per-Second (IPS) degrades only slightly. The IPS per core degrades somewhat proportionally to the total number of cores, which is to be expected from a single threaded simulation. Additionally, the simulation performance of the single-core simulation was competitive with other major simulators such as SimpleScalar [16].

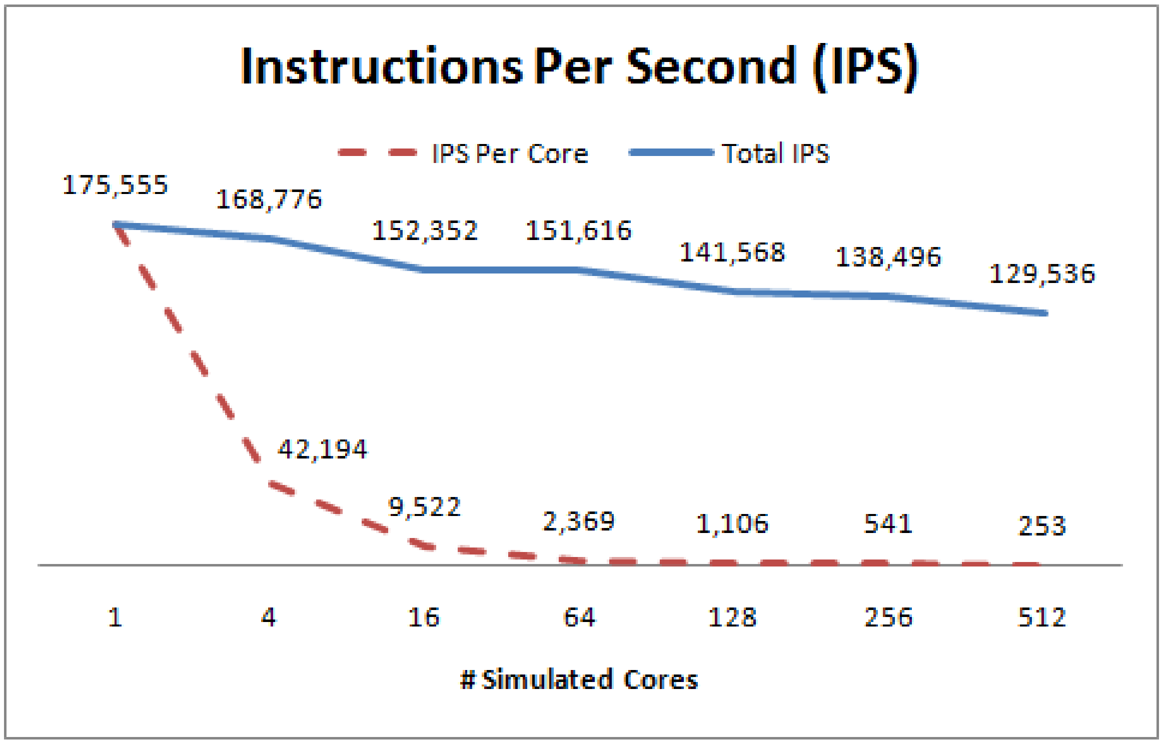


Fig. 7.8. Simulation performance results as the number of concurrent cores executing is increased

7.7 Performance Comparison Between Execution Methods

We performed several experiments to explore the benefits of multithreading with and without slack. Each experiment tested both the OTRN and thread pool multithreading approaches. The thread pool approach was tested with four threads (because the simulation host has four cores), and the OTRN approach used five threads (four resources, and one network-on-chip) in the four simulated core experiments, and seventeen threads in the sixteen simulated core experiments.

Figure 7.9 shows the performance results of a four core MIPS64 configuration with a network, and Figure 7.10 shows a similar experiment with sixteen simulated cores. In this experiment, we can see that the thread pool approach is superior to the OTRN approach in almost all cases. In the case of thread pool approach, the simulation performance exceeded the single threaded performance with three cycles of slack, while the ORTN approach required four cycles of slack.

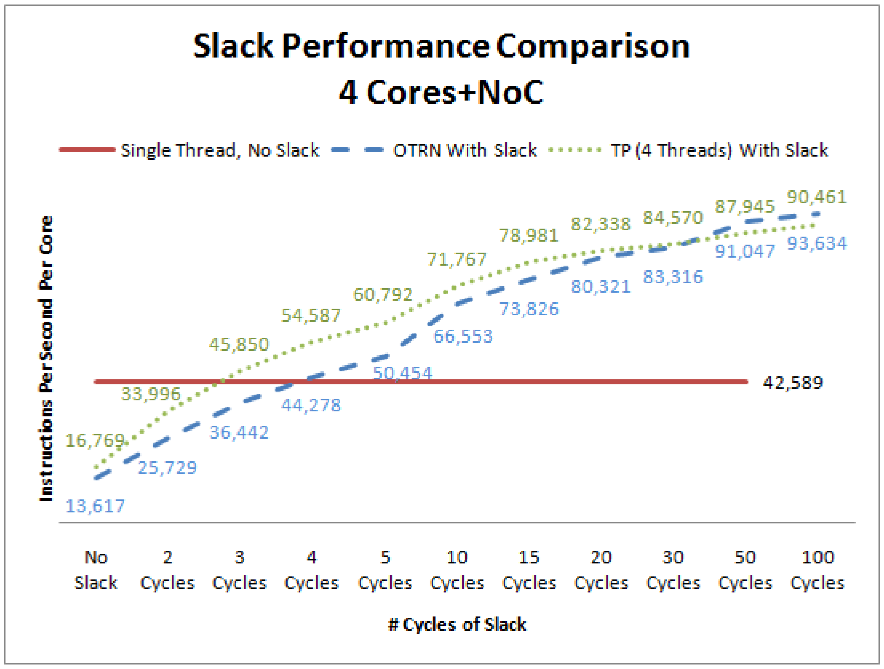


Fig. 7.9. Four core simulation performance comparison between multithreaded with slack and single threaded execution

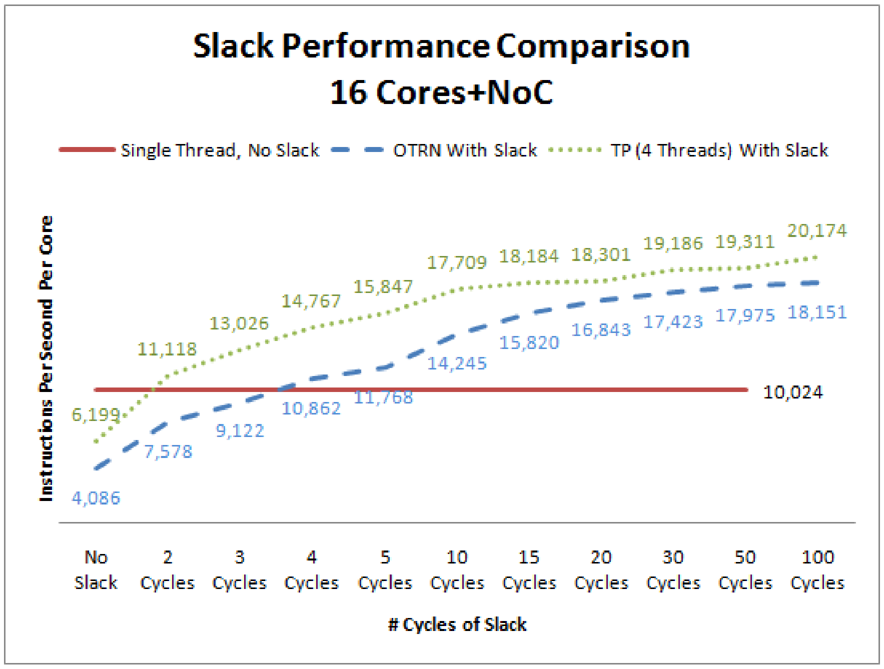


Fig. 7.10. Sixteen core simulation performance comparison between multithreaded with slack and single threaded execution

We believe that this occurred because the MIPS64 configuration that was tested did not require enough processing time to overcome the overhead associated with the context switching and synchronization (i.e., the modeling of components is not complex enough). When the threads are allowed to execute multiple cycles at a time, the benefits of multithreading become more apparent. The framework combines several smaller cycles into one group, making the overhead of context switching and synchronization small relative to the behavioral execution. The performance results show diminishing returns as the amount of slack increases, indicating that the simulator approached the peak performance of the simulation host.

We theorized that multiprocessing would be more beneficial in a situation where each cycle required a more significant amount of time to process. To test this theory, we performed another experiment with the four core (with NoC) program where we added a module to the MIPS64 core which introduced a delay in each cycle. The purpose of the delay was to mimic the additional processing time required in more complex simulations. The delay module was implemented with a simple for-loop that counted up to varying amounts, which we will refer to as the delay loop for this experiment. (A *Sleep* [70] function could not be used as it would deschedule the thread to process other threads, which would cause inconsistent performance and would be less realistic.)

The results, shown in Figure 7.11, confirmed our theory. As the amount of delay (number of times looped in the delay module) increases, the benefit of multithreaded simulation becomes more significant. The thread pool approach showed better performance than single-threaded performance over a delay loop count of 2000, and the OTRN approach improved with a delay count of 4000, even without slack. In other words, in sufficiently complex simulations, multiprocessing will be faster than single threaded execution in every case. Figure 7.12 shows the most extreme case that we tested, which used a 15,000 delay loop count, and utilized slack. With these results, we see that even the 'no slack' case is faster by over 2000 IPS (per core), and the speed improves as the amount of slack increases.

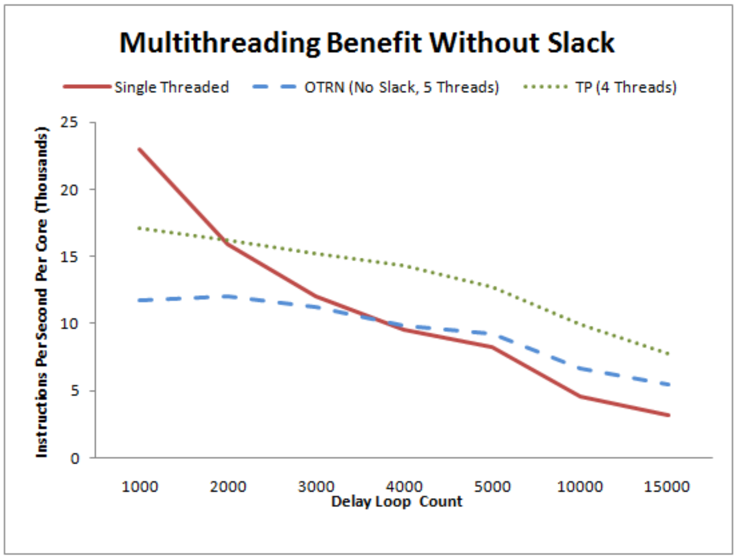


Fig. 7.11. Multithreaded performance comparison by varying the amount of work in the delay module (OTRN uses five threads, TP uses four threads)

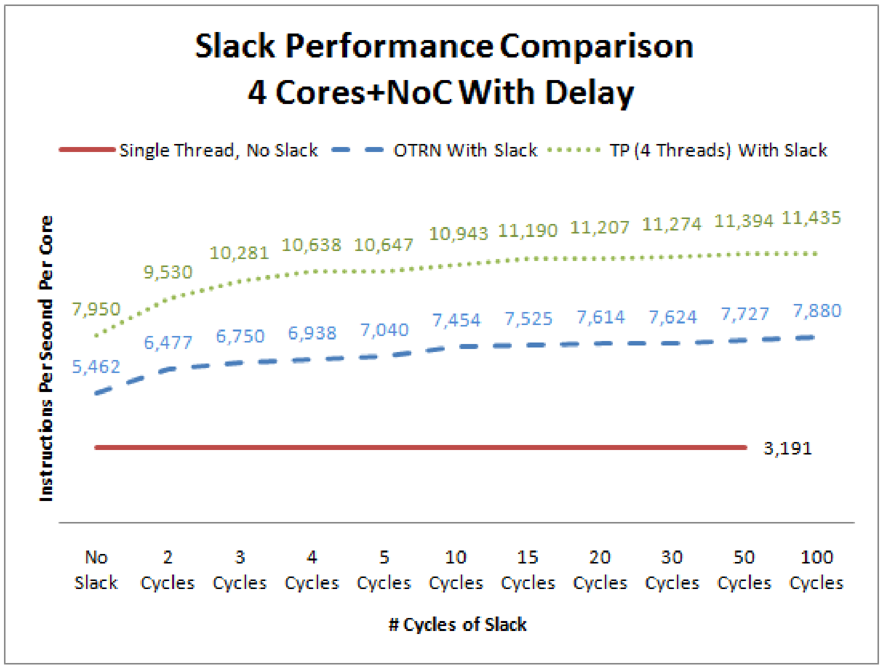


Fig. 7.12. Four core simulation multithreaded performance comparison with large granularity and 15,000 delay loop count (OTRN uses five threads, TP uses four threads)

7.7.1 Evaluation of Simulation Error Introduced With Slack

In this experiment, we evaluated how much simulation error is introduced as we varied the amount of slack in a simulation. We used the four simulated core dot product program with the thread pool multithreaded approach. We observed the difference in the total execution cycles of the slack simulations against the synchronized simulations (i.e., no slack) to produce the percentage of error, and the results are shown in Figure 7.13. The percentage of error will likely vary with the application, though we can observe from the figure that a small amount of slack yields a small amount of error, despite the significant speed increase. Thus, slack is a useful method for increasing multithreaded simulation speed where a small amount of simulation error is acceptable.

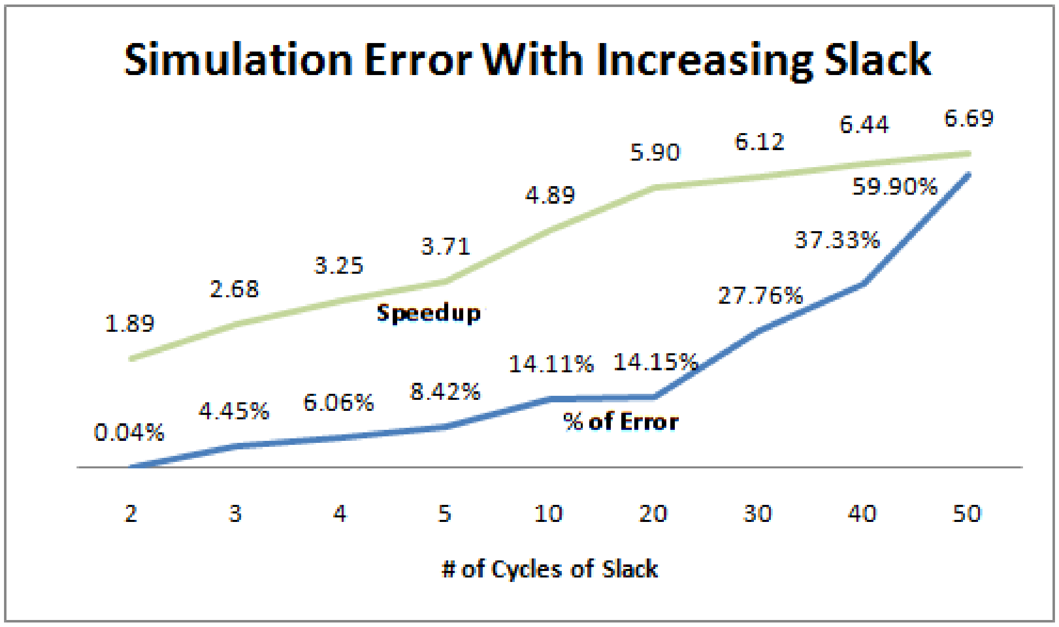


Fig. 7.13. The simulation error introduced as the amount of slack increases

7.7.2 Discussion of Multithreading Approaches

In our performance comparison (Section 7.7), we can see that the thread pool multithreading approach consistently outperforms OTRN. This result occurs because the thread pool approach makes better utilization of the available threads than OTRN. In the OTRN approach, when a resource/network has completed execution, the thread must wait at the barrier until all other threads have arrived. In the case of the thread pool approach, once a thread has completed its work, it proceeds to work on another resource/network to be executed within the same cycle without a context switch. Additionally, there is less overhead from the operating system and synchronization since there are less threads to manage and synchronize. Thus, the thread pool approach is able to utilize the multiple cores better, while minimizing overhead, which yields superior simulation performance.

Another justification for the thread pool approach is its ability to simulate very large amounts of resources. Figure 7.14 demonstrates the performance of many-core configurations using both single threaded and thread pool multithreaded execution methods. This figure shows similar results to Figure 7.12; however in this case we are testing with large amounts of resources. Again we introduced a delay to mimic a more detailed simulation; however in this experiment we only used a 500 delay loop count (half of the smallest delay previously tested). Figure 7.14 demonstrates that when executing with a single thread, the simulation performance is more sensitive to the modeling detail. When executing with a thread pool, the performance was virtually identical with or without the delay module. Additionally, the thread pool performance beat the single thread performance when the delay module was introduced.

With these results, we can see that both single threaded and multithreaded performance have their advantages and disadvantages depending on the simulation host and modeling detail. The simulation framework gives the user the flexibility to make their own decision regarding the total performance and accuracy of their simulations. In many cases multithreaded execution is beneficial with today's multicore hosts, and multithreading will also ensure that the simulation framework will be able to take advantage of additional computing power as the amount of processing cores increase in the future.

Slack provides an interesting method of increasing the amount of processing required per synchronization by introducing some amount of timing error into the simulation. The increased processing time enables the simulator to take better advantage of multiple cores available on the simulation host. Slack may be unnecessary to increase performance with multithreading if the modeling detail of the simulation is great enough to require a significant amount of processing.

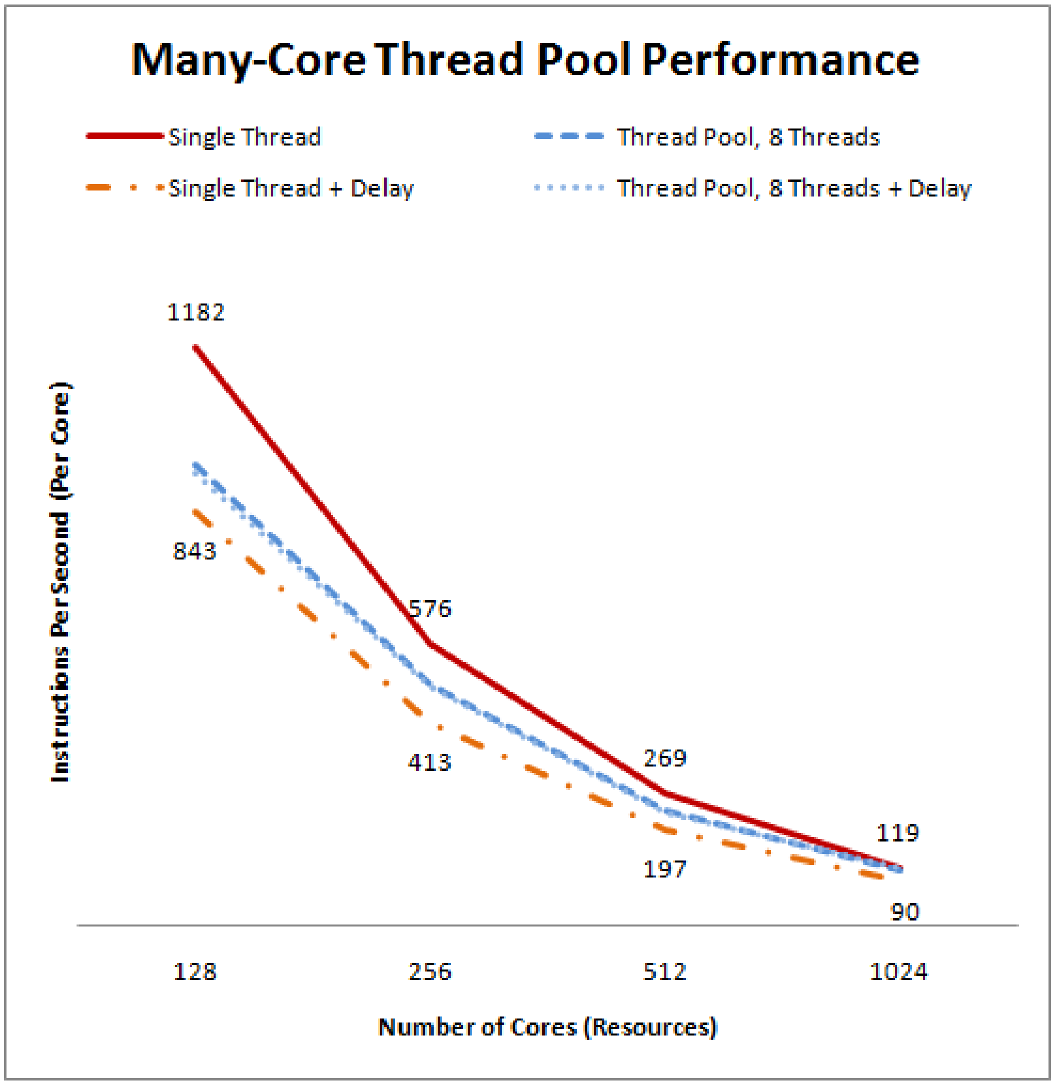


Fig. 7.14. Performance comparison between single-threaded and thread pool multithreaded execution methods with 500 delay loop count

8. SUMMARY AND CONCLUSION

8.1 Summary

In this thesis, we have discussed Mhetero, a simulation framework for dynamically configurable discrete event simulators for many-core heterogeneous Chip Multiprocessors(CMPs). We began by analyzing the previous work in the area of retargetable simulation. Then we discussed the organization of the framework as to how to configure, construct, and execute simulations within a unified interface, and the details behind the framework's implementation. Furthermore, we discussed how we applied our configurability approach to implement a NoC infrastructure into the framework to facilitate communication between resources. We also introduced two approaches to multithreaded simulation, and examined some of the advantages and disadvantages that multithreaded simulation can offer. Finally, we performed several experiments to validate our framework, and showed very useful results that can help simulation designers build, configure and execute their simulations. We envision the framework will be used to further computer architecture research and education.

8.2 Future Work

A natural evolution for this simulation framework is to enhance its GUI with more visual elements, such as resource and network diagrams, and visual performance results. For example, a resource diagram could display a visual representation of the resource's modules and the communication channels between them. Visual performance results of a simulation could also be presented to a user in the form of graphs that show data such as cache hit/misses and instructions per second. The framework could also store previous values, which could be graphed to compare current and previous results. These visualizations would be more stimulating and motivating to usefsrs, and in particular, computer architecture students.

Another possible direction for future development would be to improve the framework's debugging capabilities. One improvement in this area would be to add the ability to pause and step through a simulation to examine the variables in a module's communication channel. This would be very helpful for simulation designers and it would be complementary to the visual diagrams described. The user could use the diagram to inspect values as they progress through the resource/network, which would be useful as for developments as well as educators. Teachers could also use this tool to demonstrate how instructions flow through a processor pipeline and how the simulated processor is affected as events occur.

Since the area of network-on-chip simulation is still in its infancy, it would stand to reason that additional improvements to the NoC infrastructure could help differentiate Mhetero from other simulation frameworks. To this end, the level of detail in network-on-chip simulation could be increased to enable packet timing, power, and bandwidth evaluation. Power consumption measurement could be added to the framework by implementing functions in the *Router* class that would allow simulation designers to tally and log power consumption figures based on usage. Additionally, the *Network* class could also be augmented to record connection duty-cycle (bandwidth) statistics.

Finally, the instruction format information that is collected in the RCE could be used for the automatic construction of assemblers. This type of tool is available with other ADLs such as EXPRESSION [19] and LISA [20], and could be used to speed up the process of developing test programs to execute in simulations. Much of the information required for such a tool is being collected by the framework in its current form, and thus this would be a natural direction for Mhetero in future work.

8.3 Conclusion

The simulation framework discussed in this thesis provides several contributions in an effort to improve discrete event and processor simulation for research and education. The dynamic compilation technique produces fast simulations that compile quickly, with virtually unlimited configurability. The techniques that we described here enable the framework to maintain the same easy-to-use and capable interface throughout the user experience, from simulation configuration to execution. This cohesive and seamless interface provides a tool that is approachable by novice as well as expert users alike. The framework's modular design allows users to easily test new implementations and extend their simulator's functionality. External modules provide a means for simulation designers to extend their simulators and make their work available to others. Additionally, the network-on-chip infrastructure builds on the framework's configurability and compilation capabilities to provide a structured environment for intra-chip communications. Combined, these features create an interesting and powerful simulation platform that provides an exciting computer architecture research and education experience.

LIST OF REFERENCES

LIST OF REFERENCES

[1] R. Kumar, D. Tullsen, and N. Jouppi, “Heterogeneous chip multiprocessors,” *Computer*, vol. 18, Nov. 2005.

[2] H. Meyr, “Heterogeneous mp-soc–the solution to energy-efficient signal processing,” in *Multiprocessor SoC MPSoC Solutions/Nightmare*, Design Automation Conference, 2004.

[3] Intel, “Futuristic intel chip could reshape how computers are built.” http://www.intel.com/pressroom/archive/releases/2009/20091202comp_sm.htm. Visited Jan. 2010.

[4] AMD, “The future is fusion — amd.” http://sites.amd.com/us/fusion/Pages/index.aspx. Visited Jan. 2010.

[5] W. Wolf, “The future of multiprocessor systems-on-chips,” in *Multiprocessor SoC MPSoC Solutions/Nightmare*, Design Automation Conference, 2004.

[6] MSDN, “Compiling to msil.” http://msdn.microsoft.com/en-us/library/c5tkafs1

[7] V. Lee, “A framework for comparing models of computation,” *IEEE Trans. on Computer-Aided Design of Integrated Circuit and Systems*, pp. 1217–1223, Dec. 1998.

[8] M. Yourst, “Ptlsim.” http://www.ptlsim.org/. Visited Jan. 2010.

[9] “M5.” http://www.m5sim.org. Visited Jan. 2010.

[10] “bochs: The open source IA-32 emulation project.” http://bochs.sourceforge.net/. Visited Jan. 2010.

[11] J. Emer, P. Ahuja, and E. Borch, “Asim: A performance model framework,” *Computer*, pp. 68–76, 2002.

[12] “Gxemul.” http://gxemul.sourceforge.net/. Visited Jan. 2010.

[13] P. M. et al., “Simics: A full system simulation platform,” *Computer*, vol. 35, pp. 50–58, 2002.

[14] D. Wallin, H. Zeffer, M. Karlsson, and E. Hagersten, “Vasa: A simulator infrastructure with adjustable fidelity,” *Parallel and Distributed Computing and Systems*, 2005.

[15] M. M. et al., “Multifacets general execution-driven multiprocessor simulator (gems) toolset,” *SIGARCH Computer Architecture News*, pp. 92–99, 2005.

[16] “Simplescalar LLC.” http://www.simplescalar.com/, 2010.

[17] S. M. et al., “Wisconsin wind tunnel ii: A fast and portable parallel architecture simulator,” *Workshop on Performance Analysis and Its Impact on Design*, Jun 1997.

[18] V. Pai, P. Ranganathan, and S. Adve, “Rsim: An execution-driven simulator for ilp-based shared-memory multiprocessors and uniprocessors,” *Third Workshop on Computer Architecture Education*, Feb 1997.

[19] A. H. et al., “Expression: A language for architecture exploration through compiler/simulator retargetability.” http://www.cs.ucr.edu/ vahid/-courses/269_w00/date99_dutt.pdf, 1999.

[20] V. Zivojnovic, S. Pees, and H. Meyr, “Lisa machine description language and generic machine model for hw/sw co-design,” *Proceedings of the IEEE Workshop on VLSI Signal Processing*, Oct. 1996.

[21] M. Freericks, “The nml machine description formalism,” *Fachbereich Informatik*, 1991.

[22] M. Reshadi and N. Dutt, “Generic pipelined processor modeling and high performance cycle-accurate simulator generation,” vol. 2, pp. 786–791, 2005.

[23] C. Barnes, P. Vaidya, and J. Lee, “An xml-based adl framework for automatic generation of multithreaded computer architecture simulators,” *Computer Architecture Letters*, vol. 8, Apr. 2009.

[24] N. Honarmand, H. Sohofi, M. Abbaspour, and Z. Navabi, “Processor description in apdl for design space exploration of embedded processors,” *Proc. EWDTS*, 2007.

[25] “Rexsim: A retargetable framework for instruction-set architecture simulation.” http://www.cecs.uci.edu/technical_report/TR03-05.pdf. Visited Jan. 2010.

[26] G. H. et al., “Isdl: An instruction set description language for retargetability,” *In Proc. Design Automation Conference*, pp. 299–302, 1997.

[27] Y. S. Chandra and T. The, “Retargetable functional simulator,” 1999.

[28] P. Dickens, M. Haines, P. Mehrotra, and D. Nicol, “Towards a thread-based parallel direct execution simulator,” 1996.

[29] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, “Garnet: A detailed on-chip network model inside a full-system simulator,” *In Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[30] “Noxim - NoC simulator.” http://noxim.sourceforge.net/. Visited Jan. 2010.

[31] M. Palesi, S. Kumar, and R. Holsmark, “A method for router table compression for application specific routing in mesh topology noc architectures,” *SAMOS VI Workshop: Embedded Computer Systems: Architectures, Modeling, and Simulation*, pp. 373–384, 2006.

[32] G. Ascia, V. Catania, M. Palesi, and D. Patti, “A new selection policy for adaptive routing in network on chip,” in *International Conference on Electronics, Hardware, Wireless and Optical Communications*, 2006.

[33] V. Puente, J. Gregorio, and R. Beivide, "Sicosys: An integrated framework for studying interconnection network performance in multiprocessor systems," *In. Proc. IEEE 10th Euromicro Workshop on Parallel and Distributed Processing*, Jan. 2002.

[34] J. Jump, "Yacsim reference manual," *Rice University, Electrical and Computer Engineering Department*, Mar. 1993.

[35] E. G. et al, "Elements of reusable object-oriented software," *Addison-Wesley Professional Computing Series*, 1995.

[36] M. Chidester and A. George, "Parallel simulation of chip-multiprocessor architectures." http://www.hcs.ufl.edu/pubs/PARSIM2002.pdf, 2002.

[37] L. Eeckhout and K. Bosschere, "Efficient simulation of trace samples on parallel machines," *Parallel Computing*, vol. 30, no. 3, pp. 317–335, 2004.

[38] A. N. et al., "Accuracy and speed-up of parallel trace-driven architectural simulation," 1997.

[39] G. Lauterbach, "Accelerating architectural simulation by parallel execution of trace samples," 1993.

[40] MSDN, "Xmldocument class (system.xml)." http://msdn.microsoft.com/en-us/library/system.xml.xmldocument.aspx. Visited Jan. 2010.

[41] MSDN, "Xmlnode class (system.xml)." http://msdn.microsoft.com/en-us/library/system.xml.xmlnode.aspx. Visited Jan. 2010.

[42] MSDN, "Xmltextwriter class (system.xml)." http://msdn.microsoft.com/en-us/library/system.xml.xmltextwriter.aspx. Visited Jan. 2010.

[43] MSDN, "Httputility class (system.web)." http://msdn.microsoft.com/en-us/library/system.web.httputility.aspx. Visited Jan. 2010.

[44] MSDN, "switch (c# reference)." http://msdn.microsoft.com/en-us/library/06tc147t.aspx. Visited Jan. 2010.

[45] MSDN, "Command-line building with csc.exe." http://msdn.microsoft.com/en-us/library/78f4aasd.aspx. Visited Jan. 2010.

[46] MSDN, "Csharpcodeprovider class." http://msdn.microsoft.com/en-us/library/microsoft.csharp.csharpcodeprovider(VS.85).aspx. Visited Jan. 2010.

[47] MSDN, "Codedomprovider compileassemblyfromsource method." http://msdn.microsoft.com/en-us/library/system.codedom. compiler.codedomprovider. compileassemblyfromsource.aspx. Visited Jan. 2010.

[48] MSDN, "Assembly class (system.reflection)." http://msdn.microsoft.com/en-us/library/system.reflection.assembly.aspx. Visited Jan. 2010.

[49] MSDN, "Compilerparameters class (system.codedom.compiler)." http://msdn.microsoft.com/en-us/library/system.codedom.compiler. compilerparameters.aspx. Visited Jan. 2010.

[50] MSDN, "List(t) class (system.collections.generic)." http://msdn.microsoft.com/en-us/library/6sh2ey19.aspx. Visited Jan. 2010.

[51] MSDN, "interface (c# reference)." http://msdn.microsoft.com/en-us/library/87d83y5b.aspx. Visited Jan. 2010.

[52] MSDN, "Dynamic link libraries." http://msdn.microsoft.com/en-us/library/ms682589.aspx. Visited Jan. 2010.

[53] MSDN, "Compilerparameters referencedassemblies property (system.codedom.compiler)." http://msdn.microsoft.com/en-us/library/system.codedom.compiler.compilerparameters.referencedassemblies.aspx. Visited Jan. 2010.

[54] "string (c# reference)." http://msdn.microsoft.com/en-us/library/362314fe.aspx. Visited Jan. 2010.

[55] "Microsoft visual studio 2008." http://www.microsoft.com/visualstudio. Visited Jan. 2010.

[56] "How to: Debug dlls." http://msdn.microsoft.com/en-us/library/c91k1xcf Visited Jan. 2010.

[57] W. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," *In Proc. Design Automation Conference*, pp. 684–689, 2001.

[58] "Arteris." http://www.arteris.com/. visisted Jan. 2010.

[59] "Sonics, inc.." http://www.sonicsinc.com/. visisted Jan. 2010.

[60] "Tilera corporation." http://www.tilera.com/products/TILE-Gx.php. Visited Jan. 2010.

[61] W. Dally and J. Poulton, *Digital Systems Engineering*. Cambridge University Press, 1998.

[62] MSDN, "Queue class (system.collections.generic)." http://msdn.microsoft.com/en-us/library/7977ey2c.aspx. Visited Jan. 2010.

[63] MSDN, "try-catch (c# reference)." http://msdn.microsoft.com/en-us/library/0yd65esw.aspx. Visited Jan. 2010.

[64] MSDN, "Thread class (system.threading)." http://msdn.microsoft.com/en-us/library/system.threading.thread.aspx. Visited Jan. 2010.

[65] MSDN, "Semaphore class (system.threading)." http://msdn.microsoft.com/en-us/library/system.threading.semaphore.aspx. Visited Jan. 2010.

[66] J. Chen, M. Annavaram, and M. Dubois, "Slacksim: A platform for parallel simulations of cmps on cmps." http://ceng.usc.edu/assets/001/60870.pdf, Aug. 2008.

[67] D. Patterson and J. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2004.

[68] A. Gabriel, A. Fagg, and A. Bosilca, “Open MPI: Goals, concept, and design of a next generation MPI implementation,” in *Proceedings, 11th European PVM/MPI Users’ Group Meeting*, (Budapest, Hungary), pp. 97–104, September 2004.

[69] V. Sunderam, “Pvm: A framework for parallel distributed computing,” *Concurrency: Practice and Experience*, pp. 315–339, Dec. 1990.

[70] MSDN, “Thread.sleep method (int32) (system.threading).” http://msdn.microsoft.com/en-us/library/d00bd51t.aspx. Visited Jan. 2010.