

We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800

Open access books available

122,000

International authors and editors

135M

Downloads

Our authors are among the

154

Countries delivered to

TOP 1%

most cited scientists

12.2%

Contributors from top 500 universities



WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index
in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com



A dynamically configurable discrete event simulation framework for many-core chip multiprocessors

Christopher Barnes and Jaehwan John Lee
Indiana University Purdue University Indianapolis
U.S.A.

1. Introduction

1.1 Background

Processor simulation is often a cornerstone in the research of new processor concepts and the education of computer architecture students. Simulators are used by researchers to validate architecture designs and explore new concepts before actual implementation. 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, easy to understand, and fast.

Simulation speed and configurability are two important aspects in the design of processor simulators. In the past, fast simulations were typically made with 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 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 its functionality. This approach is easier but still requires the user to undergo a lengthy learning curve to begin generating useful results.

As the industry moves toward merging many different, highly specialized processor resources on one physical chip, there is a need for a highly configurable discrete event simulation environment for the study of heterogeneous processor designs. Introduced in this chapter is Mhetero, a novel simulation framework that enables users to easily construct and perform discrete event simulations that meet this need.

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 library in two ways. First, we use dynamic compilation to produce simulations based on configuration information gathered through an easy-to-use GUI. The entire process is a seamless and user-friendly experience, meaning that the user does not

leave the framework to execute external compilers, write source code, or edit configuration files. Second, the simulations produced by the framework are compiled to an intermediate language (Compiling to MSIL, 2010), resulting in quick compilation time as well as execution speeds matching that of other compiled .NET programs. While the overall performance of C# does not match that of C++, there are numerous advantages of utilizing C# for scientific computing (Gilani, 2004), which are leveraged in our framework. 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 tasks associated with a typical processor simulator. Although here we describe the techniques used in the design of Mhetero, the techniques are also applicable to other types of discrete event simulators.

Mhetero's simulation infrastructure is similar to other discrete event/time simulators with a few notable differences that facilitate processor simulation. First, instead of using a single, global event queue, Mhetero maintains several, separate event queues each modeling a communication channel between any two entities/modules of simulation. Second, instead of activating modules when certain events occur, entities/modules are activated during each 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 (Lee & Vincentelli, 1998). Hence, Mhetero's simulation infrastructure can be categorized as a synchronous, discrete time-simulation infrastructure which by definition itself is a discrete event simulation infrastructure (Lee & Vincentelli, 1998). 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, educators, and students.

In this chapter, we will discuss the design and construction of our simulation framework. We will begin by reviewing some of the previous work in the area of computer architecture simulation. We then discuss our configuration interface (Sections 2 and 4), dynamic compilation technique (Section 3), and intra-resource communication (Section 4). Finally, we will discuss several experiments that were conducted to verify the framework's design (Section 5).

1.2 Previous Work

Over the past several decades a considerable amount of research has been done in the area of computer architecture simulation. SimpleScalar (SimpleScalar LLC., 2010) and its variations have been used mostly for single processor simulation and research while the SimpleScalar multiprocessor version (Univ. of Minnesota, 2010), GEMS (Martin, et al, 2005), RSIM (Pai, et al, 1997), VASA (Wallin, et al, 2005), and WWT-II (Mukherjee, et al, 1997) (as well as its earlier versions) have been used mainly for multicore or chip-multiprocessor (CMP) simulation. While these simulators are very fast, they are not intended to produce retargetable simulations; i.e., these simulators are monolithic and cannot simulate other architectures beyond the originally intended architecture. Other simulators such as Simics (Magnusson, et al., 2002), Bochs (Bochs, 2010), and GxEmul (GXEmul, 2010) are full-system simulators for both single and multiprocessor simulation. These simulators are typically

used for the development and testing of software on various platforms, and are also not designed to be easily retargetable.

A previous approach to retargetable simulators is investigated through the use of computer Architecture Description Languages (ADLs) such as Expression (Halambi, et al, 1999), LISA (Zivojnovic, et al, 1996), nML (Freericks, 1991), and RCPN (Reshadi & Dutt, 2005). These tools have been proposed primarily for automatic generation of computer architecture simulators. Although these tools produce retargetable simulators, their respective ADLs can often be difficult for new users to learn. Additionally, the generation of simulators is typically a disjointed and error-prone process that depends on external compilers and programs to function.

Asim (Emer, et al., 2002), a framework for modeling the performance of a processor (e.g., timing delays and signal propagation delays), most closely resembles Mhetero as it is a retargetable simulation framework that segments functional units into modules and includes two graphical tools for generating and viewing configuration files. However, Asim includes a separate controller program used to execute simulations. On the contrary, Mhetero builds on the concept of using a single unified GUI for both configuration and simulation, creating a seamless environment. This approach, enabled by the techniques described in this chapter, allows the user to focus on developing their simulations without being burdened by the inner workings of the simulator's configuration.

Our simulation framework is built to minimize the difficulties associated with retargetable simulators by providing an easy-to-use GUI intended to offer a minimal learning curve. Additionally, simulators built by our framework are compiled using a technique that is completely concealed from the user, avoiding any compiler configuration concerns. Finally, simulators generated by our framework are capable of being competitive with other major simulators in terms of instructions per second. The performance of the resulting simulations is addressed in Sections 3.7 and 5.5.

1.3 Definitions

Before we proceed with the explanation of our simulation framework, we will take a moment to explain some of the terminology used throughout this chapter.

Resources represent 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. Note that the network is treated separately from a resource by the framework, and is explained in detail in Section 4.

Modules represent functional units within a resource, such as processor stages, branch predictors, and forwarding units.

Simulation designer is the user who is using the simulation framework for the purpose of producing or revising a processor simulator.

Configurability refers to the process of creating or customizing a new simulator by changing the settings (e.g., cache configuration) and source code of modules, resources, and routers in the simulation configuration.

2. Resource Configuration Interface

2.1 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 can 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 1 depicts the organization of the editors for the design and configuration of simulations.

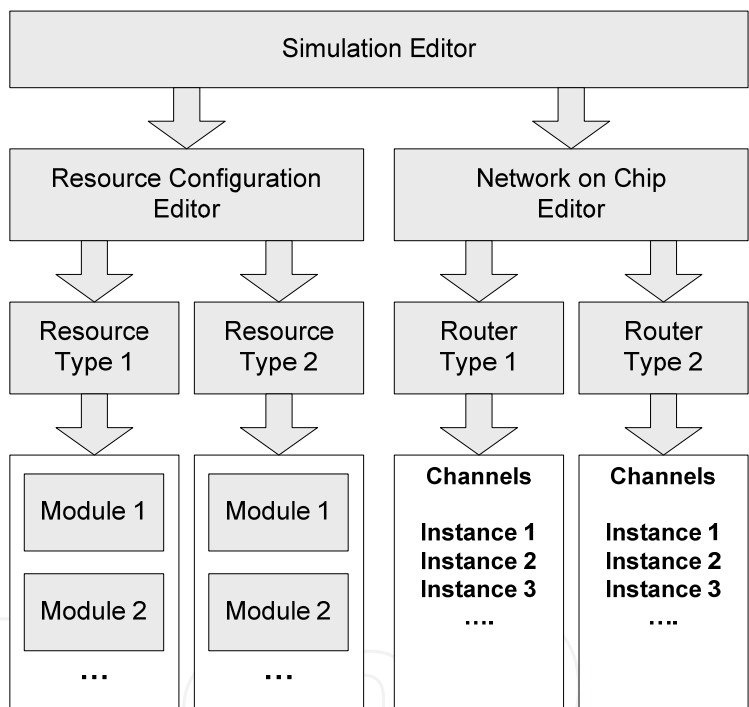


Fig. 1. Organization of editors within the simulation framework.

2.2 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 layer 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. The remainder of Section 2 details the Resource Configuration Editor, and the NoC Configuration Editor is discussed in Section 4.

2.3 Resource Configuration Editor (RCE)

The *Resource Configuration Editor (RCE)* is the central location for editing the function or structure 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 2 shows the RCE interface. Several of the more simple tabs are discussed in this subsection, and the remaining tabs are described in Sections 2.4 – 2.7.

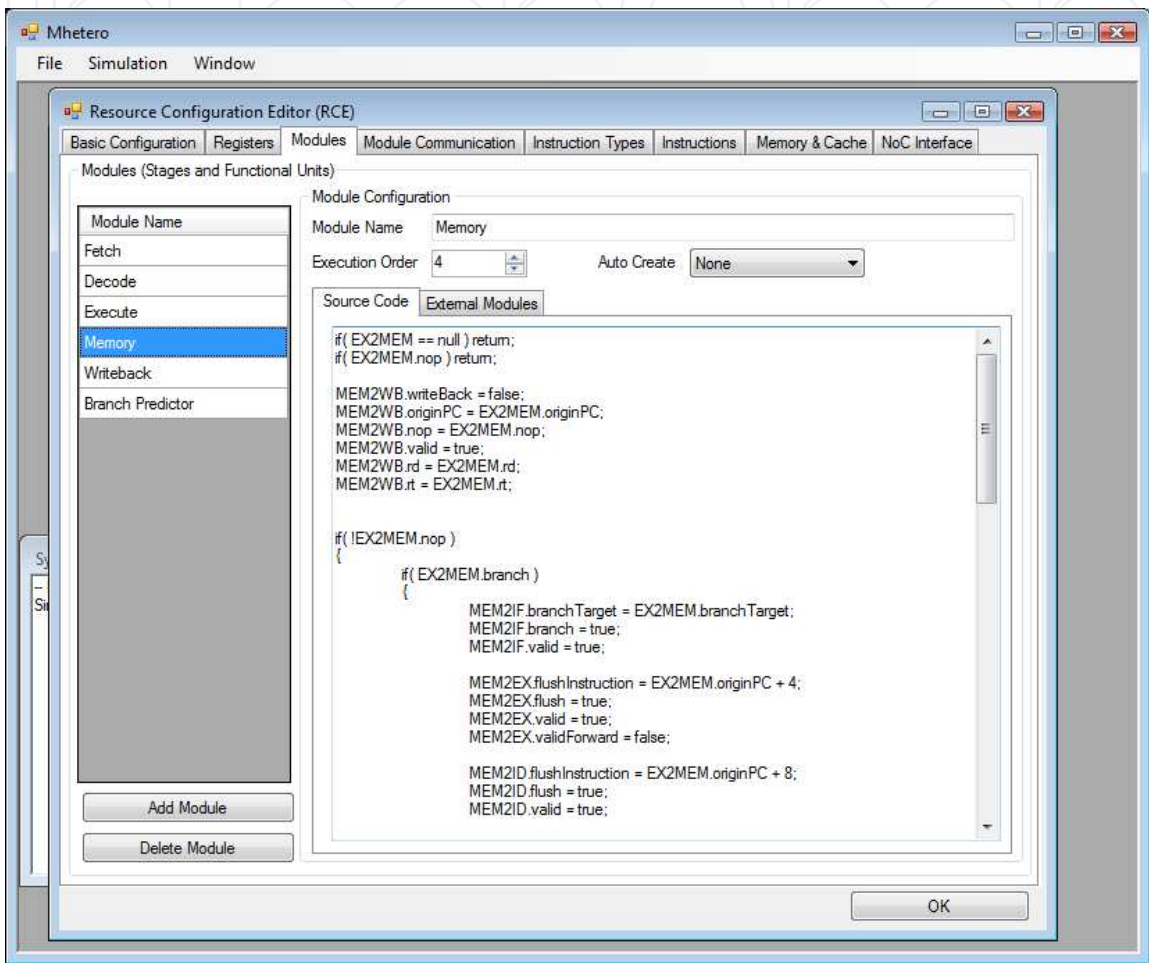


Fig. 2. A screenshot of the RCE Interface.

The *Basic Configuration* tab contains fields for the name of the resource type, the 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 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 provides an interface for the user to specify the register names, number of registers, and data types. The *Instruction Types* tab allows the user to specify the instruction format which is 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.

2.4 Module Editor

Modules are a core concept to the extendibility and configurability of the framework. A set of modules forms a resource. 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 facilitates 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 2) allows the user to input the module's name, execution precedence (i.e., order), 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 in the resource. Details on how external modules are linked to the resource are given in Section 3.5.

2.5 Module Communication

Under the *Module Communication* tab, 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 are combined into 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 as produce valid outputs, if necessary. The management of communication data between modules is handled by the framework through the use of *Queues* (MSVC Dev. Center, 2010).

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.

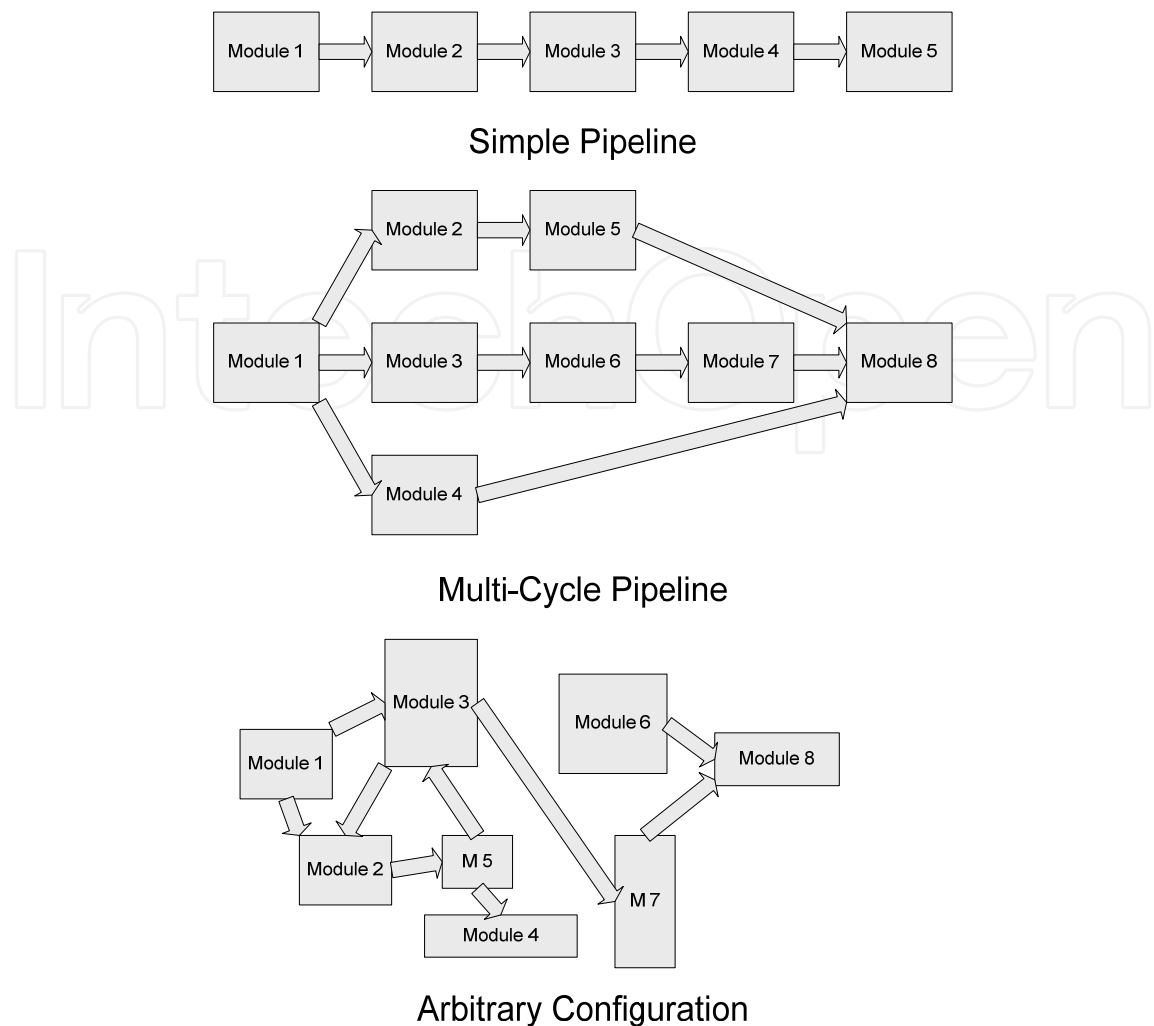


Fig. 3. Potential pipeline configurations.

2.6 Instructions

The *Instructions* tab 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. If desired, the instruction source code may be used to automatically generate execution stage source code during compilation (detailed in Section 3.3).

2.7 Memory and Cache

The *Memory & Cache* tab enables the user to specify the size and type of the data and instruction memory as shown in Figure 4. 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.

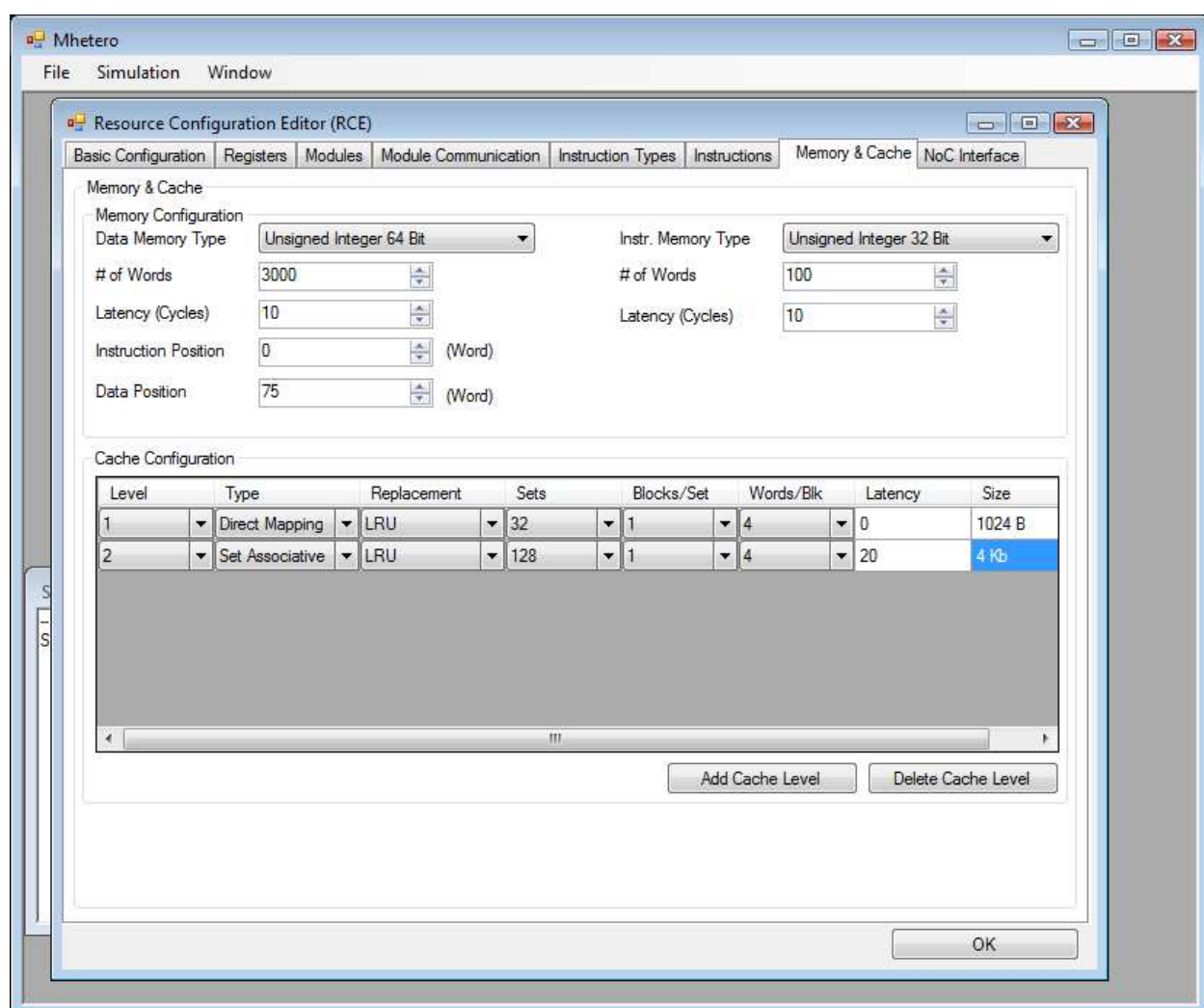


Fig. 4. A screenshot of the *Memory & Cache* tab in the RCE.

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. For example, this would be useful for emulating a shared cache/memory. In this case, the simulation designer must implement a network and its protocol to access a resource modeling a memory module. Details about intra-core networks are explained in Section 4.

2.8 Simulator Configuration Data File

Information regarding the simulator's configuration is loaded and saved in an XML format utilizing the .NET Document Object Model (DOM) XML classes *XmlDocument*, *XmlNode*, and *XmlTextWriter* (MSVC Dev. Center, 2010). 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 XML file, and inserts its settings.

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 that must be instantiated and loaded. *Simulator* instantiates the appropriate number of resources and networks, and then invokes the *LoadConfiguration()* function. The *LoadConfiguration()* function is sent a reference to the appropriate portion of the XML tree to load as an *XmlNode*, which it uses to read 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* (MSVC Dev. Center, 2010) 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 organization of configuration data allows the framework to store and load entire simulation configurations, including multiple heterogeneous cores and networks, into a single file.

3. Dynamic Compilation

3.1 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 MSIL (Compiling to MSIL, 2010). 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. Thus, 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 structure the framework to support this behavior, how the dynamic compilation is implemented, and how the framework communicates with the newly generated simulator executed inside the framework.

3.2 Framework Structure

Two classes, *Simulator* and *Network*, make up the core of the dynamic compilation implementation. Figure 5 shows the organization of these two classes within the framework. The simulation executes in a different thread (referred to as “Simulation Thread” in Figure 5) from a thread of the framework and its GUI (together referred to as “Framework Thread”). The *Simulator* class was constructed to interface the simulation framework to the chip’s resources and networks. *Simulator* 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. *Network* handles the router compilation process, which is initiated after *Simulator* has compiled all of the resources. Since *Network* must execute during the simulation, it is executed in the simulation thread, similar to other resources, instead of the framework thread (details about the simulation execution are provided in Section 3.6).

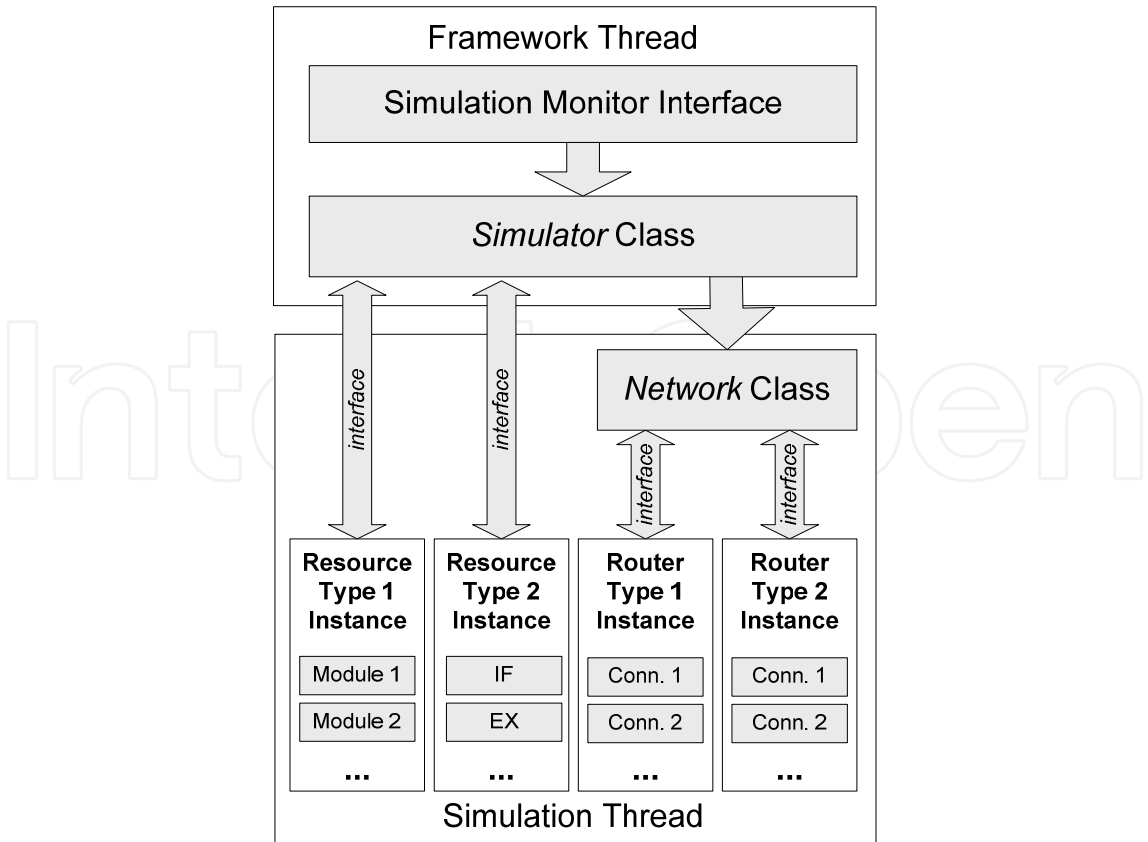


Fig. 5. Organization of the framework structure and communication interface.

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. 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. Additionally, some NoC implementations may include a memory system modeled as a chip resource, so networks for accessing memory may also be necessary.

3.3 Implementation of Dynamic Compilation

Before the compilation process can begin, the source code of the simulator must be gathered by the framework. Figure 6 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 a new class that implements the behavior of the resource. 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 they are 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 every instruction's source code into a *switch* 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 instruction source code to execute.

Once the simulator source code has been pieced together, the program is compiled. The compilation utilizes the C# Compiler (CSC.exe) 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.

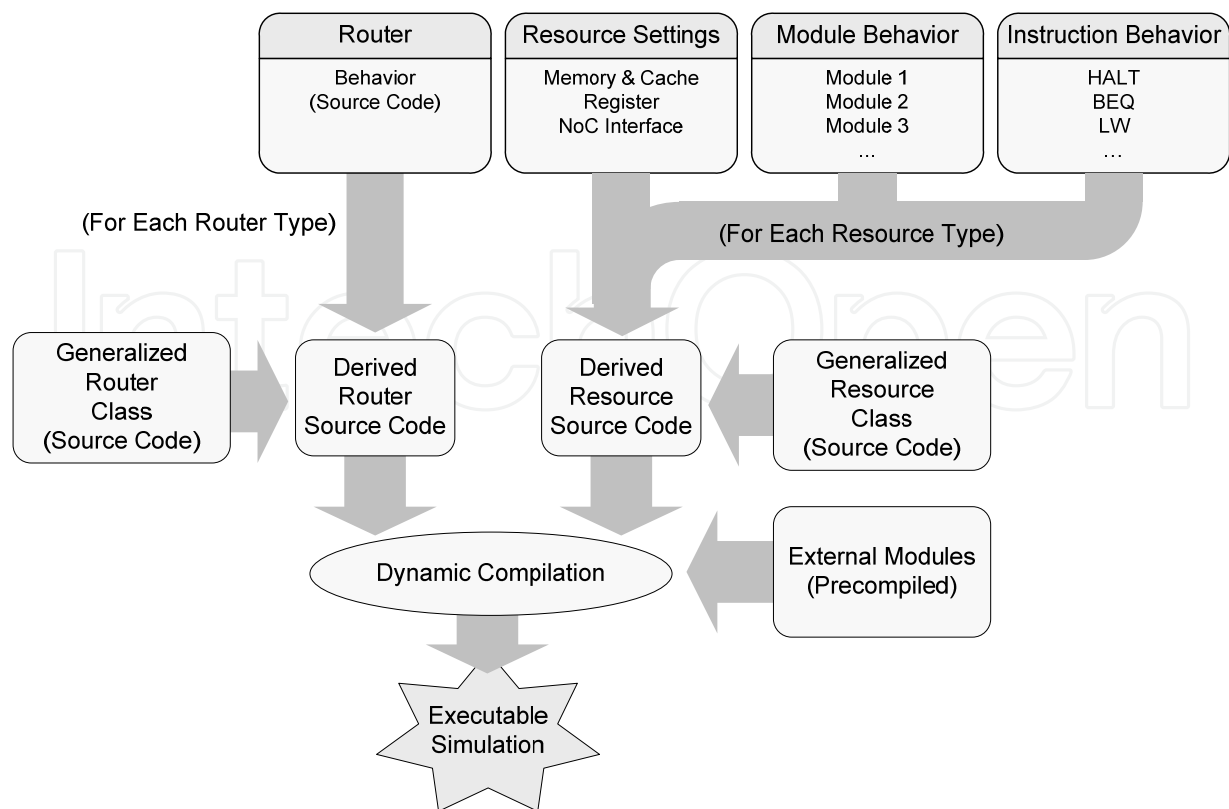


Fig. 6. Flow chart of the dynamic compilation process.

The execution of the C# compiler is managed by the *CSharpCodeProvider* class (MSVC Dev. Center, 2010). We use the *CompileAssemblyFromSource()* function included in the *CSharpCodeProvider* class to produce an *Assembly* (MSVC Dev. Center, 2010) which represents the compiled code. *CompileAssemblyFromSource()* takes two parameters, the source code and *CompilerParameters*. *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()* returns compilation results which provide a reference to a compiled *Assembly* if the compilation was successful or a list of error messages (i.e., module or router source code compilation errors). The compiled *Assembly* data structures are stored in a *List* and used for instantiating the new resource and router classes.

3.4 Communication Between the Framework and Simulator Components

Communication between the framework and the resources and routers is facilitated by the *interface* capability which is provided in C#, as well as other object oriented languages. *Interface* enables developers to generalize the signature of function calls which may be included into a compiled *Assembly*, the result of the dynamic compilation process (discussed in Section 3.3). 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 6) implement standard 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. Communication between resources and routers is discussed in Section 4.

3.5 External Modules

External modules are precompiled Dynamic-Link Library (DLL) files containing a class that implements the functionality of a module. During the compilation process (described in Section 3.3), any external modules specified in a resource are loaded and linked into the compiled code. This is accomplished by referencing the external module in the *ReferencedAssemblies* property of the *CompilerParameters* class, which 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, 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 variables, and/or additional functions, 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 (in the same resource) 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#.

Although regular (internal) modules provide less flexibility than external modules, they require less expertise to implement since the simulation designer is primarily tasked with developing the module's behavioral source code. Thus, external modules should be considered as a more advanced configuration option.

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 and used within the module, or to be called from other modules.

Figure 7 illustrates how two external modules could be integrated into a resource's execution loop.

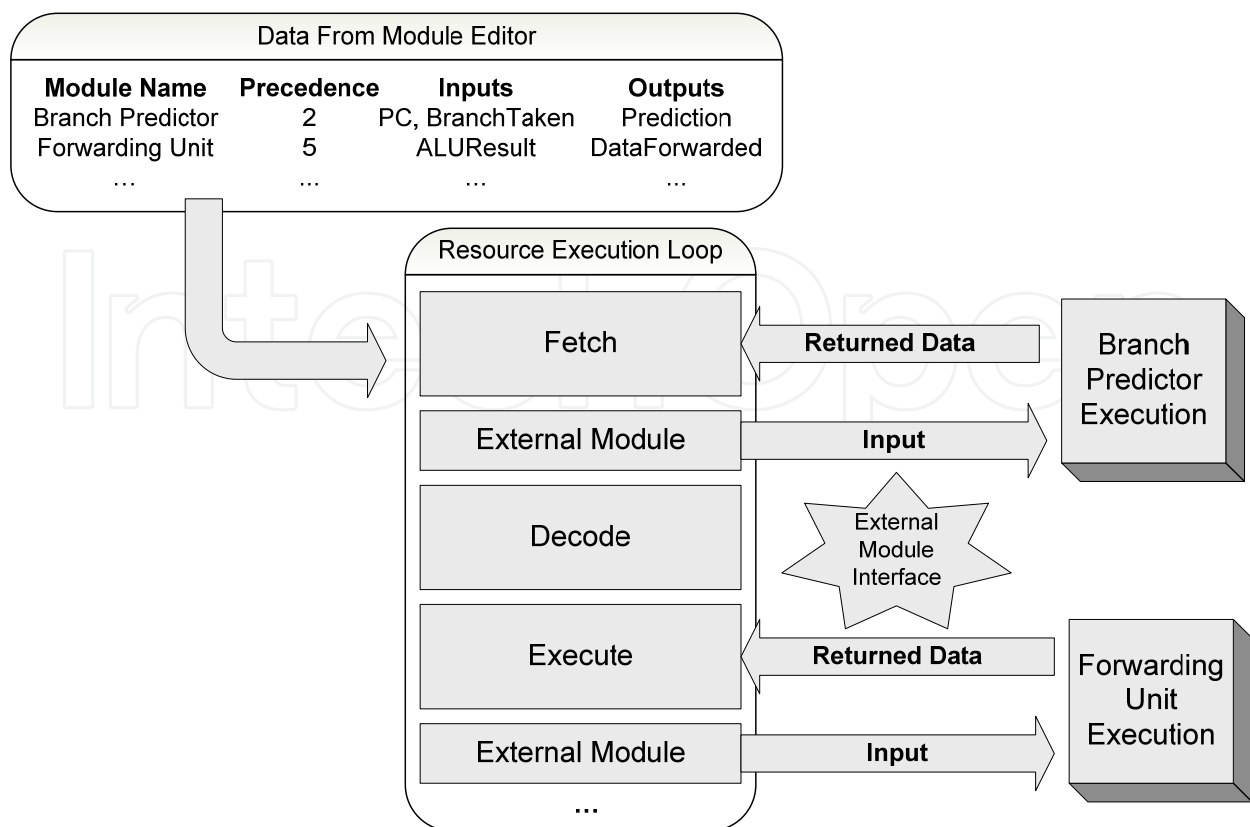


Fig. 7. Two external modules integrated into a resource’s execution loop.

3.6 Execution

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 8. Executing the Simulation Monitor instantiates the classes and prepares the execution of the simulation thread. The user must press the “Start Sim” button to begin the simulation.

Once the simulation is started, the simulation thread is initiated and every instance of the resources and networks is executed. They are executed one cycle at a time, repeatedly, until each resource has completed executing their assigned program (i.e., the program that the simulated resource is running). During a resource’s cycle, all of its modules are executed within a try-catch 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 simulated program, 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.

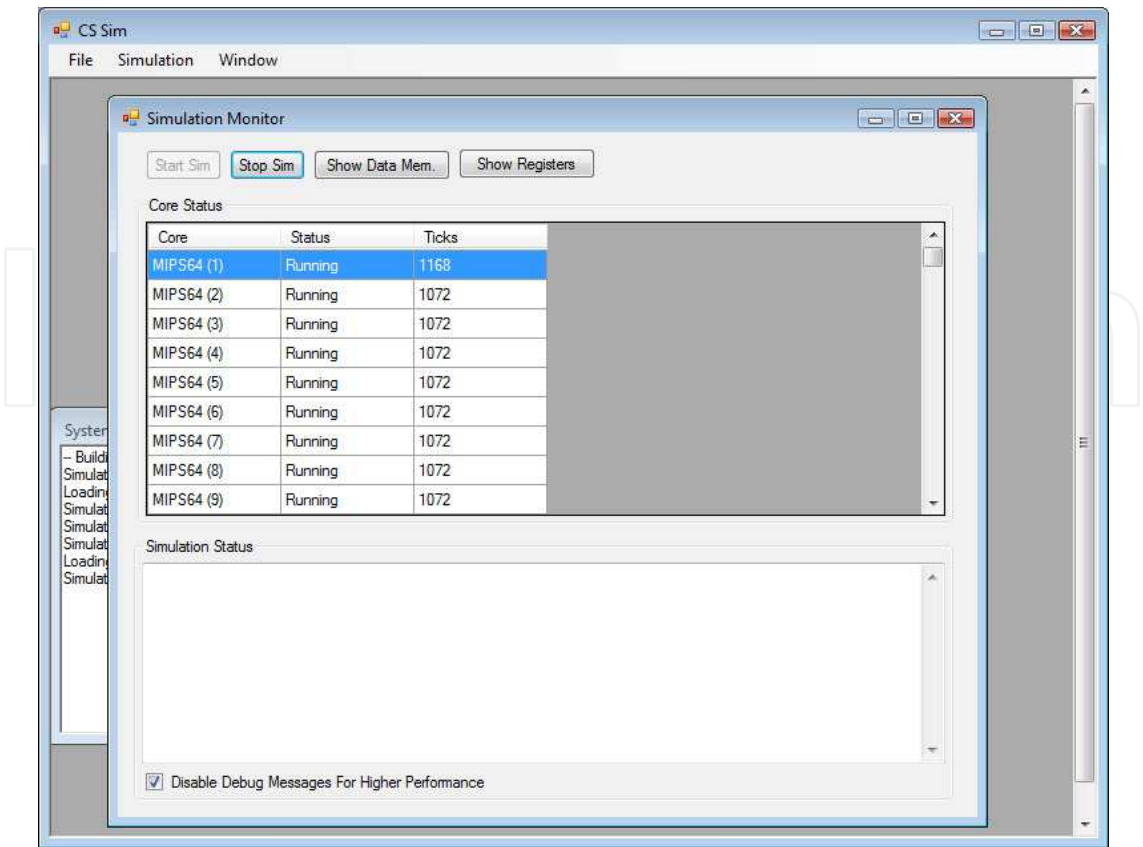


Fig. 8. A Screenshot of the Simulation Monitor Interface.

3.7 Performance Concerns

Due to the nature of the framework, the performance of the resulting simulation can vary greatly depending upon the simulation configuration and modeling detail. During the development of the framework, every effort was made to keep the simulation overhead to a minimum. In Section 5.5, we show that simulators generated using our framework can be competitive with other major simulators.

4. Network-on-Chip

4.1 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. Additionally, 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 discuss our NoC implementation, the NoC Configuration Editor, and explain how the NoC executes within the simulation framework.

4.2 Network-on-Chip Structure and Execution

Routers and resources interface with the network using inputs and outputs, which are implemented using the FIFO queue .NET class, *Queue*. Connections (described in more

detail below) 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 3.2) manages the flow of data through the connections and executes the routing functions for each *Router* instance.

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 type and even router types existing in different networks. This results in an extremely flexible NoC implementation that can simulate arbitrary network topologies. Figure 9 shows an example 2D mesh network.

During each cycle while 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 that it iterates through 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, which are available to their behavioral source code. 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 register mapping for I/O, or memory mapping, or instruction mapping (creating and using user-defined instructions for I/O).

IntechOpen

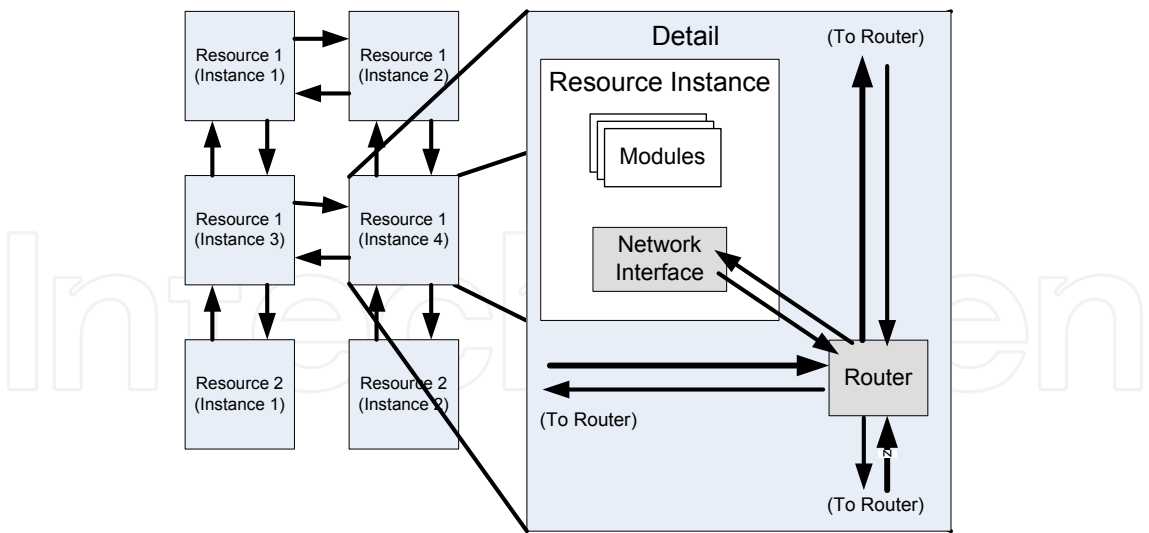


Fig. 9. An example of a 2D mesh network topology.

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.

4.3 Network-on-Chip Configuration Editor

The NoC Configuration Editor (similar to the RCE shown in Figure 2) allows users to define the router types and connections between the routers and resources. Router types have a name, the 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 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 the router’s 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 pieces, known as flits, a common occurrence in current NoC designs.

5. Experimentation

5.1 Overview

The goal of these experiments was to demonstrate and verify the configurability of our framework, as well as its ability to produce cycle-accurate discrete event simulators. Four 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.

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.

5.2 Cache Simulation Experiment

The purpose of this experiment was to demonstrate the framework’s cache system. One level of 1KB cache was used with three different mapping schemes: 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 sort algorithm 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 10) demonstrate that the cache’s performance varies with different configurations and the accuracy responds in a manner that is in line with expectations.

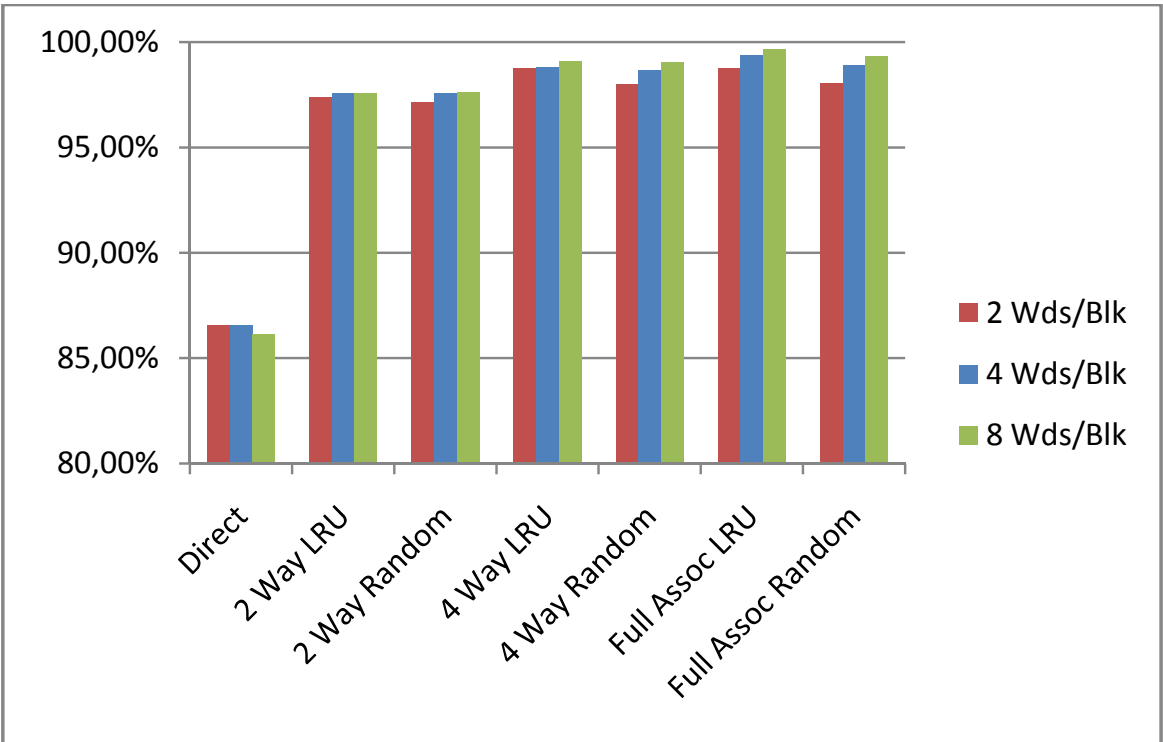


Fig. 10. Cache simulation results.

5.3 Branch Prediction Algorithm Comparison Experiment

This experiment was conducted to demonstrate the capability of using external modules with 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 3.5. 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 typical programs, but does not perform any specific function.

Results across all of the students were similar. The branch prediction results from one project are shown in Figure 11(a) and Figure 11(b). Figure 11(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 11(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 the simulation. This method provides an easy-to-use and standardized environment for testing and comparison.

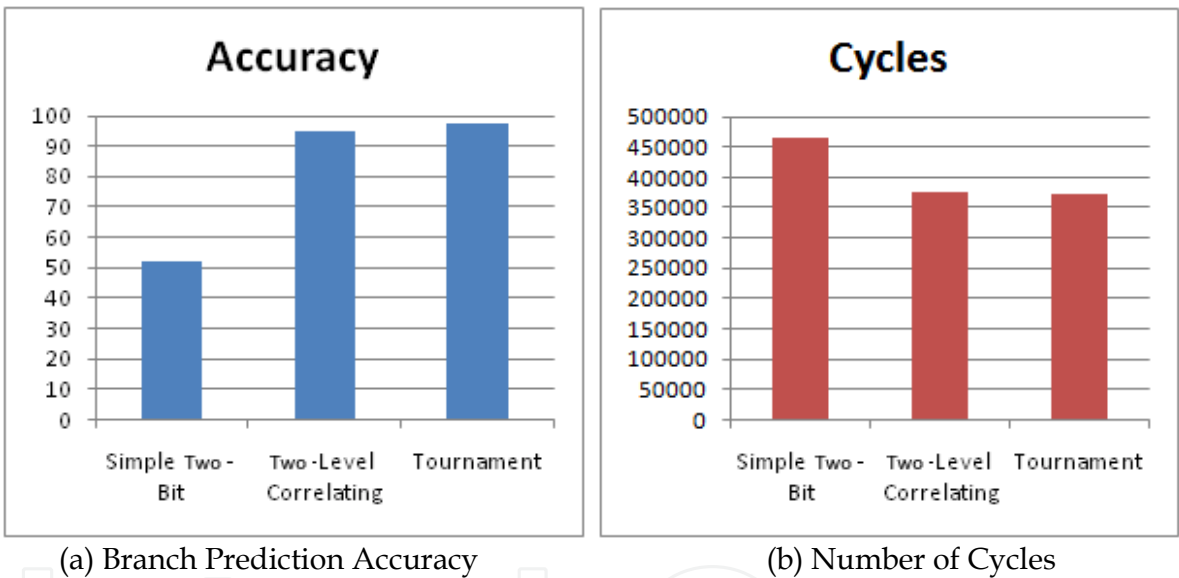


Fig. 11. Branch Prediction accuracy and number of cycles required by each scheme.

5.4 Network-on-Chip Experiment

This experiment is a brief demonstration of the NoC capabilities of 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. This process repeats until all of the calculations have been completed. This is similar to how MPI (A. Gabriel, et al, 2004) or PVM (Sunderam, 1990) processes.

To demonstrate the capabilities of the NoC, we implemented a 2D mesh network topology (similar to the one shown in Figure 9) and then varied the number of slave cores performing the calculations and observed the number of cycles needed to aggregate all of the results. In this experiment, 600 pairs of 64-bit values were used to perform a dot product calculation on each slave core. The cores interacted with the network through registers mapped to network inputs and outputs.

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 simulated programs running on the slave cores.

The results (shown in Figure 12) 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 of routers that each packet must traverse increases, reducing the benefit of additional cores. As can be seen in the figure, an especially large speedup occurs after increasing the processing cores from 4 to 16 with a large workload due to the high ratio of slave core processing time to communication overhead. With 512 cores, the total execution times for the small and large workloads became nearly identical.

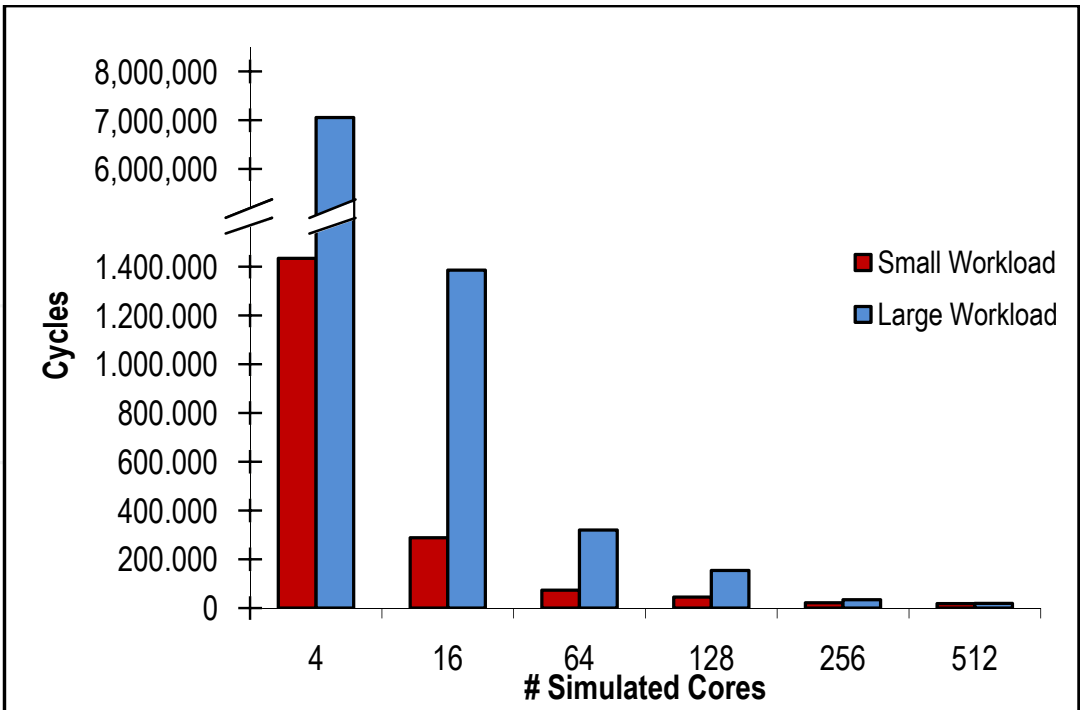


Fig. 12. Number of cycles required with varying number of cores.

5.5 Simulation Performance Experiment

The purpose of our last experiment was to examine the performance of a simulator generated by the framework. A MIPS64 configuration was executed several times with a varying number of cores, each executing an insertion sort application. There was no network executing during this experiment.

The results of the experiment are illustrated in Figure 13, which shows that as the number of cores increases, the total Instructions-Per-Second (IPS) degrades only slightly, while the IPS per core degrades proportionally to the number of cores. Additionally, the simulation performance for a single-core simulation is competitive with other major simulators.

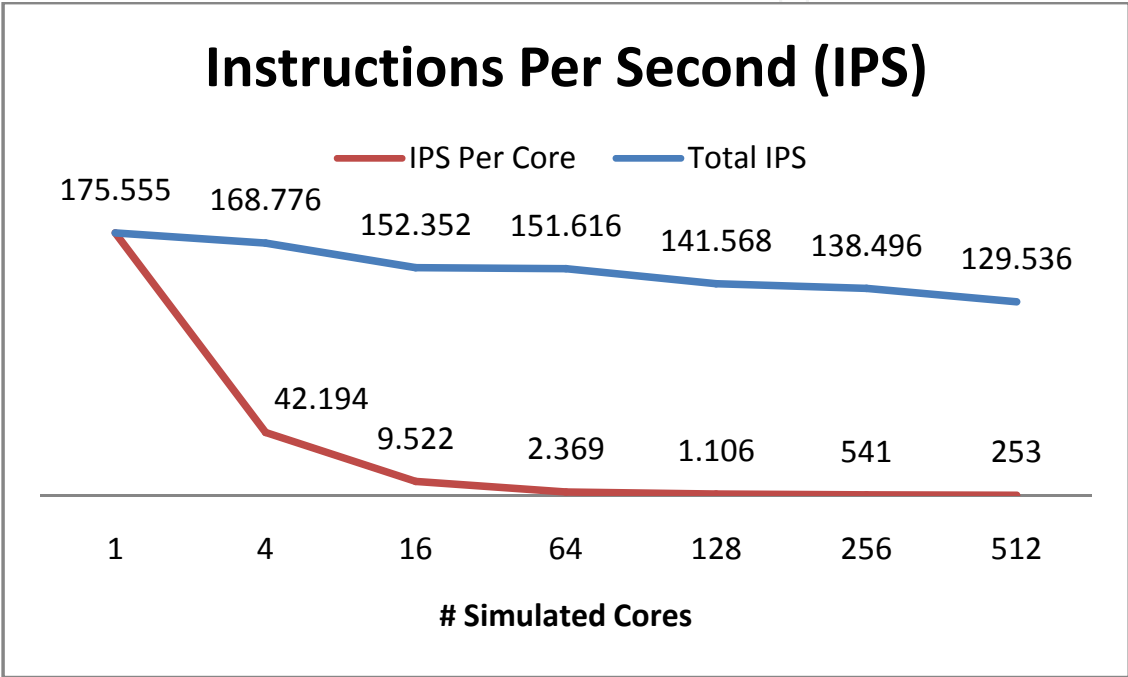


Fig. 13. Performance results with increasing number of cores executing concurrently.

6. Summary and Conclusion

6.1 Summary

In this chapter, we have discussed a simulation framework for dynamically configurable discrete event simulators for many-core chip-multiprocessors. In particular, we have discussed how users can use the framework to configure, construct, and execute simulations, and the details behind the framework’s implementation. We also discussed how we applied our configurability approach to a NoC implementation in the framework. Finally, we performed several experiments to verify our framework, and showed how it can be used to further computer architecture research and education.

6.2 Conclusion

The simulation framework discussed in this chapter provides several contributions in an effort to improve discrete event and processor simulation for the purpose of research and education. The dynamic compilation technique produces fast simulations and quick

compilation with nearly unlimited configurability. The techniques that we described here allow the framework to maintain the easy-to-use and capable interface for simulation configuration and execution, producing a cohesive and seamless experience that is approachable by novice and expert users alike. The framework's modular design allows users to easily test new implementations and extend a simulator's functionality. 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.

The framework can be accessed from:

<http://www.ece.iupui.edu/~johnlee/index.php?section=tools>

7. References

- A. Gabriel, A. F. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. *Proceedings, 11th European PVM/MPI Users*, 97-104.
- Bochs: *The open source IA-32 emulation project*. (2010). Retrieved from SourceForge: <http://bochs.sourceforge.net/>
- Emer, J., Ahuja, P., Borch, E., Klauser, A., Luk, C., Manne, S., et al. (2002). Asim: A Performance Model Framework. *Computer*, 2, 68-76.
- Freericks, M. (1991). The nML machine description formalism. *Fachbereich Informatik*.
- Gilani, F. (2004). *Harness the Features of C# to Power Your Scientific Computing Projects*. Retrieved 2010, from MSDN: <http://msdn.microsoft.com/en-us/magazine/cc163995.aspx>
- GXEmul. (2010). Retrieved from SourceForge: <http://gxemul.sourceforge.net/>
- Halambi, A., Grun, P., Ganesh, V., Khare, A., Dutt, N., & Nicolau, A. (1999). EXPRESSION: a language for architecture exploration through compiler/simulator retargetability. *Design, Automation and Test in Europe Conference and Exhibition 1999*, 485-490.
- Lee, A. and Vinentelli, A. (1998). A framework for comparing models of computation. *IEEE Trans. on Computer-Aided Design of Integrated Circuit and Systems*, 1217-1223.
- Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Larsson, F., et al. (2002). Simics: A full system simulation platform. *Computer*, 35, 50-58.
- Martin, M. (2005). Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, 92-99.
- Microsoft Corporation. (2010). *Compiling to MSIL*. Retrieved from MSDN: <http://msdn.microsoft.com/en-us/library/c5tkafs1>
- Microsoft Corporation. (2010). *MSVC Dev. Center*. Retrieved from Microsoft Developer Network: <http://msdn.microsoft.com/en-us/vcsharp/default.aspx>
- Pai, V., Ranganathan, P., & Adve, S. (1997). RSIM: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors. *Third Workshop on Computer Architecture Education*.
- Reshadi, M., & Dutt, N. (2005). Generic pipelined processor modeling and high performance cycle-accurate simulator generation. *Design, Automation and Test in Europe*, 2, 786-791.

- S. Mukherjee, S. R. (1997). Wisconsin Wind Tunnel II: A Fast and Portable Architecture Simulator. *Workshop on Performance Analysis and Its Impact on Design*.
- SimpleScalar LLC. (2010). *SimpleScalar Overview*. Retrieved from <http://www.simplescalar.com/>
- Sunderam, V. (1990). PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 315-339.
- Univ. of Minnesota. (2010). *SIMCA, the Simulator for Superthreaded Architecture*. Retrieved from ARCTiC Labs: <http://www.arctic.umn.edu/SIMCA/index.shtml>
- Wallin, D., Zeffer, H., M.Karlson, & Hagersten, E. (2005). Vasa: A simulator infrastructure with adjustable fidelity. *Parallel and Distributed Computing and Systems*.
- Zivojnovic, V., Pees, S., & Meyr, H. (1996). Lisa machine description language and generic machine model for hw/sw co-design. *Proceedings of the IEEE Workshop on VLSI Signal Processing*.

IntechOpen

IntechOpen

IntechOpen



Discrete Event Simulations

Edited by Aitor Goti

ISBN 978-953-307-115-2

Hard cover, 330 pages

Publisher Sciyo

Published online 18, August, 2010

Published in print edition August, 2010

Considered by many authors as a technique for modelling stochastic, dynamic and discretely evolving systems, this technique has gained widespread acceptance among the practitioners who want to represent and improve complex systems. Since DES is a technique applied in incredibly different areas, this book reflects many different points of view about DES, thus, all authors describe how it is understood and applied within their context of work, providing an extensive understanding of what DES is. It can be said that the name of the book itself reflects the plurality that these points of view represent. The book embraces a number of topics covering theory, methods and applications to a wide range of sectors and problem areas that have been categorised into five groups. As well as the previously explained variety of points of view concerning DES, there is one additional thing to remark about this book: its richness when talking about actual data or actual data based analysis. When most academic areas are lacking application cases, roughly the half part of the chapters included in this book deal with actual problems or at least are based on actual data. Thus, the editor firmly believes that this book will be interesting for both beginners and practitioners in the area of DES.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Jaehwan Lee and Christopher Barnes (2010). A Dynamically Configurable Discrete Event Simulation Framework for Many-Core Chip Multiprocessors, Discrete Event Simulations, Aitor Goti (Ed.), ISBN: 978-953-307-115-2, InTech, Available from: <http://www.intechopen.com/books/discrete-event-simulations/a-dynamically-configurable-discrete-event-simulation-framework-for-many-core-chip-multiprocessors>

INTECH
open science | open minds

InTech Europe

University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China

Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

© 2010 The Author(s). Licensee IntechOpen. This chapter is distributed under the terms of the [Creative Commons Attribution-NonCommercial-ShareAlike-3.0 License](https://creativecommons.org/licenses/by-nc-sa/3.0/), which permits use, distribution and reproduction for non-commercial purposes, provided the original is properly cited and derivative works building on this content are distributed under the same license.

IntechOpen

IntechOpen