Evaluating the Design and Performance of a Single-Chip Parallel Computer Using System-Level Models and Methodology

Patrick Anthony La Fratta

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in a partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

James M. Baker, Co-chair
Cameron Patterson, Co-chair
Mark Jones
Thomas Martin

April 27, 2005

Blacksburg, Virginia

Keywords: System-level Design, Methodology,
Parallel Computing, System-on-a-Chip, SCMP

Evaluating the Design and Performance of a Single-Chip Parallel Computer Using System-Level Models and Methodology

Patrick Anthony La Fratta

James M. Baker, Ph.D., Committee Co-chair

Cameron Patterson, Ph.D., Committee Co-Chair

Department of Electrical and Computer Engineering

Abstract

As single-chip systems are predicted to soon contain over a billion transistors, design methodologies are evolving dramatically to account for the fast evolution of technologies and product properties. Novel methodologies feature the exploration of design alternatives early in development, the support for IPs, and early error detection – all with a decreasing time-to-market. In order to accommodate these product complexities and development needs, the modeling levels at which designers are working have quickly changed, as development at higher levels of abstraction allows for faster simulations of system models and earlier estimates of system performance while considering design trade-offs.

Recent design advancements to exploit instruction-level parallelism on single-processor computer systems have become exceedingly complex, and modern applications are presenting an increasing potential to be partitioned and parallelized at the thread level. The new Single-Chip, Message-Passing (SCMP) parallel computer is a tightly coupled mesh of processing nodes that is designed to exploit thread-level parallelism as efficiently as possible. By minimizing the latency of communication among processors, memory access time, and the time for context switching, the system designer will undoubtedly observe an overall performance increase. This study presents in-depth evaluations and quantitative analyses of various design and performance aspects of SCMP through the development of abstract hardware models by following a formalized, well-defined methodology. The performance evaluations are taken through benchmark simulation while taking into account system-level communication and synchronization among nodes as well as node-level timing and interaction amongst node components. Through the exploration of alternatives and optimization of the components within the SCMP models, maximum system performance in the hardware implementation can be achieved.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

On September 7, 2004, Intel Corporation unofficially announced a switch from frequency to parallelism as its primary philosophy for microprocessor design as the company plans to release its first dual-core processor [39]. The increasing costs of single-processor systems due to design complexity, increased verification times, high power consumption, and a host of other obstacles have lead to a widespread recognition of the diminishing returns of investments in these design efforts. Furthermore, the amount of instruction-level parallelism (ILP) that can be extracted from most applications has limits that single-processor systems will soon reach.

As these challenges lead to an expected change of paradigms in computer architecture, Moore's law predicts that technologies will soon achieve over a billion transistors on a single die [20]. This increasing number of transistors consequently results in heightened design complexity, and as the widespread use of single-processor systems diminishes, new methodologies and system-level design tools are becoming essential to keeping costs low in the planning and construction of multiprocessor systems. Architects must perform accurate analyses of design alternatives of these systems in early design stages to avoid unreasonable costs and time-to-market (TTM). This chapter includes an overview of the objectives of current system-level design techniques, a description of the Single-Chip Message-Passing (SCMP) computer whose optimization can be achieved by use of these techniques, and a presentation of the motivation for this thesis.

1.1 System-level Design

As systems on chips (SoCs) grow increasingly complex, and time-to-market becomes exceedingly important, the primary problem in the design of these systems is no longer the limit on the number of transistors able to be placed on the die. Rather, the development of improved *methodologies* is the challenge that lies ahead for system designers. Methodologies are defined as techniques, sets of models, and transformations that allow the efficient conversion of an abstract or behavioral specification to a detailed definition of the system ready for hardware implementation [21].

The need for innovative methodologies has become more evident. Over the past twenty years, by Moore's Law, the complexity of chips has seen an average increase of 58% (transistors/chip) per year, while designer productivity over the same period of time has seen an average increase of only 21% (transistors/designer/day) per year [21]. As this gap continues to widen, designers will be unable to utilize the resources available on the die within a reasonable TTM. This trend reveals that the conventional RTL-to-gates strategies are no longer useful or suited for the complex SoC projects currently underway. By postponing the overwhelming details that are inherent within the models at RTL, a much welcome shift in the design paradigm will open many avenues early in the design process including performance evaluation, greater opportunities for IP assembly, consideration of design tradeoffs, and many other benefits, resulting in a decreased TTM [20] [48]. With this shift in paradigm comes an inevitable incursion of new tools and languages. While VHDL and Verilog lend themselves well to the RTL-to-gates approach, modern designers recognize that the risks and effort required at this level of design are unnecessary, and the acquisition of new tools that will close the gap between what they can fabricate and what they can design is essential to a successful design [28]. Additionally, these dated HDLs are not well suited for satisfying the current need of IP protection. Since design at lower levels of abstraction is usually carried out with a specific implementation in mind, early system development at these levels prevents design space exploration and thus tradeoff

evaluation [24]. Finally, considering unnecessary details in system models yields slow simulations, further inhibiting thorough performance evaluation. In summary, five key aspects of today's industry necessitate innovative design methodologies [12]:

- Fast evolution of product properties
- Fast evolution of technologies, requiring frequent adaptations of methodologies
- New application domains appearing
- Decreasing TTM
- Demand for decreasing cost in the market encountering price erosion

1.2 The SCMP Parallel Computer

As the number of transistors able to be placed onto a single die is well into the hundreds of millions [46] and by Moore's Law will continue to rise, innovative designs of SoCs continue to emerge to achieve best performance from these increasing resources. Underlying these innovations is the evidence that the conventional objective in the performance improvements of microprocessor design, the exploitation of ILP available in applications, must be altered. Single-core processor designs aimed at this objective, equipped with superpipelines and hardware support for branch prediction, speculation, and out-of-order instruction execution, have become exceedingly complex and consequently unreasonably expensive to design and verify. Furthermore, studies show that the reaching of the limit on the amount of ILP available in most applications is imminent. Recent designs, such as IBM's Blue Gene [22], based on research showing that a substantial amount of coarse-grained parallelism is actually present in modern applications, have been built for extracting the parallelism available at the thread level [52] [41].

To develop a new approach for performance improvement and utilize chip resources more effectively, single-chip multiprocessors have emerged as promising alternatives to superscalar

designs for a great number of reasons [41]. While some of these consist of multiple processing elements sharing one main memory, such as Stanford's Hydra system [26], others have emerged that assign separate memories to each processor [22]. One such distributed memory system in particular, the Single-Chip Message-Passing (SCMP) computer, addresses arising problems with conventional superscalar architectures and offers substantial potential for performance improvements in many applications [2]. Rather than use available transistors to lengthen pipelines or to support advanced dynamic scheduling techniques in pursuit of more ILP, the SCMP architecture is comprised of simple processing nodes replicated across the chip up to sixty-four times. The nodes are arranged in a 2-D mesh configuration, each having network connections to its four nearest neighbors as shown in Figure 1.1. Since all nodes are identical and have a relatively simple design with only a five-stage pipeline and no complex hierarchical memory system, the development and verification costs are greatly lessened. With each node having support for up to sixteen contexts and the ability to perform context switches in as little as three or four clock cycles, SCMP is an extremely well-suited architecture for exploiting thread-level parallelism (TLP).

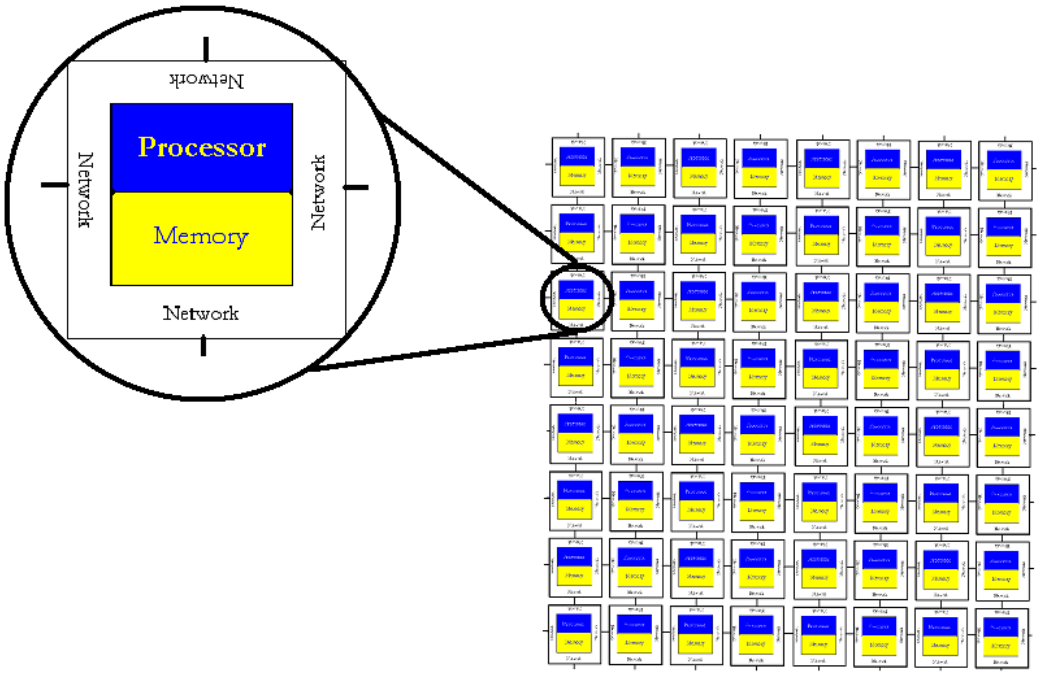


Figure 1.1: An 8x8 Configuration of SCMP Nodes.

The integration of the entire SCMP system onto a single chip has many benefits in the realm of high performance computing. First, each node has two to eight megabytes of DRAM embedded on-chip, so there are not large numbers of clock cycles wasted while waiting for data to come in from off-chip memory locations. Next, the tightly coupled nature of neighboring nodes offers some substantial benefits and solutions. The latency costs involved in the communication that is inherent in parallel applications are lessened. Additionally, the increased wire delays that result as a consequence of shrinking wires and the rising resistance of these wires is curtailed. The nodes are physically closer to each other and the channels can be kept as short as possible, since inter-nodal connections are only given to nearest neighbors. Furthermore, each node only needs a local clock, and since clock signals need not be propagated across the entire chip, the problems that arise due to clock skew are consequently diminished [27] [44].

1.2.1 The SCMP Node

The SCMP tile can be broken down into ten primary component blocks, as shown in Figure 1.2. The five-stage pipeline, networking components, memory, and support for multiple threads of execution that comprise the node are relatively low in complexity. The 32-bit RISC pipeline, similar to the design of the pipeline found in MIPS [42], divides control into five stages. Observe that there is no hierarchical memory structure in the design. While a modest instruction cache is available to the pipeline, all data fetches are directly to main memory through the memory controller. The node is designed to lessen the overheads involved in the execution of multithreaded applications as much as possible, with support of up to sixteen contexts in the Context Register File (CRF) and the Context Management Table and a context switch possible in as little as three to four clock cycles. The Network Interface Unit (NIU) governs the distribution of incoming and outgoing data and thread messages, while the router links messages to other nodes and accepts messages to its residing node via wormhole routing and virtual channel flow control [23]. In-depth descriptions of

the functionality and implementations of the SCMP node's components are presented in Chapter 3.

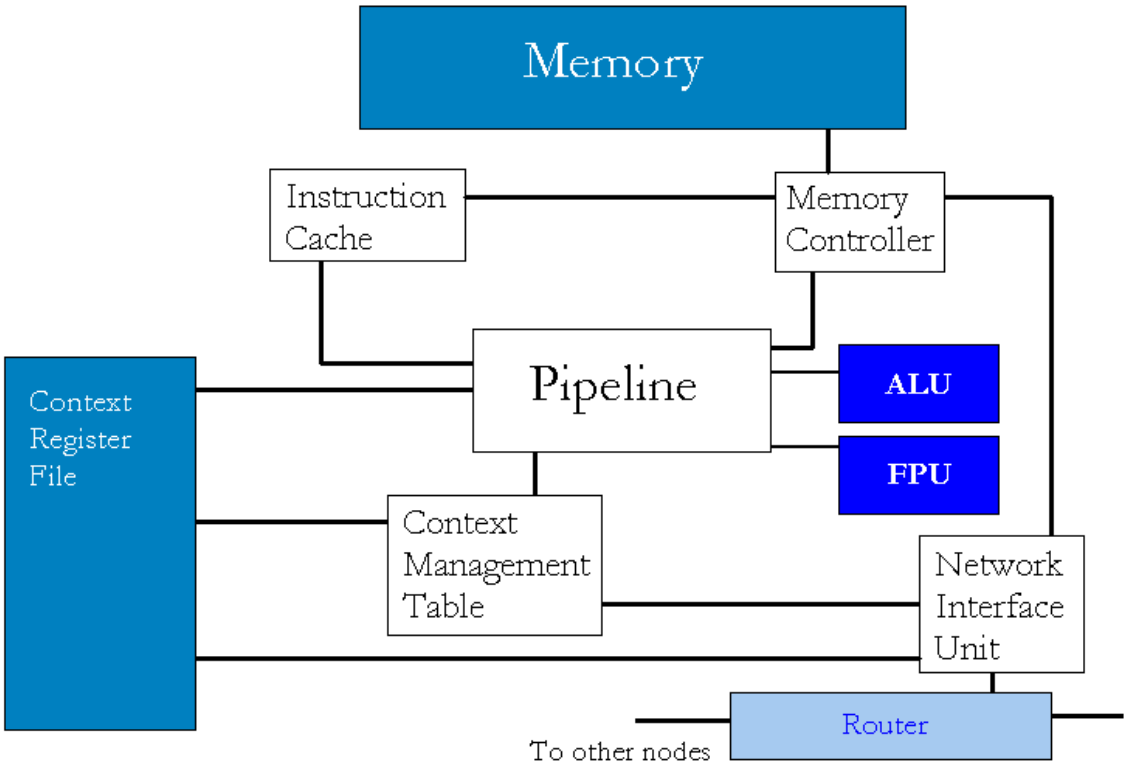


Figure 1.2: Block Diagram of the SCMP Node.

1.3 Organization of Thesis

The objective of this thesis is to present the development of various system-level hardware models of SCMP at different levels of abstraction through a formalized methodology. The models are constructed to assist in the evaluation of the feasibility of design options and are utilized in the analysis of design alternatives. The components of the SCMP node have been loosely defined, and a system-level analysis of the design alternatives is in order. This thesis investigates effective system-level design methodologies and applies them through the

development of various SCMP models. As necessary for design space exploration, the strengths of SCMP, the challenges in its implementation, and options available to overcome these challenges are presented.

Chapter 2 presents background information and previous work required to carry out the studies found in the following chapters, including an exploration of current design methodologies as well as previous SCMP modeling efforts. The background required for the primary design option under consideration, a memory-partitioning scheme for power conservation that requires a profile of memory usage in typical applications, is also explored in this chapter. The third chapter includes a thorough discussion of the SCMP system as it has been developed prior to this study. Chapter 4 provides information regarding the design of the various SCMP models developed through this project. Chapter 5 begins with a description of the algorithm that was used to develop an application for benchmarking and evaluation, and concludes with a presentation of the results found by running this benchmark on the models. Finally, chapter 6 summarizes this thesis work and the results, closing with a description of possibilities for future studies in this field.

Chapter 2

Background and Previous Work

2.1 Design Techniques and Terminology

In order to explore recently emerging system-level design methodologies in the field of computer architecture and hardware design, one must have at least a moderate background of certain techniques and terminology. This section provides the information in these areas necessary to understand the material found in the following sections. Section 2.1.1 describes the roles of the various levels of abstraction that are widely used in system-level hardware modeling. The purpose of section 2.1.2 is to provide a brief background on the essential topic of *Models of Computation*, which are a modeling language's underlying definitions of the domains at different levels of abstraction.

2.1.1 Modeling Levels and Abstraction

The entry point of design flows in recent methodologies is no longer at RTL, and the modeling involved, sometimes referred to as the *System to RTL flow* [40], at the abstract levels necessitate explicit definitions of new characteristics of these levels. The usefulness and significance of a model at a particular abstraction level is determined by how accurate that

model is with respect to different aspects of the final implementation. The purpose of this section is to define the nomenclature of models at various abstraction levels by establishing criteria for each in terms of what details of the final implementation they must take into account.

When a model is described as *functionally accurate*, this is simply stating that the behavior of the system is reflected by and incorporated into the behavior of the model. From a system-level perspective, all models constructed for this project are functionally accurate. *Structural accuracy* refers to how well the model embodies the physical layout of the final implementation. Inclusion of component blocks and partitioning of hardware and software modules are details having to due with a model's structure. A facet of structural accuracy is *pin accuracy*, which implies that the signal-to-pin assignments amongst the components of the model match those of the end product. The *timing accuracy* of a model illustrates how similarly it behaves to the constraints of the final design with respect to time, including those constraints explicitly defined in the specification as well as those inherently imposed by the final implementation such as critical paths due to propagation delays. Although certain types of models may use clock-driven simulation simply for process activation, a model must closely reflect the timing of the target implementation with respect to number of clock pulses in order to be considered completely *cycle-accurate*.

A high-level representation of a system that is the direct transformation of a textual specification into a model in a System Design Language (SDL) is appropriately named the *executable specification*. The executable specification must be functionally accurate and meet all criteria imposed by its textual description, including any timing constraints. Similar to the executable specification is the *untimed functional model*. Communication is implemented with primitive channels, with no consideration to protocols or implementations of busses. No timing is incorporated at all into this type of model, while synchronization between modules is usually implicit with blocking first in, first out (FIFO) buffers used for communication. This type of implicitly synchronized, FIFO-based model, the *dataflow model,* is frequently used for untimed functional modeling. Modules performing computations are woken up when all

data necessary to perform a task is available, and input data tokens are always mapped to corresponding output tokens, guaranteeing that no data is lost. The *timed functional model* is usually constructed with FIFOs, but synchronization between modules is no longer implicit within the communication channels. With unblocking FIFOs and a clock to wake up the computational modules, this model embodies the function and timing constraints imposed by both the specification and target implementation. The emphasis of a *transaction-level model* (TLM) is the explicit separation of the details of communication amongst components and the functionality of components themselves [24]. Within a module, transaction requests are made with calls to abstract interface functions, so that modifications to the channels through which the data is passed are independent of the interfaces that the modules use directly [11] [7]. While the TLM need not have any structural accuracy, the *platform transaction level model* uses the TLM communication style, should have a corresponding module for every component block in the final design, and is effective in gauging overall system performance and modeling hardware-software interactions early in the design process. Although the term *behavioral hardware modeling* has been overloaded among different sources, in the context of system-level design this technique implies that the representation of a system is pin-accurate and functionally accurate at its boundaries but not cycle-accurate. All of these characteristics, including cycle-accuracy are achieved in the *pin-accurate, cycle-accurate model*, but in both of these models the final implementation's internal structure is not represented. Finally, the *register-transfer level* (RTL) *model* is accurate with respect to all the characteristics listed in the previous model, and in addition has internal structure that is representative of that of the target implementation. This implies that functional units, data storage, and data path of the final design should be included in this model. Data storage takes the form of register files, and the data path is usually governed by a central controller and synchronized to a clock [24].

2.1.2 Models of Computation

An underlying concept of all SDLs and hardware models at all levels of abstraction is the *model of computation* (MOC). Each level of abstraction must have a set of implicit rules of the domain and operational semantics implied by the language by which the models that reside at that level must obey and follow. The three key modeling characteristics that are determined by these rules and semantics are:

- The model of time.
- The rules for process activation.
- The communication semantics.

These rules and semantics are established by the MOC, and it is important to note that an effective SDL is able to support many MOCs, including those novel MOCs defined by a user of the language, as well as heterogeneous models in which all modules do not necessarily use the same MOC. A few examples of MOCs useful to this study are listed in Table 2.1, along with their definitions of the modeling aspects listed above.

MOC	Model of Time	Communication	Process Activation	Application
Kahn Process Networks (KPN) [30]	Untimed.	Infinite length FIFOs carrying data "tokens".	Processes activated upon reception of data tokens.	Algorithmic models of signal-processing applications. High level untimed functional modeling.
Static Dataflow	Untimed.	Finite length FIFOs.	Processes activated upon reception of data tokens.	Special case of KPNs used for more efficient simulation.
RTL Hardware Modeling (type of Discrete Event MOC)	Integer-valued with delta delays.	Signals.	Modules activations' sensitive to input signals (e.g. clock edge).	Low level modeling. Highly accurate performance simulation.
Transaction-Level Model (TLM) (type of Discrete Event MOC)	Various: untimed, integer-valued, et. al.	Calls to abstract interface functions.	Depends on timing method used. Two-phase synchronization [24].	Useful in achieving fast simulations while evaluating communication design trade-offs.

Table 2.1: Examples of Typical Models of Computation.

Now that model types, abstraction levels, MOCs, and SDLs have been generally defined, a brief overview of how they relate and are independently defined is in order. An example of how the typical flow of development of these entities follows. Once system designers establish an MOC that is useful for the abstract representation of systems, an SDL must be developed or modified to include constructs and mechanisms that support that MOC. A level of abstraction for the model, i.e. a level of detail with respect to the final implementation, is chosen that is well suited for this MOC. The model type that corresponds to this abstraction level is then chosen and constructed. Although the order of events and choosing of each of these aspects of the design process is not necessarily as described above, this example illustrates that one particular MOC need not be bound to a single level of abstraction. Furthermore, different types of models may be implemented with different underlying MOCs [24] [40].

2.2 Methodologies

The System Development Life Cycle (SDLC) of a product includes all stages of its design from specification to development to manufacturing. The objective of methodology is to minimize the length of the SDLC while maximizing the quality of the final implementation [17]. Recent studies have shown that the conventional RTL-to-gates approach to system design will soon be impractical. Key features such as IP reuse and abstract modeling techniques that are incorporated into recent methodologies are critical to the success of new designs as the gap between complexity and productivity widens [38] [14]. Optimal methodologies will vary among systems, but in order to establish the most optimal for SCMP, recently established general system design methodologies that have been shown to be effective were studied.

2.2.1 SpecC

The SpecC language is a superset of ANSI-C and seeks to satisfy the industrial need of a universal system design language that supports concurrency, timing, synchronization, IP integration, and hardware/software codesign, among other features. Although the SpecC SDL was not used for this project, its developers present a design methodology that can be used with the language (although the language supports the use of many methodologies – an important feature of any SDL) for transforming the specification model into an implementation model for fabrication.

The SpecC methodology is based on the development, refinement, and transformations of four well-defined models: the specification model, the architecture model, the communication model, and the implementation model. The *specification model* is very similar to the executable specification described in section 2.1.1. There are five key goals of the specification model's construction:

- Separate communication and computation.
- Expose parallelism.
- Use hierarchy to group related functionality.
- Choose proper granularity.
- Identify system states.

The transformation from this specification model to the architecture model is through a three-step process called *architecture exploration*. Through these steps the designer attempts to explore in more detail his or her options for the structure of components, communication mechanisms, and more specific details of how components' functionality will be implemented. Table 2.2 includes descriptions of the steps of *allocation*, *partitioning*, and *scheduling* that are involved in the architecture exploration phase.

Name of Transformation Phase	**Objective**	**Steps**
Allocation	Refine structure.	Determine the number and types of components. Choose which will be IP components.
Partitioning	Refine behavior, communication, and storage units.	Introduce hierarchy, bind behaviors to components, and introduce timing estimates and synchronization.
Scheduling	Refine timing.	Serialize behavior hierarchy, Optimize control.

Table 2.2: The Steps of Architecture Exploration.

In the *communication refinement* phase, the protocols of communication channels are determined, and the virtual busses present in the architecture model are modified to include timing information. In the first step of this phase, *protocol insertion*, two layers for all communication channels are implemented. The lower layer contains the implementation of the bus protocol, while the application layer provides an interface for components using the bus. The *transducer synthesis* step is needed most frequently when using IP components with a fixed protocol and an interface not compatible with other system components. Transducer creation and synthesis is the process of creating adapters or wrappers around the incompatible interfaces for use with the desired modules. The final step in communication refinement is *protocol inlining*, which constitutes the incorporation of the protocols into the system's functional modules.

By the final stage of design, the objective is to perform necessary minor modifications to the model of the previous phase so that back end tools can be used to compile software components into executable code and synthesize hardware components into RTL models ready for manufacturing. The *implementation model* should be accurate with respect to function, timing, and structure [21].

2.2.2 IP Assembly

The support of efficient integration of intellectual property (IP) or virtual components (VCs) within designs at abstract levels of design is an essential aspect of languages and methodology. IPs are parametrizable modules that deliver a desired functionality, and they usually also hide the details of how that functionality is implemented with sufficient security and protection against violation of the rights to the IP. Furthermore, by establishing a standard for IP core interfaces – the objective of the Open Core Protocol International Partnership (OCP-IP) – entire methodologies have been developed based on the integration and evaluation of systems at abstract levels based on previously developed components [24]. The general IP Assembly methodology is based on the four steps of specification development, functional partitioning, mapping these partitions onto different components selected from a library of IPs, and verification. This methodology is similar to platform-based design, in which the library of components consists of a family of architectures satisfying a set of constraints imposed to allow the reuse of hardware and software [24]. Since in IP Assembly the platform is not predetermined, the selection of IP components and mapping of functional partitions must be performed simultaneously. This constitutes a more difficult design approach, but with the advantage that the final design more closely matches the criteria of the specification [21]. Certain aspects of the IP Assembly methodology are key to the design phase of this project.

2.2.3 SystemC

SystemC has many strengths as an SDL [1] [49], but it was chosen for this project for two primary reasons. First of all, previous hardware models of SCMP components [10] were created using this language, so these modules can be reused or modified for the new models. Secondly, the language is very simple to learn with backgrounds in just C++ and event-driven hardware modeling. Although SystemC does not have one particular methodology to

which the developers subscribe, there is literature available on recommended techniques and modeling styles to use with the language [24] [1] [40] [51].

An effective MOC for creation of a system representation that is untimed but functionally accurate is dataflow modeling. In SystemC, a module's functionality is represented by SC_THREADs, which are threads of control that can contain or call functions that include wait statements for suspending until an event occurs. Blocking FIFO channels are then used for communication, and the threads will implicitly be synchronized through the passing of data. To create a timed functional model, timing delays are incorporated into the model via wait statements with a literal time value as an argument.

When synchronous behavior for behavioral model synthesis is desired, the SC_CTHREAD process type is used. These processes are restricted to a single signal sensitivity, which is usually the system clock that will be used to synchronize all components. State machine creation begins in this phase, and mechanisms are provided for synchronous resets and handshaking. Dynamic sensitivity is available in any of the above models through calls to wait with an event or combination of events as the argument.

At RTL, SC_METHODs are used for modeling components' behavior. These processes are usually sensitive to a clock and other signals needed for any desired asynchronous behavior. No wait statements are allowed, as an SC_METHOD always executes from beginning to end upon being awoken. However, dynamic sensitivity is still available through calls to the next_trigger function. SC_METHODs usually model state machines, where state information is contained within a set of member variables owned by the module that contains the method.

2.3 SCMP Modeling

2.3.1 C Simulator

The primary model for SCMP performance evaluation up to this point has been a simulator written in the C programming language. The execution of SCMP applications with this model is driven by a top level function, *simulate_cycle*, that performs a series of function calls to govern the behavior of each processing node depending on the node's previous state as defined by sets of variables it owns. The application on which the model is based is a single-threaded process and written entirely in C. No event-driven process activation of components occurs in simulations, making for very fast execution times.

Although the C simulator plays an important role in the methodological development of SCMP, additional modeling is needed for a variety of reasons, especially with an eventual hardware implementation of the system in mind. First of all, the new models should lend themselves well to further refinement and adaptation to novel design methodologies. Although just about any paradigm of modeling can be implemented with C, much time can be wasted by implementing constructs essential to system-level hardware design such as clocking and synchronization mechanisms. Since semantic implementations of such tools in the form of recently developed SDLs are already available, there is no need to implement them in C. Secondly, the new models should be structurally accurate with respect to component blocks of the node and communication channels within and among the nodes. Most of the components in the C model are represented with a struct to contain data and a group of functions to carry out desired behavior. Data is passed amongst components via function parameters or by directly assigning variables within the structs. Although efficient, the lack of modularity and encapsulation of the components does not support IP well, which is an important feature if a third party block needs to be incorporated into the design. Additionally, evaluation of design tradeoffs of communication channels in the final implementation including the busses connecting blocks within the node and the network

connecting the nodes across the chip would be quite difficult with communication modeled in this manner. Finally, tools for synthesizing C code into hardware are at a very early stage of development. In order to accomplish the objective of implementing SCMP in hardware, more refined models are needed. Using new SDLs and system-level design methodologies, these intermediate models can be efficiently developed.

2.3.2 Modeling of the Node's Components

Some hardware modeling and consideration of design options have been completed in an earlier thesis project by Mark Bucciero, "The Design of the Node for the Single Chip Message Passing Parallel (SCMP) Computer" [10]. In this project, various components of the SCMP node, including the memory controller, the instruction cache, the CMT, and functional units were modeled at RTL using SystemC. Developing testbenches to verify the components as autonomous entities and then as interacting blocks, the exploration of design options by gathering timing data and behavioral results was carried out. Some of the models from the project were incorporated into the pin- and cycle-accurate model developed for this project, and much of the information in the design chapters of the thesis was referenced. In a few aspects, this project is an extension of Bucciero's project. However, the design for this project begins at the system level involving the interaction of multiple SCMP nodes and consequently an increased amount of detail, which in turn necessitates the use of more abstract modeling levels and different methodology.

2.3.3 Evaluation of Power Consumption

While one primary objective of constructing refined models of SCMP is, eventually, the construction of a hardware implementation, others are to obtain more accurate performance metrics, to uncover performance bottlenecks from a system perspective, and to ensure that

constraints set by the specification are satisfied. In "Microarchitectural Level Power Analysis and Optimization in Single Chip Parallel Computers," Ramachandran reveals the challenges in meeting the power consumption constraints in single chip multiprocessors through the use of power estimation models [44]. The results of this work illustrate that one of the primary factors contributing significantly to the overall power consumption of the SCMP system is the amount of embedded DRAM built onto the chip.

The problem of power consumption of memory on single-chip systems has been revealed in many sources [34] [9] [4] [5], and solutions to this problem are currently being presented through further research [19]. The next section opens with a discussion of various proposals that have been studied as solutions to this problem.

2.4 Profiling Memory Accesses

In order to overcome the barrier of the increasing power consumption of on-chip memories, many researchers are studying methodologies for the design of novel memory architectures. One source seeks to design a highly customized memory architecture by selecting modules from a library of IPs. The selection is based on a rigorous analysis of memory access pattern information in the application [25]. Another paper presents a methodology that partitions memory into both on-chip and off-chip sectors, caching as much frequently used data as possible to the on-chip memory to lower overall power consumption based on the principle that on-chip accesses consume less power per transfer [50]. Yet another popular method of reducing overall power consumption is to partition the memory into modules and switch the modules among a set of states based on recent usage [13] [47] [36] [32]. The size of the modules and frequency of state switching are parameters that can be varied based on empirical data gathered and analyzed with memory access pattern algorithms [31]. A theme that is common to all of these ideas is the necessity for profiles of memory usage in common applications to be used in considering the feasibility of new memory architectures and

performing their optimizations. The next section presents tools that have been recently developed to create these profiles.

2.4.1 Memory Profiling Tools

Many tools, SDLs, simulators, and methodologies have recently been developed and modified to provide the architect with system-level power consumption estimates [37] [29] [8]. A primary objective of the SCMP modeling efforts in SystemC is the incorporation of metrics that will be useful for design trade-off analysis involving the main power sinks of the system. Very similar models have been developed as parts of past research projects. For example, Coumeri and Thomas present a methodology for low-power memory design which includes the profiling of bus activity to memory modules and memory access patterns using behavioral models of memory arrays in Verilog [15]. Addressing the issue of high power consumption of memories, Benini et. al. have explored the automated synthesis of a multi-bank SRAM architectures for application-specific systems based on dynamic execution profiles of memory usage [5]. The model developed in this work will be constructed for the use of gathering memory usage profiles to explore the feasibility of the use of novel memory architectures on SCMP.

Chapter 3

SCMP Architecture

The SCMP system is a novel architecture in the realm of single-chip multiprocessors in many aspects, and many of its novel characteristics have been previously explored, optimized, and specified with consideration to the challenges of SoC integration. The development of the refined models of the SCMP system is based on this past work that includes specification and high-level modeling. Implementing the functionality and interconnect at the system- and node-levels requires a strong background in this previous work, as key design features should be consistent with previous specification. This chapter presents the aspects of the SCMP architecture with which the designer should be familiar prior to proceeding with methodological development. The first section presents lower-level features of the architecture that were not discussed in the previous overview of SCMP. The next sections include brief backgrounds of the RISC pipeline and the primary networking components found on the node. Finally, the chapter closes with a presentation of the key components supporting multithreaded execution on the node.

3.1 ISA Overview

The SCMP node is a RISC-based, 32-bit architecture with an ISA of 108 instructions for performing common tasks such as arithmetic operations, logic operations, loads, and stores,

as well as tasks more specific to the SCMP system such as context management and message passing. It is not necessary to run an operating system on SCMP to execute applications, as the implementation of the entire ISA in hardware including context management and network functionality yields very fast instruction execution. All components of the node including memory, register files, and contexts are integrated on-chip, and the ISA and compiler [6] were developed with this in mind. This paper focuses on the modeling and evaluation of an implementation of this ISA, and illustrates that the SCMP hardware is ready to execute applications extremely efficiently. Table 3.1 lists the types of instructions available to the compiler, a few examples of each type, a list of the components involved, and the approximate number of cycles spent in the EX stage of the pipeline for those instructions (assuming no unexpected stalls or exceptions occur).

Instruction Type	Examples	Components Involved	Approx. # of Cycles to Execute
Arithmetic- Logic	ADD, SUB, AND, LSH	ALU, CRF	1
Control-Transfer	BRA, BEQ, BNE	ALU	Untaken: 1 Taken: 3 (Pipeline flush required)
Context Management	END, SUSPEND, ALLOC	ALU, CMT	ALLOC: 1 Others: 3 (Pipeline flush required)
Load/Store	LDB, LDW, STB, STW	Memory Controller, Memory, CRF	2
Network Interface	SENDH_R, SEND, SENDM	Memory controller, Memory, NIU	1 or 2

Table 3.1: Types of Instructions in the SCMP ISA.

All instructions are 32-bits, and there are six different formats in which they can be delivered. A record of these formats, listed in Table 3.2, is necessary when the implementation of the ID stage of the pipeline is carried out.

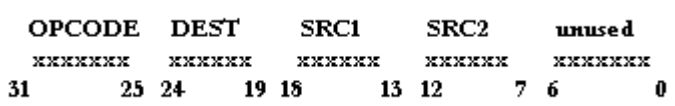
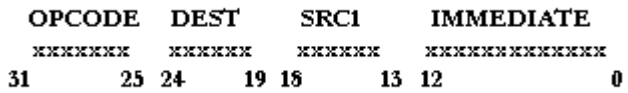
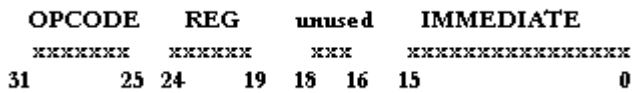
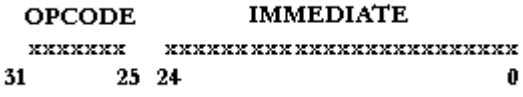
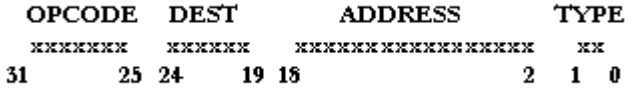
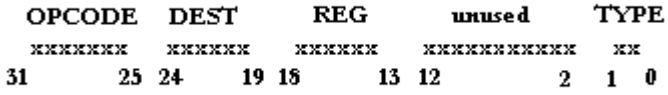
Name	Format
Three-Register	
Two-Register & Immediate	
One Register and Immediate	
Immediate Displacement	
Send Message Header (Immediate Address)	
Send Message Header (Register)	

Table 3.2: SCMP Instruction Formats.

3.2 Pipeline

The control logic for each node in an SCMP system is partitioned into a five-stage pipeline. The nature of the high-level functional partitioning of the control is similar to that of the pipeline found in the MIPS architecture [42], with the stages of instruction fetch (IF), instruction decode (ID), execution (EX), memory access (MEM), and write back (WB).

While a substantial amount of reference material is available that illustrates the general challenges of the implementation of pipelined control, the pursuit for ILP in any novel system such as SCMP will undoubtedly present new difficulties. The primary source of these challenges in SCMP is that each stage of the pipeline interacts directly with a set of components on the node, and in some cases, the sets with which multiple stages interact intersect. As a result, in addition to the stalling, timing issues, and exceptions that inevitably occur and inhibit maximum performance, structural hazards caused by contention amongst the components used by the stages present further design challenges. While possible solutions to these problems are presented in chapter 4 as incorporated into the models, this section presents general functional and structural characteristics of the SCMP pipeline.

The IF stage is essentially in charge of delaying for a clock cycle while the instruction cache responds with the next instruction to be executed, or to indicate a stall in the case of a cache miss. The icache is two-way associative and 32KB in size, with 8 instruction words/set. This study does not explore optimization of the icache parameters, as previously chosen values are used. The icache is one of the three components contending for the memory controller, and as a result long stalls when cache misses occur can be greatly detrimental to performance of the pipeline. Other than this issue, the IF stage is relatively simple.

The instruction is passed from the IF stage to the ID stage whose responsibility is to decode the instruction, determine which register values are needed from the CRF, send the requests for these values to the CRF, and check for data hazards amongst the current instruction and any others before it. The potential for a structural hazard exists in the ID stage, since there are two other components that could possibly be making requests to the CRF while ID is, including the MEM stage. Read-After-Write (RAW) are the only data hazards for which the ID stage must check. Out-of-order execution is not supported by the current SCMP design, so an instruction writing to a register cannot do so before another that comes before it in the instruction stream. Consequently, Write-After-Write (WAW) and Write-After-Read (WAR) hazards need not be remedied.

The EX stage of the pipeline carries out the task of the instruction, sending requests to the available functional units (FUs) or other components. There are numerous events that can occur to cause stalls in the EX stage. Now that the register values are available, the EX stage can test whether a branch instruction will yield a control hazard. In the event of a taken branch, the pipeline is flushed and a request is sent to the icache with the address of the first instruction in the new stream. If a context switch needs to occur due to a SUSPEND or END instruction, a request is sent to the CMT for the instruction pointer (IP) of the new context and the pipeline is flushed. Finally, if an instruction is reached that requires interaction with the NIU, a few problems might be encountered. For instance, the NIU may be busy servicing a previous request, or the data buffer to the NIU may be full in the case of very large data messages. In most of the cases in which EX is using another component on the node, a structural hazard may be encountered, potentially causing an increased number of stalls. Note, however, that in chapter 4, it is shown that contention for the memory controller cannot cause stalls in this case, since memory accesses due to loads and stores are given highest priority in the memory controller's arbitration logic.

The MEM stage is present to deal with delays due to memory accesses and interaction with the memory controller. Finally, when the instruction reaches WB, the writing of results back to the CRF, if necessary for the current instruction, is handled. It should be noted here that when the lower-level timing and structure of the pipeline is explored in chapter 4, each stage of the pipeline turns out to have more responsibility and a little more complexity than just that which is considered here.

3.3 Network

Much of modern day computer architectural design encompasses aspects of computer systems other than handling the control and data local to a single processing element. Rather, it is essential that architects of this era have a strong background in the design of

networking components and topologies [16]. The SCMP network is a 2-D mesh topology, and the two networking components are the network interface unit (NIU) and the router. Messages are passed through the network in the form of 34-bit flow control digits, or flits.

Various design options for the SCMP network were evaluated, and an optimized design was presented in *Balancing Performance, Area, and Power in an On-Chip Network* [23]. Key design choices that are particularly relevant to this study were those involving the routing algorithm, switching mechanism, and flow control techniques. The path that a message takes when traveling from source to destination is determined by the routing algorithm. The routing algorithm chosen for SCMP was dimension-order routing which, in the case of a 2-D mesh topology, requires that each flit of a message travels in one direction along the chip until it reaches the desired row of nodes, makes a turn, and travels the remainder of its path. This algorithm was chosen primarily for its simplicity and because it has been proven that it is deadlock-free as long as all messages entering the network are eventually ejected. The switching mechanism of a network determines how the input channels are connected to output channels along the path of the message. The mechanism chosen for SCMP, wormhole switching, is a technique in which a special flit is sent before each message that reserves the resources that will be required to pass the message. In the case that network traffic requires the message to block, the flits remains strewn out in place across the network. This avoids the need to store the entire message at one given location while traffic clears. Flow control is the method that the routers use to synchronize data between them as they move through the network. Virtual channel flow control is used in the SCMP network. This technique maps multiple lanes of buffers, called virtual channels, to each single physical channel that connects pairs of nodes. As a message moves through the network, the physical channel carries the additional information of which virtual channel is being used by the message. However, with this increase in bandwidth consumption comes the advantage that the entire physical channel is not bound to one message, which is an important feature when a particular message becomes blocked in the network for a long period of time.

The NIU is responsible for passing flits to the router to be injected into the network, and receiving messages from the router to perform the necessary task associated with the incoming flits. The two message types that can be sent are thread messages for creating new threads on the destination node, or data messages for writing a block of words to the memory of the destination node. In order to specify the type and destination of the message, a special flit called the header is sent at the beginning of each message. Figures 3.1 and 3.2 show the formats for the header flits of the two message types.

H	T	X Disp		Y Disp		Thread	unused	
1	0	xxxx		xxxx		0	xxx...x	
33	32	31	28	27	24	23	22	0

Figure 3.1: Format of a Thread Message Header Flit.

H	T	X Disp		Y Disp		Data	unused		Stride Value	
1	0	xxxx		xxxx		1	xxx...x		xxxxxxxxxxx	
33	32	31	28	27	24	23	22	12	11	0

Figure 3.2: Format of a Data Message Header Flit.

The first bit of every flit is an indicator of whether or not it is a header flit, while the second bit is set if the flit is a tail flit. As shown in figures 3.1 and 3.2, the header bit of each of these flits is set, while the tail flit is cleared. Bits 24 through 31 are used by the routers for directing the message to its appropriate destination, as well as for reserving the virtual channels required for passing the message through the network. Because dimension-order routing is used, the routers simply decrement the value contained in bits 28-31 as it moves along the X dimension of the nodes. Once this value reaches zero, the proper column of nodes has been reached, and the flit is transmitted in the appropriate direction along the Y dimension. Once the value contained in bits 24-27 has reached zero, the destination node has been reached, and the flit is ejected from the network and into the node's NIU. Bit 23 of the header flit indicates whether the message is a thread message for initiating a new

context on the destination node, or a data message for modifying the memory contents of the destination node. The stride value in the data message header flit is the address displacement that should be used between each data write by the NIU.

Following the header flit in either message type is an address flit. The address flit in a thread message indicates the starting IP of the new context, while in a data message the address flit indicates the first address to which the NIU should begin writing data. The format of the address flit is shown in figure 3.3.

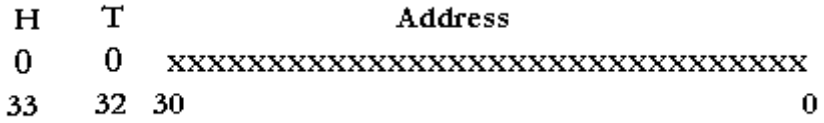
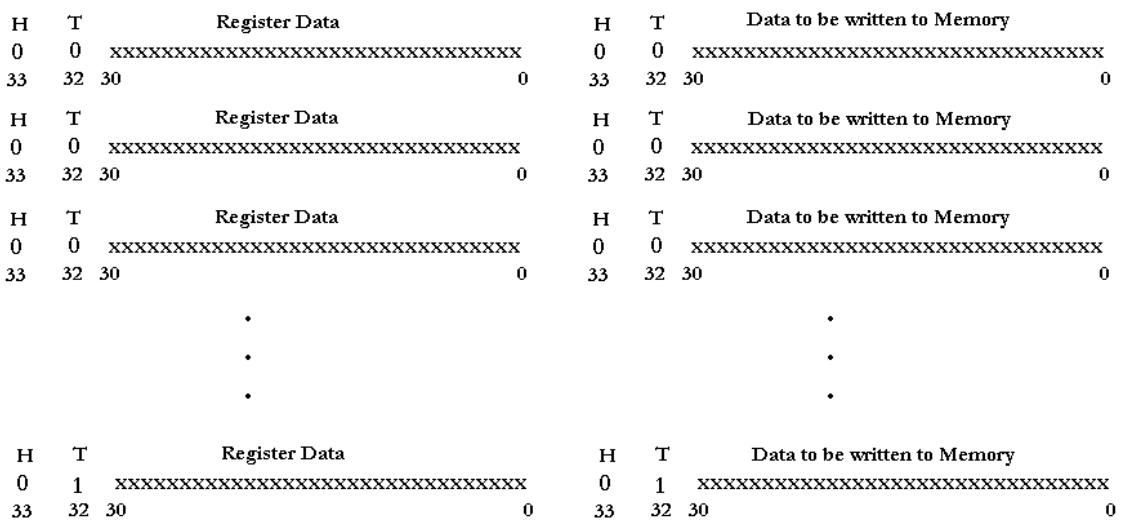


Figure 3.3: Format of the Address Flit.

Finally, following the address flit of the message is the payload. The payload of thread messages consists of register values that should be written to the CRF in the registers associated with the context being created. The payload of data messages contains the data that should be written to memory by the NIU. The last flit in the payload should have the tail bit set, indicating to the routers that the resources that have been allocated for sending the message can be freed. Figure 3.4 illustrates the format of the payload for thread and data messages.



(a) Thread Message (b) Data Message

Figure 3.4: Format of the Messages' Data Content.

Once the message reaches its destination, the router ejects it to the NIU and begins carrying out the appropriate task for the message, based on bit 23 of the header flit. In the case of a thread message, the NIU sends a signal to the CMT indicating that it needs to create a new context of execution. The NIU then writes the IP value that is contained in the address flit into the appropriate location in the CMT, and begins writing register values to the new context's registers in the CRF. Once the register values have been written, the *thread* bit in the new context's field is set, indicating to the pipeline that the new thread is ready for execution.

If the message is a data message, the NIU sends a signal to the memory controller indicating that it needs to write values to memory. The address contained in the address flit is sent to the memory controller, along with the data in the first flit of the payload. The stride value is added to the address and then sent, along with the data in the next flit, to the memory controller. The NIU continues in this fashion until the tail flit of the message is received.

This covers the basic background of the SCMP networking components and functionality necessary to understand the modeling of the components in Chapter 4. The details of

message injection and the finer details of component implementation will be considered in that chapter. For a more complete study of the SCMP network including consideration to higher levels of abstraction, see *Support for Send-and-Receive Based Message-Passing for the Single Chip Message-Passing Architecture* [35].

3.4 Context Management

A key feature of the SCMP system is the efficient execution of multithreaded applications. Each node contains hardware support for storing the necessary information for up to sixteen threads of execution at a given time. These contexts consist of entries in the context management table (CMT) that contain thirty-two bits each. Figure 3.5 shows the breakdown of these bits in an entry of the CMT.

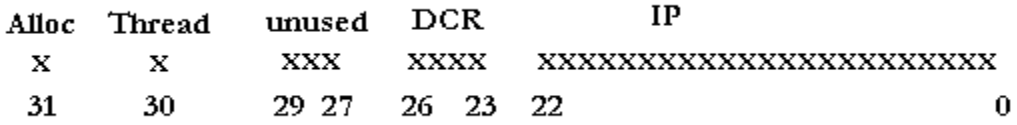


Figure 3.5: Format of an Entry in the CMT.

Bits 23-26, the data context register (DCR), indicate which set of registers in the CRF is associated with this context. Four bits have been allocated for this value, because there are sixteen sets of registers in the CRF that can be used with the contexts. The *Alloc* bit indicates that this entry in the CMT is in use, and the data context register (DCR) has been loaded. Although this bit is set, this doesn't necessarily indicate to the pipeline that this context represents a thread that is ready to execute. For instance, if the context is being used only to store data in the general purpose registers (GPRs), or if GPRs are currently being loaded with the appropriate values by the NIU, the *Alloc* bit should be set in both of these cases. Once the DCR and the GPRs have been loaded, and the context is a new thread that

should be executed by the pipeline, the *Thread* bit is then set. The *IP* field indicates the address location in memory of the first instruction that should be executed when a thread context begins or resumes.

Because the pipeline is only executing the instruction stream associated with one of the contexts at a time, as well as writing to the register values of one of these contexts at a time, two special registers are present to indicate which of the sixteen in the blocks is currently in use. The *Active Thread Register* (ATR) contains the index in the CMT of the thread that is currently being executed, while the *Data Context Register* (DCR) points to an additional context of thirty-two registers that can be accessed by the thread. When a thread is suspended and another is available, the address following the last instruction that was executed is written to the IP of the appropriate CMT entry, the IP of the new context is fetched and sent to the icache for instruction retrieval, and the ATR and DCR are loaded with the values from the CMT entry of the new thread. The simplicity of this process is an important feature of the SCMP system, as the completion of context switches can take place in a very small number of clock cycles.

The pipeline, NIU, and memory controller all have write access to the CMT and CRF, so some form of arbitration must be present. The ALLOC instruction is available in the instruction set so that a new context can be directly created from the current instruction stream. The SUSPEND instruction is given when the currently executing thread should be halted, and another thread context, if any is available, should begin or resume execution. When an END or FREE instruction is given, an entry in the CMT will be removed. The NIU must have access to the CMT and CRF in order to create new thread contexts that are being initiated by thread messages ejected from the network. Finally, the CMT is memory-mapped and can be accessed from the memory controller. This feature could be useful in case an operating system is implemented for SCMP that needs to access the CMT directly via memory writes. However, this type of CMT access is not necessary, and because the models discussed in chapter 4 assume that no operating system is necessarily being used, the CMT is not memory-mapped in these models. The options for arbitration between pipeline

and NIU access of the CMT are considered in the various models of these component blocks in the next chapter.

Chapter 4

Design of the System-Level Models

In the previous chapters, the products of past work with SCMP modeling have been discussed, including a simulator written in C and pin-accurate models of various components written in SystemC. In order to proceed with further modeling to achieve the final objective of a fully optimized hardware implementation, a *methodology*, or a well-defined series of refined models and transformations, should be in place. The previously developed models of the SCMP system and components in conjunction with the detailed descriptions of the behavioral and structural nature of the system can be used as the foundation for the refined models. The topic of this chapter is the in-depth development of intermediate models that will be used for the performance evaluation discussed in the following chapters. Furthermore, these models will also serve as a basis for the production of a more refined model that can be used for implementing the system in hardware. Figure 4.1 presents an overview of the steps required to reach the hardware implementation of SCMP from the system models that are currently available.

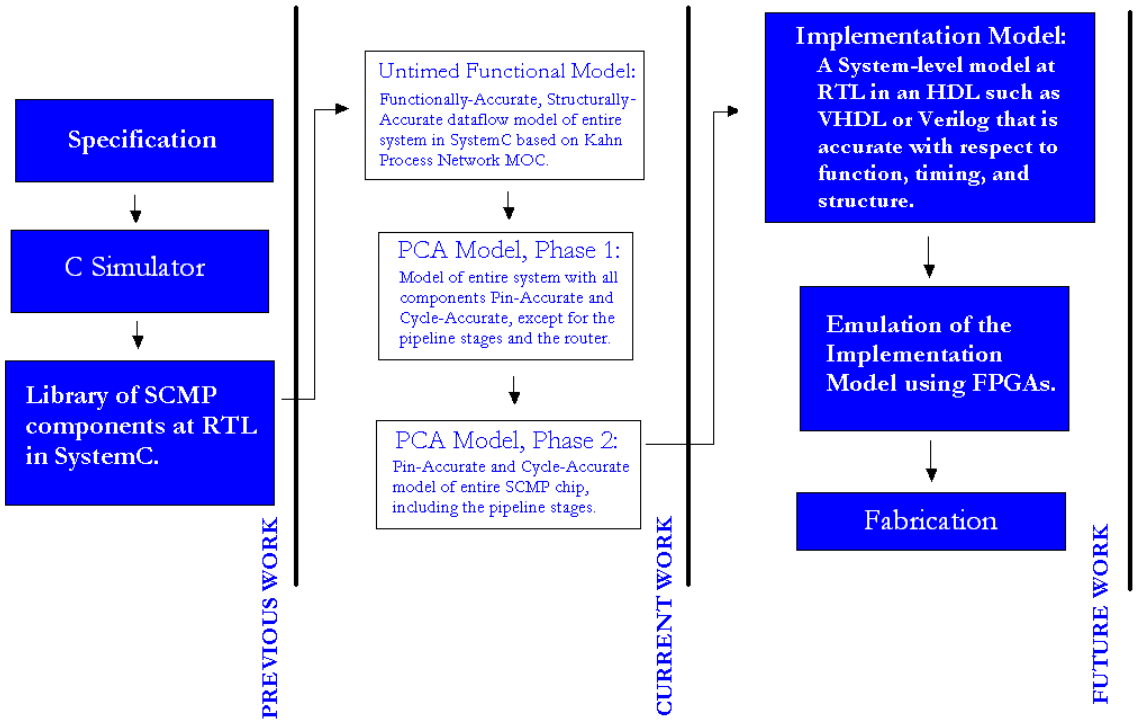


Figure 4.1: SCMP Modeling and Design Flow.

The first section in this chapter presents the development of the untimed functional model of a complete SCMP system, with synchronization amongst the components implicit within the communication channels. The model is created using a dataflow modeling scheme based on the Kahn Process Network (KPN) MOC in which blocking FIFOs are used to represent interconnect amongst the components. Each component is modeled using threads that sleep until all input data needed to carry out the appropriate task has arrived. The second section involves the transformation of this untimed functional model into a pin- and cycle-accurate model, as shown in the second step of the design flow in figure 4.1. Because the simulation of the pipeline is very computationally intense, the first phase of the pin- and cycle- accurate model leaves the pipeline in its functionally accurate representation as modeled in the untimed functional model. All components of the final model are pin- and cycle- accurate except for the router and interconnect amongst the nodes. The router is a pipelined component whose implementation is rather complex, and the incorporation of a cycle-accurate router into the refined models would be greatly detrimental to simulation

performance. Furthermore, a layout of a possible implementation of the SCMP router has been completed as part of a previous project [23]. For these reasons, the router and network of the system in the refined models are pin-accurate, but not cycle-accurate. The reader will notice that in the first section, the interfaces of the component blocks are presented graphically, while such figures are not present for all components in the second section. The connectivity of the untimed functional model was part of the design phase of this project and serves as a foundation for the less abstract models. However, the interfaces of some component blocks at RTL are illustrated in depth in previous research [10], so they are left out here.

A primary application of these refined SCMP models is the gathering of metrics to uncover bottlenecks and realize potential optimizations, as well as to consider the feasibility and effectiveness of new design features. The final sections of this chapter discuss how these models have been constructed with this objective in mind, including the incorporation of mechanisms necessary to gather data at run-time to produce a profile of the memory usage of applications.

4.1 Untimed Functional Model

The first complete model of an entire SCMP system in an SDL is presented in abstract form as an *untimed functional model* (UFM), in which all node-level and system-level interconnect are represented as blocking FIFOs. All functionality in the UFM is implemented with software threads that execute indefinitely, sleeping while waiting for all data to be present on inputs. Figure 4.2 presents the structure of the untimed functional model at the node level.

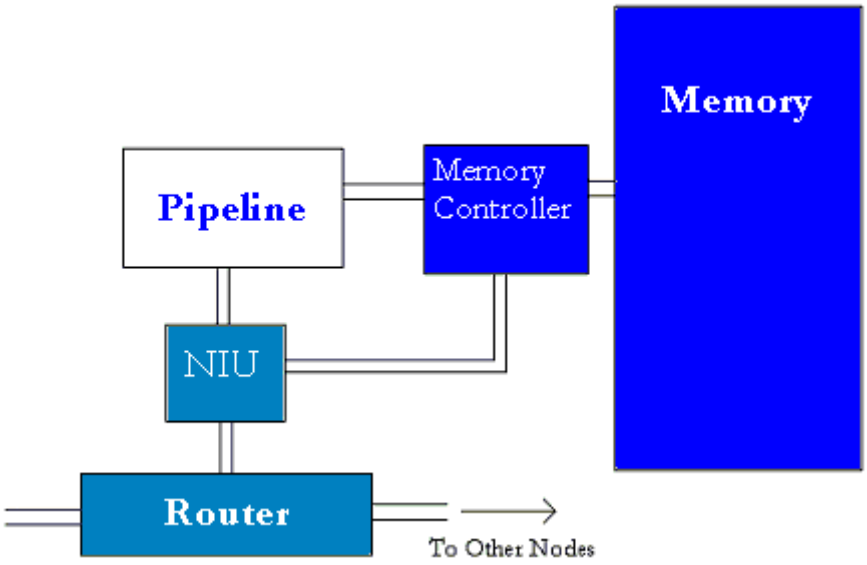


Figure 4.2: Block Diagram of the Node's Untimed Functional Model.

Notice that in this representation, a few very important components of the node are left out of the block diagram because they were not set apart as separate entities, illustrating another extremely powerful feature of high-level designs. The tasks of the functional units, the ALU and the FPU, are carried out directly within the thread embodying the pipeline behavior, while the context management table and context register file are actually represented as data structures within the pipeline block. Since the latter two components are simply blocks that service requests for data, there is no need to separate them into autonomous units in the functionally accurate model. The functionality of the system without these blocks is still perfectly accurate, and the layout of components is accurate enough that the next step of model refinement is clear enough. Important to note is that in this model, the *Pipeline* block is only labeled such because as the model is refined to include timing, the block will be partitioned into the separate stages for employing ILP. In this model, all tasks corresponding to instructions are performed as atomic, sequential tasks. The instruction cache was not incorporated into this model at all, since it is simply a component for performance improvement that can easily be integrated into the design later. As the refinement process progresses, the functionality that should be assigned to other blocks such

as the CMT and instruction cache will be extracted from the pipeline block and implemented in separate components, as illustrated by the methodology.

4.1.1 Chip-Level Design

As opposed to a previous project whose objective was to model and verify components of the node as autonomous entities, the goal of this project is to develop a working model of the entire SCMP system that can execute the binaries produced by the compiler. In order to accomplish this new objective, the design process begins at a new level in the hierarchy of the SCMP system. The first step in the chip-level design of the model is to establish how the nodes will appear to one another and define the channels that will be used to represent the network for passing flits amongst the nodes. Figure 4.3 presents the external interfaces of each node.

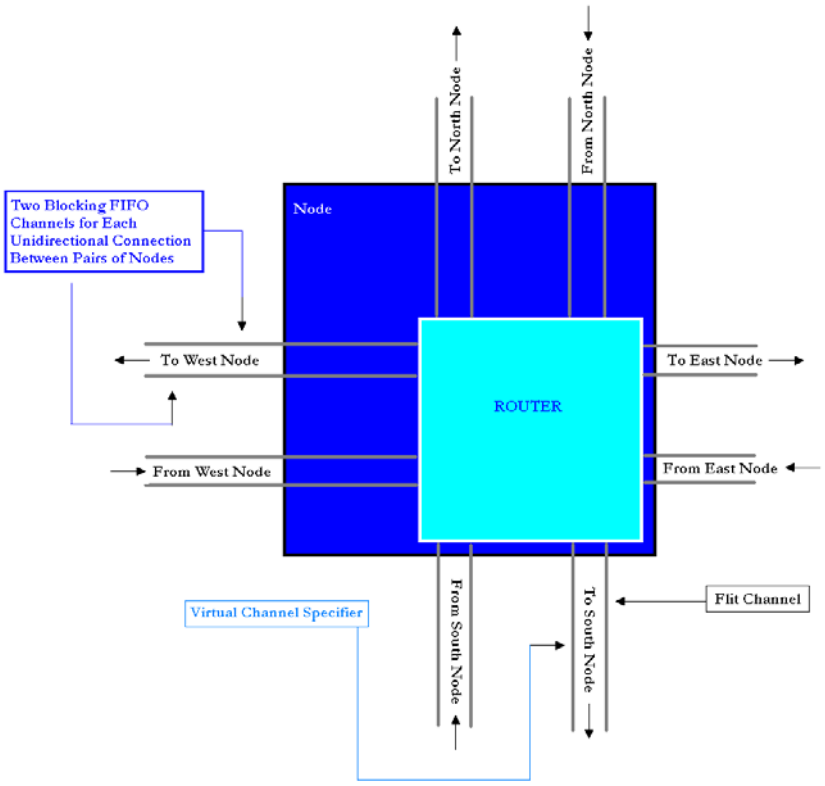


Figure 4.3: The Interfaces of Nodes at the Chip Level.

The establishment of the node at this level is the first step in the development of the untimed functional model, and the representation of the entire chip in SystemC is based on this interface definition. Notice in the figure that there are actually two FIFO channels between a pair of nodes. Because virtual channel flow control is being used, the flits of multiple messages will be attempting to cross the same physical channel in the middle of messages. Each time a flit is passed from one node to another through the flit channel FIFO, the index identifying the virtual channel being used is also specified via the other FIFO. Since each message begins with the passing of a header flit and wormhole switching has been chosen, the virtual channels needed to link the message from source to destination are preemptively reserved for the remainder of the message as the header flit travels from router to router. The virtual channels that are reserved are not made available again until the tail flit of the message passes through. As a message travels through the network, each flit of the message is accompanied by its virtual channel identifier. Any number of flits belonging to other messages being passed simultaneously whose paths overlap with this message may share its physical channels. A wormhole has been spawned from the header flit of each message, and the resources associated with this wormhole are protected from disruption by other messages. The details of how dimension-order routing, wormhole switching, and virtual channel flow control are implemented in the router model are discussed in section 4.1.2.

4.1.2 Networking Components

The router plays a few different roles in the passing of messages, including supplying a link from one node to the next in a message that does not belong to the node on which it resides, recognizing when flits belong to its node and ejecting them from the network into the NIU, and injecting flits from the NIU into the network. Because there are four channels that are very frequently receiving messages simultaneously, the untimed functional model of the

router contains more behavioral blocks – SC_THREADs in the case of dataflow modeling – to model its behavior than the average SCMP component.

For creating a functionally accurate component, it is necessary to clearly establish which of the tasks the component is performing may be performed concurrently. The module representing the component should own an individual SC_THREAD for each of these tasks. In the case of the router, only one flit will be injected at a time, so a single thread watching for incoming data from the NIU is sufficient for flit injection. However, the four incoming channels will certainly be receiving incoming messages at the same time, and so the router module must own four separate threads for ejecting flits from the network.

Complications arise in all concurrent task performance when these concurrent tasks must share the same resources. This is definitely the case with the task of flit ejection, since more than one incoming message may be destined for the router's owner. These flits must all be passed off to the NIU in some organized fashion to avoid message corruption. One solution is to provide four separate FIFO buffer connections from the router to the NIU. The benefit to this approach is that all threads are not contending for an individual buffer. However, a significant amount of extra buffer space is required with this approach, not only because there would be four simultaneous writers to the buffers, but also because there exists only one NIU working at a certain rate servicing the flit ejections. Another disadvantage is that the NIU must be slightly more complex, since it is piecing together messages from four sources rather than one. A second approach is to view the NIU ejection channel as just a fifth channel that contains an interface identical to the outbound channels. With this method, the virtual channel mechanisms handle the arbitration among incoming flits just as it is performed for outgoing flits being linked to other routers. For the untimed functional model in this project, a single channel for incoming flits is connected from the router to the NIU. Figure 4.4 presents the connections of the router in this model required for the injection and ejection of flits.

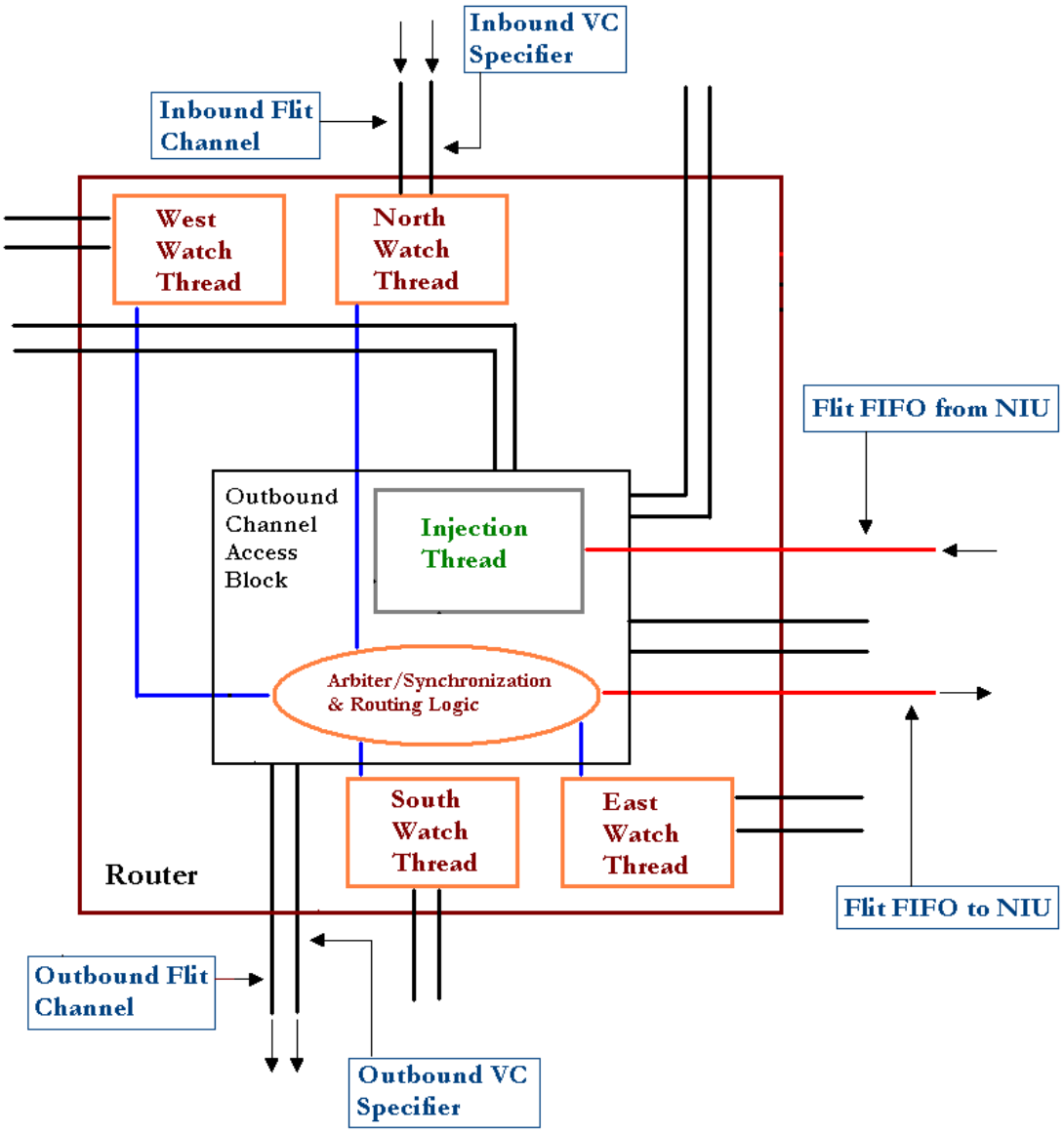


Figure 4.4: External Connections and Internal Structure of the UFM of the Router.

The path of a message begins when the source node's router receives a header flit from the NIU. The message begins its journey when the router searches for a free virtual channel and assigns to that virtual channel the direction in which the entire message will travel. Whenever a router receives or wants to inject a header flit, the correct direction in which the flit should be passed must be determined by means of the dimension-order routing algorithm. The

logic to implement this algorithm is quite simple, since the format of the flit has been created to accommodate dimension-order routing. Recall from chapter 3 that there are two fields in every header flit that specify the distance remaining for the message to travel until reaches its destination. See figure 4.5 for a flowchart describing the implementation of the general dimension-order routing algorithm.

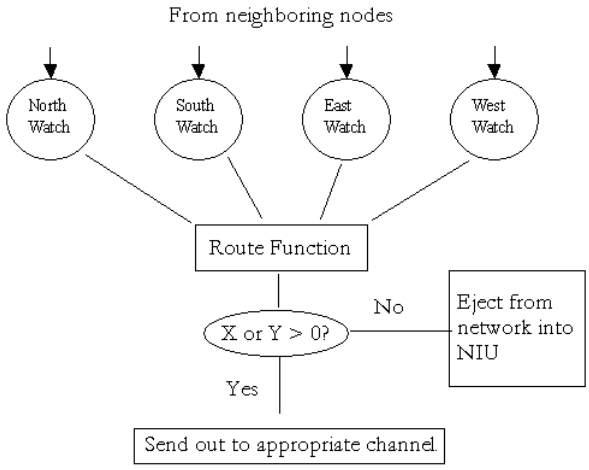


Figure 4.5: Flowchart of Dimension-Order Routing Implementation.

Prior to reaching the router of the source node, the header flit's X- and Y- offset fields contain the distance the message must travel along each dimension in order to reach its destination. To bind a direction to a virtual channel, the X-offset of the header flit is checked first. If this doesn't equal zero, then the column of nodes on the chip containing the destination node has not yet been reached. If the value is positive, the value in this field of the flit is decremented, the virtual channel is bound to the east direction, and the flit is sent through the connection to the neighboring node to the east. If the value is negative the value in the X-offset field is incremented, the virtual channel is bound to the west direction, and the flit is sent to the neighboring node to the west. If the X-offset field is zero, the column of nodes containing the destination node has been reached. A similar check to the one previously described is made to the Y-offset field, and if non-zero, the value is updated, the virtual channel is bound to a direction, and the flit is passed. If both the X- and Y-

offset values are zero, the destination node has been reached, and the virtual channel is bound to the node's NIU input FIFO.

Note that once the direction is assigned to the virtual channel, the wormhole through which remaining flits will travel continues to claim that virtual channel and its associated resources until the tail flit of the message is passed. The index of the virtual channel through which the message is being passed is received from the VC Specifier channel shown in figure 4.4. Consider the case that a virtual channel is currently in use by another message being linked from another neighboring node. If this occurs on the source node that is attempting to inject a flit into the network, the virtual channels continue to be checked until a free one is found, the outbound message direction is bound to that channel, and the flit is passed off in the appropriate direction. If this virtual channel conflict occurs on a node other than the source of the message, however, the second incoming message attempting to use the virtual channel that is currently occupied must block in place until the earlier message finishes passing through. In the future, a more complex switching technique that offers lower average message-passing time may be implemented. For example, this networking scheme does not contain a router that supports dynamic virtual channel assignment as a given message travels through the network, but incorporating such a mechanism into the model would be fairly straightforward. The model of the router included here offers a simple implementation and the correct functionality needed for the UFM.

Once the header flit of the message is ejected at the destination node, the NIU determines what type of message it is by checking bit 23. In the case that this bit is cleared, indicating that a thread message is being received, the NIU block needs an outbound connection to the pipeline block, since the pipeline block owns the data structures that represent the CMT and the CRF. On the other hand, if bit 23 is set, the message is a data message. The NIU is given direct access to the Memory Controller block so that regular instruction execution in the pipeline need not be disrupted. The connectivity and internal structure of the NIU is shown in Figure 4.6.

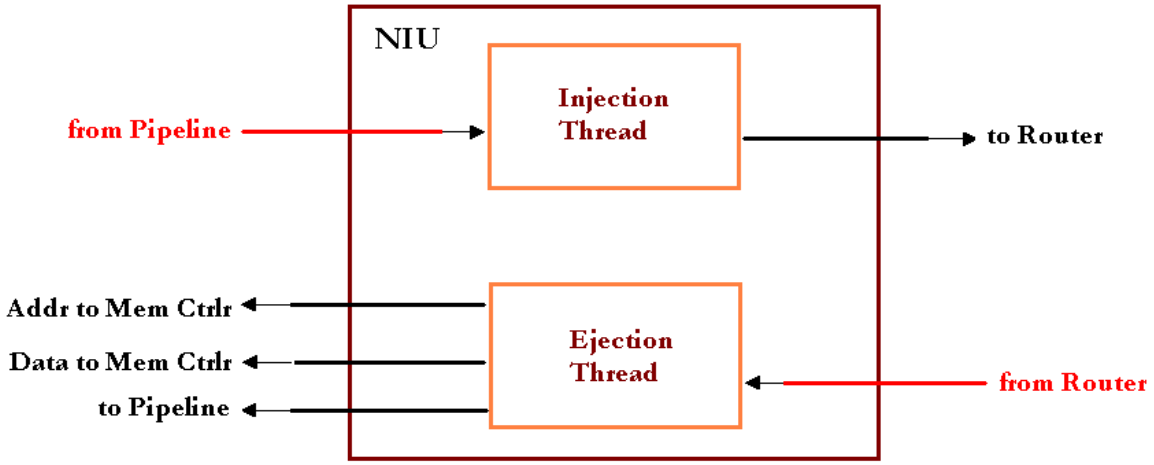


Figure 4.6: External Connections and Internal Structure of the UFM of the NIU.

The NIU's ejection thread is constructed on the assumption that the router will pass to it the flits of a message contiguously. If two messages need to be ejected from different directions, the router's outbound NIU channel arbiter logic will ensure that all flits of one message in its entirety are given to the NIU before any flits of another message are given. Upon the arrival of a message, the NIU must simply extract the necessary address from the second flit and send it to the FIFO buffer of either the pipeline or the memory controller. Since the synchronization in the UFM is implicit within the blocking FIFO channels connecting the components, the NIU's ejection thread simply goes into a loop that extracts the data from the remaining flits of the message and then loads them into the appropriate buffers. If either of these components' SC_THREADs is waiting for data to be present in these FIFOs, they will awaken and begin the necessary behavior. In the other case that the components' threads are occupied with another task, the data will be waiting in the buffers until the threads reach their reading phase.

The injection thread simply passes flits that it receives from the pipeline off to the router. As appropriate for the UFM, this behavior has been greatly simplified, as many of the details that will need to be considered when timing is introduced are ignored. Noteworthy, though, is that in the refined models, the NIU's injection machine has direct access to the memory

controller for the construction and injection of the flits of data messages. Complexity of the interconnect and functionality of this model is spared by distributing all flits of both thread and data message to the pipeline. No state information or state logic is necessary in this model of the NIU, keeping it appropriately simplistic for fast simulation, easy verification, and making it a solid foundation for the refined models.

4.1.3 Instruction Execution

The *Pipeline* block of the UFM contains all of the data structures and functions necessary to perform the context management, save register data, fetch instructions, decode instructions, execute instructions, and construct flits. It is important to realize that the *Pipeline* block in this model is only named as it is because some of the functionality of this block will eventually be partitioned into various stages to form a pipeline in the timed models. For the UFM, all of the functionality of this block is implemented in a single thread, which begins with a phase that reads a thread message from the NIU. The first issuance of a read from the NIU blocks until a thread message is received, in which case the thread reads the IP and register values, if any, from the NIU. Instructions are then fetched by requests directly to the memory controller – no instruction cache is present in this model – decoded, and executed until a context switch occurs or an END instruction is reached. The main function at the top level injects the necessary thread message into the network that initializes the execution of the rest of the application. Thread message ejection, instruction execution, and context switching continue to occur as described above until all spawned threads reach END instructions, and the application is complete. This is the basic function of the *Pipeline* block in the UFM whose interconnect and internal structure is pictured in figure 4.7.

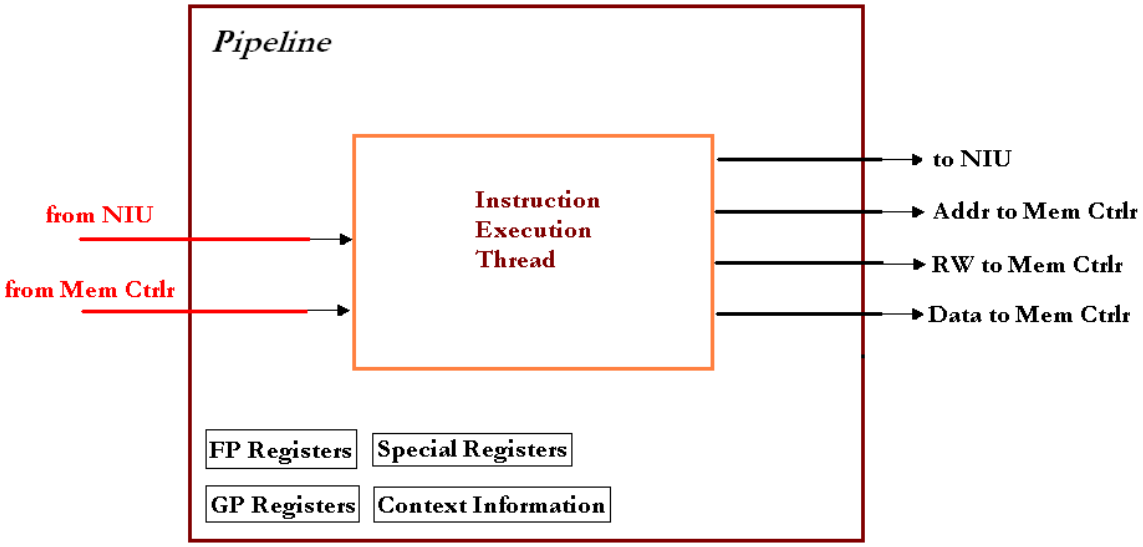


Figure 4.7: External Connections and Internal Structure of the UFM of the *Pipeline* Block.

The connectivity of this block is fairly straightforward. FIFOs are present for flit injection, flit ejection, memory loads, and stores. Both sets of registers are simply represented with *m* x *n* blocks of values, where *m* is the number of contexts that the node supports at a given time and *n* is the number of registers in each context. The context information is simply an array of *m* integers that store the IP of each context, with a special value indicating that the context is not ready for execution. A list of DCRs associated with the contexts is not necessary, since in this model all contexts' own both an IP and a set of registers, and the corresponding registers are the same index into the array of register values as their index into the array of IPs. The ATR is a special register, is used as an index into all data structures containing context information, and is updated in the event of a context switch. The 108 instructions in the ISA are implemented within a switch case. All functionality that can be directly incorporated into the module, such as arithmetic and logical operations, are modeled as such. When a context update instruction is encountered, the data structures containing the context information are modified appropriately.

4.1.4 Memory Controller

The memory controller is a rather simplistic entity in the UFM. Since no timing is present, tasks are carried out instantaneously, and this model is only functionally accurate, contention amongst the components competing for access to the memory is ignored. There are two separate threads for servicing requests from the pipeline and the NIU. Requests for memory queue up in the input FIFOs, the memory controller fetches the necessary data in order from the memory module, and returns the data to the requesting component, which will block, if necessary, on a FIFO read waiting for the memory controller to service its request. The memory module is essentially a container for a large array of 32-bit values, with wrapper functions and FIFOs necessary to serve out the data it contains to the one component that has direct access to it, the memory controller. The connectivity of this model of the memory controller and memory module is presented in figure 4.8.

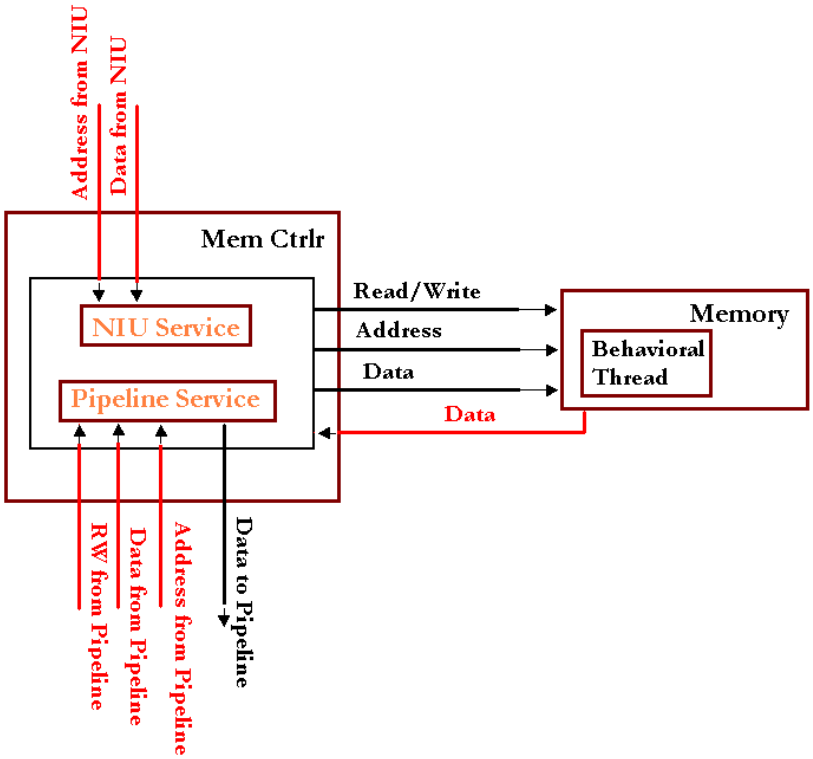


Figure 4.8: Block Diagram of the UFM of the Memory and Memory Controller.

4.2 Pin-Accurate, Cycle-Accurate Models

4.2.1 Heterogeneous Models

Upon completion of the UFM, it is time to begin incorporating timing into the models of the components. A primary feature of SystemC as an effective SDL is the support of heterogeneous models. This is to say that the architect can declare a system clock and synthesize one or more components from the UFM to a Pin- and Cycle-Accurate Model (PCAM) and leave all of the other components unchanged at their higher levels of abstraction. The UF blocks will complete tasks in delta time – instantaneously as far as the timed blocks are concerned – and can communicate with the refined blocks through either the previously used high-level communication mechanisms or the lower-level signals into which they will be transformed during this process. This powerful capability embodies the very definition of modularity of design, making the component-by-component refinement process of the node straightforward and simple.

One method of interfacing the UFM of one component with the initial version of the PCAM of another is to substitute unblocking FIFOs as communication channels and use wait statements for synchronization in the UFM. Since SC_METHODs, which always execute from beginning to end upon being woken up, are used to model the behavior of the PCAM of a component, a wait statement in the read call to a blocking FIFO in this module type will cause the simulation to stop and flag an error, so these channel types should be generally avoided. Another approach to interfacing a UFM with a PCAM is to use signals with wait statements in the UFM for synchronization. This is similar to having a non-blocking FIFO with a depth of one, so there must be buffers explicitly incorporated into the models if needed. In this project for most of the components, the blocking FIFOs in the UFMs were first substituted with non-blocking FIFOs, and then replaced with signals once the intermediate model's functionality was verified.

4.2.2 Model Refinement

As mentioned in a previous section, the functionality of a UFM block is comprised predominantly of thread processes running concurrently that are synchronized implicitly by blocking communication mechanisms. The first step in synthesis is to transform these threads into another class of SystemC processes called *methods*. Methods cannot stop in the middle of execution as threads do, waiting for data to come in through a FIFO or waiting on a semaphore to be signaled by another component's thread. Rather, a method is always executed from top to bottom whenever an event occurs on a signal appearing in its sensitivity list. Because information must be stored between method excitations, this implies that these methods must have state. All of the components designed in this project besides the memory (whose interaction with the rest of the system is governed by the memory controller – this is discussed further in the next section) are synchronous, meaning that their corresponding methods are only sensitive to the system clock and a reset signal. With each clock pulse, the stimuli that the module receives on its inputs govern the changes that will be incurred on its state and hence the behavior of its method on the next clock pulse. In addition to this refined model of functionality that more closely represents hardware components, the communication mechanisms need to also be modified to match that of the hardware.

In the world of hardware components, when one block needs to hand data off to another, it cannot simply place that data in a queue and count on it being there when the receiving component needs it unless such a mechanism has been explicitly designed. Taking up valuable chip space, resources, and time, queues should only be implemented when necessary. Although, it should be noted here that the FIFO channels can still be used when working with timed models. There are key times when the queue has a place in these models, especially when a timed block is communicating with another untimed entity, or the future incorporation of a FIFO in hardware is in the design plans. At any rate, for the PCAM, it is desirable to use a communication system much more closely to that of actual

hardware by requiring that components rely on request lines, data buses, and acknowledge signals to transfer data amongst each other. However, because the functional model already contains the major communication channels that are necessary for correct system behavior, the designing of these channels is, in most cases, simply a matter of transforming the already present FIFOs into their timed counterparts. The changes in request, acknowledge, and data signals necessary to move information from one place to another are incorporated into the state machines along with the functionality of the components.

Ensuring that a particular state of a component has the data it needs from all other components to complete its desired task can be a challenge, especially when the source of the desired data is currently experiencing contention amongst other components. This challenge presents the most difficulties when dealing with the memory controller, and is discussed further in the next section.

The following sections explain how the refinement process described above was followed to incorporate timing and structure into each component of the node, including the design aspects that needed to be considered at this level of abstraction and the challenges that were presented throughout. A block diagram of the PCAM of the node is shown in figure 4.9.

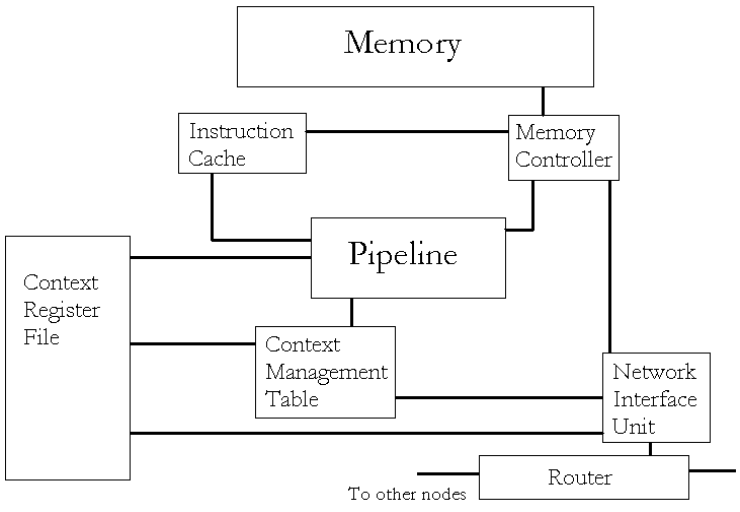


Figure 4.9: Block Diagram of the PCAM of the SCMP Node.

4.2.3 Networking Components

The design of both the NIU and the router involve rather advanced design considerations. The NIU must be equipped with DMA and access to the CMT for the efficient creation, injection, and ejection of data and thread messages. The router is a pipelined entity consisting of hardware support for virtual channels, a crossbar switch, routing mechanisms, and other subcomponents.

As mentioned previously, an optimized design of the router has been presented in a previous study. Because the router is a rather complex component, it remains only functionally accurate in the refined model of the node. This modeling decision has positive and negative implications on the simulation of SCMP applications. Two advantages are that the modeling efforts could be focused on blocks of the node that have not been studied in depth at this point, such as the pipeline, and also that simulations will execute in a significantly less amount of time. A disadvantage is that the execution time estimates gathered from the simulations will be skewed, since messages are passed through the network in delta time. In spite of this disadvantage, estimates of the additional time required to pass messages can be made and considered with the timing results of simulations until a more refined model of the router is created.

The NIU was implemented as two synchronous state machines: one for receiving flits being ejected from the network by the router, and another for injecting flits into the network through the router. The NIU is given access to read and write directly from memory. By implementing the NIU with DMA, the pipeline does not have to be stalled to read data from memory when data messages are being sent out to the network. The pipeline only has to provide the NIU with a starting address, count, and stride value, and the NIU can take care of memory reads autonomously while the pipeline continues its work. Similarly, when a data message is being received from another node, the pipeline need not be involved at all. The header and address flits of the data message indicate the starting address and offsets

needed to write to memory the data of each flit that is contained in the payload. Hence, no intervention of the pipeline is needed. Figures 4.10 and 4.11 illustrate the ejection and injection state machine designs for the NIU.

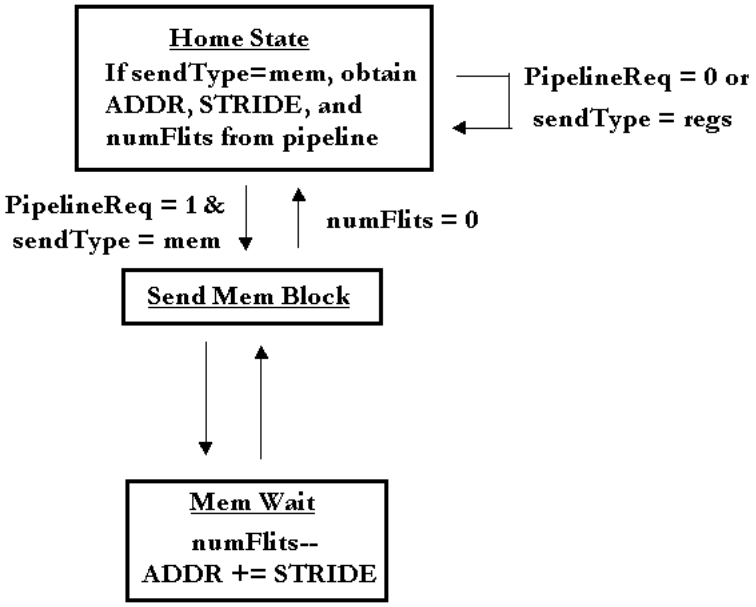


Figure 4.10: Diagram of the State Machine for Message Injection from the NIU.

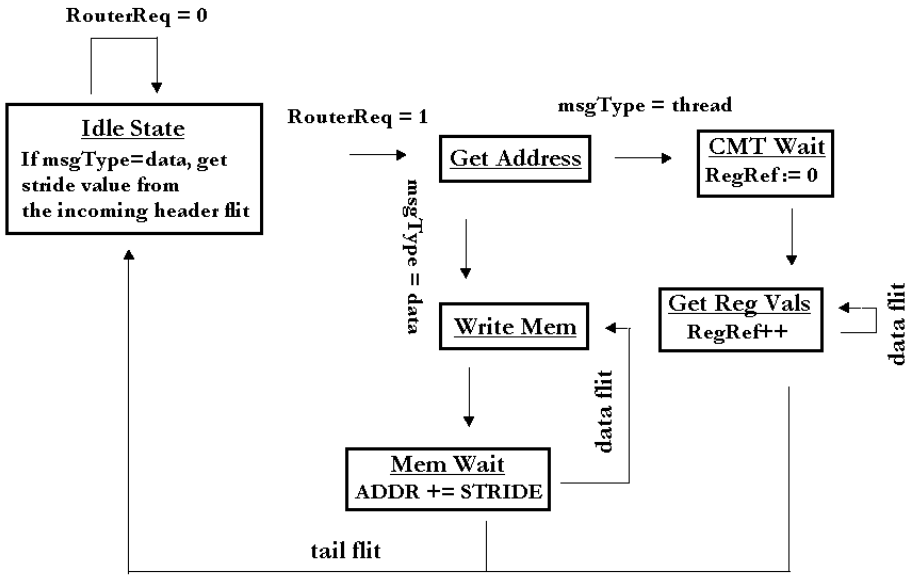


Figure 4.11: Diagram of the State Machine for Message Ejection through the NIU.

4.2.4 Context Management

As mentioned previously, the context register file and thread context data were simply stored as data structures in the pipeline in the UFM. The thread context data is a block of information containing which of the sixteen contexts is currently allocated and the associated IP addresses that should be used. The context register file contains the corresponding integer and floating point registers associated with these contexts.

In the PCAM, this data needs to be extracted from these structures in the pipeline and placed into autonomous components whose data can only be retrieved through synchronous communication with these components. The task of moving this data from the pipeline to separate components is a fairly simple one, requiring that the pipeline make requests to obtain desired information rather than having it readily available.

However, in the system diagram at the beginning of this section, it is evident that the NIU also needs access to both the CMT and context register file. The reason for this is that when thread messages are received from the network, both instruction pointer addresses and register values are contained within these messages. In the previous model, this data was received and simply passed from the NIU to the pipeline where it was stored appropriately in the structures containing the context information. Because both the pipeline and the NIU will now be making requests to the new components, there must be some arbitration within these blocks' state machines.

There are two ways to deal with this problem, and while both will provide accurate functionality, the decision will have an affect on the accuracy of the performance estimations. The first and simpler approach is to assume that the blocks can service more than one request in a single clock cycle. In this case, one of two assumptions is made. First, that the access logic must be able to make multiple reads/writes to the storage block at a time, in which case it must be designed well enough to deal with simultaneous accesses to

the same locations. Or second, that the access logic is fast enough to perform more than one access to the storage block. For designing the CMT and the CRF components, the second assumption is made and they are designed to deal with both pipeline and NIU accesses in a single clock cycle. Whether or not this assumption is valid will be determined with further study of the timing details of lower-level hardware implementations of these components.

The second and more difficult way to deal with the contention problem is to only service one incoming request at a time, flagging the requests from the components waiting to be serviced and determining a priority scheme for the contending requests. Although this method is not used with these components, this approach is necessary when dealing with multiple memory requests, and will be discussed further in the following section that presents the design of the memory controller.

Because the CMT and CRF simply provide access to the data stored within and it is assumed that multiple requests for this data can be serviced in one clock cycle, there is no need to design a state machine for these components. However, the interaction amongst the NIU, CMT, CRF, and pipeline when receiving a new thread message from the network was a key design aspect of this project. The orchestration of this functionality required a significant amount of time and design effort, and a flow chart illustrating the steps is shown below.

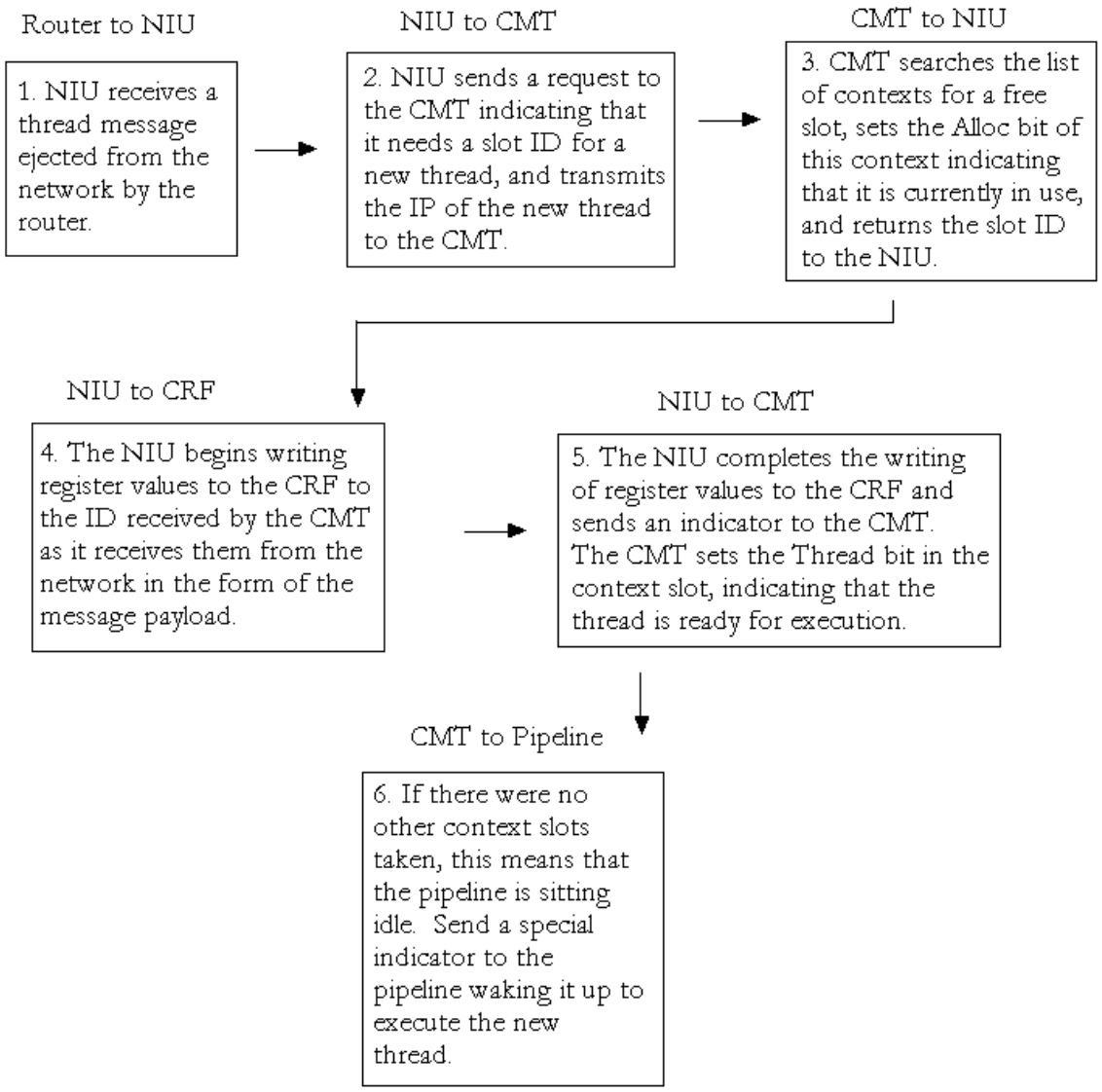


Figure 4.12: Steps in the Node's Thread Message Reception Process.

4.2.5 Memory Controller and Instruction Cache

The memory controller governs all memory accesses to components of the node, including the pipeline, NIU, and the instruction cache. Because memory accesses are assumed to be very fast due to the integration of the memory directly on chip, no data cache is present.

The instruction cache is available to provide instructions to the IF stage of the pipeline to lessen the amount of memory contention due to instruction fetches and loads and stores. Efficient designs of these two components will undoubtedly play a vital role in improving overall system performance.

The pipeline sends the address of a desired instruction to the icache, and the icache in turn searches for the address within its storage. If the address is found, the corresponding data is returned. Otherwise, the pipeline must be stalled to wait for the memory controller to service a request from the icache. The optimization of the icache parameters is not a focus of this study, as much work is available on the evaluation of performance while varying cache configurations [18]. Previously determined values were used in the model for cache size, block size, and associativity, but these values can easily be modified for performance evaluation purposes if desired. The functionality of the icache is fairly simple if the underlying concepts of caches are well understood. The state diagram of the icache is shown below in figure 4.13.

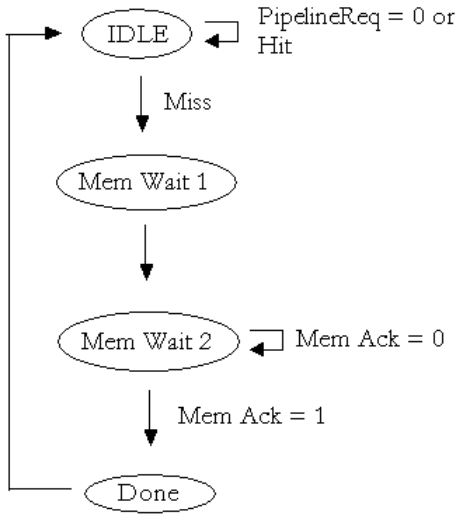


Figure 4.13: Diagram of the Icache State Machine Design.

The memory controller is one of the more complex components of the SCMP node. Because the memory controller is the owner of data that multiple components need to access, it is the memory controller's job to govern this contention. As mentioned in the last section, creating the relatively simple model through the assumption that more than one of these requests can be serviced within one clock cycle is not an option. Although there may be memory components that allow for this performance such as DDR, the amount of time that memory reads and writes take place in the PCAM is already very small and so these designs are not considered.

Because, in the model, the memory controller's access to memory happens asynchronously and instantaneously (all delays in memory accesses are currently incorporated into the memory controller), there is no need to design a state machine for the controller. However, the complexity arises in that three different components contending for access to memory means that there are three sets of requests, data, addresses, memory commands (specifying the read/write data size), and other signal lines coming in, and three sets of acknowledge and data signals going back out. The designer must assert that the input lines of a particular component are read on the precise clock cycle that the controller will service that request. All other input lines must be ignored. The correct acknowledge signal and data lines must be set, while all other acknowledgement lines are lowered and all other data output lines left unwritten. Additionally, if more than one request comes in at a time, the controller must raise a flag indicating that another component is waiting, and service the requests with correct priority. Although the algorithm is fairly straightforward, the design of such a controller requires the careful study of timing diagrams to ensure that components receive their corresponding data correctly and at the time at which they are acknowledged. An algorithm illustrating the design of the memory controller in the PCAM is listed below.

0. Set inService := -1. Set all ack lines low.
1. Wait for the clock to go high. Set all ack lines low. If inService = -1, go to step 2. Otherwise, go to step 10.
2. If pipelineReq = true & NIUWaiting = false & icachewaiting = false & inService = -1, then go to step 3. Otherwise, go to step 4.
3. Set address := pipelineAddr. Set the inService := 0. If pipelineReadWrite = true, set memDataIn := pipelineDataIn, raise pipelineAck, and assert that all other ack lines are low. Go to step 10.
4. If (NIUReq=true & ICacheWaiting=false) | NIUWaiting=true, go to step 5. Otherwise, go to step 8.
5. If inService!=-1, set NIUWaiting=true and go to step 10. Otherwise, go to step 6.
6. If NIUReadWrite = true, set memDataIn := NIUDataIn, raise NIUAck and assert that all other ack lines are low. inService := 1. Go to step 10.
7. If ICacheReq=true | ICacheWaiting=true, got to step 8. Otherwise, go to step 10.
8. If inService !=-1, set ICacheWaiting=true and go to step 10. Otherwise, go to step 9.
9. inService := 2. Go to step 10.
10. If memAck=true, go to step 11. Otherwise, go to step 1.
11. If inService = 0, retrieve data from memory and set pipelineDataOut := dataFromMem. Raise PipelineAck and assert that all other ack lines are low. Go to step 14. Otherwise, go to step 12.
12. If inService = 1, retrieve data from memory and set NIUDataOut := dataFromMem. Raise NIUAck and assert that all other ack lines are low. Go to step 14. Otherwise, go to step 13.
13. Retrieve data from memory and set icacheDataOut := dataFromMem. Raise IcacheAck and assert that all other ack lines are low. Go to step 14.
14. Set inService = -1. Go to step 1.

Figure 4.14: Memory Controller's Algorithm for Servicing Incoming Requests.

4.2.6 Pipeline

The subsystem of the timed models whose design presented the most difficulties was the pipeline. For this reason and also due the significant increase in simulation time upon its completion, the first timed model in this work actually leaves the pipeline as only a functionally accurate block. Although a model with only a functionally accurate pipeline can

allow for fast simulations and may provide useful estimates of certain performance metrics, many architectural optimizations and design challenges are difficult or even impossible to consider without a model including a cycle-accurate version of the pipeline. Many unexpected challenges arose in the development of the pipeline in addition to the classic, fundamental problems that are inevitably encountered with ILP-extraction. A great deal of reference material is available that discusses most of these problems, their solutions, and designs of pipelines in general [42] [33] [45], so the focus of this section will be primarily the issues that are specific to SCMP such as the specific structure of the stages and the difficulties that arise with the stages' interaction with the other components on the node. Each stage is presented with a discussion of its function and the main problems encountered in its design.

Because the stages generally perform the same task on each cycle, no state machine design is needed for the individual stages. Generally speaking, a given stage does not need to have information about what occurred during the last clock cycle, unless the pipeline is being stalled. Rather, the pipeline itself is sort of a distributed state machine, and the state information is transferred from one stage to the next through registers. In the PCAM, these registers were modeled with FIFOs. This is an excellent example in which a timed model uses an abstract, untimed communication channel for representing a mechanism other than lower-level communication signals, illustrating one of the many powerful features of an effective SDL.

Most architects possessing any background in pipelining are familiar with the classic, five-stage MIPS pipeline consisting of the IF, ID, EX, MEM, and WB stages. The SCMP pipeline is similar to this configuration in many ways, but an important note should be made as the IF stage is presented regarding a point of confusion that may arise with the names of the stages. These names may be misleading in some aspects, a result of the fact that one clock cycle is not generally sufficient to perform the necessary interactions with the components with which a stage is working. Consider the fetching of an instruction, for example. In figure 4.15 showing the IF stage's I/O pins, notice that there is no request line

to the instruction cache. However, the WB stage's I/O pins, shown in Figure 4.19, do include this line as well as address lines to the icache. In a simple model of the pipeline, the WB stage is only assigned the task of storing the results of an operation back to the register file, a task which does not necessitate interaction with the instruction cache at all. However, SCMP's instruction cache needs at least half of a clock cycle to return the requested instruction back to the IF stage. Since the instruction cache is active high and the IF stage needs the instruction to pass off to the ID stage before the rise of the next clock, the request signal and IP must be given to the instruction cache prior to the previous rising clock edge. Hence, the task of setting up these signals is assigned to the WB stage. This is a recurring theme amongst the design of the stages, as similar pin assignment issues will arise throughout this section.

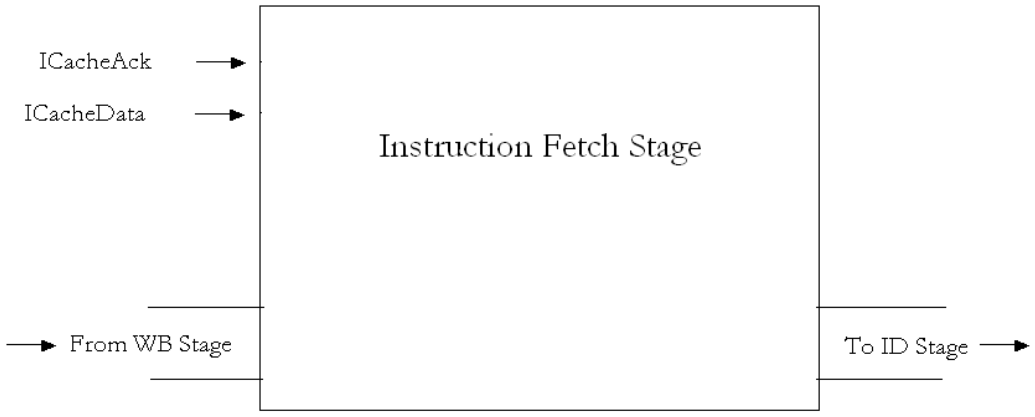


Figure 4.15: I/O of the Instruction Fetch Stage of the pipeline.

The Instruction Fetch stage is essentially a stage that constitutes a stall in the pipeline while waiting for the latency of the instruction cache. Its functionality simply entails receiving the instruction word and handing it off to the ID stage. Every clock cycle during which the pipeline is not being flushed and there is no stall in a following stage, the IF stage will ideally be receiving an ack from the instruction cache on every cycle. Since cache misses inevitably occur, the IF stage must be prepared for the case in which the ack is not received. Icache misses generally incur an eight-clock stall penalty while the icache retrieves the block

containing the desired instruction word from memory. This penalty may be increased if the memory controller is busy servicing another component with higher priority.

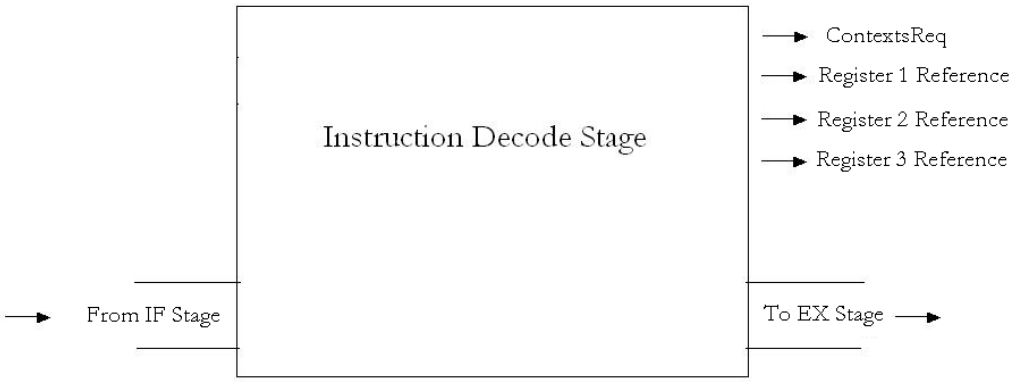


Figure 4.16: I/O of the Instruction Decode Stage of the pipeline.

The Instruction Decode stage receives the instruction word from the IF stage. It splits up the instruction into its parts: register references, immediate values, store address displacements, etc. It sends a request to the CRF for the registers specified in the word. Although another component or stage may currently be using the CRF, the CRF is assumed to be able to service multiple requests in a single clock, as mentioned in section 4.2.4. Hence, no stalls should occur due to contention with the CRF. However, the ID stage is responsible for checking for data hazards in its current instruction and the instructions ahead of it in the pipeline. If the destination register of any instructions that have not yet reached WB matches either of the source registers of ID's current instruction, the pipeline must be stalled while the results of the prior operation are written to the CRF. In its most recent model, forwarding has not yet been incorporated into the pipeline.

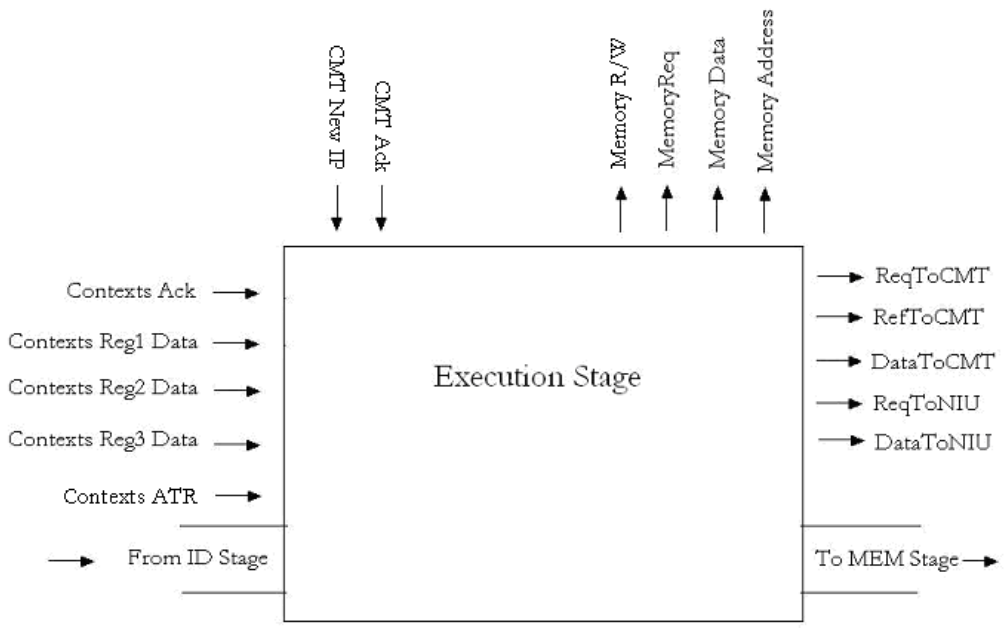


Figure 4.17: I/O of the Execution Stage of the pipeline.

The Execution stage has many responsibilities that take place upon the execution of different types of instructions. In order to execute these instructions, it needs the register data that was requested from the CRF in the ID stage, and the instruction word that it receives from the incoming FIFO. The ALU and FPU functionality are currently embedded within the EX stage, and so the interface to them is not shown here. These additions to the model require very little modification and are essentially insignificant at this point in the design. If a store or load instruction is being executed, the necessary signals are set up to the memory controller to read or write from memory. If the SUSPEND, END, ALLOC, FREEC, or other context-related instruction is being executed, the appropriate command is sent to the CMT. Finally, if a thread or data message needs to be sent, the necessary signals are set up to the NIU. The IP and register references are sent through the output FIFO to the MEM stage.

The input signals from the CMT are used when the pipeline is being flushed in the case of a context switch. Similar methods are used in the event of a taken branch, a subroutine call, or a context switch when the pipeline needs to be flushed to ensure that the next instruction

executed is actually the first instruction after the branch or the first instruction of the new context. The current IP value during regular functioning of the pipeline is passed from one stage to the next through the pipeline registers since it is frequently needed, such as in the case of a branch instruction whose destination is relative to the IP's current value, for example. When the pipeline needs to be flushed due to a branch or subroutine call, the first IP of the destination is known since it is calculated in the EX stage. On the other hand, when a context switch is occurring, the IP of the first instruction in the new context is obtained from the CMT via the input signals from that component. All instructions that come into the EX stage are ignored until the destination IP that is pending is reached. Inevitably, the destination IP is sometimes encountered before the first instruction that should actually be executed is encountered. Recall that the address space of SCMP is at maximum eight megabytes, while the IP value is a 32-bit value, meaning that there are nine bits in the IP that aren't being used. When the new IP is initially injected into the first pipeline register from the WB stage, bit thirty-one is set to indicate that it is the IP of the new context or branch destination. This approach to pipeline flushing has its disadvantages, since the IF and ID stages are working on instructions that won't be used, but optimizations can be made on future designs if desired.

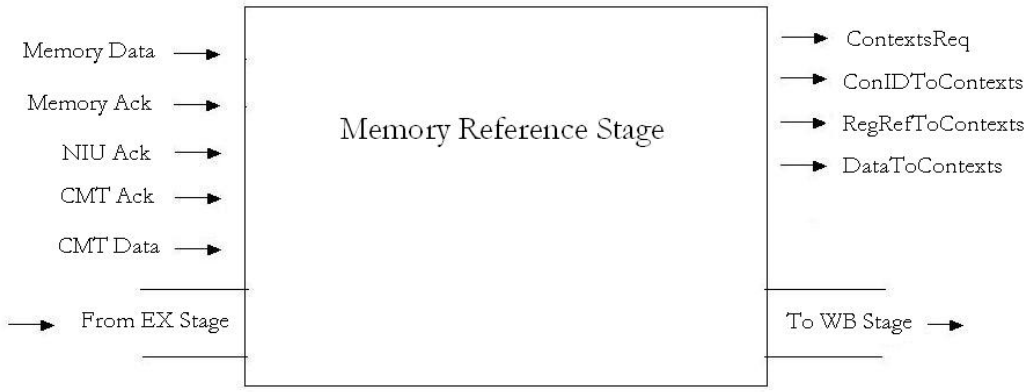


Figure 4.18: I/O of the Memory Reference Stage of the pipeline.

The Memory Reference stage needs to be able to receive data from the memory controller that was requested in the EX stage, so the necessary acknowledgement and data lines are

provided. If an NIU request was made, the ack signal is coming back to assert that the operation was completed successfully. It is possible that the NIU may be busy with another task, or that a buffer in the NIU may be full. If the ack signal does not come back after a send instruction, the MEM stage will stall the pipeline. If a context switch is occurring, the MEM stage must be able to receive the new IP from the CMT, which is then sent through the FIFO to the WB stage. The lines that are needed to write the results that were received, if any, back to the register file are provided.

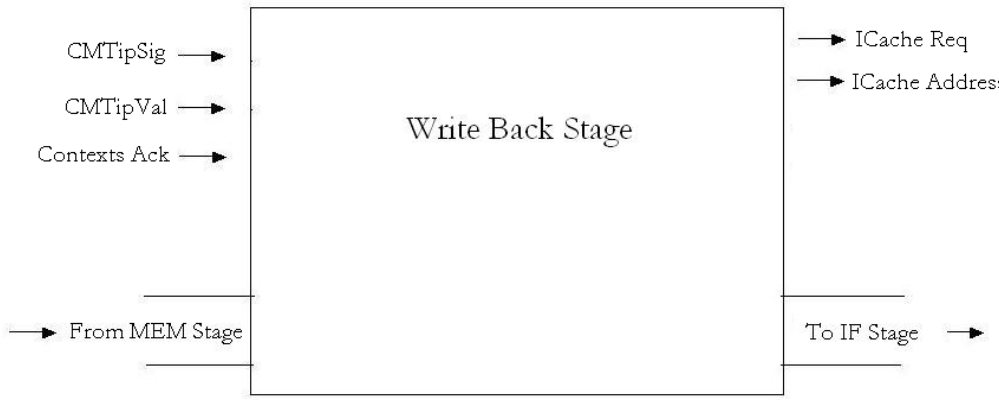


Figure 4.19: I/O of the Write Back Stage of the pipeline.

Because the Write Back stage is in charge of setting up the instruction retrieval lines to the icache, it is the job of this stage to wake up the pipeline when it is idle. The CMT recognizes that a thread message that has been received is the only one available, and notifies the pipeline at the WB stage accordingly. The signals to the icache are then set up, as they are when the pipeline is executing instructions during a normal operation.

4.3 Profiling Hardware Performance

Following the development and verification of the models, it is time to begin using them to obtain performance numbers that can be used to uncover bottlenecks and reveal potential

optimizations for the design. The first metric that is absolutely necessary since the network is only functionally accurate and the timing of message transit time must be incorporated into the execution times in some fashion is the number of messages passed through the network. It may also be useful to see how the number of messages passed varies with problem size for different benchmark applications to understand how significantly lower latency or higher bandwidth goals in the designs of the networking components will affect overall performance. This metric is incorporated into both the untimed and timed models.

Another detriment to performance in addition to message latency and bandwidth limitations is the occurrence of components sitting idle while waiting for a response from another block due to contention amongst the node for that block, also known as a structural hazard. The primary service for which components must wait due to contention is the memory controller. A useful metric for performance evaluation purposes is how the number of clock cycles that the NIU and icache must remain idle while waiting on memory requests varies with problem size for various benchmarks. Recall that the number of pipeline stalls while waiting on loads and stores should not be significant, since it has highest priority for memory requests. These numbers can only be taken from the timed models, since there is no useful way of measuring the idleness of a component in the untimed models. Some of these results were gathered from benchmark runs and are presented graphically in the next chapter.

4.4 Memory Usage Evaluation

A useful application of these models is for the use of gathering performance data for the exploration of possible design modifications that will result in an overall decrease in power consumption during the execution of typical applications. Previous work has shown that the embedded memory on SCMP is a primary contributor to energy consumption of the system, and much research has been contributed to lowering the power usage of embedded memory

architectures. One technique in particular extends the performance benefits of multi-bank memory systems by minimizing a cost function through a compiler-directed approach involving the choosing of an optimal execution strategy through three tasks: variable assignment to the banks, scheduling of memory bank accesses, and switching of the banks among operating modes [36]. The operating modes have associated power consumption costs, with lower power consumption costs implying more limitations on functionality. Minimizing the cost function for various applications to produce optimal variable assignments, access schedules, and bank operation modes could be quite useful in the design of a high-performance, low-power memory system for SCMP. Unfortunately, such a task is fairly rigorous, would certainly be a joint effort among the compiler designers, the OS developers, and the architects, and is a possibility for future work with SCMP. A valuable contribution to this work for the consideration of its feasibility is the development of models that generate a profile of memory usage for benchmark applications. This contribution is an objective of this thesis work, and the models are used for gathering such data from benchmark runs in the next chapter. Within the models at compile-time, the designer is able to specify memory bank sizes, as well as a time resolution for how frequently to reset the usage data for each bank. Once these parameters are specified, the model can produce, at run-time, a file of data regarding how frequently each unit of memory was accessed for every block of time. This data file can then be used to produce a graphical representation in three-dimensions – time, numbers of accesses, and bank number – of the memory usage profile. The architect and compiler designer can then use these graphics to assist in the optimization of the cost function and the memory architecture for achieving maximum performance and minimum power consumption.

Chapter 5

Performance Evaluation

This chapter presents the results gathered from the various metrics that were incorporated into the models. The first section explains how benchmark applications are developed for the SCMP computer using the C compiler that was developed by Sidney Bennett [6]. The second section gives a brief explanation of the benchmark that was implemented by the author. The final section provides the data in graphical form that was observed from numerous runs of the benchmark on each of the models.

5.1 Application Development for SCMP

Given that a previous member of the SCMP team has developed a C compiler that produces executable files using the SCMP instruction set [6], the development of applications for the SCMP system is very straightforward. To actually develop programs that yield the parallel execution of threads, the programmer writes directives into the C code to indicate to the compiler which functions can be used to create thread messages. A library of function calls has been developed for common tasks such as the passing of data and thread messages, obtaining node ID's, and suspending the execution of threads, to name a few.

Loading the executable into the model is as simple as parsing the file and copying the data into the memory block for retrieval by the components in the node. A thread message with the IP of the entry point is then manually injected into the network to node 0, and the program begins execution.

5.2 Floyd's Algorithm

Floyd's algorithm provides a solution to Transitive Closure, also known as the Shortest-Paths problem. Given an adjacency matrix constructed from a directed graph containing N vertices and E weighted edges among them, the shortest distance from any of the vertices to each of the others can be found. The sequential algorithm is of $O(n^3)$ time complexity:

```
for k = 0 to N-1
    for i = 0 to N-1
        for j = 0 to N-1
            a[i,j] = min(a[i,j],a[i,k]+a[k,j])
```

Floyd's Algorithm [43].

For each of the N^2 pairs of vertices i and j, each of the N values for k is considered, comparing the current shortest path's distance between i and j to the distance from the path from i to k plus the distance from k to j. In other words, each iteration of the inner loop is asking: *Is the current path from i to j longer than the path to j through point k?*. At first glance, the algorithm looks as if it cannot be parallelized, because Bernstein's conditions state that two operations cannot be performed simultaneously if the input of one of the operations, in this case the *min* function, intersects with the output of the other. If all N^2 iterations of the inner two loops for a given node are executed in parallel at some point in time, the a[i,k] and a[k,j] elements will, in all likelihood, be updated at the same time that a[i,j] is being updated, which

is a violation of Bernstein's conditions. However, there is proof that the a[i,k] and a[k,j] elements will not change during the following update:

$$a[i,k] = \min(a[i,k], a[i,k]+a[k,k])$$
$$a[k,j] = \min(a[k,j], a[k,k]+a[k,j])$$

These are the assignments being made to the elements in the *k*th row and the *k*th column of the adjacency matrix during the *k*th iteration of the inner two loops. Since the distance from any vertex to itself is zero, these assignments will always yield no change in the previous values of a[i,k] and a[k,j]. Therefore, the inner two loops may be executed in parallel for a given value of *k* [43].

To parallelize this algorithm, each node must have a copy of the data it needs to perform the *min* operation; namely, the values from the *k*th column for all rows in the adjacency matrix that it owns, and all values from the *k*th row for all columns in the adjacency matrix that it owns. To keep the algorithm simple, the nodes can be synchronized at the beginning of each iteration of the outermost loop. If this synchronization is not present, there needs to be some other mechanism that asserts that the nodes making updates with the new value of *k* do not receive other nodes' elements that have not been updated yet, and that those nodes are not changing the elements contained in the old rows and columns corresponding to the previous value of *k*.

In row-wise data decomposition, no communication of the *k*th column takes place, because each node owns all the values in its row, including those of the *k*th column. However, the disadvantage to this approach is that on every one of the *k* iterations of the outermost loop, one node owns all the values of the *k*th row, and so that node must make *p*-1 transmissions of size *n* to every other node in the system. This node becomes a hot spot, which can be very detrimental to performance on the SCMP system. The problem can be curtailed significantly by using checkerboard decomposition. If there are *x* processors in the x-dimension of the mesh, and *y* processors in the *y*-dimension of the mesh, *x* processors must

make y-1 transmissions of size n/x, and y processors must make x-1 transmissions of size n/y. If the number of processors is a perfect square, $x = y = \text{sqrt}(p)$, then $2*(\text{sqrt}(p)-1)*\text{sqrt}(p)$ transmissions must be made on each iteration of the outermost loop. This is a higher number of messages than are sent with row-wise decomposition, but the greater latency costs are most likely negligible compared to the time saved by distributing those transmissions over more processors and hence avoiding hot spots in the network.

For this project, only Floyd's Algorithm with row-wise decomposition has been developed for the SCMP system. Below is an algorithmic description of the kth row broadcast and parallel shortest paths function instantiation used in this implementation. The code for this algorithm is run on the processor that owns the kth row. The method used may not seem straightforward, as the broadcasting process sums up the array of synchronization points repeatedly. However, since the SCMP system will only use sixty-four processors at most, this is a fairly cheap operation. The reason the summing is done is so that all shortest paths function instantiations can happen virtually at once, and the instantiating process can then wait for the all of the processors to check back in by clearing their corresponding synchronization points. This way, the execution of the inner loops will occur as fast as the slowest processor, plus the small amount of time that it takes for the instantiating process to sum up the synchronization points.

0. Allocate *p* integers for *sync* array
1. Allocate *p* integers for *addr* array
2. Load *Krow* array with the data in the *k*th row
3. for i = 0 to *p*-1 do steps 4-8
    4. *sync*[*i*] := 1
    5. Initialize *sync2* on the *i*th processor to 0
    6. Send the address of *sync*[*i*] to the *i*th processor
    7. Send the address of *addr*[*i*] to the *i*th processor
    8. Set *sync2* on the *i*th processor to 1
9. end for
10. for all *p* processes, create a thread that does the following:
11. When the *i*th process has received *sync2* = 1, it will send (to this processor) its address of the location of space to store the data for *Krow* to *addr*[*i*], and then send the value 2 to *sync*[*i*]
12. *sum* := 1
13. while *sum* /= 0 do steps 14-25
    14. for i = 0 to *p*-1 do steps 15-24
        15. if *sync*[*i*] = 2 do steps 16-22
            16. if *i* /= myid do step 17
                17. Send data in *Krow* to *addr*[*i*]
            18. end if
            19. *sync*[i] := 3
            20. Set *sync2* on the *i*th processor to 1
            21. Create a thread on the *i*th processor that does the following when it receives *sync2* = 1:
            22. Calculates the shortest paths for all its vertices, and sets *sync*[*i*] on this processor to 0
        23. end if
        24. *sum* += *sync*[*i*]
    25. end for
26. end while
27. Set main synchronization variable to 1

Figure 5.1: Broadcast and Thread Creation Algorithm for Owner of *k*th Row.

5.3 Results

This section includes summaries and graphical presentations of quantitative data gathered relevant to the metrics presented in the previous chapter, including measurements of network traffic, memory access contention, and profiles of memory usage. The metrics are gathered from runs of the Transitive Closure benchmark (TCB) to observe possible relationships and trends between the data and numbers of nodes, problems sizes, and other parameters.

5.3.1 Simulation Times

The first set of measurements, presented in the charts below, reveals the significant amount of variance among the simulation times of the application with different size adjacency matrices on the models. The drastic difference in these times illustrates the computational costs of including the timing and structural details within a model and one of the many key benefits of performance evaluation at higher levels of abstraction. In the untimed functional model, simulation times are brief enough for the analyst to await completion, even for larger problem sizes with many nodes. The next model, referred to for the remainder of the section as *PCACC1*, is accurate with respect to timing and structure for all components except for the pipeline, router, and internodal connections. The pipeline's interface was modified for connections to the other components, but its functionality remains the same as in the UFM in which the execution of all instructions completes in delta time. The final model, *PCACC2*, includes the cycle-accurate pipeline. From these charts, it is apparent that if timing need not be considered for a certain phase of analysis, there is a substantial benefit to utilizing the UFM. Likewise, if PCACC1 proves to suffice for gathering estimates of some performance metrics, its use will be preferred over its refined counterpart. Results that follow in this section provide comparisons of sets of performance numbers recorded from the different models.

Node Configuration: 2x2 Grid			
Size of Adjacency Matrix (vtcs)	Execution Times (min : s)		
$\log_2 n$	UFM	PCACC1	PCACC2
3	:00	:02	:08
4	:01	:06	:15
5	:03	:20	:45
6	:18	1:54	6:27
7	2:07	13:56	28:52
8	18:16	156:12	-

Table 5.1: Execution Times for TCB on Three Models of a 2x2 Grid.

Node Configuration: 4x4 Grid			
Size of Adjacency Matrix (vtcs)	Execution Times (min : s)		
$\log_2 n$	UFM	PCACC1	PCACC2
3	:01	:19	1:00
4	:02	:32	1:39
5	:05	1:06	3:27
6	:21	3:24	24:32
7	2:13	20:57	53:47
8	21:51	-	-

Table 5.2: Execution Times for TCB on Three Models of a 4x4 Grid.

Node Configuration: 8x8 Grid		
Size of Adjacency Matrix (vtcs)	Execution Times (min : s)	
$\log_2 n$	UFM	PCACC1
3	:07	12:00
4	:09	19:08
5	:15	35:40
6	:40	74:15
7	2:42	-
8	27:23	-

Table 5.3: Execution Times for TCB on Two Models of an 8x8 Grid.

5.3.2 Network Activity

With any set of models, an important reciprocal relationship exists between the high-level, abstract representations of the system and those that are more refined. One element of this reciprocity revealing the importance of the abstract models to the refined ones is verification. An abstract model is functionally accurate but inherently less complex, and so it is consequently easier to verify. Once this model has been verified, certain features of it can then, in turn, be used to verify the functionality of the refined models. An example of this type of verification is illustrated in the next set of results.

The number of messages passed through the network versus problem size for different node configurations is an important metric for estimating how tradeoffs in network performance will affect the performance of the overall system. More importantly for this project in particular is that this metric for many applications should not vary from the abstract models to the refined models. In the Transitive Closure benchmark, for instance, the number of messages passed for a given application is a function of only the implementation of the

algorithm and problem size, and not on the timing characteristics of the components that will only be known at run-time. Hence, it is most practical to run simulations for the purpose of observing trends with respect to this metric on the UFM, since simulations complete much more quickly and, in the case that a need for design modifications arise, they can be more easily addressed with easily verified solutions.

Verification of the refined models' functionality through numbers of messages passed was carried out for different numbers of nodes with varying problem sizes. These numbers were found to be identical to the abstract models' recorded values in all cases, offering further assurance that the detailed models were indeed functionally accurate. Figure 5.2 below illustrates the closeness in values for the data gathered from two models of an 8x8 node configuration. Note that the lines representing the data taken from the different models overlap.

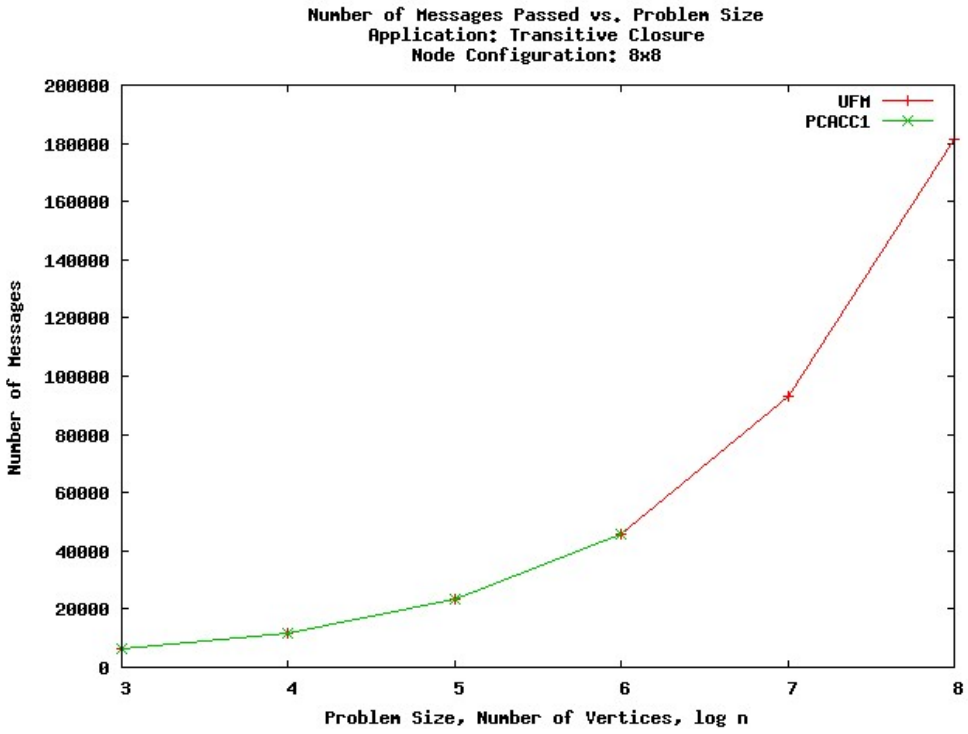


Figure 5.2: Number of Messages Passed vs. Problem Size for Two Models of an 8x8 Grid Running TCB.

5.3.3 Memory Contention

The other element of the reciprocal relationship between the abstract and refined models illustrating the importance of the more detailed models to those that are less detailed is the confirmation of sufficient accuracy. In general, whenever an abstract model provides sufficient accuracy for a particular phase of design evaluation, it should be utilized. However, the analyst must assert that the abstract model does indeed provide sufficient accuracy. This confirmation comes as a result of iterative comparisons of the metrics obtained from the refined and abstract models. Once the analyst is satisfied that the high-level model provides sufficient accuracy for a particular metric, the model can then be utilized for further evaluation. An instance of this type of comparison is made here as the number of NIU stalls due to memory contention versus problem size is considered. As discussed in chapter 4, the NIU is given a low priority in the memory controller's arbitration logic that governs the contention for contents of the memory. Since both PCACC1 and PCACC2 contain timed models of the memory controller and the NIU, the total number of clock cycles that the NIUs of all nodes must wait due to memory contention can be observed. However, as the graphs below reveal, the accuracy of the numbers taken from PCACC1 vary a great deal from those seen in PCACC2, especially when problems size is increased. Hence, simulations on the less refined model are not useful for evaluations with respect to this particular performance metric.

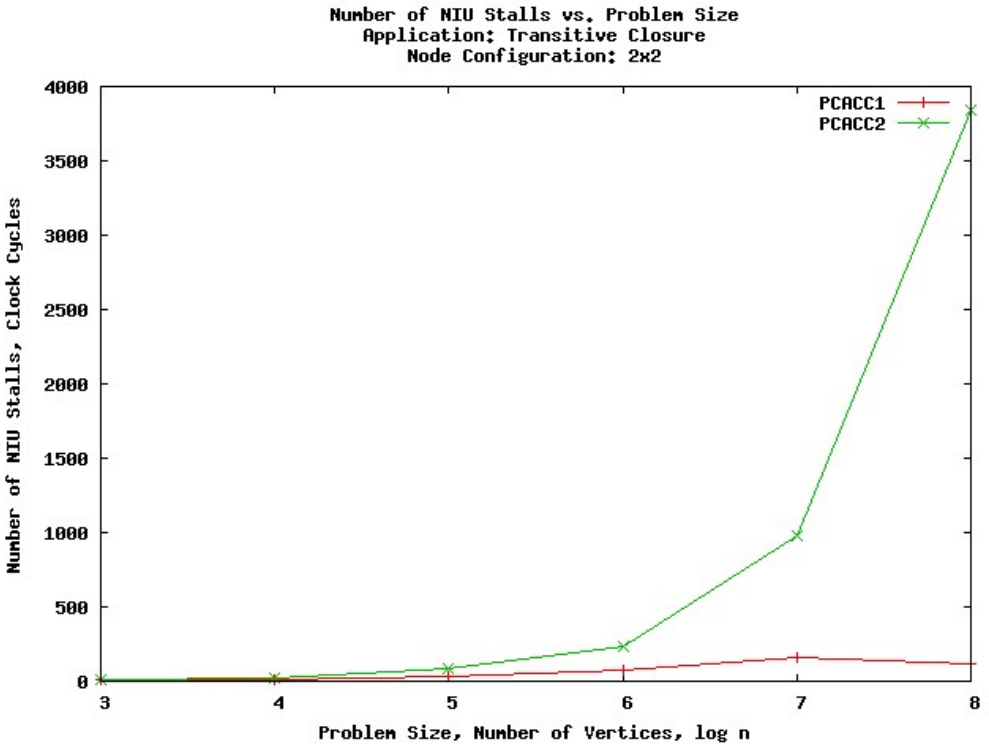


Figure 5.3: Number of NIU Stalls vs. Problem Size for Two Models of a 2x2 Grid Running TCB.

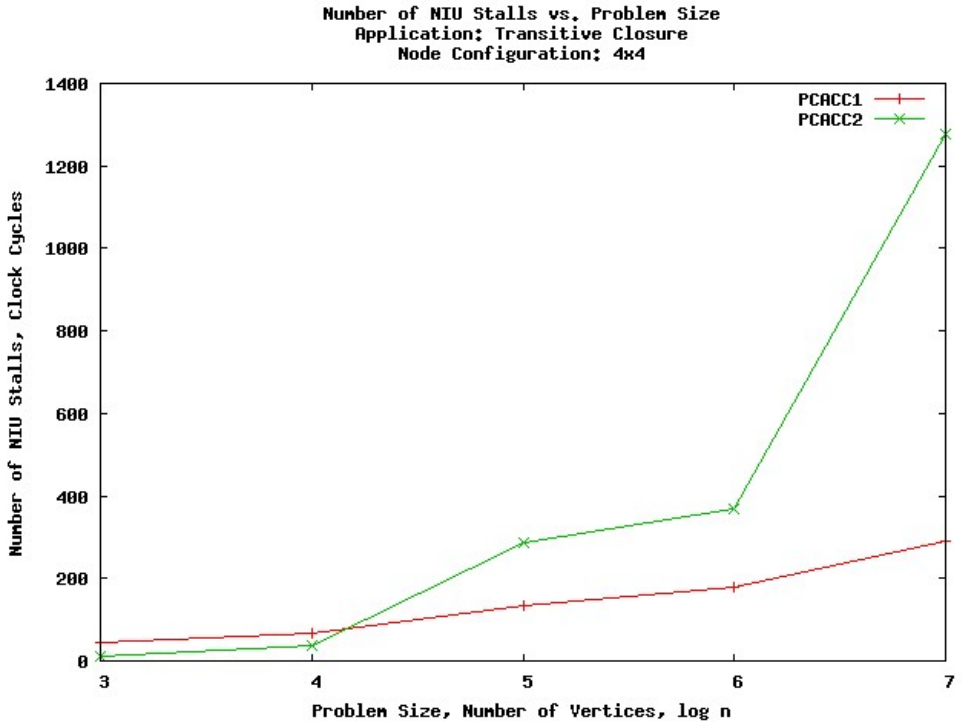


Figure 5.4: Number of NIU Stalls vs. Problem Size for Two Models of a 4x4 Grid Running TCB.

5.3.4 Memory Usage Profiles

Tools have been incorporated into PCACC2 for estimating the amount of potential power benefits available from the use of a partitioned memory architecture, as discussed in the previous chapter. In the Transitive Closure benchmark with row-wise decomposition, the nodes can fall into one or two of three sets. The first set contains only node 0, the home node, which is responsible for managing the distribution of the initial matrix values and collecting the results at the end of execution. The second set contains the current owner of the *k*th row from which all nodes must receive these values for the updating of the rows that they own. The last set contains the nodes that are not the current owner of the *k*th row.

The figures below present the data recorded showing the memory accesses during two different phases of the application for nodes falling into the different sets listed above. The initial task of partitioning the adjacency matrix and passing them to their owner takes place in the distribution phase. As shown in figure 5.5, the home node accesses each group of rows in the array for distribution during this phase. If each of these chunks of memory were divided into banks, there would be no need to have banks seven through fifteen activated throughout the execution of this phase. For instance, banks fourteen and fifteen have no activity at all until the final rows are distributed. As the next figures are considered, results appear even more promising. In fact, it seems that during the distribution of the initial matrix and only for the home node do sporadic bank accesses requiring a substantial amount of bank state switching occur. Consider node 1, a typical node that simply sits idle while waiting for its rows to arrive during the distribution phase. By the data given in the profile, it appears that the entire memory system is not being used at all during most of this phase, except for early on when it is receiving its rows. The large number of accesses at the tail end indicates that the node has begun fetching instructions as it transitions into the next phase of execution, the solution phase.

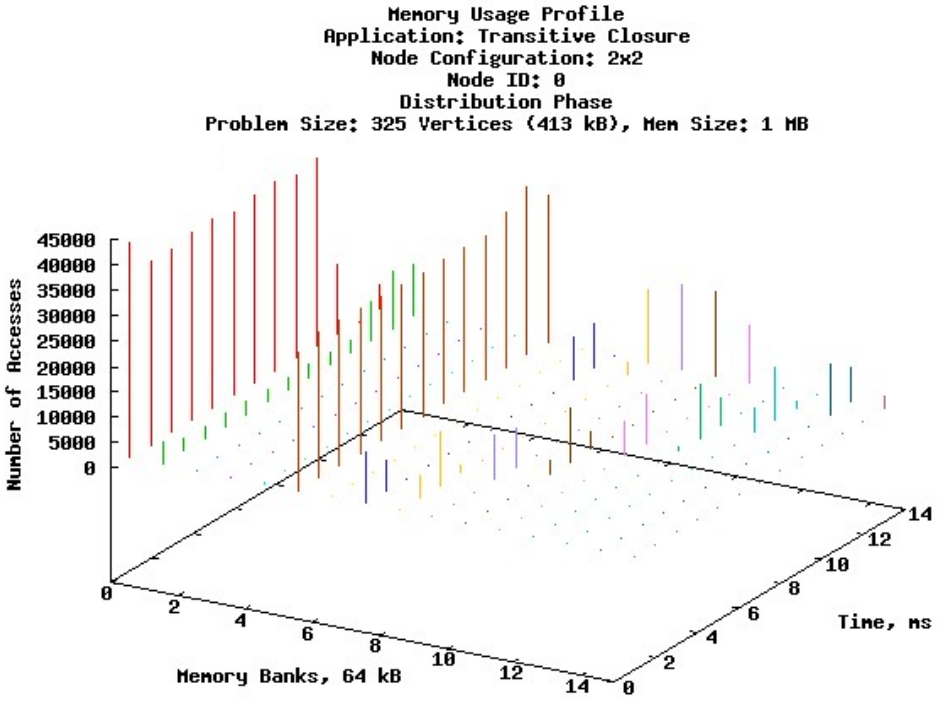


Figure 5.5: Memory Usage Profile of the Home Node during the Distribution Phase of TCB.

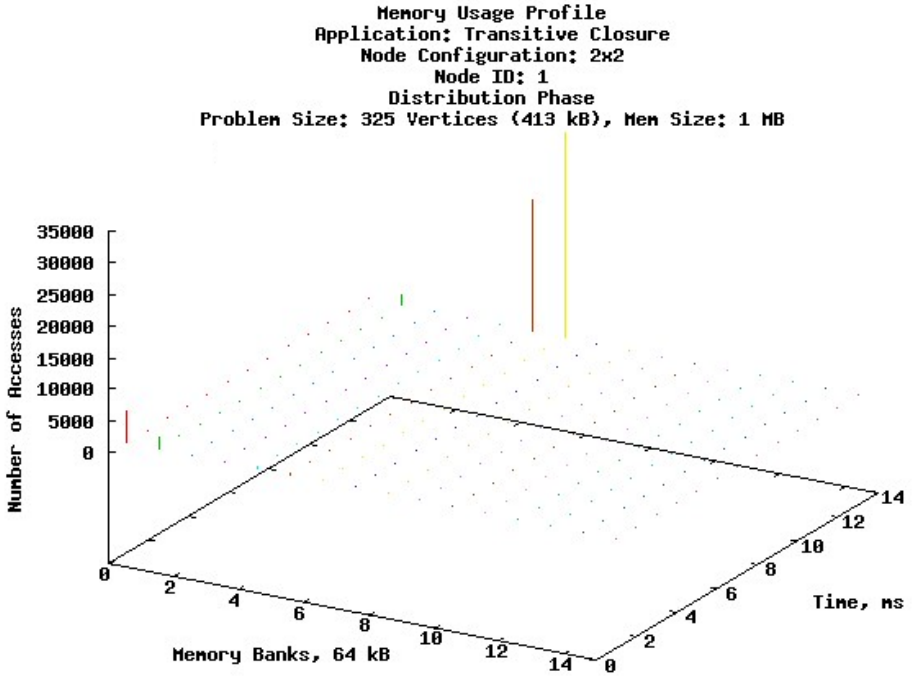


Figure 5.6: Memory Usage Profile of Node 1 during the Distribution Phase of TCB.

A graphical presentation of memory bank usage is given below for nodes falling into each set type during the solution phase. During the section of the application that is profiled here, node 1 happens to be the owner of the *k*th row, the row that all nodes must receive in order to update the values contained in the rows that they own. This ownership is characterized by a large number of accesses to the bank containing this row. Note, however, that only this bank and the two others containing sections of instruction code currently being executed indicate any usage at all. The home node has similar bank activity, except that in its case the small amount of activity in bank zero is most likely the main control thread accessing a few variables containing progress information each time is awakens. Finally, observation of figure 5.9 reveals that a node that is neither the home node nor the owner of the *k*th row generates even less bank activity. These results indicate that the unnecessary dissipation of a significant amount of power may be taking place in large numbers of idle memory banks during most of the execution time of this application, and the usage of a partitioned memory system may result in substantially less overall power consumption.

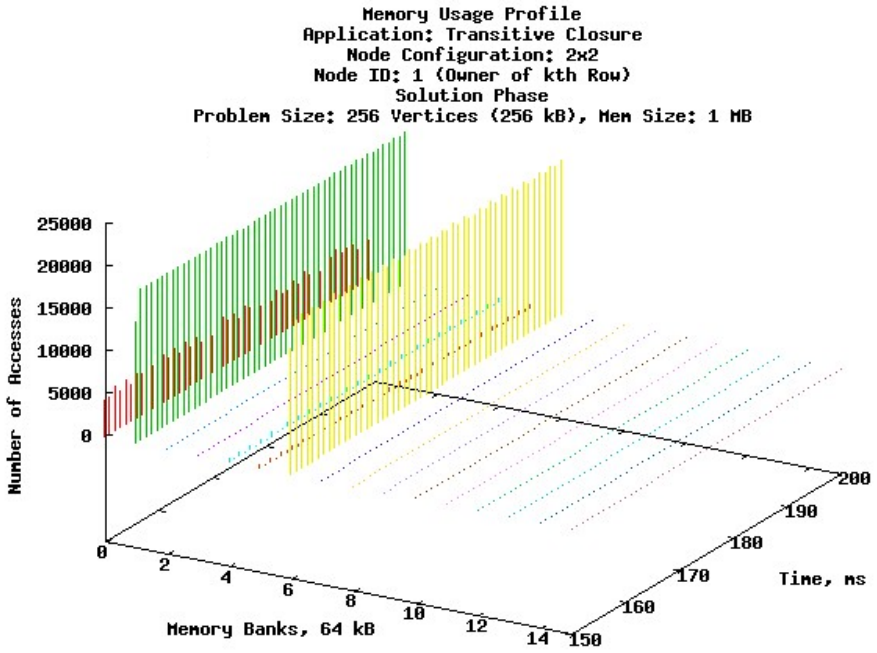


Figure 5.7: Memory Usage Profile of the Owner of the *k*th Row during the Solution Phase of TCB.

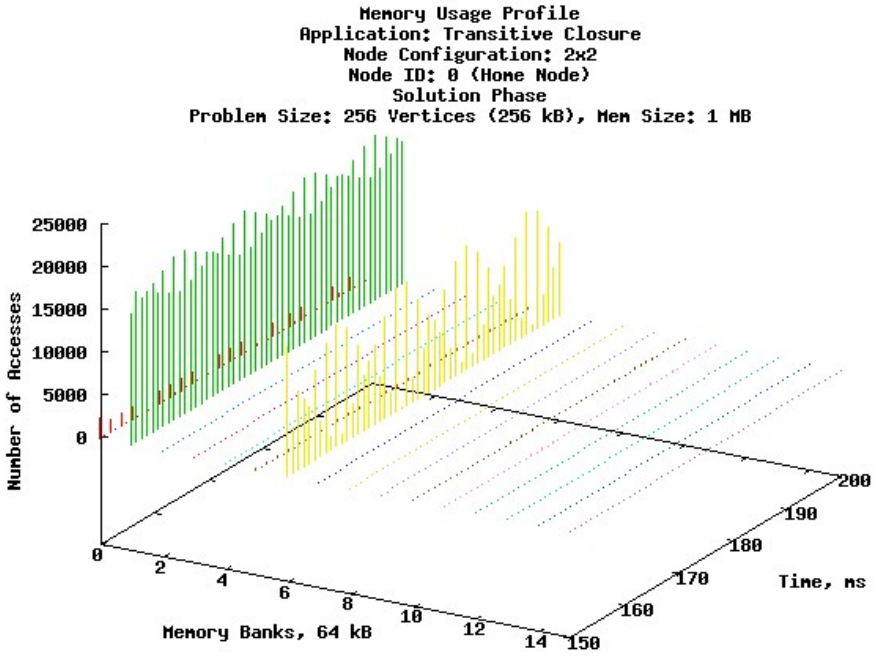


Figure 5.8: Memory Usage Profile of the Home Node during the Solution Phase of TCB.

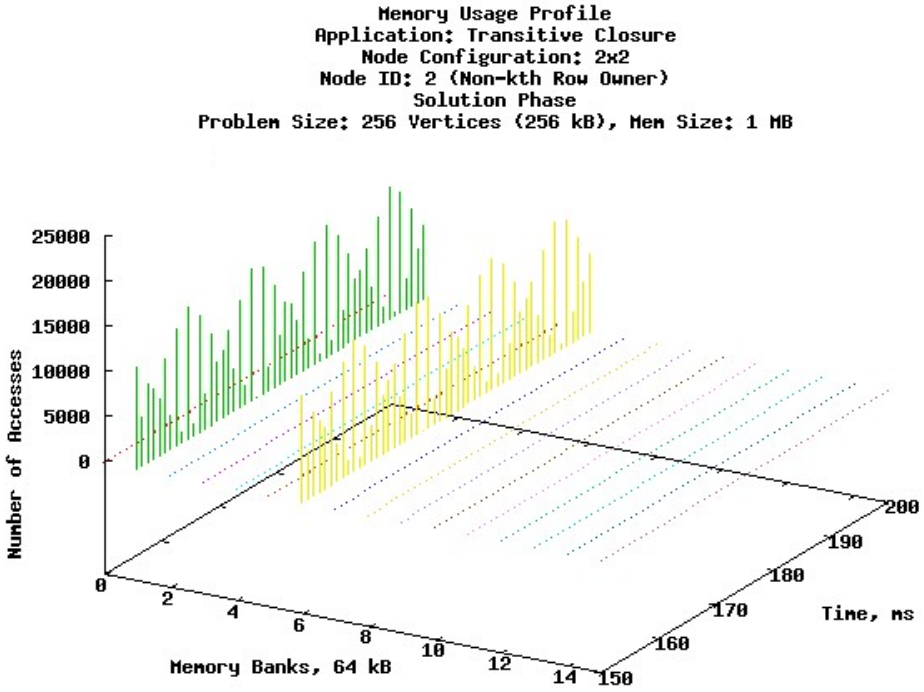


Figure 5.9: Memory Usage Profile of Node 2, a Non-*k*th Row Owner during the Solution Phase of TCB.

Chapter 6

Conclusions

6.1 Summary of Thesis

This thesis has presented the development of various models of the SCMP system in a system design language at various levels of abstraction. These models were constructed through the study and application of various system-level methodologies for the purpose of evaluation of design and performance. Various SDLs were studied for their strengths and weaknesses, and SystemC was chosen as the modeling language.

The previous model of SCMP that has been used for benchmarking was implemented in C and simulated clock cycles through a single-threaded application that carried out the tasks of components with single-cycle granularity. Although this model provides very efficient simulations and estimates of various performance metrics, more refined models are needed to achieve the final objective of a fully optimized hardware implementation of SCMP. The constructs and data types incorporated into modern SDLs allow for the support of models of computations well suited for the representation of systems at various levels of abstraction. The use of these models of computation and even abstract hardware modeling itself are not supported by a language such as C, so the application of an effective design methodology to a model of SCMP such as the C simulator would not be practical. As design trade-offs and

optimizations are considered for the SCMP system, the best approach to beginning the modeling and performance analysis of these new designs may not be at the cycle-accurate level. Furthermore, the C simulator lacks the structure that a refined model should embody including explicit communication channels and well-defined component interfaces. Finally, the Open Core Protocol International Partnership (OCP-IP) is developing of a new standard that will be taken into consideration during the development of new SDLs [3]. A primary motivation for following this standard arises due to the growing necessity of IP reuse in the design of systems at various levels of abstraction, a feature that is not supported by hardware design in C.

Various components of the SCMP node were previously modeled at RTL with SystemC, and this work has extended those modeling efforts by creating functional models of the entire system. The first model is based on the dataflow model of computation by using Kahn Process Networks to create an untimed, functionally accurate representation of the system that can execute applications with different configurations of nodes. This model was then refined one component at a time until all facets of the system besides the router and the pipeline were pin- and cycle-accurate. Finally, the pipeline was broken down into its stages and modeled with timing and structural accuracy. Once the models were completed and verified, some measurement utilities were incorporated into them. These metrics were used to confirm that the abstract models can be sufficiently accurate to be useful for performance analysis in some, but not all cases. The results were also used to assist in the verification of the correctness of the functionality of the refined models. A key limitation of the system, the power consumption of the DRAMs embedded on the chip, was uncovered as part of a previous project. In order to consider the feasibility of new memory architectures that lower this power consumption, tools for gathering data during the execution of an application to generate a profile of memory usage were incorporated into the model. Results from the execution of the Transitive Closure benchmark were presented to illustrate that a multi-bank memory system in which banks have various operating modes that trade off power consumption for limited functionality may be a reasonable consideration.

6.2 Future Work

Although this work is in many aspects a continuation of the products of previous modeling efforts for SCMP, it is also a prerequisite for future work involving the system's optimization and construction. First of all, some design features have been studied and incorporated into the SCMP simulator for evaluation. The models found in this work include many of the fundamental architectural design features that will enable SCMP to achieve high performance, but some of the more advanced modifications were not incorporated into the models. For instance, in one past project, the Active-Messages messaging layer originally chosen for the SCMP message-passing paradigm was abandoned to explore the benefits of a new programming model based on a Send-and-Receive messaging layer. Since this change involved modifications to the ISA as well as direct additions to SCMP architecture, this new paradigm would require quite a few changes to the models presented in this work. In another project, a custom VLSI layout of a low-power, optimized router design was presented. Because the router is a pipelined component, modeling it with cycle-accuracy would make already slow simulations on the refined models even more computationally intense. However, as the power of platforms on which these simulations are run increases, a model of the router embodying the design presented in this past project may be more reasonable. Another approach might be to construct a model that is abstract with respect to all components except for the router to achieve acceptable simulation times. Although the C simulator has provided useful analytical data for these design modifications, further evaluation of these modifications at the system-level with the refined models should be carried out.

There also exist avenues for future research with regard to the study of the feasibility of new memory architectures that offer solutions to the problem of high energy consumption. In this study, a memory system in which the storage is divided into multiple banks that can switch among a set of states offering varying levels of power and functionality is considered. In order to fully explore this option, more benchmarking should be carried out to optimize power-performance trade-offs by varying memory bank size and the frequency of state

switching. If evaluation through benchmarking suggests that this may be a reasonable design, an implementation of such a memory architecture could then be studied and incorporated into the models. This approach also necessitates studies in the realm of advanced operating system support for SCMP. The OS will play a vital role in the optimal timing of bank activation and deactivation, as well as the efficient servicing of memory allocation requests. Of course, other design strategies of reducing the power consumption of the memory or of the other components may be studied and built into the models as well.

Another objective that may be considered for a future project stems from the need for more efficient simulations. The power of the abstract models lies in the principle that as the designer delays the consideration of the details that must be considered in the hardware implementation, he or she gains simpler verification, less complexity, and faster simulations. However, the weakness of abstract models is that they may offer unacceptably poor estimates of the performance of the final implementation. Hence, in order to utilize the power of the abstract models, the refined models can be used to verify their accuracy. The C simulator offers particularly efficient simulations with performance estimates that are "cycle-accurate." But the question of whether or not these estimates are sufficiently accurate to provide useful data remains until the simulator's numbers are compared with the refined models of the system. A study of the accuracy of the C simulator's performance estimations will involve a great deal of analysis of its representation of the SCMP components, but the tuning of these estimates will definitely prove useful for future evaluation of the system.

Chapter 7

Acknowledgements

I would like to thank my advisors Dr. Baker and Dr. Patterson for their support and guidance over the course of my graduate work at Virginia Tech. Thank you also to my committee members Dr. Jones and Dr. Martin for their time and efforts contributed towards my thesis project and defense. I must offer my appreciation to Dr. Jae Park, Dr. Scott Harper, and Brian Marshall for their counsel, advice, and games of foosball during the closing months of my M.S. program. Finally, my deepest sense of gratitude goes out to all of my family and friends, especially to my parents and to Jenn Walters, without whom my invaluable experience at Virginia Tech would not have been possible.

This work was partially supported by the National Science Foundation
under Grant No. CCR-0113948.

Bibliography

[1] Guido Arnout. "SystemC Standard," *Asia and South Pacific Design Automation Conference 2000*, January 25-28, 2000.

[2] James M. Baker, Jr., Brian Gold, Mark Bucciero, Sidney Bennett, Rajneesh Mahajan, Priyadarshini Ramachandran, and Jignesh Shah. "SCMP: A Single-Chip Message-Passing Parallel Computer", *The Journal of Supercomputing*, vol. 30, pp. 133-149, 2004.

[3] Joe Basques. "Sonics, Nokia, Texas Instruments, MIPS and UMC Launch OCP-IP to Standardize IP Core Socket Interface," December 3, 2001, http://www.ocpip.org.

[4] L. Benini, A. Macii, M. Poncino. **"**Energy-aware design of embedded memories: A survey of technologies, architectures, and optimization techniques," Volume 2, Issue 1, pp. 5-32, ACM Transactions on Embedded Computing Systems, ACM Press, New York, NY, Feb. 2003.

[5] L. Benini, A. Macii, M. Poncino. "A Recursive Algorithm for Low-Power Memory Partitioning," *Proceedings of the Int'l Symp. on Low Power Electronics and Design*, ISLPED00, Rapallo, Italy, 2000.

[6] Sidney Page Bennett. Designing a Compiler for a Distributed Memory Parallel Computer System. Master's thesis, Virginia Polytechnic Institute and State University, November 2003.

[7] A. Bernstein, M. Burton, F. Ghenassia. "How to bridge the abstraction gap in system level modeling and design". Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference, Nov. 7-11 2004. pp. 910 – 914.

[8] A. Bona, V. Zaccaria, R. Zafalon. "System Level Power Modeling and Simulation of High-End Industrial Network-on-Chip**,"** *Proceedings of the conference on Design, automation and test in Europe – Volume*, p. 30318, 2004.

[9] J. Bond. "Power Considerations in Today's SoC Memories," Tech. Rep., MoSys, Monolithic System Technology, 2003.

[10] Mark Bucciero. The Design of the Node for the Single-Chip Message-Passing (SCMP) Parallel Computer. Master's thesis, Virginia Polytechnic Institute and State University, March 2004.

[11] L. Cai, D. Gajski. "Transaction Level Modeling: An Overview," *Proc. of CODES-ISSS*, pp. 19-24, Oct. 2003.

[12] Edited by P. Cavalloro, C. Gendarme, K. Kronlöf, J. Mermet, J. van Sas, K. Tiensyrjä, N. S. Voros. *System Level Design Model with Reuse of System IP.* Kluwer Academic Publishers, 2003.

[13] A. Chandrakasan and R.W. Brodersen. *Low power digital CMOS design*, Kluwer Academic Publishers, Boston (1998).

[14] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, L. Todd. *Surviving the SOC Revolution: A Guide to Platform-Based Design.* Kluwer Academic Publishers, 1999.

[15] S. L. Coumeri, D.E. Thomas. "An environment for exploring low power memory configurations in system level design," *Computer Design*, pp. 348 – 353, ICCD '99, Austin, TX, October 1999.

[16] D. Culler, J. P. Singh, A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, 1998.

[17] A. Dennis, B. H. Wixom, D. Tegarden. *Systems Analysis and Design: An Object-Oriented Approach with UML.* John Wiley & Sons, Inc., 2002.

[18] T.J. Dysart, B.J. Moore, L. Schaelicke, P.M. Kogge. "Cache Implications of Aggressively Pipelined High Performance Microprocessors." *Performance Analysis of Systems and Software*, 2004, pp. 123-132.

[19] A. Efthymiou. "Asynchronous Techniques for Power-Adaptive Processing." Doctoral thesis, University of Manchester. December 2002.

[20] D. Gajski, A. Wu, N. Dutt, S. Lin. *High-level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1994.

[21] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, S. Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[22] A. Gara et. al. "Overview of the Blue Gene/L system architecture," in *IBM Journal of Research and Development*, vol. 49, no. 2/3, March/May 2005, pp. 195-212.

[23] Brian Gold. Balancing Performance, Area, and Power in an On-Chip Network. Master's thesis, Virginia Polytechnic Institute and State University, July 2003.

[24] T. Grötker, S. Liao, G. Martin, S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[25] P. Grun, N. Dutt and A. Nicolau. *Memory architecture exploration for programmable embedded systems*, Kluwer Academic Publishers, Boston, 2003.

[26] L. Hammond, B.A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, K. Oluklun. "The Stanford Hydra CMP," *Micro, IEEE, Volume 20, Issue 2*, pp. 71- 84, March – April 2000.

[27] R. Ho, K.W. Mai, M.A. Horowitz. "The Future of Wires." *Proc. IEEE*, vol. 89, no.4, April 2001, pp. 490 - 504.

[28] A. Jantsch. *Modeling Embedded Systems and SoCs*. Morgan Kaufmann Publishers, 2004.

[29] A. K. Jones, D. Bagchi, S. Pal, X. Tang, A. Choudhary, P. Banerjee. "PACT HDL: A C Compiler with Power and Performance Optimizations," *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pp. 198 – 197, 2002.

[30] G. Kahn. "The semantics of a simple language for parallel programming," Information processing, pp. 471-475, Stockholm, Sweden, Aug 1974.

[31] M. Kandemir, U. Sezer, V. Delaluz. "Improving memory energy using access pattern classification," *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pp.201 – 206, ICCAD, San Jose, CA, 2001.

[32] M. T. Kandemir. "Exploiting Bank Locality in Multiprocessor SoC Architectures," *Parallel and Distributed Processing Symposium. Proceedings. 18th International*, p. 92. 26-30 April 2004.

[33] P. M. Kogge. *The Architecture of Pipelined Computers.* Hemisphere Pub. Corp., 1981.

[34] P. Landman. Low-power architectural design methodologies. Doctoral Dissertation, U.C. Berkeley, 1994.

[35] Charles W. Lewis Jr. Support for Send-and-Receive Based Message-Passing for the Single Chip Message-Passing Architecture. Master's thesis, Virginia Polytechnic Institute and State University, April 2004.

[36] C. Lyuh, T. Kim. **"**Memory access scheduling and binding considering energy minimization in multi-bank memory systems," *Proceedings of the 41st annual conference on Design automation - Volume 00*, pp. 81-86, DAC, San Diego, CA, 2004.

[37] D. Marculescu, A. Iyer. "Application Driven Processor Design Exploration for Power-Performance Trade-off Analysis." IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001.

[38] Edited by G. Martin, Henry Chang. *Winning the SoC Revolution: Experiences in Real Design.* Kluwer Academic Publishers, 2003.

[39] Rick Merritt. "Intel will demo its first multicore CPU at IDF," in *Electronic Engineering Times*, September 7, 2004, http://www.eetimes.com/.

[40] W. Muller, W. Rosenstiel, J. Ruf (Eds.). *SystemC – Methodologies and Applications*, Kluwer Academic Publishers, 2003.

[41] K. Olukton, B. Nayfeh, L. Hammond, K. Wilson, K. Chang. "The case for a single-chip multiprocessor," *Architectural Support for Programming Languages and Operating Systems*, pp. 2-11, Proceedings of the seventh international conference on Architectural support for programming languages and operating systems, Cambridge, MA, 1996.

[42] D. A. Patterson, J. L Hennessey. *Computer Architecture: A Quantitative Approach, 3rd Edition.* Morgan Kaufmann Publishers, 2002.

[43] M. J. Quinn. "Floyd's Algorithm," in *Parallel Programming in C with MPI and OpenMP*. New York, NY: McGraw Hill, 2004, pp.137-154.

[44] Priyadarshini Ramachandran. Microarchitectural Level Power Analysis and Optimization in Single Chip Parallel Computers. Master's thesis, Virginia Polytechnic Institute and State University, July 2004.

[45] C. V. Ramamoorthy, H. F. Li. "Pipeline Architecture." ACM Computer Surveys, Volume 9, Issue 1, 1977, pp. 61-102.

[46] Semiconductor Industry Association. *International Technology Roadmap for Semiconductors, 2004 Update,* 2004.

[47] Wen-Tsong Shiue. "Low power memory design." Application-Specific Systems, Architectures and Processors, Proceedings of the IEEE International Conference, 17-19 July 2002, pp. 55 – 64.

[48] S. Schulz. *System-level Design Brings New Methodology*. http://www.EEDesign.com. CMP Media, LLC, 2002.

[49] S. Stuart, D. Vermeersch, et. al. *Functional Specification for SystemC 2.0*. Synopsis, Inc., Coware, Inc., Frontier Design, Inc., et al. January 17, 2001.

[50] K. Tatas, M. Dasygenis, N. Kroupis, A. Argyriou, D. Soudris, A. Thanailakis. "Data memory power optimization and performance exploration of embedded systems for implementing motion estimation algorithms." *Real-Time Imaging*, vol.9, no.6, Dec. 2003, pp.371-86, Academic Press, UK.

[51] M. Thompson and A. D. Pimentel. "A High-level Programming Paradigm for SystemC," *Proc. of the 4th Int. Workshop on Systems, Architectures, Modeling, and Simulation*, LNCS, Samos, Greece, July, 2004.

[52] D. W. Wall. "Limits of instruction-level parallelism," *Architectural Support for Programming Languages and Operating Systems*, pp. 176-188, Proceedings of the fourth international conference on Architectural support for programming languages and operating systems, Santa Clara, CA, 1991.

Vita

Patrick Anthony La Fratta was born on April 10, 1982 in Dallas, TX. In 2000, he graduated from Brookville High School in Lynchburg, VA. He then enrolled at Virginia Tech, completing his Bachelor of Science in Computer Engineering in December of 2003.

While in graduate school, Patrick has worked as a research assistant under Dr. James Baker, and will complete his Master of Science in Computer Engineering at Virginia Tech in the spring of 2005. After graduation, he will continue his work as a software engineer at Adaptive Genomics Corporation in Blacksburg, VA. In the fall of 2005, he will attend the University of Notre Dame in Indiana, where he will pursue a Ph.D. in Computer Science and Engineering. His research interests include computer architecture, parallel computing, hardware modeling, and digital design.