SIMULATION OF MULTI-CORE SYSTEMS AND INTERCONNECTIONS AND EVALUATION OF FAT-MESH NETWORKS

by

Yu Zhang

B.S. Information Science and Electronic Engineering,

Zhejiang University, 2006

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2008

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

Yu Zhang

It was defended on

November 10th 2008

and approved by

Alex K. Jones, Assistant Professor, Department of Electrical and Computer Engineering

Steven Levitan, John A. Jurenko Professor, Department of Electrical and Computer Engineering

Jun Yang, Assistant Professor, Department of Electrical and Computer Engineering

Thesis Advisor: Alex K. Jones, Assistant Professor, Department of Electrical and Computer Engineering

SIMULATION OF MULTI-CORE SYSTEMS AND INTERCONNECTIONS AND EVALUATION OF FAT-MESH NETWORKS

Yu Zhang, M.S.

University of Pittsburgh, 2008

Simulators are very important in computer architecture research as they enable the exploration of new architectures to obtain detailed performance evaluation without building costly physical hardware. Simulation is even more critical to study future many-core architectures as it provides the opportunity to assess currently non-existing computer systems. In this thesis, a multiprocessor simulator is presented based on a cycle accurate architecture simulator called SESC. The shared L2 cache system is extended into a distributed shared cache (DSC) with a directory-based cache coherency protocol. A mesh network module is extended and integrated into SESC to replace the bus for scalable inter-processor communication. While these efforts complete an extended multiprocessor simulation infrastructure, two interconnection enhancements are proposed and evaluated. A novel non-uniform fat-mesh network structure similar to the idea of fat-tree is proposed. This non-uniform mesh network takes advantage of the average traffic pattern, typically all-to-all in DSC, to dedicate additional links for connections with heavy traffic (e.g., near the center) and fewer links for lighter traffic (e.g., near the periphery). Two fat-mesh schemes are implemented based on different routing algorithms. Analytical fat-mesh models are constructed by presenting the expressions for the traffic requirements of personalized all-to-all traffic. Performance improvements over the uniform mesh are demonstrated in the results from the simulator. A hybrid network consisting of one packet switching plane and multiple circuit switching planes is constructed as the second enhancement. The circuit switching planes provide fast paths between neighbors with heavy communication traffic. A compiler technique that abstracts the symbolic expressions of benchmarks' communication patterns can be used to help facilitate the circuit establishment.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

I first thank my advisor, Dr. Alex Jones. Without his relentless support, instruction, patience, and advice, I could never have done this work. Even more importantly, his enthusiasm in innovations, attitude towards research and optimism deeply impressed and encouraged me. I am grateful to Dr. Jones on a professional and personal level.

I want to thank Dr. Rami Melhem for his co-advice on my research. Throughout my master study and research, his outstanding experience on research, broadness, and inspiration always guide me to the right way I should pursue. I also thank Professor Steve Levitan and Professor Jun Yang for joining my committee and giving me precious advice on my thesis. I further thank my fellow researcher, Shuyi Shao for his help to accelerate my research.

Finally, I thank my parents for providing me an intensive overseas education opportunity and their regardless support, and my girlfriend, Zhexi Pan for her constant care and support.

1.0 INTRODUCTION

Moore's Law [41] is pushing us away from instruction level parallelism (ILP), which shows limited parallelism in current applications [27, 30] and increases complexity in design validation and testing, towards multi/many-core design. Chip Multiprocessor (CMP) is a popular design paradigm that explores the parallelism of multi-threaded applications to achieve higher performance within a given power envelope. Inevitably, some design challenges have emerged during this architectural turnaround from single-core processor to multi/many-core processor. Two of the challenges are the construction of a powerful yet fast multi-core simulation tool and the design of an efficient interconnection on the chip. This thesis provides three contributions which tackle these problems.

Contribution 1: A multiprocessor simulation platform. The platform is based on a cycle-accurate microprocessor simulator called SESC [50]. It extends the provided interconnection and cache structure with more recent standard CMP configurations.

Contribution 2: A non-uniform "fat-mesh" interconnection. Following the idea of a fat-tree structure, fat-mesh provides additional bandwidth to the heavy traffic links. The analytical models of two fat-mesh schemes using different routing algorithms are derived.

Contribution 3: A hybrid network consisting of a packet switching plane and multiple circuit switching planes. A symbolic expression compiler analysis technique is also extended to handle expressions contained in loops to facilitate the construction of circuit switching network.

Architecture research relies on simulation because it is too expensive to build prototypes to demonstrate each architectural innovation. In order to produce reliable results when evaluating new architectures, cycle-accurate simulations are necessary. However, trade-offs need to be made between simulation speed and accuracy as simulation times may increase exponentially with the need to simulate multiple processors in the design of parallel systems. For example, simulation of a Splash-2 benchmark with 16 processors in Simics [38] can take days to finish. Thus, a fast yet accurate multiprocessor system simulator is a necessity in the computer architecture community.

The first contribution addresses the need of a powerful and efficient simulation tool by the extension of the baseline simulator, SESC, with the architecture features of proposed many-core architectures including distributed shared caches (DSC) and distributed interconnections. First, SESC's shared cache is replaced with a popular DSC model. The shared cache increases both the memory contention and the access latency in large multiprocessor systems. On the other hand, DSC is logically shared, which means all nodes share one memory address, but cache banks are physically distributed at each processor node. The advantage of such a cache system is that it provides a transparent interface and convenient programming environment for a programmer, and at the same time keeps some data locally to fully take advantage of the computational capacity of a multi-core system. A fully featured directory-based cache coherence policy is also implemented using the Modified Shared Invalidate (MSI) protocol to keep the data consistent in DSC. A prevailing distributed network with various routing topologies and routing algorithms is also integrated into the simulator to replace the existing bus structure. The buses can easily become shared data access bottlenecks when the systems get larger. Instead, distributed networks, especially mesh networks, provide scalable inter-processor communication to the system.

While the first contribution provides a multi-core simulation infrastructure for various processor and network architectural studies, the next two contributions focus on the interconnection design, which occupies a high priority in CMP study as the interconnections can impede the CMP speedup due to additional communication and coordination overhead [25].

In the second contribution, a non-uniform "fat-mesh" is presented as an augment to the traditional mesh topology. Mesh is a dominant network structure used in today's CMP with its scalability to large numbers of processors. However, in heavy traffic, links on the interior of the structure are more heavily utilized than those on the exterior. A similar phenomenon can be seen in other interconnection topologies, the most famous of which is the fat tree [33].

An effective method to build a fat-mesh is to add links in proportion to the traffic requirement in each link of the mesh under the expected traffic pattern. We suggest that the typical traffic pattern of popular architecture and, in particular, cache organization, of many-core systems can be reasonably modeled as all-to-all personalized communication. A popular cache configuration is to utilize relatively small, private level one (L1) caches and larger, distributed/shared level two (L2) caches where the addresses are interleaved. Based on our experiments, the interleaving in such cache system tends to destroy locality of reference and results in a similar traffic pattern to personalized all-to-all communication.

The traffic requirement for a particular link in a mesh can be further impacted by the routing algorithms employed in the system. We explore the best link arrangement policy by examining two routing algorithms. One is the popular dimensional ordering algorithm commonly known as XY routing. XY routing is simple, deterministic, avoids deadlock, and is reasonably good at distributing the traffic evenly in the network. However, many non-deterministic routing algorithms have been proposed in an attempt to improve on XY routing. We consider an example of these algorithms called XY-YX routing, which is the default routing scheme in SESC. At each switch point XY-YX routing randomly selects the traversal dimension (i.e. X versus Y) in an effort to more evenly distribute the traffic of a biased workload that favors a smaller number of communication pairs.

The thesis derives the analytical models for two fat-meshes schemes using XY and XY-YX routing algorithms. It compares the scalability of the proposed non-uniform fat-meshes with traditional uniform meshes. The results show that our fat-mesh provides a significant benefit in terms of message latency compared to a traditional mesh.

In the third contribution, the thesis implements a multi-layer network system with one packet switching plane and multiple circuit switching planes in the simulator. While packet switching and circuit switching are both commonly used switching techniques, they both have unavoidable drawbacks. A hybrid network comprising both of them and overcoming the disadvantages of both sides is drawing more attention in high performance computing. The idea is to let the regular, predictable communications using the fast circuit switching while some random, irregular or conflict communications can still utilize the traditional packet switching plane to avoid deadlock or other unexpected situations. The hybrid network is implemented in the the simulator and shows its potential performance gain.

Furthermore, as the benefits of circuit switching can only outweigh its drawbacks when the communication exhibits locality and when an application's traffic locality is appropriately explored, a compiler analysis technique is explored to extract the communication pattern in a parallel application to better utilize circuit switching. In particular, a loop structure analysis procedure is presented with the help of Mathematica, a symbolic expression manipulating tool, to solve the problem which cannot be tackled by the common compiler techniques. Though the target applications we apply our compiler techniques on are Message Passing Interface (MPI) programs, and they are more dominant in a distributed memory environment, we believe that these compiler techniques have the same benefits to shared memory applications and can be leveraged with proper modification.

The remainder of this thesis is organized as follows: Section 2 provides a background and explores related previous work. More specifically, Section 2.1 focuses on the development of multi-core simulator in the architecture community; Section 2.2 introduces the mesh network and its variants; Section 2.3 describes hybrid network systems intended to better utilize the network. Section 3 provides several aspects of original SESC, followed by the extensions made to SESC, including distributed shared cache, directory based coherence protocol and network module. Section 4 provides the derivation of the fat linear array, followed by the fat-mesh structure using XY routing and XY-YX routing. After briefly introducing the simulation parameters and benchmarks, the fat-mesh results are discussed at the end of this chapter. Section 5 presents the implementation of a hybrid network and a compiler aided technique to extract benchmarks' communication pattern. Some concluding remarks are presented in Section 6 and finally, future directions of this work are explored in Section 7.

2.0 BACKGROUND AND RELATED WORK

2.1 ARCHITECTURE SIMULATION

Simulators have become an integral part and one of the most important tools of computer architecture research and the processor design process. Simulators allow architects to quickly evaluate the performance of different processor configurations without having to fabricate a physical processor. Moreover, physical processors can only takes one configuration, may take years to develop and are extraordinarily expensive. Therefore, architects use simulators to guide design space exploration and to quantify the efficiency of an enhancement. Additionally, a simulator is much more flexible than fabricating the processor since it can accurately determine the expected performance of a processor enhancement without having to undergo all the necessary circuit-level design steps. Consequently, without simulators, designing processors would either be too expensive or would yield very poor designs.

Multi-core designs have become commonplace in the processor market, and are hence a major focus in modern computer architecture research. Industrial and academic architects have developed many multi-core simulators. SimpleScalar [1], as one of the most popular single-processor simulators, has been used as a major processor research and computer architecture course tool since its debut in 1995. During the current multi-core trend, SimpleScalar was also extended to multi-core system simulation in [10, 39, 14]. The idea is to spawn the simulator to multiple threads or processes, each models one processor and cooperates with others.

Rsim [47] is a shared memory multiprocessor simulator from Rice university. It aggressively explores ILP in the processor to increase the simulation accuracy at the expense of speed. It provides a fixed memory hierarchy system with directory-based coherence. It also has a wormhole routing mesh network. Because the simulation of program execution with RSIM is very slow (several thousands times slower than program execution on the real machine), it is not suitable for evaluation of various designs of real-world programs.

Simics [38] is a full system simulator that models an entire system and is accurate enough to run an operating system(OS). Several projects have extended the Simcis infrastructure. SimFlex [23] is a component-based computer architecture simulator based on Simics which enables full-system timing-accurate simulation of multiprocessor systems and facilitates both model integration and compile-time simulator optimization. It also adds directory based coherence components to the original Simics infrastructure.

A full system simulator like Simics effectively provides virtual hardware that is independent of the nature of the host computer. It simulates computer systems at such a level of detail that complete software stacks from device drivers to operating systems can run on the simulator without any modification. Its potential ability to simulate several applications simultaneously as opposed to an instruction set simulator's ability to simulate only one program is one of its valuable features.

Unfortunately, a major drawback of cycle accurate simulators especially the full system ones is their slow simulation speed. With the ever-growing size and complexity of modern microprocessors, detailed cycle-accurate simulation has become orders of magnitude slower than their hardware counterparts. For example, doing cycle-level simulation using SimpleScalar for a SPEC2000 application takes up to a month [65] and newer benchmark suites will have even longer run times. It has also been reported that Intel's [16] and AMD's [6] fastest true cycle-accurate x86 simulators run at 1KHz to 10KHz which translates to two minutes of execution time in approximately one to ten years of simulation time. At such speeds, it is impractical neither to explore, evaluate and refine microarchitectures using full benchmark simulations nor to validate new applications on the simulated architectures.

To solve this problem, statistical sampling techniques have recently been used to reduce the amount of code that has to be simulated in detail in order to get accurate performance estimates. Wunderlich et al. [64] proposed SMARTS, which only simulates some small execution regions of the benchmark in detail while fast-forwarding with only functional simulation between the simulation points. But still, the execution time can be several hours as reported in [64] to one month as reported in [65].

The baseline simulator used in this thesis, SESC, is a microprocessor architectural simulator designed with performance in mind: it is up to ten times faster than competing simulators, with the capability of executing over 1.5 million instructions per second (MIPS) on an Intel Pentium 4 3GHz processor [50]. Besides utilizing a rabbit mode to skip the warmup code region without detailed timing simulation to accelerate the simulation, SESC decouples the functional and timing model. The actual instructions are executed in an emulation module built from MINT [62], a MIPS emulator. The emulator returns instruction objects to SESC, which are then used for the timing simulation. The justification for this is twofold. First, it is much faster to have the actual instruction executed in a simple emulator. Second, it is easier to program and debug when execution and timing are separate. The timing simulator, which is very complex, does not need to be 100% accurate if it does not affect the computation of instructions. A similar decoupling concept is also seen in the FAST simulator [11] developed by UTAustin. The FAST's functional model simulates the computer at the functional level, including the instruction set architecture (ISA) and peripherals; it also executes the application, operating system and BIOS code. The timing model simulates only the microarchitectural structures that affect the desired metrics and is implemented in a single Field Programmable Gate Arrays (FPGA) to accelerate the simulation.

2.2 MESH NETWORKS AND THEIR VARIANTS

A mesh is a dominant interconnection network in a multi-core system on chip. Unlike being utilized in the 2D/3D torus in the super-computing area, long wrapped around wires would create a significant area, power, and performance overhead and thus are not practical in CMP design. In a typical mesh implementation, processors are connected through a network made up of a series of routers. Each processor has a dedicated connection to one router. In the middle of the network, each router has dedicated connections to each of its four neighbors (North, South, East, and West). In the corners and in the edges of the network the routers do not have all of the neighbors and therefore, a portion of their ports are unusable and their degree is smaller. The SESC simulator is augmented with such a mesh network.

There have been many efforts to optimize or improve the performance of mesh-based topology interconnects by adding or removing links. Carlson explored adding one or more global mesh structures to the processing array. The modification showed improved performance for low to medium inter-processor communication requirements and no speedup for higher communication volume such as sorting [8]. Samatham develops a technique to extend a De Bruijn network into a mesh with a minimal number of added links to achieve the benefits of both types of networks [51]. Ziavras proposes a method to uniformly reduce the number of links from a hypercube while largely retaining the advantageous properties of the hypercube such as low diameter to overcome the drawback of the hypercube in increasing the number of communication channels for each processor with the increasing total number of processors in the system [67]. Lin et al. propose Mesh with Hybrid Busses (MHB) by augmenting a mesh with 1-bit row and column buses. MHB occurs no increase in the fabric area overhead or complexity, but improves performance of digital geometry tasks [37]. Bhagavathi et al. describe an architecture, commonly referred as mesh with multiple broadcasting, constructed by endowing each row and each column with its own dedicated high-speed bus and used for sorting [5].

Moreover, there have been several types of "fat" topologies proposed in addition to the traditional fat tree. For example, Ohring et al. describe generalized fat trees and introduce the extended generalized fat tree (XGFT) [46]. The XGFTs are designed for greater scalability than traditional fat trees in the number of leaf processor nodes and amount of routing resources, particularly due to their construction from non-uniform switches that contain different numbers of links [29]. A fat pyramid is an extension of the fat tree which adds additional links to complete a mesh [19]. The fat-pyramid structure with area $\theta(A)$ requires only $O(logA)$ slowdown to simulate any competing network of area A under very general conditions. Fat-stacks [9] proposed by Chen and Sha are a multi-level network, where each level contains one or more subnetworks comprised of rings, and the link capacities double up the levels of the network. The augmented fat-stack resembles the fat-tree and the fat-pyramid in hardware structure, but it is the minimal universal network for an $O(logA)$ overhead in terms of hardware usage while still approaching the performance of the fat-pyramid and having far better performance than that of the fat tree.

Fat-meshes have traditionally been defined to include multiple links per hop that allow data to travel in parallel for improved performance [34, 24]. According to [34], a d-dimensional $p_1 \times p_2 \times$

$\ldots \times p_d$ fat-mesh is defined by $\Delta \times \sqrt[d]{P}$, where Δ is the number of links per hop, $P = p_1 p_2 \ldots p_d$ denotes the set of nodes in the mesh. By this definition of fat-meshes, the number of links is uniform throughout the network.

However, we are proposing to increase the number of links based on the needs of worst case all-to-all personalized communication, thus constructing a non-uniform fat-mesh. The all-to-all personalized communication pattern we imposed on our non-uniform fat-meshes is reasonable. Unlike traditional parallel computing systems for which communication patterns are often regular [55], multi-core processors with many cores on the chip are dominated by highly random traffic [17] which can approach personalized all-to-all communication. This change is not due to the organization of the parallel program data but rather the memory caching. For example, when using a DSC system, multiprocessor simulators such as Simics [38] and SESC [50] interleave the memory caching based on cache line size or some other regular block regularities. Unless the application has been developed with this in mind, the interleaving in the system will not match how the program is parallelized. Thus, significant network traffic is required for the processors to access the appropriate data, resulting in seemingly random or all-to-all style traffic.

There are several research efforts attempting to optimize personalized all-to-all and all-to-all broadcast in mesh networks [59, 66]. Huang et al. highlights the non-uniformity of traffic in mesh-based NoCs and proposes non-uniform allocation of virtual channels for wormhole traffic to match the requirements of the application [26]. While this method does not actually increase the bandwidth of the heavily utilized links, it is an attempt to understand the bottlenecks in those heavily utilized links and to prevent under utilization due to link contention blocking. In contrast, this thesis presents a theory for construction of a non-uniform fat-mesh that does provide additional bandwidth for heavily utilized links.

2.2.1 Routing Algorithm on Mesh

The construction of a non-uniform fat-mesh is further impacted by the type of routing algorithm employed in the system. A routing algorithm can be defined as the path taken by a packet between source and target switches. It can affect the traffic load of the network fundamentality and thus deserves careful study and selection in network research. Routing algorithms can be classified into deterministic ones and non-deterministic ones according to their adaptivity [15].

A deterministic algorithm always supplies the same path between a source/destination pair while adaptive one uses information about network traffic and/or channel status to avoid congested or faulty regions of the network. XY routing [52, 12] is an example for a deterministic routing algorithm which routes packets first along the x-dimension and then along the y-dimension until the packet reaches the destination.

Adaptive or non-deterministic algorithms are more responsive to dynamic network conditions. For example, a routing table may include several options for an output channel based on the destination address. A specific option may be selected randomly, cyclically or according to the network status. There have been many research works exploring such non-deterministic random routing algorithms. A randomized routing analytical model on fat-trees is presented in [20] to prove that fat-trees are universal routing networks that any network can be efficiently simulated by a fat-tree. A randomized on-line algorithm introduced in [32] is a general paradigm for the design of packet routing algorithms for fixed-connection networks. The algorithm is proved to route n^2 messages on a $n \times n$ mesh in $2n + O(logn)$ parallel communication steps with very high probability.

Nesson and Johnsson develop a class of randomized routing algorithms called ROMM [44]. ROMM randomly chooses an intermediate node within the minimum rectangle defined by the source and destination nodes and routes packets via the intermediate node. The two phases of routing (i.e., from source node to intermediate node and from intermediate node to destination node) may use some variation of deterministic routing (i.e., XY-order or YX-order). While ROMM has been shown to offers a potential for improved performance compared to deterministic algorithms under light loads, recent work has demonstrated that in the worst case, ROMM may saturate at a lower throughput in 2D-torus networks [60] and 2D mesh [53].

Seo et al. propose the design of an oblivious randomized routing algorithm, O1TURN, which offers optimal worst-case throughput in two dimensional mesh networks [53], O1TURN allows packets to traverse one of two possible dimension-ordered routes that differ only in the order of dimension traversal (i,e, X-first Y-next or Y-first, X-next). Ramanujam and Lin expanded this design to 3D mesh [48]. While these algorithms provide randomized algorithms on general cases, we consider our XY-YX routing algorithm, which makes the random routing decision at each router, under a non-uniform fat-mesh network and in an all-to-all personalized traffic context, and show the performance improvement in the simulation.

2.3 HYBRID NETWORK

Many high performance computing systems use packet switching networks to interconnect system processors, this is also the default switch technology used in the SESC simulator. Packet switching networks send limited sized data through the network by adding routing information to the front of the data, thereby creating a data packet. Each data packet is independent of others. When a large amount of data needs to be communicated, multiple packets are created and sent through the network. A switch must read the header of each incoming packet in order to determine which output channel to use to send the outgoing message. For packet switched networks, the size and quantity of the buffers, the arbiter and the applications communication contention impact performance [25]. As the system gets larger and bit rates increase, a packet switching network can consume a disproportionately high portion of the system cost when striving to meet the twin demands of low latency and high bandwidth.

Circuit switching, on the other hand, creates hard circuits between endpoints in response to an external control plane and before the data are sent, obviating the need to make switching decisions at runtime. Because circuit switches create hard circuits instead of dynamically routed virtual circuits, they contribute almost no latency to the switching path aside from propagation delay. However, the establishing of such a direct circuit route incurs a high latency cost and once the circuit is established, it may block other circuits from forming and is difficult to re-configure at run time to suit the timely traffic pattern. Therefore, the performance of this method is impacted by the latency in establishing a circuit and by contention within the communication pattern [25].

Recently, many research efforts have been focused on exploring the possibility of applying a hybrid network with both a packet switching plane and a circuit switching plane in order to overcome the limitation of single switching technology. From the theory point of view, the authors of [31] investigate the effect of adding random links to mesh and torus networks under uniform traffic. Ogras and Marculescu present a design methodology to insert application-specific long-range links to standard grid-like networks [45]. they propose a link insertion algorithm to achieve throughput improvement and to ensure dead-lock free in the system. HFAST [28] is a hybrid switching architecture that uses circuit switches to dynamically reconfigure lower degree interconnects to suit the topological requirements of a given scientific application. As optical switching technologies become more practical in fabric technology advancement, they show the best promise of providing highest switch radix and throughput. Barker et. al. discuss the feasibility of an optical circuit switching network handling long-lived bulk data transfers and a secondary lower bandwidth electronic packet switching network [3]. Figure 1 shows the overview structure of this approach.

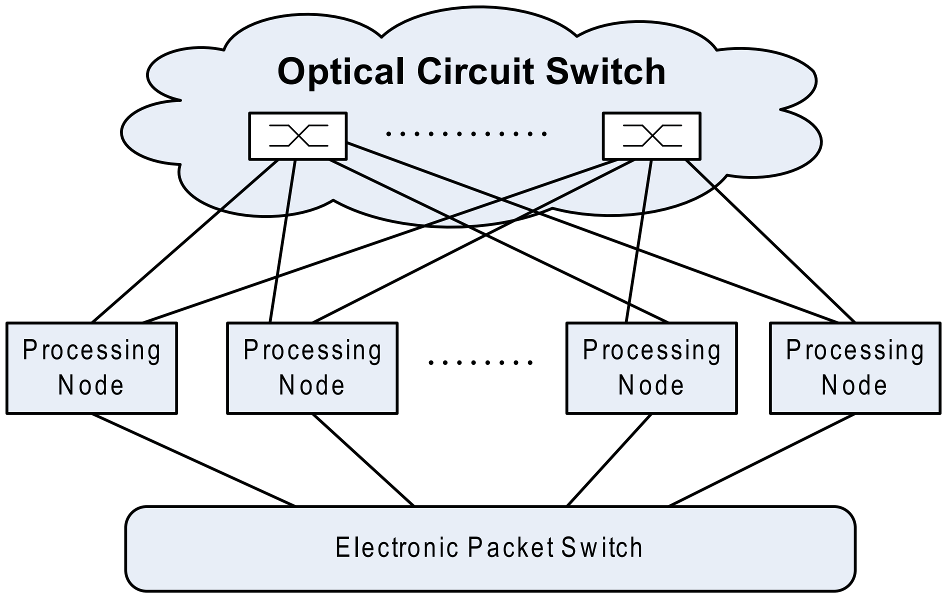
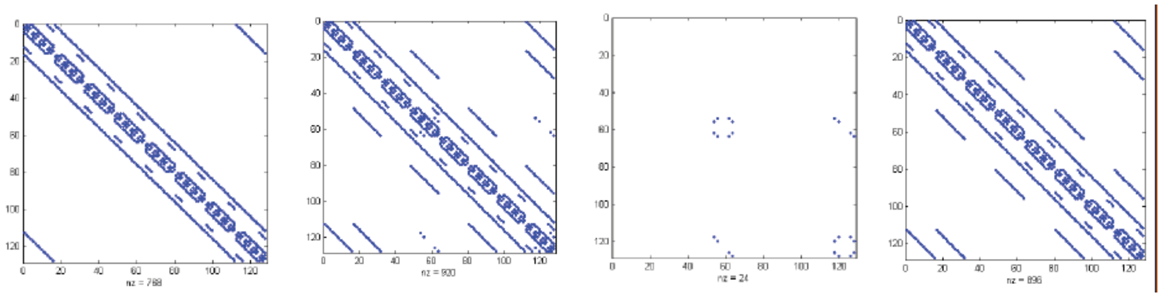


Figure 1: High performance computing with optical circuit switching

2.3.1 Communication Pattern Extraction

Enabling circuit switching in multiprocessor systems has the potential to achieve more efficient communication at lower cost compared to packet switching, but due to the circuit establishment and configuration overhead, communication in a parallel application needs to exhibit locality, and this locality needs to be appropriately exploited to reveal the communication pattern.

Shao et al. find a large portion of applications' communications which are classified as dynamic (communication cannot be determined until run-time when it actually occurs) to be actually persistent (communication's topology cannot be determined at compiler time, but the locality can be determined) [54], and this gives us a good opportunity to identify the traffic pattern and thus reduce the communication overhead based on the pattern. Figure 2 visually shows the regular communication matrix exhibited by the application MG from NAS benchmark suite [2]. A p-matrix represents all the point-to-point communications in an execution phase. Each entry of a p-matrix represents the weight of the corresponding point-to-point communication. From the figure, we can see each execution phase has one persistent traffic pattern represented by a matrix.



(a) p-matrices 0,1,6-10. (b) p-matrices 2,11. (c) p-matrix 3. (d) p-matrix 4.

Figure 2: Predicted p-matrices of MG. MG communication depend on run-time input data. 12 p-matrices were during the execution. Each sub-figure represent one or more phases of the execution.

Many interesting research projects acquire information about communication patterns by compiler techniques. For example, Cappello and Germain propose an approach to associate compiled communications and a circuit switched network [7]. Their compiled communication technique requires that a large portion of static communications be identified at compile-time. Dietz and Mattox study the Flat Neighborhood Network (FNN), which uses the communication patterns to determine the design of the network [13]. Liang et. al. describe a compiler which supports compile-time scheduled communication for their adaptive System On Chip (SOC) communication architecture [36].

However, in many cases the expression for a communication destination or volume will contain unresolved variables that cannot be solved at compile time. Thus, our approach manipulates symbolic expressions within the compiler to determine how these values are calculated. Symbolic analysis was a hot topic in parallel compiler research in the middle 1990s. Several researchers studied symbolic analysis techniques in the context of parallel compilation for High Performance Fortran (HPF)-like programs [22, 61]. Most of these works emphasized the parallelization of the program with the assistance of symbolic analysis, with a focus on loops and arrays.

Additionally, the target applications to which our compiler techniques are applied are MPI programs. Even though the message passing scheme is the dominant parallel programming model in distributed systems, there are few significant researches concerning applying symbolic analysis to this class of applications. However, Shires et al. did present an algorithm for building a program flow graph representation of message passing applications [57].

This thesis, based on an MPI parallel compiler we developed [54], explores using symbolic expression analysis techniques to identify and represent the communication pattern in MPI parallel applications at compile time and also to assist in the circuit switching establishment. In particular, symbolic analysis for variables embedded in loop structure is proposed as a complement for the existing compiler aided communication pattern analysis.

3.0 CONSTRUCTION OF MULTI-CORE SIMULATOR

We developed our multi-core simulator based on SESC. SESC is a microprocessor architectural simulator developed primarily by the i-acoma research group at the University of Illinois Urbana-Champagne (UIUC). It models different processor architectures such as single processors, chip multiprocessors and processors-in-memory, and thread-level speculation.

An emphasized feature of SESC is the partition of functional emulation and timing simulation, resulting in a fast simulation speed. We choose SESC as our baseline simulator not only because of its fast speed, but also for its ease for extension. SESC is written in a modular fashion to facilitate extend additional architectures upon the given package. SESC also provides a configurable cache system, which is easy to reconfigure and extend.

This chapter will first go through some important features and aspects of SESC that relate to our simulator extension. After that, several extensions made to the SESC will be presented, including a DSC system with directory based cache coherence protocol and a distributing network module.

3.1 SESC SIMULATOR BACKGROUND

Figure 3 is a concept diagram of SESC. The simulator has three major components. An execution flow controlled by MINT that actually executes the input binary instructions. The processor module models the processor pipeline which fetches the instructions from the execution flow and lets the instructions flow through. Each processor has its own memory system, which consists of local caches and buses connecting to other nodes' caches. A memory requests is generated at the processor when the instruction needs load or store operation and injected into the memory system. It acts as an information exchanger and keeper in the interaction between the processor and memory system. The next three sections describe these major components in detail.

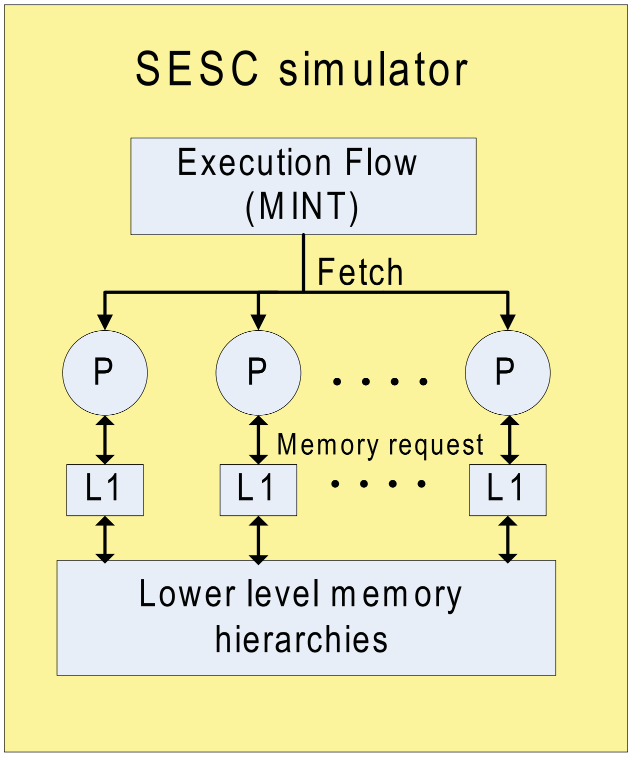


Figure 3: SESC original architecture view.

3.1.1 Core System Overview

SESC utilizes a MIPS emulator, MINT, to actually execute the instructions. The emulator returns instruction objects to SESC, which are then used for the timing simulator. These instruction objects contain all the relevant information necessary for accurate timing. The generic processor fetches the information to calculate how much time it will take for the instruction to be executed through the pipeline.

The processor object coordinates interactions between the different pipeline stages. It does this by first calling a function to fetch one or more instructions into the instruction queue. SESC supports several different branch predictors which are handled in the fetch stage. The choice of predictor and its size is occurs at run-time. Processor then calls a function to issue instructions from the queue into a scheduling window. There are two clusters that schedule and execute instructions, one for integer and one for floating-point instructions, and each has its own scheduling window. Instruction dependencies are solved in the issue stage. Instruction are scheduled to execution until input operands are ready, which causes the out-of-order execution. The execution stage is finished upon resource availability. Finally, a function is called to retire already-executed instructions from the reorder buffer.

The core of the processor is modeled as an execution-driven simulator. In other words, functions are called to simulate some parts of the processor every cycle. The rest works as an event-driven simulator. The *CallBack* class and its subclasses let the programmer schedule the invocation of a function call at a given time in the future. This is particular useful in timing simulation of the memory system, because actual latencies in the cache and bus are not predictable; only a callback can give a cycle-accurate simulation result. The system maintains a callback queue. Every callback is inserted into the queue after being generated. The callback object records the calling object, the member function it is calling, the function parameters, and the future cycle at which we call the function. In each simulation cycle, the callbacks stored in the current cycle slot will be called.

3.1.2 Cache and Bus Interconnect Overview

In a typical modern micro-processor, there are several levels of caches. At the uppermost level are L1 caches, which usually have one I-cache for instruction and one D-cache for data. Below L1 is a larger, slower L2 cache. In many configurations, there is also an L3 cache either on-die or off-die.

The cache implementation in SESC is necessarily quite complex and uses many event-driven callbacks. This is necessary to model all the latencies and transactions involved in caches. When the processor needs to interact with the memory system, it creates a memory request. The type of memory request depends on the type of memory operation (e.g. read or write), the highest memory hierarchy (e.g. I-cache or D-cache), etc. The data itself is not actually modeled in the cache system, as we only care about the data flow and timing information. After the memory request is initialized, a function is called to access the highest level cache object.

The cache object receives the requests from the upper level, and processes operations according to the request type. When the cache line is found and the requirements are satisfied, it schedules a callback for the upper level object to call in the future, depending on the cache port's availability. If a cache miss occurs, the memory request goes down to the next level cache to fetch the data.

One key issue in a shared cache multiprocessor is the cache consistency model which defines the cache coherence. As L1 caches are private, different processors may see different values of the same memory location. Therefore, cache coherence is mandatory in the system. SESC provides bus-based snoopy coherence protocol, shown in Figure 4. Snoopy bus is a popular cache coherence policy in a small scale multiprocessor system. It has a global bus which connects all caches and memory banks. All memory addresses are broadcast on that global bus. All caches and memory banks "snoop" (listen to) that bus. The operations require broadcast since caching information is in the processors. Bus is a natural broadcast medium which also serializes requests on it.

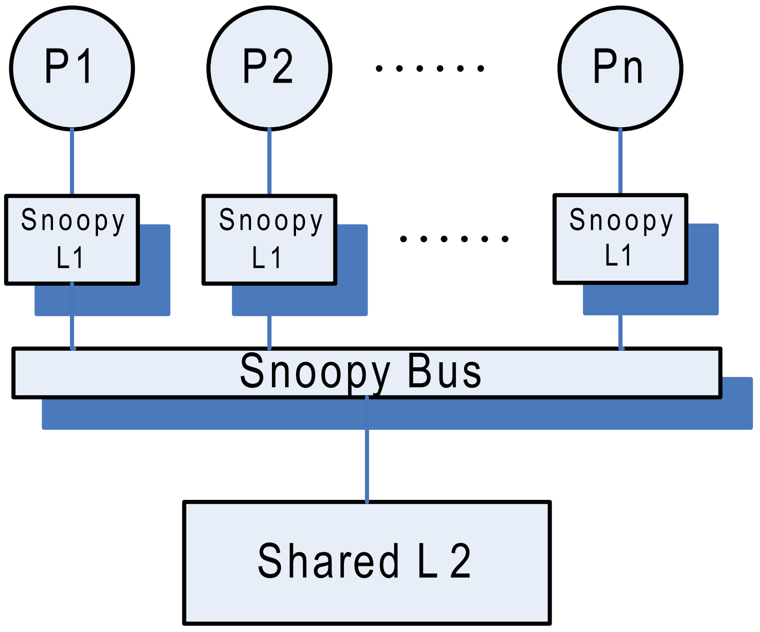


Figure 4: Bus based shared memory SMP structure provided by SESC.

SESC employs a MSI write-invalidate, write-back snoopy protocol. In a write-invalidate policy, the writing processor forces all other caches to invalidate their copies in a write operation. It produces less bus traffic than a write-update policy, in which the writing processor forces all others to update their copies. In a write back protocol, the memory is updated only when the block in the cache is being replaced. It produces less bus traffic than a write-through policy, in which the memory is updated every time the cache is updated. A snoopy bus coherence state machine is illustrated in Figure 5. In the protocol, a read miss causes all caches to snoop bus and send the data to the requestor if one has the data. A write to shared data causes invalidates to be sent to all caches which snoop and invalidate copies.

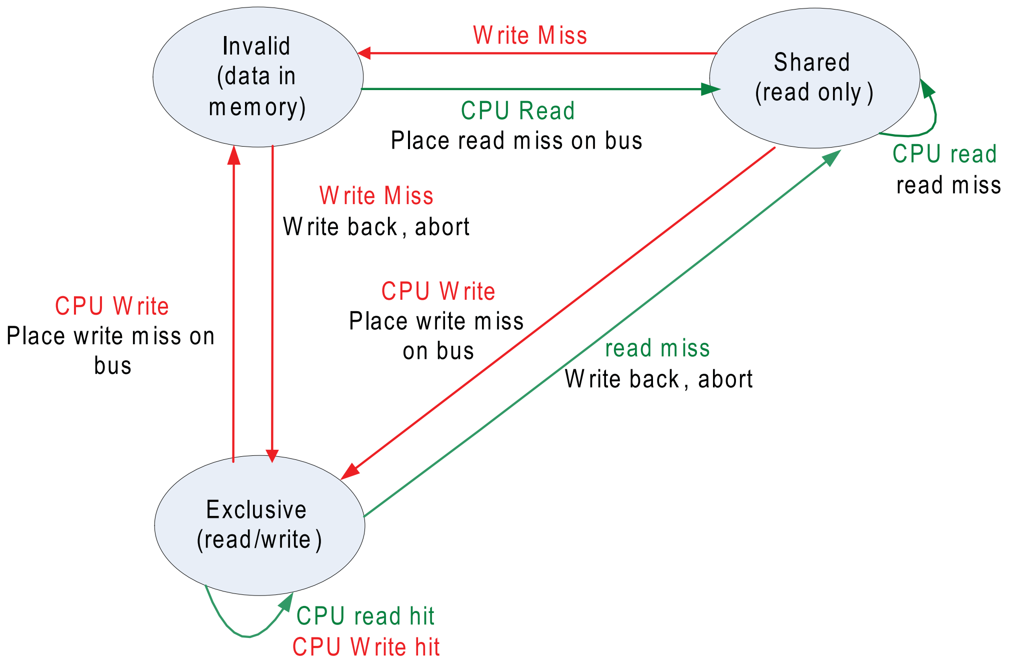


Figure 5: Bus based snoopy MSI protocol state machine.

3.1.3 Network Module

SESC provides a network module which is neither plugged into SESC nor working alone, and thus needs thorough modifications. Figure 6 shows a UML representation of network module coming with the SESC package. The InterConnection class represents the whole network layout. An InterConnection object is defined by two components:

- A set of Router objects: The Router class represents a router in an interconnection network. Each router contains a routing table represented by the RoutingTable class, which stores network path and status information. The RoutingTable has the Wire objects that connect routers to each other. The Wire class represents an unidirectional link between two routers. Each

Router is defined by an ID and a set of parameters that model the dynamic behavior of the switch, including router crossing latency, local port latency, and number of ports for each addressable local ports.

- A RoutingPolicy object: This class is in charge of building the routing tables for a given network configuration, such as fully-connected network, uni-ring, bi-ring and hypercube. This is the place to extend different routing topologies and algorithms.

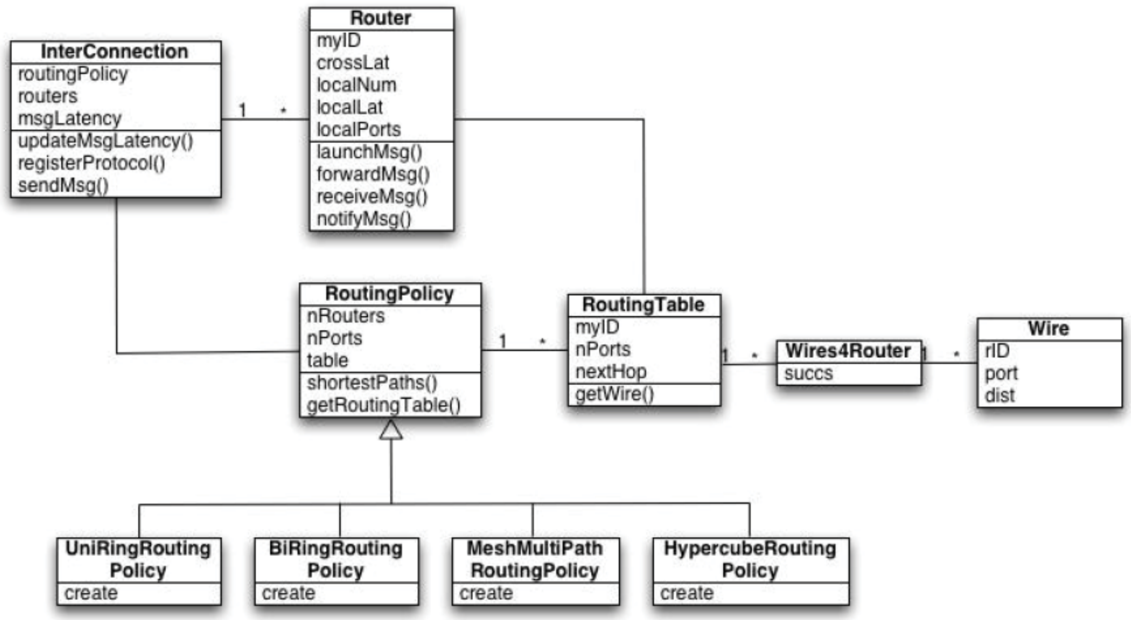


Figure 6: SESC Interconnection network class organization.

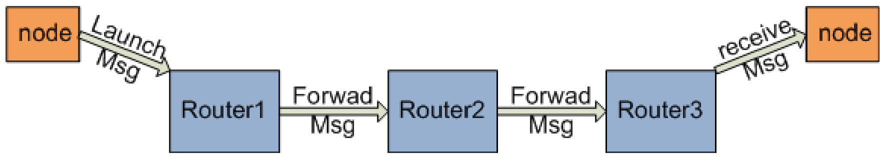


Figure 7: Message traversal procedure in SESC network.

Figure 7 conceptually shows the network traversal process. A message is injected in the network via a launchMsg function. Messages are sent from router to router using a forwardMsg function. Each router selects the proper routing path according to its routing table and the message's destination, and then it schedules a callback depending on the router port's availability. Once a message arrives at its destination, a receiveMsg is invoked.

3.2 SESC SIMULATOR EXTENSIONS

While the original SESC can adequately simulate current CMP architectures with a small number of cores, future proposals require more parallelism of multi-threaded applications to achieve higher performance. This is because data access is typically a bottleneck in the systems, as multiple processors compete for limited on-die memory resources. Two major factors impacting the latency of on-chip memory access are wire delays and contention over shared memory. Hence, the organization and management of on-chip cache memory and network become critical to system performance.

To solve this memory access latency problem, CMPs are shifting towards a Non Uniform Cache Architecture (NUCA) [4], where the cache is divided into multiple banks, and accesses to closer banks result in shorter access times. In NUCA, performance depends on the average (rather than worst-case) latency [21]. The division of the cache into multiple banks also allows multiple processors to access different banks simultaneously, thus reducing contention.

However, such a cache scheme demands a very efficient interconnection network to minimize total access time. Buses, while heavily used today (in original SESC as well), suffer from resource contention issues; as the number of nodes increases, performance degrades due to excessive conflicts. Hence, they are not considered appropriate for systems of more than about 10 nodes [35]. To overcome these limitations, attention has shifted toward NoCs, which provides distributed communication to alleviate the contention. The most popular NoC topology in use today is the mesh, in which nodes are tiled in a grid.

Based on the above observations, several extensions to the original SESC architecture are explored and implemented to suit the future many-core CMP requirement.

3.2.1 Distributed Shared Cache

While a shared cache system has a single global physical memory equally accessible to all processors, a distributed cache system consists of multiple processing nodes communicating by means of message passing. This type of system usually exhibits poor programmability and portability because all data partitioning and explicit communication must be done by the programmer.

DSC, using the idea of NUCA, combines the advantages of distributed system and shared system and overcomes their difficulties by providing a global single address space on top of physically distributed cache systems. DSC combines the ease of the shared memory programming paradigm with the scalability and constructability of distributed cache systems. Thus, DSC and its variants are extensively used in multiprocessor systems nowadays.

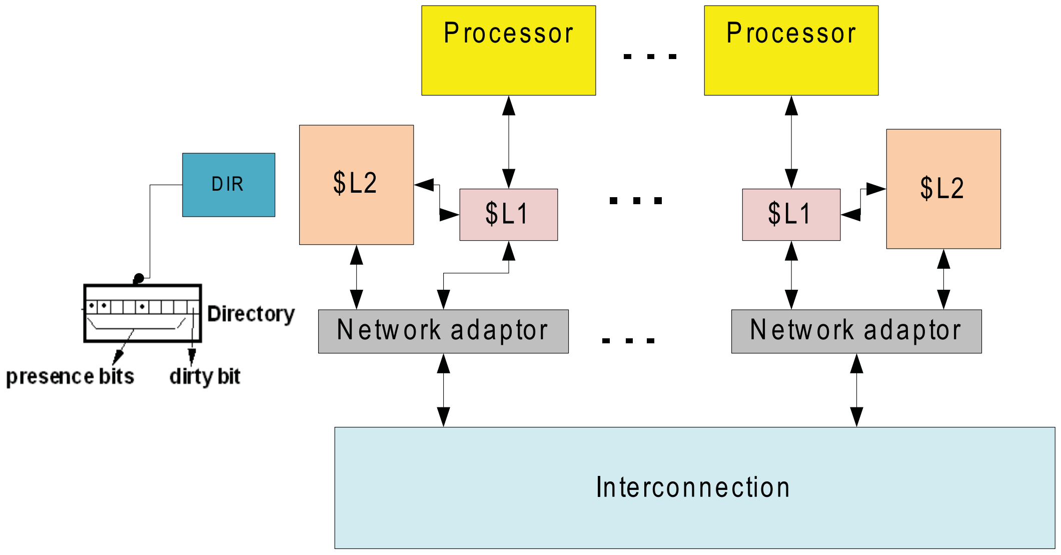


Figure 8: Distributed shared cache implemented in SESC simulator.

We implemented a DSC system in the SESC. Figure 8 displays an overview of what the distributed shared system looks like in the simulator. In such a configuration, L1 caches are private to their processors, and L2 cache is physically distributed, logically shared by all processors. Thus, each node contains one processor, its L1 cache (including I-cache and D-cache) and its portion of the L2 cache. A network adaptor is modeled so that cache misses and coherence messages can travel among nodes via interconnection. The adaptor communicates to both L1 and L2 as required by the coherence protocol.

One issue in implementing a distributed shared cache system is the interleave granularity which depends on the memory address mapping. A fine granularity destroys the natural traffic pattern of the application and leads to random all-to-all communication. A coarse granularity, on the other hand, creates a traffic hotspot and makes the computation and communication unbalanced. Take Figure 9 as an example. In the simulator, we have 32 bit memory address space, the cache line block size is set to 64 bytes, so we have 6 offset bits. The 8 way associate set, 2 Megabyte L2 makes 12 bits set index. The other 14 bits are for the tag area. Most multiprocessor simulators like Simics and SESC interleave the memory caching based on cache line size or some other regular addressing. For a 16 node multiprocessor system, we need 4 bits to determine the mapping. Utilizing bit6 to bit9, as shown in Figure 9, leads an interleave granularity at cache line size, which in turn destroys the intended traffic pattern of the application.

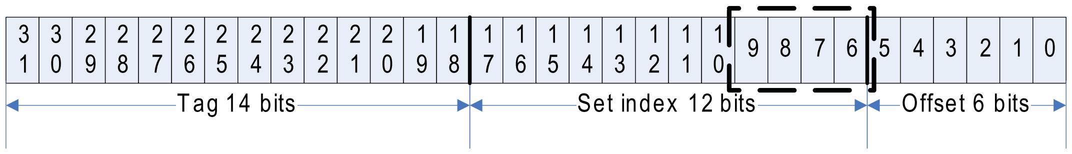


Figure 9: Memory address partition and granularity level.

3.2.2 Directory Based MSI Cache Coherence Protocol

The address space of a DSC is logically shared across the caches or memories of the processors. To reduce traffic over the network, a replication of certain data stored at a remote processor is usually created in the local cache of a processor. Multiple copies of data, while improving read performance, create a need for coherence enforcement.

The snoopy bus based cache coherence scheme in the original SESC does not scale because they rely on broadcast, hierarchical snoopy bus which is the root of bottleneck. On the other hand, directory based schemes allow scaling as they avoid broadcasts by keeping track of all processors caching a memory block(e.g. the location of the caches that have a copy of a shared data item are known). This means that a broadcast is not required and individual messages can be sent using point-to-point messages to maintain coherence. This brings flexibility in using any scalable point-to-point network such as a mesh structure as opposed to just a bus. The absence of broadcasts eliminates the major limitation of scaling cache coherence multiprocessors to large number of processors.

Directory based coherence uses a directory for each cache line to track the state of every block in the cache. Centralized directory will soon become a bottleneck when the processor number increases, therefore, a distributed directory is implemented to alleviate creating of a centralized hotspot. A fully featured distributed directory based MSI cache coherence is implemented in SESC.

The coherence protocol maintains three states:

- **Modified:** data is in only one processor's L1, and different from L2 (out-of-date).
- **Shared:** data is in one or more processors L1, and the same as in L2 (up-to-date).
- **Invalidate:** not valid in any L1.

The directory must trace each cache line's state and which processor has data when the cache line is in a Shared state. This is usually accomplished by keeping a bit vector. The vector's bit is set when a corresponding processor has the data. Another dirty bit is used to indicate if the cache line is in Modified state. Typically, three processors will be involved in one transaction:

- **Request node:** the node which issues a memory request on the target cache line.
- **Home node:** the node to which the target cache line belongs.
- **Remote node:** the node's L1 cache contains the target cache line (modified or shared).

Specific coherence messages must be created to transfer the control information and actual data among the involved nodes in the coherence transactions. Table 1 gives a detailed description of the message types we created to participate in the MSI coherence.

Figure 10 and Figure 11 illustrate the state transition from the CPU and directory point of view, respectively. Take the directory transition as an example. If a block is in the Invalid state, which means the copy in memory is the current value, possible requests for that block are read miss or write miss. Read miss causes the requesting processor to be sent data from the memory, the requestor to be added as a sharing node and the state of block to be made Shared. Write miss makes the requestor and block state Modified. If the Block is Shared, which means the memory

Table 1: Directory Protocol Messages List.

Message type	**Source**	**Destination**
Read	Local cache	Home directory
Processor P’s read request reads directory associate with address A		
Write	Local cache	Home directory
Processor P’s write request reads data at address A		
Invalidate	Home directory	Remote caches
Home directory needs to invalidate a shared copy at address A in remote L1		
Invalidate ack	Remote caches	Home directory
After invalidation, remote caches sends an ack back to home directory		
Fetch	Home directory	Remote cache
Home directory needs to fetch the block at address A		
Fetch-Invalidate	Home directory	Remote cache
Home directory needs to fetch the block at address A ; invalidate it afterwards		
Read reply	Home directory	Local cache
Read miss response, act as an acknowledgement, make P a sharer		
Write reply	Home directory	Local cache
Write miss response, act as an acknowledgement, make P Modified owner		
Write back	Remote cache	Home directory
Remote cache writes back a data value for address A to home direcotry		

value is up-to-date, a read miss has the same behavior as in Invalid block case. A write miss will invalidate all processors in the set Sharers, Sharers is set to the identity of the requesting processor, and the state of the block is made Modified. If the Block is Modified, which means current value of the block is held in the cache of the processor identified by the set Sharers (the owner), in a read miss, the owner processor gets a data fetch message, changes its state to Shared and sends data to directory, where it’s written to memory and sent to the request processor. The requesting processor is added to the set Sharers, which still contains the owner (since it still has a readable copy). In a data write-back, the owner processor is replacing the block, and hence must write it back, making the memory copy up-to-date (the home directory essentially becomes the owner); the block is now

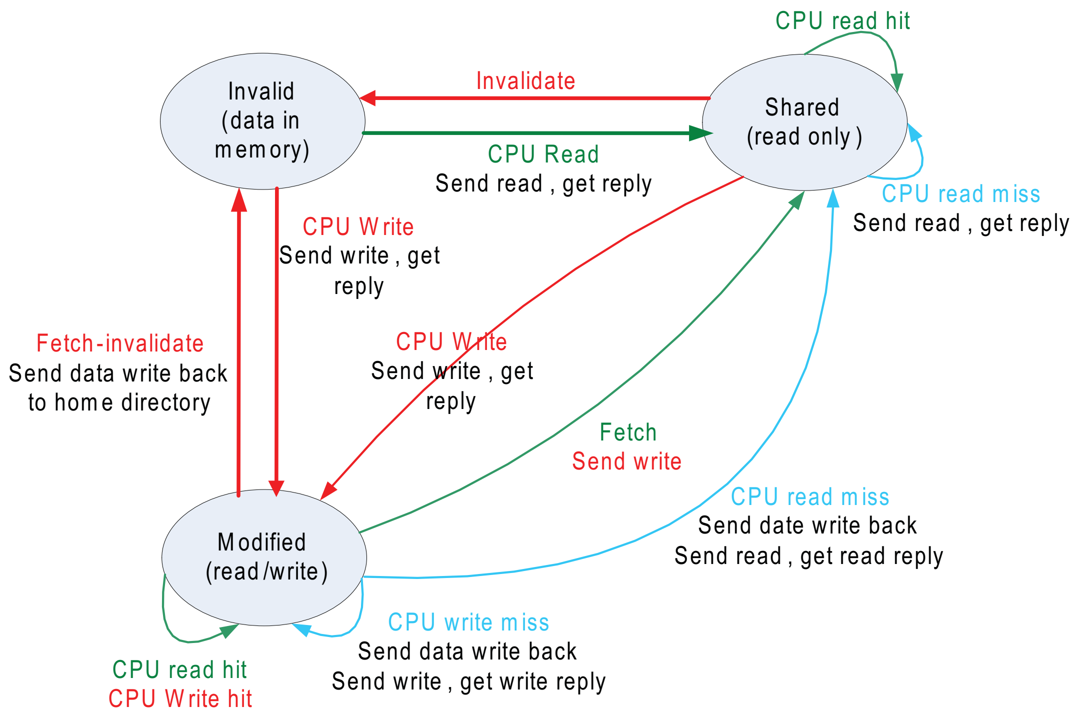


Figure 10: State machine for CPU requests for each memory block.

Invalid, and the Sharer set is empty. In a write miss which means the block will have a new owner, a message is sent to the old owner causing the cache to send the value of the block to the directory from which it was sent to the requesting processor, which becomes the new owner. Sharers is set only to the new owner, and the state of block is made Modified.

The above cache coherence is implemented in the simulator. Two cases require special mention here. First, in a write miss when the block is in Shared state, the write can only proceed after all invalidation acknowledgments come back from the sharers. A barrier must be set to synchronize all the messages. Second, in a write hit to a Shared block. When a write hit happens in L1, a similar action should be taken as changing a Shared node to a Modified one though it is a hit in the cache.

The race condition is also considered in the system. When two memory requests need to access the directory for the same cache line, the directory will serve the first coming request. After the coherence operations begin, the entry of this cache line in the directory will be locked. The second

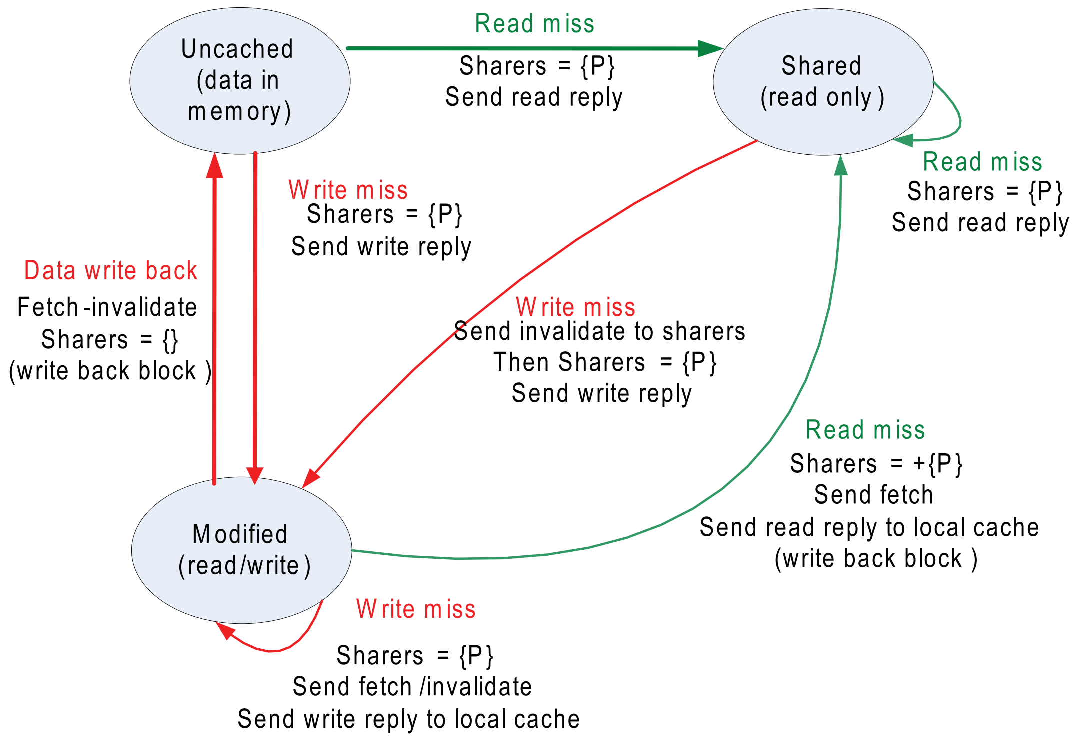


Figure 11: State machine for Directory requests for each memory block.

request will be put in a pending buffer until the first request finishes all its operations. After that, the second request will be retrieved from the pending buffer and access the corresponding directory entry which is possibly in a new state.

3.2.3 Network Implementation

As stated above, the interconnection provided by SESC is a centralized non-blocking snoopy bus which easily becomes the traffic hotspot and bandwidth bottleneck when the core number gets large. Therefore, a distributed network module is implemented to replace the bus to interact with the DSC for scalable inter-processor communication.

By modifying the network module provided in the SESC package, I managed to get two working versions of network simulator. The first one is a standalone network simulator. Input to this can be a trace of memory requests or regular memory traffic patterns such as all-to-all communication or a nearest neighbor pattern. The other version is an integrated network module that cooperates with the core SESC system. We can carry a full benchmark simulation using this combined simulator. The implementation of the network module includes several aspects, following sections walk through them.

3.2.3.1 Routing topologies and algorithms The routing topology and routing algorithm depend on each other. Some routing topologies have obvious and fixed routing algorithms, others require more complex and versatile algorithms. The network module can be configured with various routing topologies as a runtime parameter, including uni-ring, bi-ring, torus, hypercube, full-connected, and it is also extended to support mesh and fat-mesh as described in Section 4. Once the topology is fixed, a routing algorithm needs to be chosen.

All of the routing algorithms implemented in the simulator are distributed routing algorithms, in which the routing function is calculated in routers as the packets travel across the network. Headers contain the destination address, used by routers to select output channel(s). Each router knows only its neighborhood as the whole topology is encoded distributively into individual routers. When creating the routing topology, a routing table is generated for each router depending on the location of each router in the topology. Every routing table is static and different from any other. The number of table entries is O(N), where N is the number of nodes in the network. Each entry in the routing lookup tables indicates which output channel(s) correspond(s) to every destination.

In our implementation, some topologies apply deterministic routing algorithms. For fully-connected topology, the routing algorithm is obvious, each port corresponds to one other destination. For ring topology, the routing algorithm is also fixed, as the destinations on the left using the router's left port, the destinations on the right using the right port. In mesh-based topologies, a dimension-order routing (XY) [52] [12] is implemented as the default routing. It routes packets first along the x-dimension and then along the y-dimension until the packet reaches the destination. It is known to be deadlock-free in meshes and is easy to implement in hardware.

Distributed routing algorithms do not take into consideration any other information except the addresses while can provide some degree of adaptivity. For example, there may exist several shortest paths between the source and destination, the routing algorithm selects randomly or cyclically one of them. Distributed non-deterministic algorithms can uniformly spread the communication load in situations where adaptive solutions are too expensive or slow. In torus and hypercube, SESC provides such a routing alternative. First, a Dijkstra-like algorithm is leveraged to find all shortest paths in the mean time, then the routing decision is made cyclically among all the choices when the packages arrive at the router. The purpose of this routing algorithm is to distribute the traffic more evenly among the four ports of the router. When applying to mesh without wrapped-around wires, the algorithm becomes a XY-YX routing, described in Section 4.3.

3.2.3.2 Router structure The router structure also depends on the routing topology. Figure 12 is a concept diagram of the router structure modeled in the simulator for the dominant mesh topology. The router consists of four input buffers and four output ports for external input/output channels, one local buffer for processor packet injection and one local output port for packet ejection. The input buffer stores the incoming packages and output ports control the outgoing flow. The port's link bandwidth can be configured so that only a limited number of packages are allowed to proceed in each cycle. The pending packages are blocked until the port becomes available.

One of the limitations of our current network implementation is that SESC assumes an unlimited input buffer in the router. Thus, routers can always accept incoming messages from upstream. However, we still model the port contention such that the outgoing message flow is limited by the port's link width. The reason for an unlimited buffer is that for a non-deterministic routing algorithm such as XY-YX, the unlimited buffer ensures the store-and-forward packet switching network will not deadlock. When the traffic load is not too heavy the unlimited buffer size does not appear to unfairly impact the results. For example, our study of personalized all-to-all communication resulted in the routers being required to store a maximum of only eight messages at a time in a 10×10 size mesh. Smaller meshes require even smaller buffer size than that.

3.2.3.3 Switching techniques The switching technique implemented in the simulator is store-and-forward switching, alternatively called packet switching. The message is split into fixed-length packets; each packet contains header information that includes a destination address. Every packet is individually routed from the source to the destination. One step of the packet switching is called hop. It consists of copying the whole packet from one output buffer to the next input buffer. Routing decisions are made by each intermediate router only after the whole packet was completely

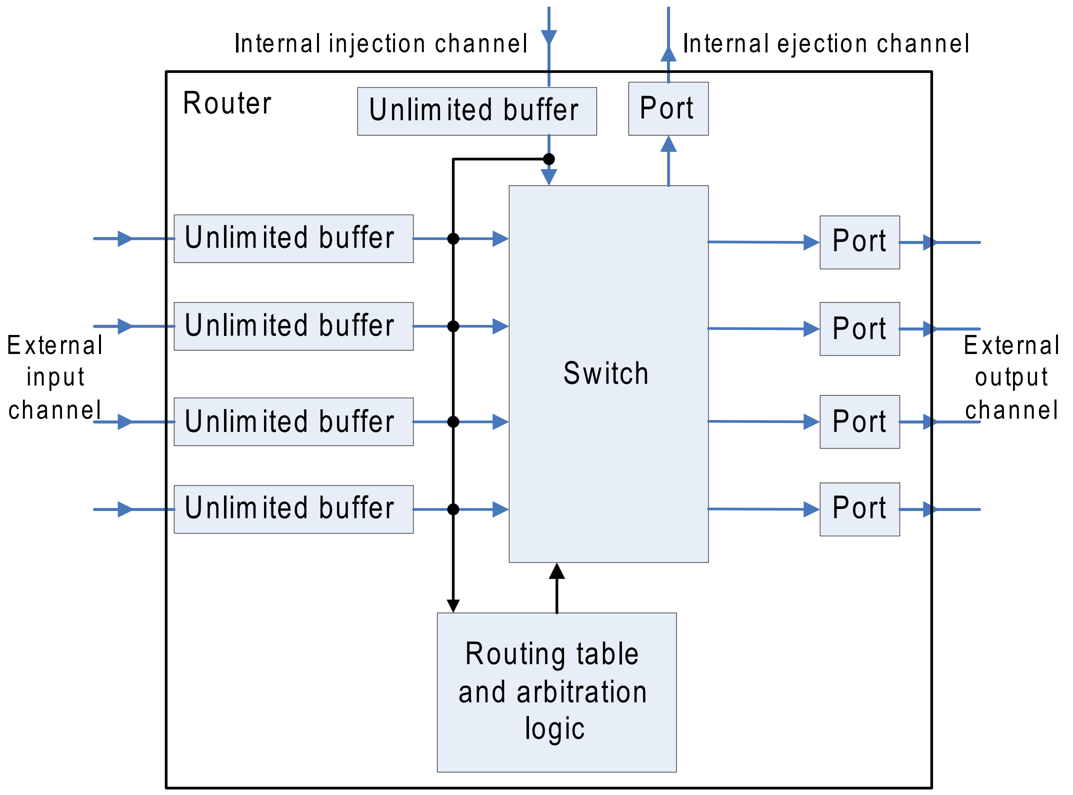


Figure 12: Mesh-based router implemented in the simulator consists of input buffers, output ports, a switch and a routing table.

buffered in its input buffer. This technique is advantageous when messages are short and frequent, since one transmission makes busy at most one channel from the whole path.

Figure 13 describes the procedure of packet switching. In this figure, we assume a scenario in which routing tables are static during the time of message transfer (i.e., all packets traverse the same path). While this is not a valid assumption under all circumstances, it serves the purpose of motivating a cost model for message transfer. Assume a message of M words is broken into packets, and the packets are assembled with header information. The message transmitted over distance d takes a time t, determined by:

$$t_{packetswitching} = d(t_r + (t_w + t_m)M) \tag{3.1}$$

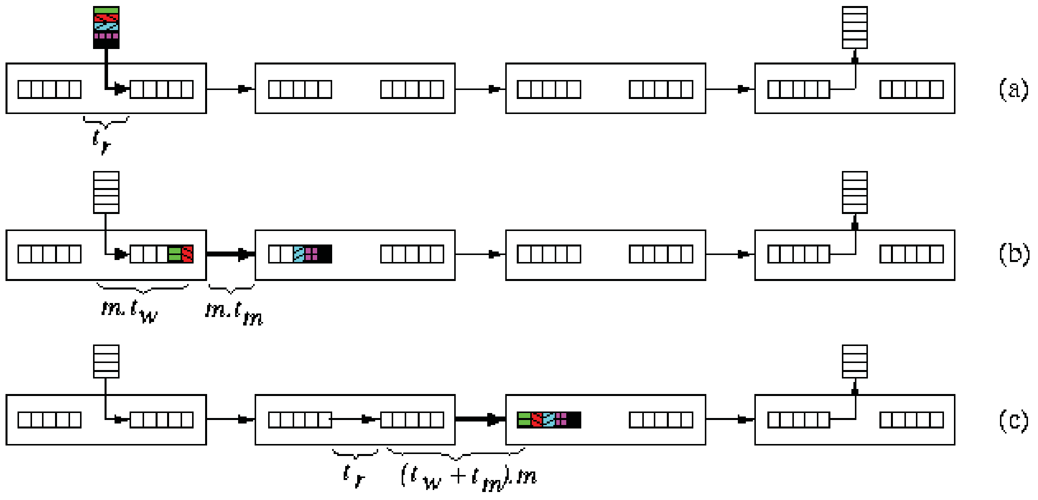


Figure 13: Store-and-forward switching of a message. (a) Routing decision is being made in the first router. (b) The packet is performing the first hop to the second router after has been copied to the output buffer of the first router. (c) The whole packet after the second hop.

where t_r is the time of the routing decision within one router while building a routing path, t_w is the intra-router switching latency once the router has made the routing decision and a path has been set up through the router, and t_m is the inter-router channel latency. At every router, routing decisions must be made, and then the packet hops to the next router, which takes $t_r + (t_w + t_m)M$ time. This repeats d times. In real implementation, we assume $t_m M$, the inter-router link traversal time, is a parameterized constant value not dependent on the size of the packet. The size of the packet M, router crossing latency t_w, and router decision latency t_r can also be configured.

As seen in Eq. (3.1), the communication latency is proportional to the product of packet size and distance. This forces the designers to seek the shortest path routing. While the equation indicates the minimum message traversal latency in the packet switching network, the practical latency could be much longer due to contention. Therefore, the message latency in a packet switching network could be fairly high in heavy traffic circumstances.

3.2.3.4 Interaction with the cache system After implementing the stand alone network simulator, we replace the bus interconnect provided by SESC with this distributed network module to enable a full benchmark simulation. Figure 14 shows the system diagram after the integration of mesh network. The processors are no longer organized in a bus 1D array, but in a mesh-based 2D grid. The network module needs interaction with both the L1 and L2 cache. As L2 is shared (logically) by all processors and there is only one instance of a L2 cache object in the simulator, L2 is a good place to create the network instance in the system. Therefore, the network module is attached to the L2 object which is in charge of creating the network and connecting it to every upper level cache (L1). In the system, cache system, either L1 or L2 bank, injects messages (listed in Table 1) into the network. After messages traverse the network, the network schedules a callback at L2 or the corresponding L1 to send messages back to the memory system. The injection and ejection of the messages are controlled by the network adaptor, which basically models a port controlling the in/out flow.

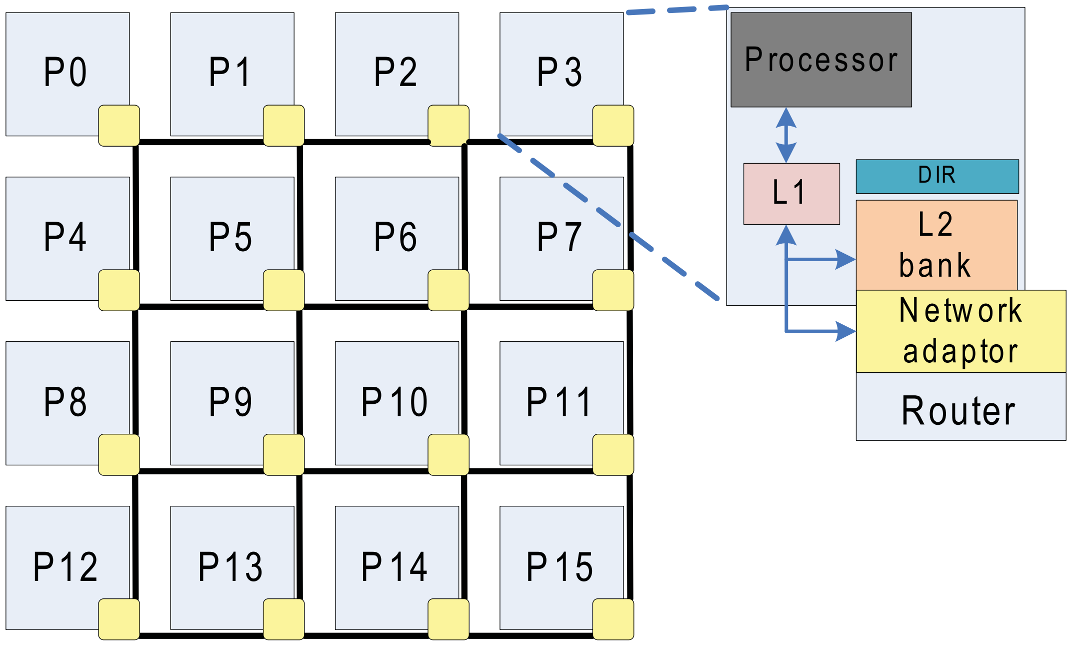


Figure 14: System diagram with network implementation.

Cooperation among the core system, private L1, distributed shared L2 and network module is critical to a complete multiprocessor simulation. Such a multiprocessor system, with distributed shared caches and a distributed network, especially a mesh network, is a popular configuration in recent CMP trend. The following two chapters, Section 4 and Section 5, introduce our two novel architecture designs based on the extended simulation platform.

4.0 FAT-MESH

In this chapter, the novel network structure non-uniform fat-mesh is presented as a method to provide additional bandwidth to highly utilized links in proportion to the requirements dictated by the traffic (e.g., all-to-all personalized communication) in many-core processors. Starting from the analysis of a linear array, we observe that by providing additional links near the heavy traffic connections will provide an opportunity for more efficient communication. We expend this idea to the mesh topology, and derive two non-uniform fat-mesh schemes using different routing algorithms. Detailed analytical models for both of them will be presented respectively by modeling all-to-all traffic.

4.1 FAT LINEAR ARRAYS

A linear array is one class of sparse network that often compared when studying dedicated interconnect topologies for a multiprocessor systems. Linear arrays are not employed directly as an interconnect paradigm, however, they can be used to construct meshes, a structure widely used in parallel computing and of significant interest for multi-core processor architectures exceeding a handful of cores. A linear array is constructed where each processor has two neighbors, one to the left and right, with the exception of the ends, which have a single neighbor.

While traditional parallel systems contain links that are typically inter-chip in multi-core systems, processor location and wire length become increasingly important. For example, a ring (or a one dimensional torus) can be constructed by connecting the ends together. However, in a chip level design a ring would increase the wire density and interconnect wire length of the chip leading to larger, higher-power consuming devices compared to a linear array. This problem becomes exacerbated when dealing with mesh interconnects and will be revisited when discussing meshes in Section 4.2. Therefore, in chip level design, linear array is more often used than ring topology.

Figure 15 shows an example of a linear array for a system with N processing nodes connected to the interconnect. The nodes are numbered from 0 to $N-1$. Additionally, each link is labeled from 0 to $N-2$. Thus the node number matches the link number to the right of each node.

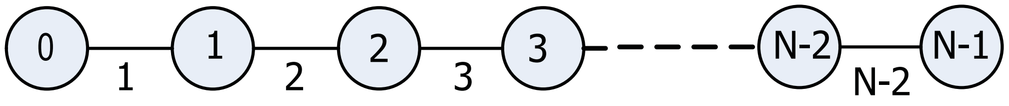


Figure 15: Generic linear array example.

Consider the case of all-to-all personalized traffic on the linear array, a worst case message transmission requirement. Typical algorithms will optimize the link utilization and sequentialize the messages (e.g. [18]). However, if multiple links are used to allow multiple messages to traverse the same hop simultaneously, the messages can be delivered more quickly.

We have used *traffic density* to indicate number of messages that must traverse a link in an all-to-all personalized communication. Figure 16 shows the traffic density for $N = 2..6$ linear array. The traffic density is non-uniform, with a much higher density near the center of the array with lower densities to the exterior. Thus, to efficiently utilize this structure this traffic suggests that increasing the bandwidth of the links toward the center compared to the outside links will allow more efficient communication.

For an all-to-all personalized communication the number of the messages to be completed in the system is dictated by Eq. (4.1), where N is the number of processors. The total messages is of the $O(N^3)$. However, the number of messages required to traverse a particular link is described by Eq. (4.2) where i is the link index shown in Figure 15 and N is the number of processors.

$$messages = \frac{N}{3}(N^2 - 1) \tag{4.1}$$

$$messages_i = 2(N - i - 1)(i + 1) \tag{4.2}$$

Thus, it seems clear that by providing additional links near the center of the linear array as shown in Figure 17 will provide an opportunity for more efficient communication. This special

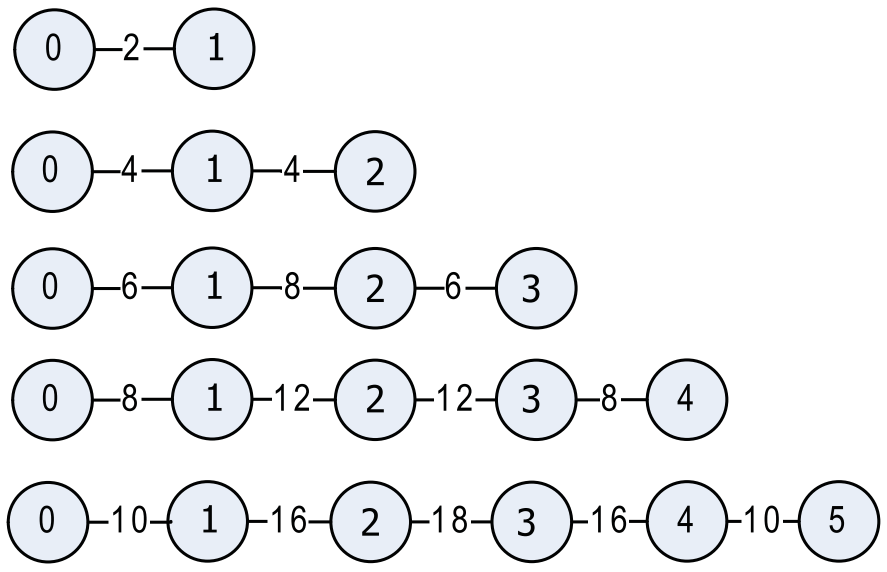


Figure 16: All-to-all personalized messages on linear arrays with various N values.

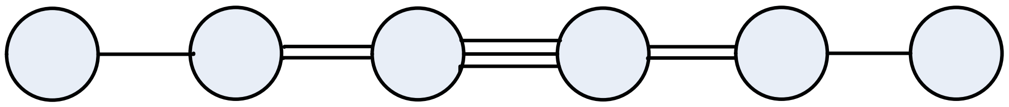


Figure 17: Example of a non-uniform fat linear array.

linear array can have multiple links at some connections (fat property), and the link size can be different depending on the locations of the connections (non-uniform property), thus we call it a non-uniform fat linear array.

However, unlike with a fat tree, the number of links at each location is not obvious. If all-to-all personalized communication is used as a guide it is possible to normalize the expression from Eq. (4.2) by the number of messages by the outermost link where $i = 0$ arriving at Eq. (4.3).

$$capacity_i = \frac{(N - i - 1)(i + 1)}{(N - 1)} \tag{4.3}$$

The normalized value from Eq. (4.3) decides the number of links in each connection of a linear array and can be used to construct the non-uniform fat linear array. This equation also shows the non-uniformity of the traffic traversing on each link in the linear array. The worst traffic scenario happens in the center link(s) of the array, which require(s) the most number of links.

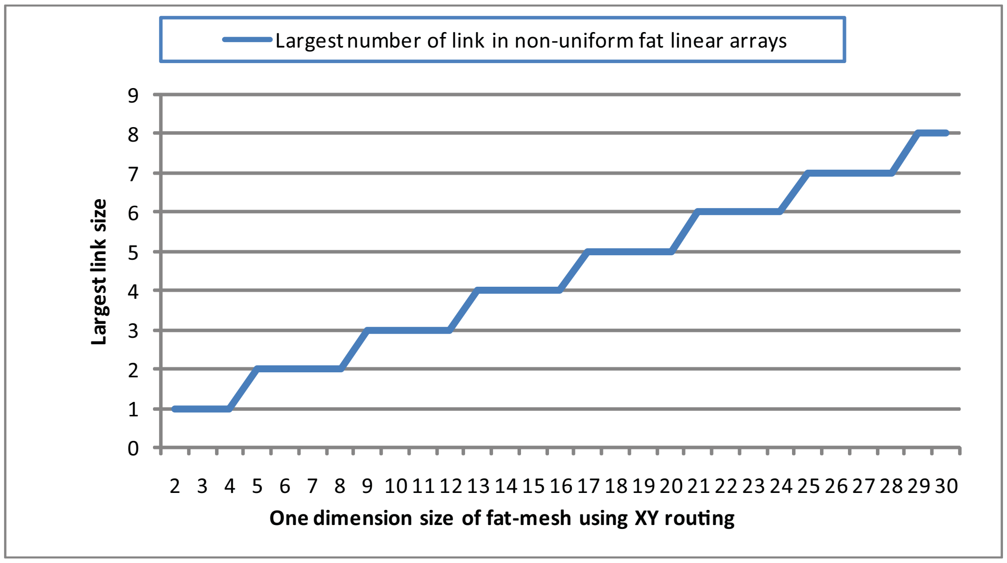


Figure 18: Number of links for the heaviest traffic link in a fat linear array.

Figure 18 shows the worst case link size in each size of non-uniform linear array. From the paper, we can see in an array of size 30, the middle link can experience 8 time more traffic than the edge link, thus is provides with 8 physical links to alleviate the traffic contention. From this observation, we can see the traffic can exhibits great variants in distribution. Using this distribution information, we can provide more bandwidth to the traffic hotspot in the hope to improve the overall performance.

4.2 FAT-MESH USING XY ROUTING

The predominant interconnection choice for emerging many-core chip multiprocessors is some variation of a mesh. For rack-style supercomputers the complexity of using a torus network instead of a mesh is not significantly different. This is due to the length of the links, particularly between racks, which is typically on the order of meters and adding wrap around links to a mesh to create a torus would not significantly change this length. However, implementing a torus in NoC instead of a mesh requires a significant increase in link distance and resources used in the device.

Similar to a linear array discussed in Section 4.1, the traffic density of the mesh structure varies tremendously based on the geographic location of the link. Figure 19 shows an example from the simulation. This is the result we get by running our stand alone network simulator configured with a 49 processor regular square mesh topology and XY routing algorithm under all-to-all personalized communication. The number on each link indicates the number of message traversing that link. We can see that the traffic density near the center is twice as many as those near the edge.

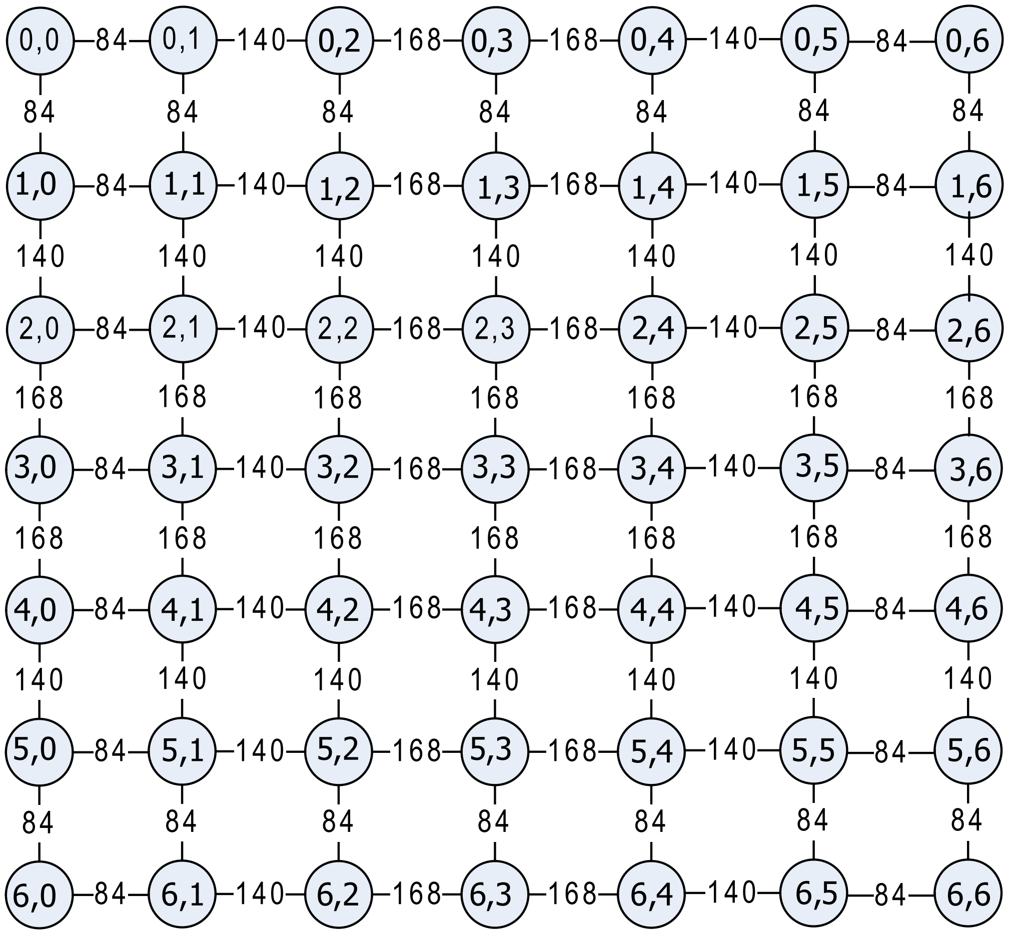


Figure 19: Messages traversing each link for an all-to-all personalized communication on a 7x7 mesh.

For all-to-all communication XY routing provides the most even distribution of messages across each link. Rerouting a message off of a heavily utilized link will only increase the use of a similarly heavily utilized link and will just create an unbalanced communication. It does not allow less heavily utilized links to be used in place of heavily utilized links.

To derive the formulas for traffic density in a $N \times N$ size mesh as shown in Figure 20,we have created a parameter i, which represents the link index similar to the linear array. However, for the mesh, this represents the link level in the horizontal or vertical direction. This is shown graphically in Figure 20. i ranges from 1 to $N-1$ regardless of which row or column of the mesh is being considered. In Figure 20 the node id is represented with an X and Y coordinate ranging from 0 to $N-1$ in both dimensions.

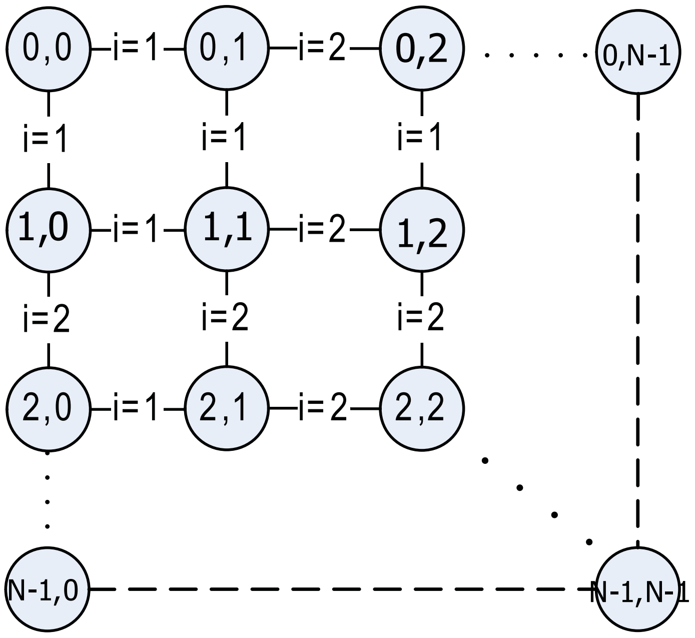


Figure 20: Generic mesh configuration with horizontal and vertical links labeled with a parameter i.

During an all-to-all communication, the total number of messages in the system is shown in Eq. (4.4). The number messages traversing each link is determined by the value of i and N in the mesh. Consider the horizontal link in Figure 21 highlighted with the heavy line. Processors involved in messages traveling from left to right over the link are shown in the shaded region. Processors involved in messages traveling from right the left are outlined. The first group of messages involve i processors each communicating with $N(N-i)$ processors resulting in $iN(N-i)$ messages. For messages traveling left to right there are $N-i$ processors communicating with iN processors also resulting in $iN(N-i)$ messages. Thus, the total number of messages for link i is shown in Eq. (4.5).

$$messages = \frac{2}{3}N^6(N^4-1) \tag{4.4}$$

$$messages_i = 2N^2i - 2Ni^2 \tag{4.5}$$

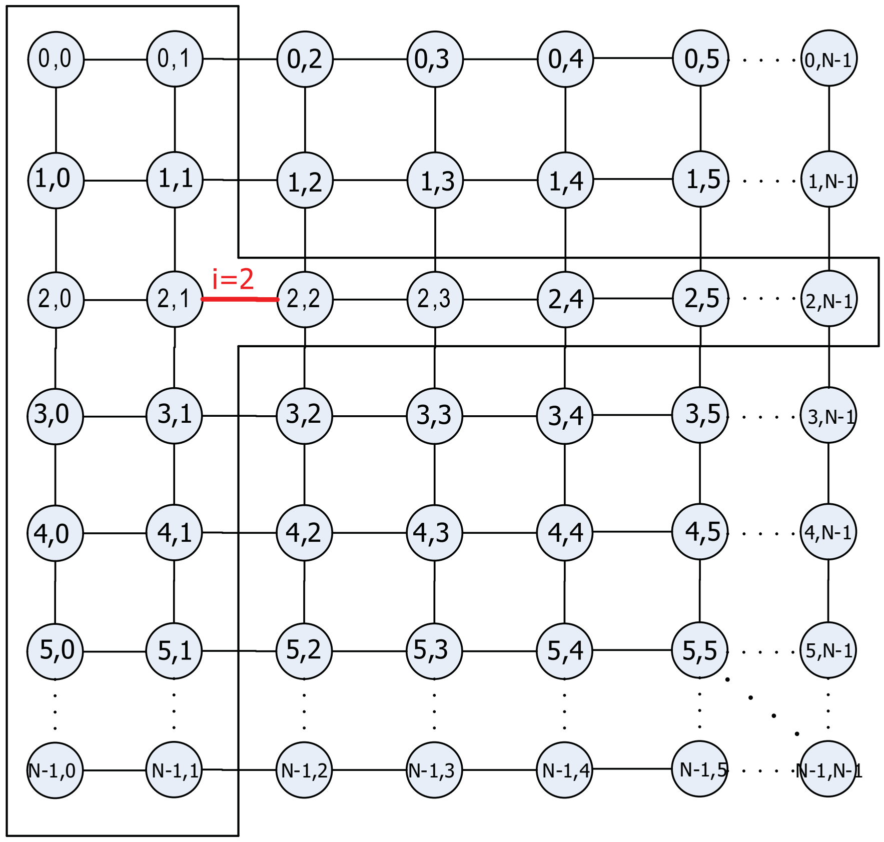


Figure 21: Messages traversing a particular link during all-to-all personalized communication on a $N \times N$ mesh with.

Similar to a fat linear array, a fat-mesh would have capacity requirements per link based on the requirements of the all-to-all personalized communication. Using a similar normalization concept as described for a linear array in Eq. (4.3) we can determine the capacity required by link i for a square mesh of $N \times N$ processors by dividing the expression from Eq. (4.5) with Eq. (4.5) where $i = 1$. The simplified normalized capacity expression for a fat-mesh is shown in Eq. (4.6).

$$capacity_i = \frac{i(N-i)}{N-1} \tag{4.6}$$

Based on Eq. (4.6), we can draw the worst case link requirement for a fat-mesh. Figure 22 shows the worst case number of links in a fat-mesh. This worst case traffic scenario happens in the center of a fat-mesh. By carefully examining Figure 18 and Figure 22, we can see the two figures are the same, and the equations to draw this worst case traffic scenario are the same. That's not out of the expectation as linear array is indeed a special 1-D case of a mesh.

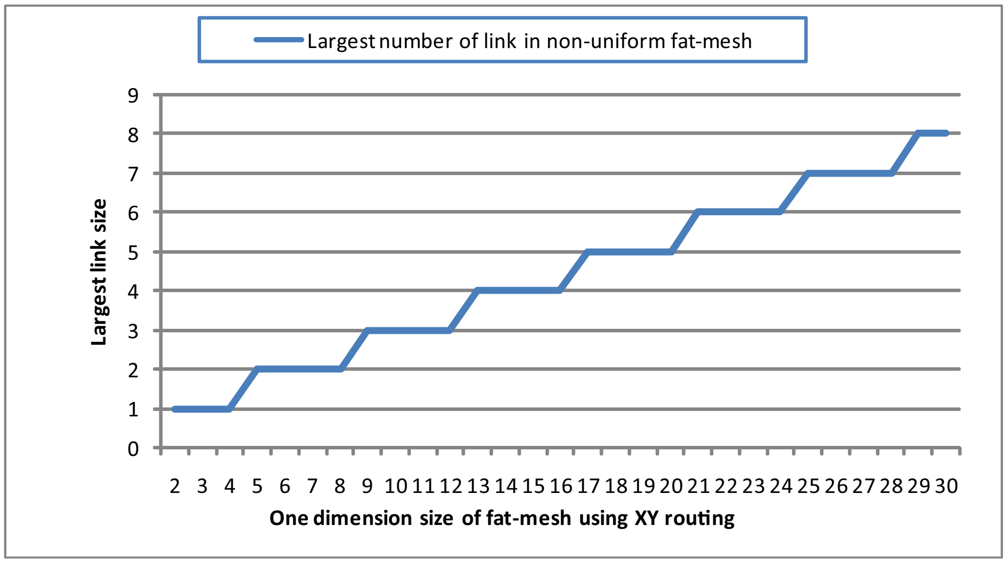


Figure 22: Number of links for the heaviest traffic link in a fat-mesh(XY).

Simulations were conducted of the traffic requirements of 500,000 random full permutations of traffic and as expected, the link utilization ratios followed the bandwidth requirements of described in Eq. (4.5). Thus, for evenly distributed traffic, the utilization ratio from Eq. (4.5) should apply. In the next section we extend this analysis for rectangular meshes.

4.2.1 Rectangular Fat-Mesh

We can extend the formulas from Eqs. (4.5) and (4.6) for a square mesh of $N \times N$ processors to work for a general mesh of $M \times N$ processors. We use a vertical index i with a horizontal index j.

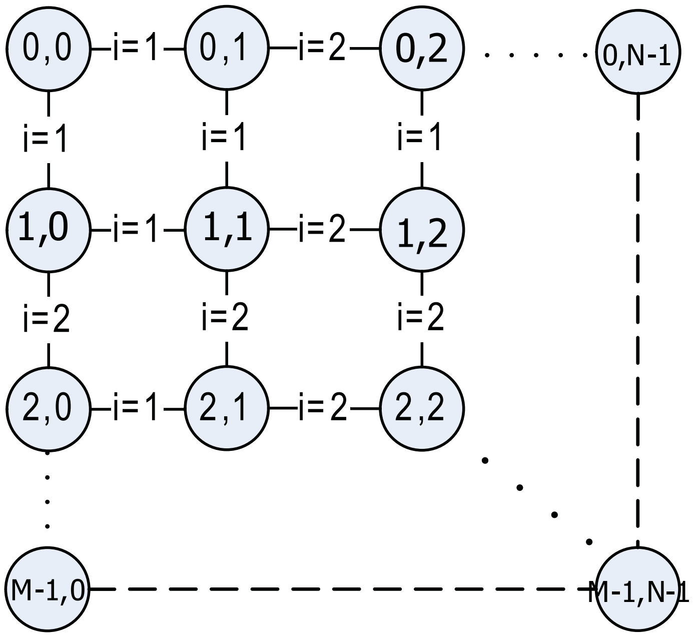


Figure 23: Generic $m \times n$ mesh with vertical links label i and horizontal links labeled j.

In the vertical direction the number of messages per link is represented by Eq. (4.7) with the capacity per link represented by Eq. (4.8). In the horizontal direction the number of messages per link is represented by Eq. (4.9) with the capacity per link represented by Eq. (4.10).

$$messages_i = 2MNi - 2Ni^2 \tag{4.7}$$

$$capacity_i = \frac{i(M-i)}{M-1} \tag{4.8}$$

$$messages_j = 2MNj - 2Mj^2 \tag{4.9}$$

$$capacity_j = \frac{j(N-j)}{N-1} \tag{4.10}$$

4.3 FAT-MESH USING XY-YX ROUTING

In order to get the intuitive central clustered fat-mesh where more traffic will be clustered in the middle of the mesh so that we can insert more links there to take advantage of traffic locality, we used another routing policy called XY-YX routing. This is a randomized non-deterministic routing algorithm. The routing between source and destination doesn't always go first straight horizontally then vertically as in XY routing. Instead, the router will choose the available paths in turn. In this way, the traffic contention of one router will be distributed more evenly among its four ports, and traffic pattern will be formed in a central clustered fashion. This routing policy still guarantees that the number of hops for each message will be minimal, that means no cycles or "go back" routes will be chosen.

Similarly, We apply all-to-all personalize traffic on a mesh using XY-YX routing. The traffic distribution is more unbalanced than the mesh using XY routing. Figure 24 shows a possible result from the simulator of the traffic density on a 7×7 mesh using XY-YX routing. As the routing decisions are made at run time, the traffic density of each link can vary in each simulation run. We can see the links in the center have significant more messages traversing on them than the peripheral nodes, which gives us potentially better opportunity to tune the performance.

We call such non-uniform fat-meshes built using XY-YX routing algorithm fat-meshes(XY-YX). In next section, we derive the analytical model of a fat-mesh(XY-YX).

4.3.1 Analytical Model of Fat-mesh(XY-YX)

The traffic density of a particular link L in a fat-mesh(XY-YX) is the summation of the probability of all messages traveling on this link. Therefore, in order to derive the formulas for traffic load of the link L from node (i, j) to node $(i + 1, j)$ (shown in red arrow in Figure 25) in a $N \times N$ fat-mesh(XY-YX), we need calculate the probability of messages from every node (a, b) in the mesh to every other node (m, n), such that $0 \leq a, b < N, 0 \leq m, n < N, and\, a \neq m, b \neq n$, travels on the link L. In order to simplify the calculation, we have grouped nodes which have similar possibility behaviors as shown in Figure 25. The nodes around the link L are partitioned into 6 regions in the mesh, namely region A_1 to region A_6.

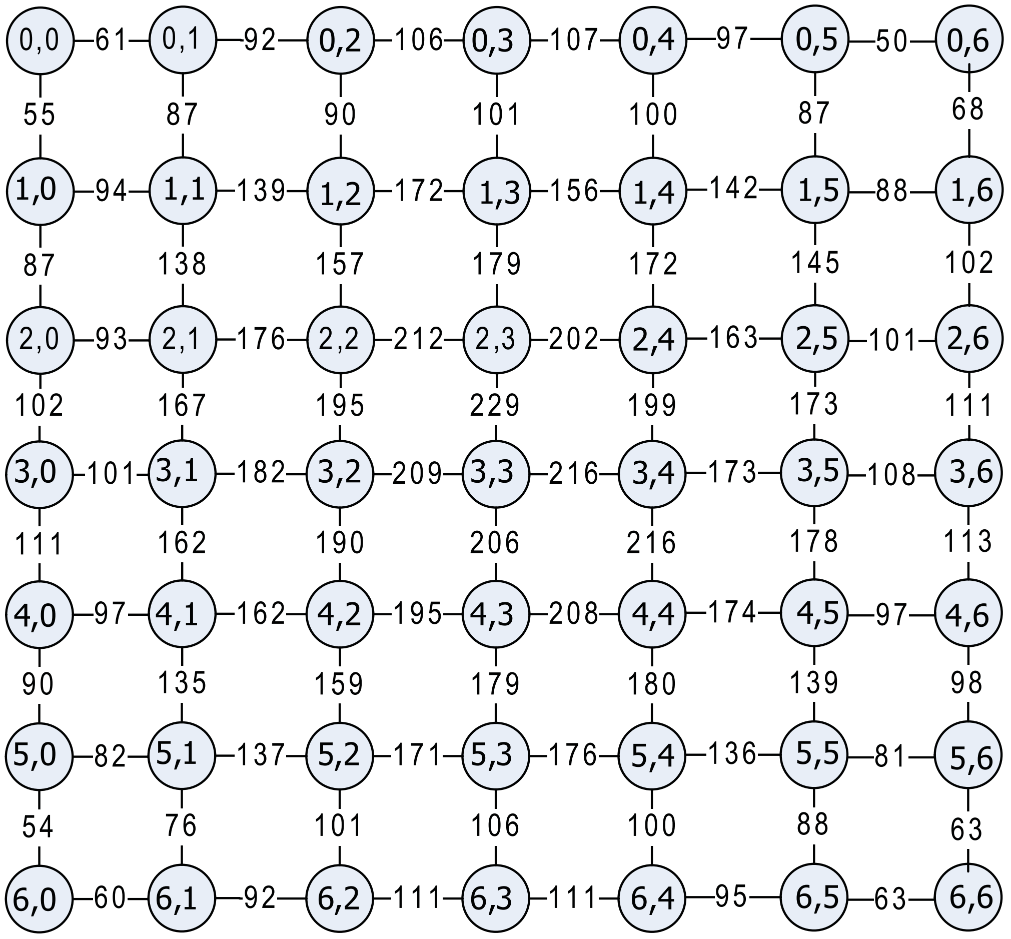


Figure 24: A possible result from the simulator of messages traversing each link during the all-to-all personalized communication on a 7x7 mesh using XY-YX routing algorithm.

Obviously, communications between region A_1 and A_4 never contribute to the link L's density, because those communications only utilize the links above link L. Similarly, region A_3 and A_6's communications don't contribute. There's another naive property of this graph, the traffic from left regions of link L to its right regions always equals the traffic from right regions to left regions as the traffic is bi-direction. Therefore, we only need calculate one way traffic on the link, for example from left to right.

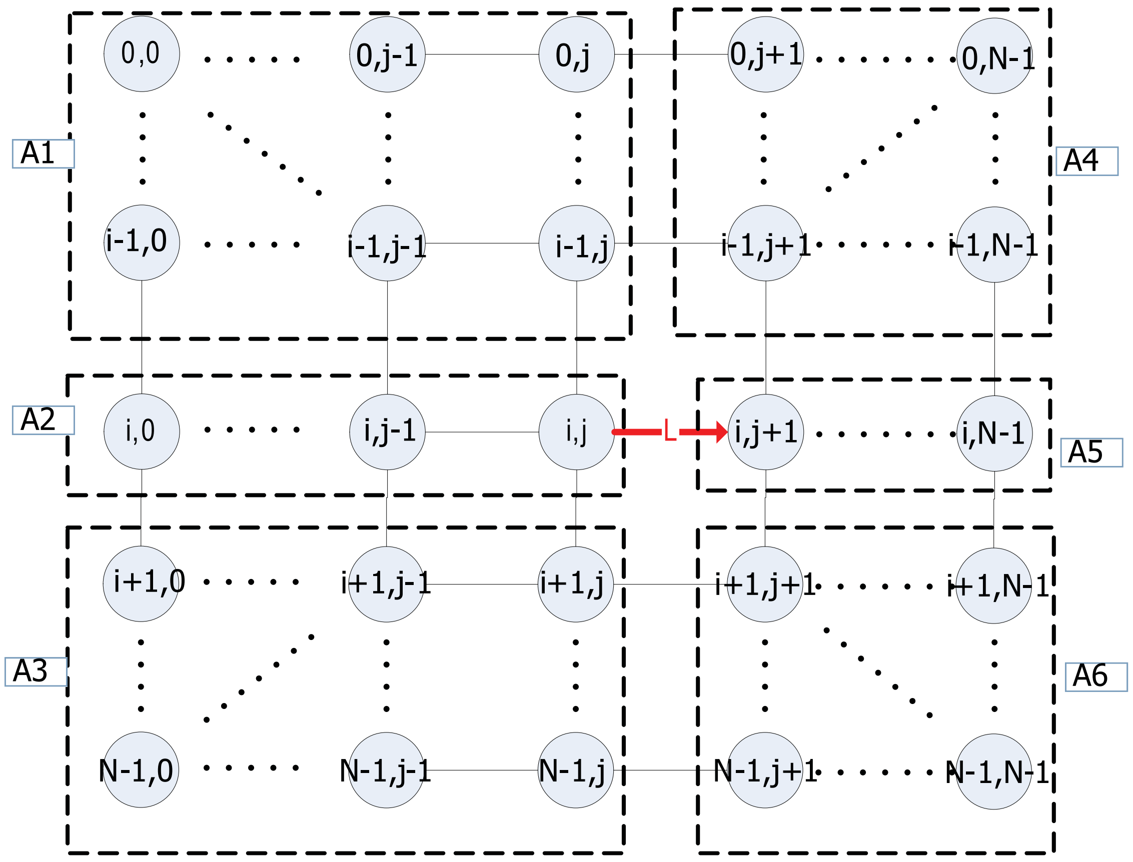


Figure 25: Area partition of a mesh topology to facilitate the fat-mesh(XY-YX) formulas derivation.

We have used $T(m,n)$ to represent the traffic contribution of region A_m to region A_n on link L, the density on link L which is from node (i,j) to node$(i,j+1)$ can be expressed as

$$Density_L = 2(\,T(1,5) + T(1,6) + T(2,5) + T(2,4) + T(2,6) + T(3,5) + T(3,6)\,) \quad (4.11)$$

Let's analyze Eq. (4.11) term by term. Take a look at some simpler terms first. Term $T(2,5)$ indicates communication from region A_2 to regionA_5 as shown in Figure 26. Each nodes in region A_2 will definitely travel on link L when sending messages to nodes in region A_5. As each node in region A_2 will send one message to each node in region A_5, the term can be calculated as

$$T(2,5) = (j+1)(N-1-j) \quad (4.12)$$

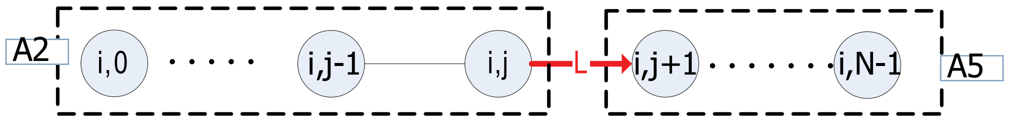


Figure 26: Communications from region A_2 to region A_4 on link L.

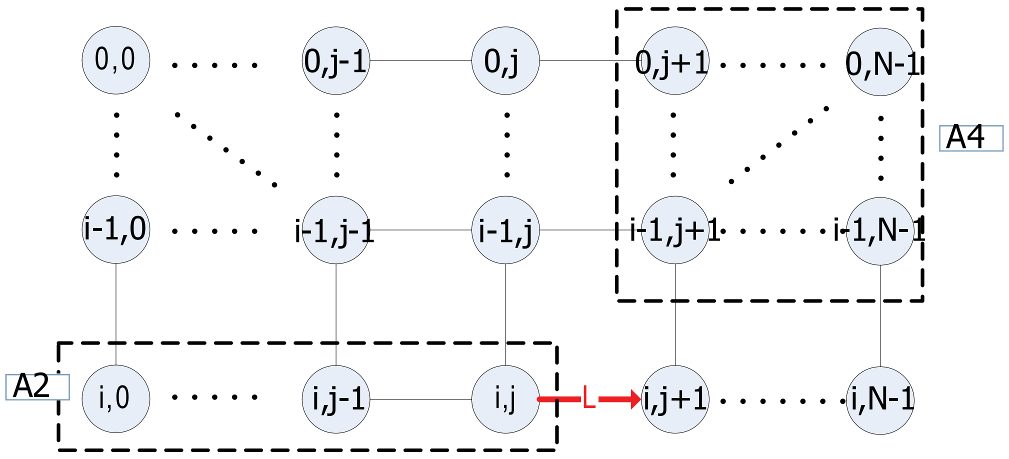


Figure 27: Communications from region A_2 to region A_4 on link L.

Term $T(2,4)$ indicates communication from region A_2 to regionA_4. As shown in Figure 27. The nodes in region A_2 may or may not use link L to reach the nodes in region A_4. For every

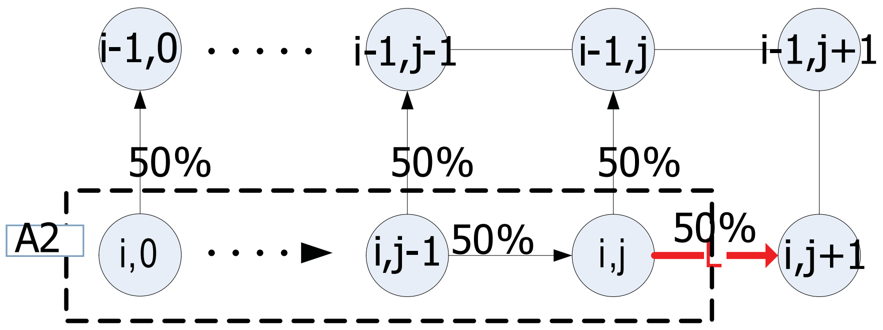


Figure 28: The probability calculation flow of communications from region A_2 to region A_4.

message from node (i,j) to region A_4, even with different destination nodes, the probability of going via Link L is always 50%. Therefore, the contribute of node(i,j) to region A_4 on Link L's load equals $\frac{1}{2}$(number of nodes in A_4), which is $\frac{1}{2}i(N-1-j)$. For node $(i,j-1)$, in order to reach the nodes in region A_4, it has 50% probability to go via node (i,j), 50% probability via node $(i-1,j-1)$. When reaching node $(i-1,j-1)$, the route will never go through Link L, when reaching node (i,j), it will have 50% probability to use Link L. So, the final probability of node $(i,j-1)$ is $(\frac{1}{2})^2 i(N-1-j)$. The above process is illustrated in the Figure 28. In general, node $(i,j-m)$ has $(\frac{1}{2})^{(m+1)} i(N-1-j)$ probability in going to region A_4, where $0 \leq m \leq i$.

It's easy to discover that region $T(2,6)$ has exactly the same calculation method as $T(2,4)$, so we can combine these two terms together, the equation for the combined term can be written as Eq. (4.13).

$$\begin{aligned} T(2,4)+T(2,6) &= T(2,(4+6)) \\ &= \frac{1}{2}(\sum_{m=0}^{j}(\frac{1}{2})^m)((N-1-j)i+(N-1-j)(N-1-i)) \\ &= \frac{1}{2}(\sum_{m=0}^{j}(\frac{1}{2})^m)(N-1-j)(N-1) \end{aligned} \tag{4.13}$$

Term $T(1,5)$ and Term $T(1,6)$ are not as obvious as above terms, and need some careful deduction. Let's first take a look at Figure 29. To get the traffic contribution from region A_1 to region A_5, A_6, the easily way may be by first obtaining overall probability of all the nodes in region A_1 getting to node (i,j), then node (i,j) will be guaranteed to use link L to get region A_5, and 50% use link L to reach region A_6.

Therefore, let's focus on up left corner of link L consists of region A_1 and region A_2. It's a rectangle area from node $(0,0)$ to node (i,j). One way to explain this problem more clearly is to change the coordinate system. Let's assume node (i,j) to be the new origin, row i to be abscissa axis, and column j to be the ordinate axis, thus node $(0,0)$ will be the new (i,j). The new coordinate system is illustrated in Figure 30. Now the goal is to calculate the probability of each node in this area going to node $(0,0)$.

Let's assume $p(m,n)$ is the probability of traveling from node(m,n) to node$(0,0)$ in a directed mesh network, where $0 \leq m \leq i, 0 \leq n \leq j$. Obviously, node $(0,0)$ always reach itself, so $p(0,0)=1$. Probability of nodes on column 0 are also easy to derive, i.e. $p(m,0)=(\frac{1}{2})^m$. Nodes

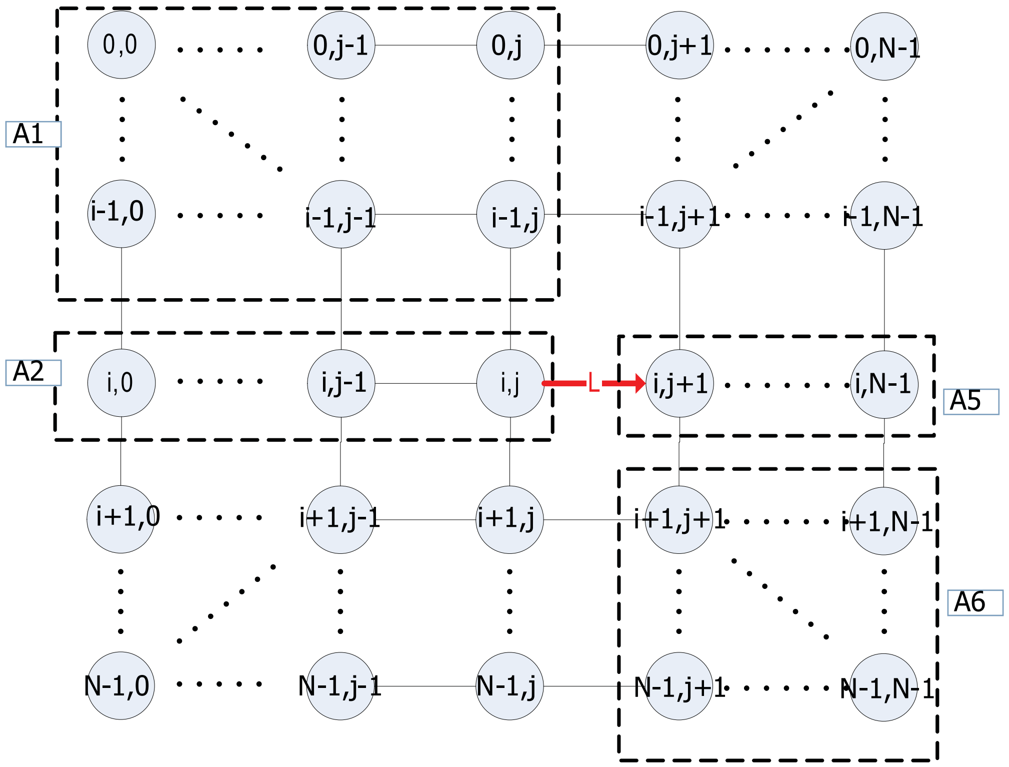


Figure 29: Communications from region A_1 to region A_5 and A_6

on the first row has two cases, when destinations nodes are in region A_6, $p(0,n) = (\frac{1}{2})^n$. On the other hand, going to region A_5 means $p(0,n) = 1$. For each other non-boundary nodes, in order to reach down right area of link L, the message will either go through neighboring down link or left link. Take the node $(1,1)$ in Figure 30 as an example, the message from it has 50% probability to reach node $(1,0)$ via the left link, and 50% possibility to reach node $(0,1)$ via the down link. Therefore, $p(m,n) = \frac{1}{2}p(m-1,n) + \frac{1}{2}p(m,n-1)$. In order to get the final value of $p(m,n)$, we need to recursively expand to the neighboring nodes until we reach the boundary nodes whose values are known as base cases. However, recursion is not good in formulas derivation. In order to transform the problem into a non-recursive one, let's take above two cases separately.

For the first case, $T(1,6)$, which is going from region A_1 to region A_6, the probability of node (i,j) to node $(0,0)$ equals unique routes between node (m,n) and node $(0,0)$ times the probability of taking each route. We define two routes between two nodes are unique if at least

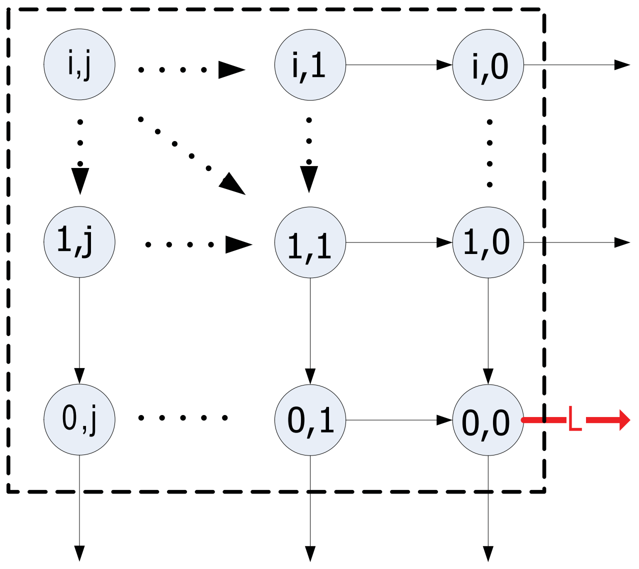


Figure 30: Transposed up left corner of link L of the fat-mesh(XY-YX).

one link between two routes is different. The probability of taking one particular route is always $(\frac{1}{2})^{m+n}$ in this case, where $m+n$ is the number of hops between two nodes. As each hop in a route will have 50% chance to get the next node in the same route, and we will always take $(m+n)$ hops from source to destination.

On the other hand, the unique number of routes between two nodes in a directed mesh network can be calculated in combination. Still taking Figure 30 as an example, the node(i,j) is $i+j$ hops away from node$(0,0)$. In each hop, it can either jump left or jump downwards, we have exactly j left hops and 1 down hops to get from node (i,j) to node $(0,0)$. Thus, the number of paths from node(m,n) to node$(0,0)$ equals picking unordered m down or n left hops from all $m+n$ hops. The combination can be computed in ${}_{m+n}C_m = \binom{m+n}{m} = \frac{(m+n)!}{m!n!}$.

Below are the non-recursive equations for nodes in region A_1 going to region A_6. we can see that the boundary values are just the special case of general cases.

$$\begin{aligned} p(m,0) &= (\frac{1}{2})^m, \;\; 0 \le m \le i \\ p(0,n) &= (\frac{1}{2})^n, \;\; 0 \le m \le i \\ p(m,n) &= (\frac{1}{2})^{(m+n)} \binom{m+n}{m} = (\frac{1}{2})^{(m+n)} \frac{(m+n)!}{m!n!}, \;\; 0 \le m \le i, 0 \le n \le j \end{aligned}$$

For the second case,going from region A_1 to region A_5, there is some difference than the first case as the first row $p(0, m) = 1$. This can be solved by expanding node (m, n) to each of the *second row* nodes it can reach, adding the possibilities of reaching those second row nodes together. Thus the probability of node (m, n) to node $(1, k)$, where $0 \leq k \leq n$, equals unique routes from node (m, n) to node $(1, k)$ times the probability of taking each route. The same, probability of taking each routes is $(\frac{1}{2})^{m+n-k-1}$, as we take $(m + n - k - 1)$ hops to reach node $(1, k)$. Then node $(1, k)$ has 50% to reach nodes $(0, k)$. Thus the total probability is $(\frac{1}{2})^{m+n-k}$. As shown above, the number of unique routes from node (m, n) to node $(1, k)$ is ${}_{m-1}C_{m+n-1-k} = \binom{m+n-1-k}{m-1} = \frac{(m+n-1-k)!}{(m-1)!(n-k)!}$.

Now we have the non-recursive equations for nodes in region A_1 to region A_6.

$$\begin{aligned} p(m,0) &= (\frac{1}{2})^m, \quad 0 \leq m \leq i \\ p(0,n) &= 1 \\ p(m,n) &= \sum_{k=0}^{n}((\frac{1}{2})^{(m+n-k)}\binom{m+n-1-k}{m-1}) \\ &= \sum_{k=0}^{n}((\frac{1}{2})^{(m+n-k)}\frac{(m+n-1-k)!}{(m-1)!(n-k)!}), \quad 0 \leq m \leq i, 0 \leq n \leq j \end{aligned}$$

Given the each node's probability to node $(0, 0)$, it's easy to get $T(1, 5)$ and $T(1, 6)$ now, which are summation of nodes' possibilities in region A_1. Formulas are shown in Eq. (4.15).

$$\begin{aligned} T(1,5) &= (\sum_{m=1}^{i}\sum_{n=0}^{j}p(m,n))(N-1-j) \\ &= (\sum_{m=1}^{i}\sum_{n=0}^{j}(\sum_{k=0}^{n}((\frac{1}{2})^{(m+n-k)}\frac{(m+n-1-k)!}{(m-1)!(n-k)!})))(N-1-j) \qquad (4.14) \\ T(1,6) &= \frac{1}{2}(\sum_{m=1}^{i}\sum_{n=0}^{j}p(m,n)) \cdot (N-1-j) \cdot (N-1-i) \\ &= \frac{1}{2}(\sum_{m=1}^{i}\sum_{n=0}^{j}((\frac{1}{2})^{(m+n)}\frac{(m+n)!}{m!n!}))(N-1-j)(N-1-i) \qquad (4.15) \end{aligned}$$

Similarly, T(3,5) and T(3,6) are shown in Eq. (4.16) using the same derivation method.

$$\begin{aligned} T(3,5) &= (\sum_{m=1}^{N-1-i}\sum_{n=0}^{j} p(m,n))(N-1-j) \\ &= (\sum_{m=1}^{N-1-i}\sum_{n=0}^{j}(\sum_{k=0}^{n}((\frac{1}{2})^{(m+n-k)}\frac{(m+n-1-k)!}{(m-1)!(n-k)!})))(N-1-j) \end{aligned} \tag{4.16}$$

$$\begin{aligned} T(3,4) &= \frac{1}{2}(\sum_{m=1}^{N-1-i}\sum_{n=0}^{j} p(m,n))(N-1-j)i \\ &= \frac{1}{2}(\sum_{m=1}^{N-1-i}\sum_{n=0}^{j}((\frac{1}{2})^{(m+n)}\frac{(m+n)!}{m!n!}))(N-1-j)i \end{aligned} \tag{4.17}$$

Finally, we can replace Eq. (4.12) to Eq. (4.16) into Eq. (4.11), and get

$$\begin{aligned} Density_L &= 2(T(1,5)+T(1,6)+T(2,5)+T(2,4)+T(2,6)+T(3,5)+T(3,6)) \\ &= 2(j+1)(N-1-j)+(\sum_{m=0}^{j}(\frac{1}{2})^{m})(N-1-j)(N-1)+ \\ &\quad 2(\sum_{m=1}^{i}\sum_{n=0}^{j}(\sum_{k=0}^{n}((\frac{1}{2})^{(m+n-k)}\frac{(m+n-1-k)!}{(m-1)!(n-k)!})))(N-1-j)+ \\ &\quad (\sum_{m=1}^{i}\sum_{n=0}^{j}((\frac{1}{2})^{(m+n)}\frac{(m+n)!}{m!n!}))(N-1-j)(N-1-i)+ \\ &\quad 2(\sum_{m=1}^{N-1-i}\sum_{n=0}^{j}(\sum_{k=0}^{n}((\frac{1}{2})^{(m+n-k)}\frac{(m+n-1-k)!}{(m-1)!(n-k)!})))(N-1-j)+ \\ &\quad (\sum_{m=1}^{N-1-i}\sum_{n=0}^{j}((\frac{1}{2})^{(m+n)}\frac{(m+n)!}{m!n!}))(N-1-j)i \end{aligned} \tag{4.18}$$

Therefore, as we can see in the final formula, there are only three input variables, N, i, j, and the equation gives the final link density from node (i,j) to node $(i,j+1)$ in a $N \times N$ mesh network. That means the size of mesh and geographic location of the nodes can fully determine the traffic density on every link of the fat-mesh(XY-YX) for the all-to-all personalized communication. Figure 22 is a comparison between the worst case number of links in a fat-mesh using XY routing and XY-YX routing respectively. This worst case traffic scenarios always happen in the center of fat-meshes. From this figure, we can see the traffic is more unevenly distributed using XY-YX routing than that using XY routing, causing the middle links require much more bandwidth than that of XY routing. In a 30×30 size mesh, the center links using XY-YX routing can have 22 times more traffic pressure than the edges links. Such unbalanced traffic pressure can will create great traffic contention in those center regions and slow down the whole network. It implicates that fat-mesh(XY-YX) can potentially achieve better performance by leveraging the more unbalance of the traffic.

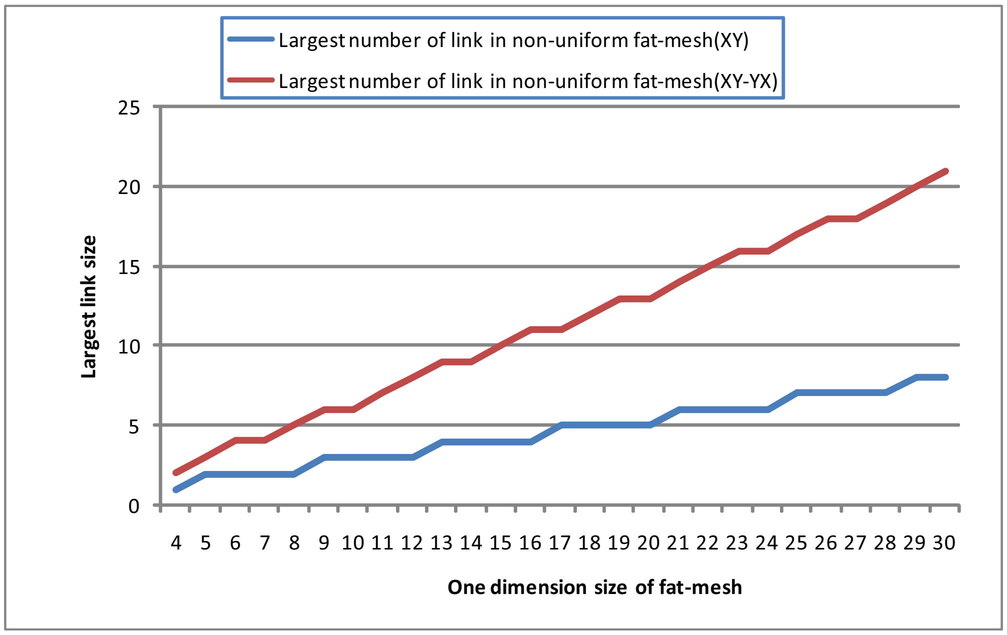


Figure 31: Number of links for the heaviest traffic link in a fat-mesh(XY-YX).

The formula is verified by Mathematica, a symbolic mathematical tool, the Mathematica outcomes accord with the simulation results. Figure 32 shows the result from the calculation of the analytical model. The link densities of the 7×7 mesh are close to the density results from the simulation shown in Figure 24. The deviations are because of the randomness in the run time. The simulation results can approach the analytical result by averaging multiple simulation runs.

In order to build the fat-mesh(XY-YX) network, the capacity requirements per link of the mesh can be determined using a similar normalization concept as described for a linear array in Eq. (4.3), we can divide the density per link expression of Eq. (4.18) by the outmost edge link's traffic density, which can be calculated with Eq. (4.18) where $i = 0, j = 0$. Figure 33 shows the normalized link size of a 7×7 mesh using XY-YX routing. From this figure, we can clearly see that there are more physical links in each connection in the middle of the mesh and less links per

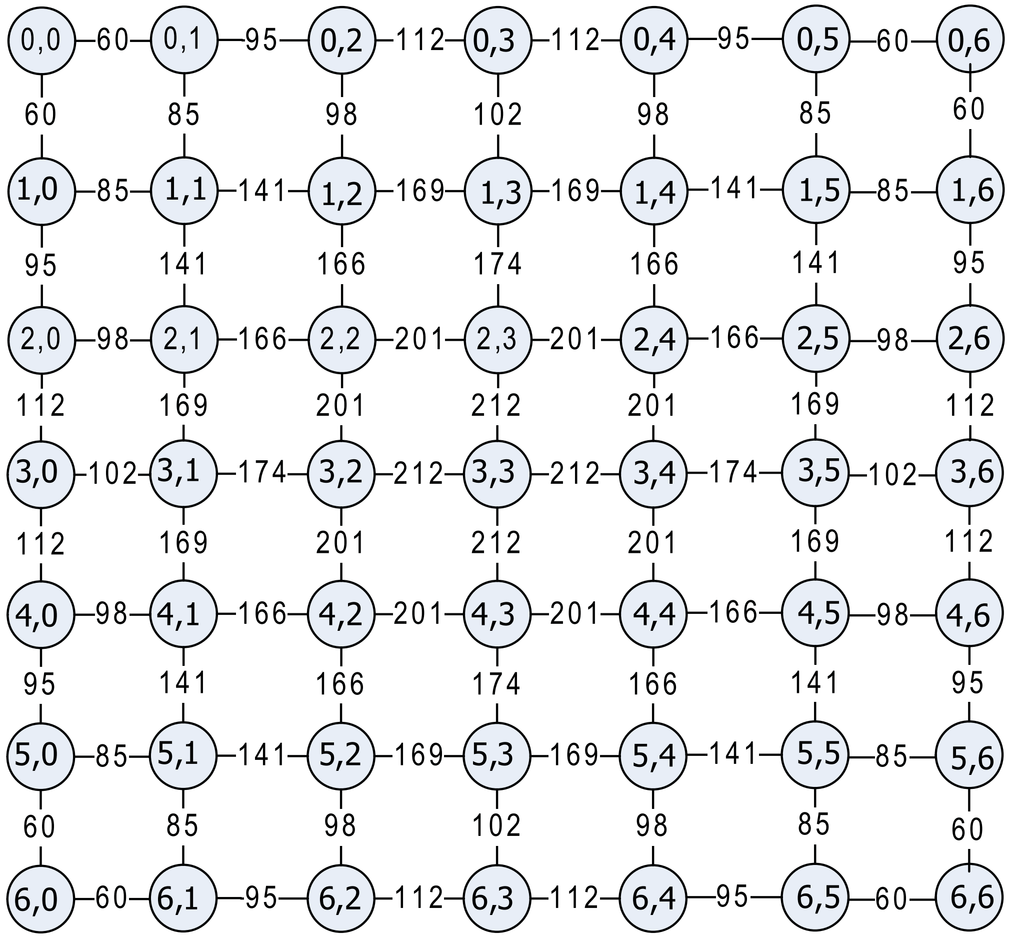


Figure 32: Analytical traffic model of a 7×7 mesh using XY-YX routing under all-to-all personalized communication. Results are rounded to closest integers.

connection on the edges. Fat-mesh(XY-YX) network is configured using these normalized number of links per connection information in the simulator. In next section, we introduce our build the fat-mesh structure in the simulator and present fat-mesh result.

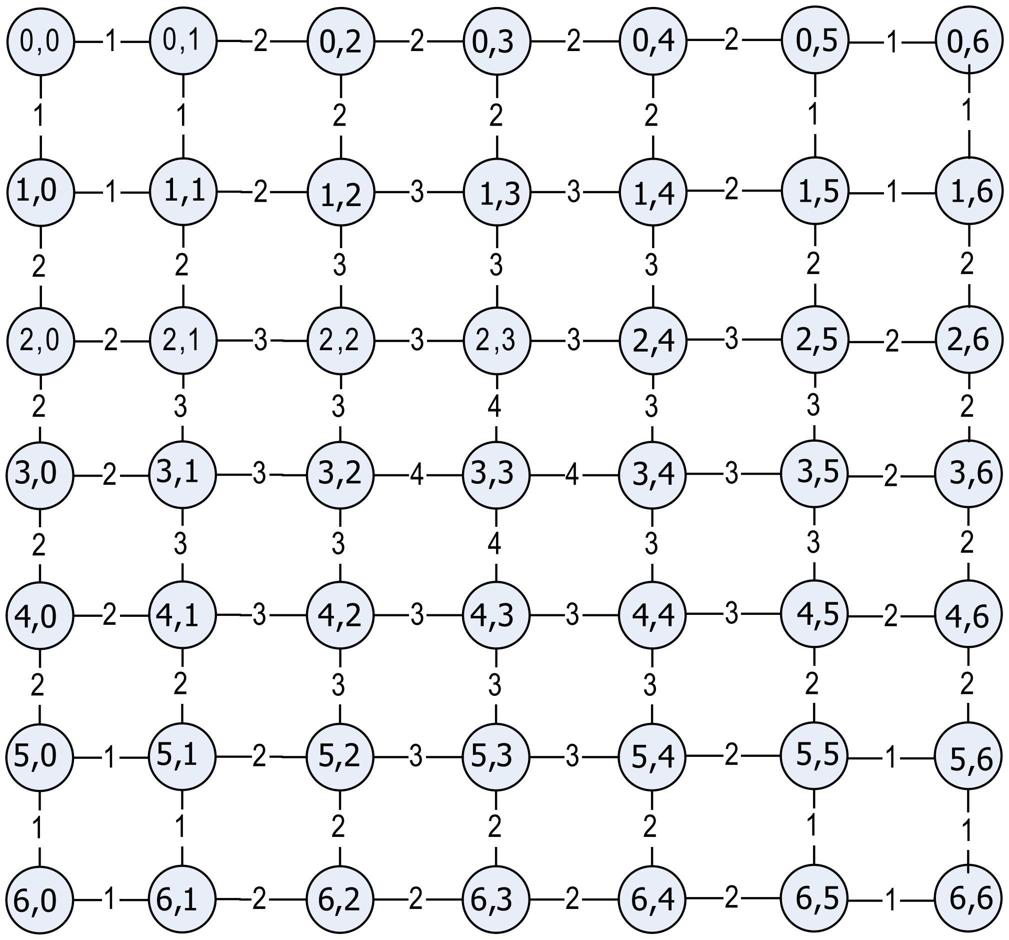


Figure 33: Analytical model of normalized link size of a 7 $\times$ 7 mesh using XY-YX routing. Fat-mesh(XY-YX) can be built based on this link size information.

4.4 FAT-MESH RESULTS

4.4.1 Simulation Environment

For our evaluation purposes we have developed a event-driven network simulator based on the network module provided by SESC. As stated in Section 3.2.3, We replaced SESC's default bus interconnect with this modified version mesh network. The mesh network module is designed to either run as a stand alone program with synthetic traffic patterns or to cooperate with other parts of simulator with compiled benchmarks as input. The modified mesh is written in a fully configurable manner so that it can flexibly model different network topologies (e.g. linear array, mesh, fat-mesh) and routing algorithms (e.g. XY and XY-YX routing algorithms).

We also implemented two normalized fat-mesh schemes based on XY and XY-YX routing algorithms in the network simulator. The normalization concept(e.g. shown for Eq. (4.6)) calculates the number of links for a particular link of the fat-mesh by dividing the number of messages traversing this link by the the number of message traversing the outmost edge link, which has the smallest number of messages traveling on it.

Table 2 shows the simulator system configuration we used to complete full system simulation of various benchmarks. Table 3 describes the network configuration we used in the experiments.

Table 2: Core System Simulator Configuration

Characteristic	**Description**
Core	Out-of-order, core number equals the mesh size one router per core
L1 I/D Caches	private, 16K, 4 way set assoc, 1 cycle lat
L2 Cache	Distributed shared, 2M, 8 way set assoc, 5 cycles lat, 64 Byte cache line
Cache coherence	Directory based, MSI cache coherence
Memory	Unlimited size, 500 cycles lat

For the workload, we examined six Splash-2 [58] benchmarks to represent real applications' performance. Splash-2 benchmark suite is a popular choice to represent typical parallel applications and traffic patterns common to CMP-like shared memory systems such as all-to-all and nearest neighbor patterns. Our simulation conducted by generating a trace from each Splash-2 application and executing the trace on the network module. For fat-mesh part, we first tested personalized all-to-all synthetic traffic on the fat-mesh, and then use splash-2 benchmarks to show the real-life application performance. Following is a brief description of the benchmarks we use and the communication patterns they represent.

- FFT - The FFT kernel is a complex 1-D version of the radix- sixstep FFT algorithm, which is optimized to minimize inter-processor communication. Data sets are organized $\sqrt{N} \times \sqrt{N}$as

Table 3: Network Configuration

Characteristic	Description
Mesh size	From 4×4 to 10×10
Switching techniques	Packet switching
Routing algorithm	XY, XY-YX
Network topology	Uniform mesh, fat-mesh(XY), fat-mesh(XY-YX)
Port width	96 bits
Wire latency	1 cycle per hop
Switch transversal latency	1 cycle

matrices partitioned so that every processor is assigned a contiguous set of rows which are allocated in its local memory. Every processor transposes a contiguous submatrix of $\sqrt{N}/p \times \sqrt{N}/p$ from every other processor, and transposes one submatrix locally. Communication require all-to-all inter-processor communication.

- LU - The LU kernel factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The dense matrix A is divided into an $N \times N$ array of $B \times B$ blocks ($n = NB$) to exploit temporal locality on submatrix elements. To reduce communication, block ownership is assigned using a 2-D scatter decomposition, with blocks being updated by the processors that own them. Communication patterns include One-to-many, many-to-one traffic.
- Radix - The integer radix sort kernel, The algorithm is iterative, performing one iteration for each radix r digit of the keys. In each iteration, a processor passes over its assigned keys and generates a local histogram. The local histograms are then accumulated into a global histogram. Finally, each processor uses the global histogram to permute its keys into a new array for the next iteration. This permutation step requires all-to-all communication.
- Ocean - The Ocean application studies large-scale ocean movements based on eddy and boundary currents. It partitions the grids into square-like subgrids to improve the communication to

computation ratio. The grids are conceptually represented as 4-D arrays, with all subgrids allocated contiguously and locally in the nodes that own them. It needs nearest neighbor interactive communication.

- Raytrace - Raytrace application renders a three-dimensional scene using ray tracing. A hierarchical uniform grid (similar to an octree) is used to represent the scene. The image plane is partitioned among processors in contiguous blocks of pixel groups, and distributed task queues are used with task stealing. The data access patterns are highly unpredictable in this application.
- Water - Water evaluates forces and potentials that occur over time in a system of water molecules. The forces and potentials are computed using an $O(n)$ algorithm. It imposes a uniform 3-D grid of cells on the problem domain, and requires communication of both nearest neighbor and irregular.

Table 4 lists the benchmarks' input dataset used in the simulation.

Table 4: Input Dataset of Six Benchmarks from Splash-2 Suite.

benchmark	**Input dataset**
FFT	4K points, 32Byte cache line
LU	256x256 matrix, 16k blocks
Radix	256k integers, radix = 1024
Ocean	258x258 grids
Raytrace	teapot.env
Water	512 molecules, 3 time steps

4.4.2 Fat-Mesh Simulation Results

We implemented two normalized fat-mesh schemes based on XY and XY-YX routing algorithms in the network simulator. This section shows the results for fat-mesh. First of all, Figure 34 shows the total number of links used by each type of mesh topology. We compare on three type of meshes and torus. "Mesh" and "2 link uniform mesh" are both uniform mesh each with one or two links connecting two neighboring nodes respectively, "fat-mesh(XY)" is the non-uniform mesh we proposed using XY routing. As we use more links where there are more traffic during all-to-all communication, the links of fat-mesh will be inevitably increasing. When the mesh size increases, the links connecting by the routers will be even more unbalanced depending on their geographical location. The simulation showed that in 10×10 fat-mesh, 15% of the links are 3 times heavier than the base case links. Therefore in this case, the total number of links in fat-mesh(XY) exceeds that of two link uniform mesh. But at most time, the fat-meshes(XY)'s sums of links are reasonably following those of two link uniform meshes.

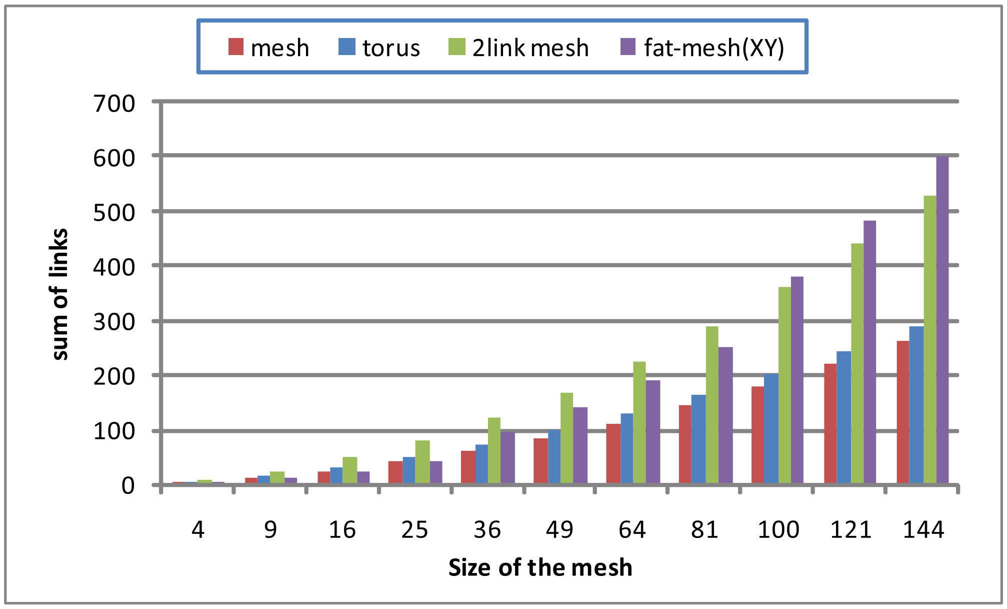


Figure 34: Comparison of sum of links with mesh size from 2×2 to 12×12 among different mesh and torus types.

Figure 35 illustrates the average message delay during all-to-all communication in each type of mesh and torus. Here, XY routing is applied in all three network structures . The green bars shows the average message delay for fat-meshes(XY). The reduction in message delay of fat-mesh(XY) over traditional meshes and toruses is considerable, from 28.4% lower in 36 nodes mesh to 46.6% lower in 100 nodes mesh. Additionally, as shown in Figure 35, two link uniform fat-meshes have more links than the fat-mesh(XY) configuration for until $n = 100$ while the fat-mesh(XY)has very competitive performance (only 6.2% worse for $n = 36$ and 1.8% worse for $n = 81$ nodes mesh.

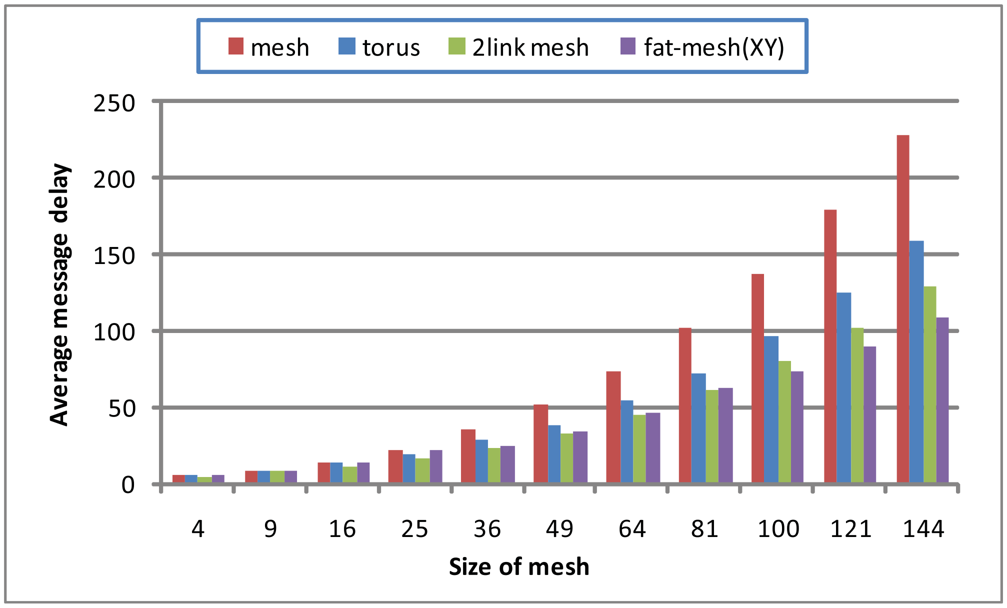


Figure 35: Average message latency during all-to-all communication with mesh size from 2×2 to 12×12 with different mesh and torus types.

In the $n = 100$ case, the number of links in fat-mesh(XY)increase by approximately 5.6% over the number of links in a uniform 2-link fat-mesh, however, there is nearly a 10% reduction in the message latency. The results show the possible performance gain the fat-mesh can get by better arranging the bandwidth of the mesh according the traffic requirement.

Figure 36 compares the link sums of various fat-mesh(XY-YX) configurations against uniform meshes and toruses. Fat-mesh(XY-YX) configurations quickly exceed two link unform meshes and even exceed three link uniform meshes after 10×10 mesh size. This is a result of the middle routers in fat-mesh(XY-YX) experience much heavier traffic load than the edge routers, which we believe is typical of many non-deterministic routing strategies. The mathematical model shows for $n = 144$ the middle routes in fat-mesh(XY-YX) have 8 times more traffic than the routers on the periphery.

Figure 37 shows the improved average message latency exhibited by the fat-meshes(XY-YX) against the uniform meshes and toruses. For fairness, we use XY-YX routing algorithm in all four cases. As expected, though we have more links in fat-mesh(XY-YX), we gain dramatically

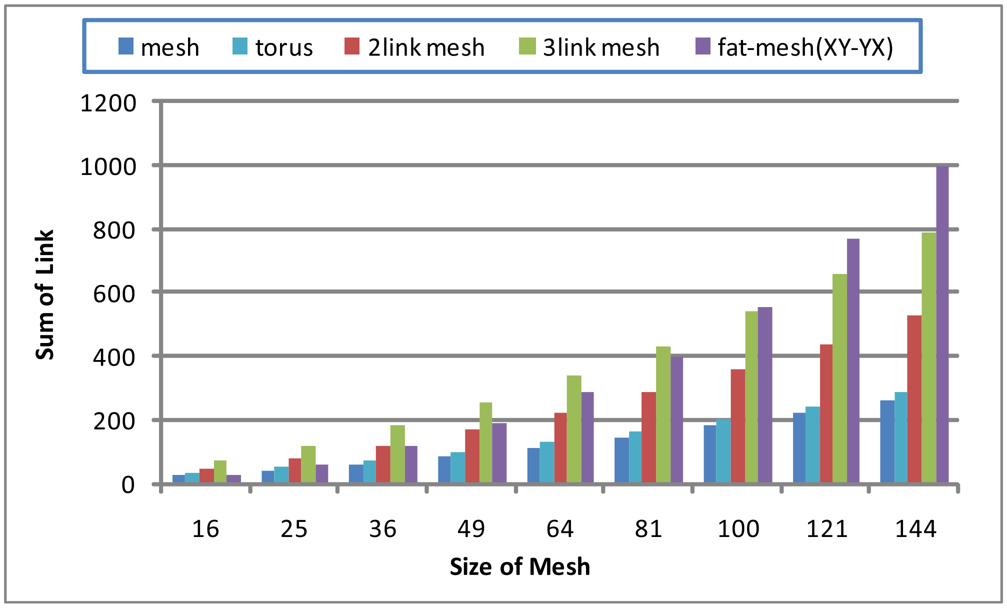


Figure 36: Comparison of sum of links with mesh size from 2×2 to 12×12 among different mesh and torus types including fat-mesh(XY-YX).

performance improvement from it. It is worth to specially mention here, in 6×6 mesh size, the fat-mesh(XY-YX) has the same number of links as two link uniform mesh, however fat-mesh(XY-YX) has lower latency than the uniform mesh; in 8×8 and 9×9 mesh size, the fat-mesh(XY-YX) has fewer links than three link uniform mesh, but still fat-mesh(XY-YX) outperforms the uniform mesh. From these results, we can see even with less resources (the physical links and ports associated with them), the fat-mesh(XY-YX) can achieve better performance (lower average message delay) because it takes the advantage of the traffic pattern and organize the network in a more efficient way for the traffic flow.

To test our findings on real applications, we considered various mesh configurations and the impact on execution of the benchmarks from Table 4 as shown in Figure 38 for an 8×8 system. We picked six representing programs from the suite, and got their communication traces by running them in the SESC simulator. Afterwards, we fed the trace to the network module with different mesh types. The results shows that in general, fat-mesh(XY-YX) has the best performance among the three mesh types. It best performs in five of six benchmarks. The only exception is LU, as

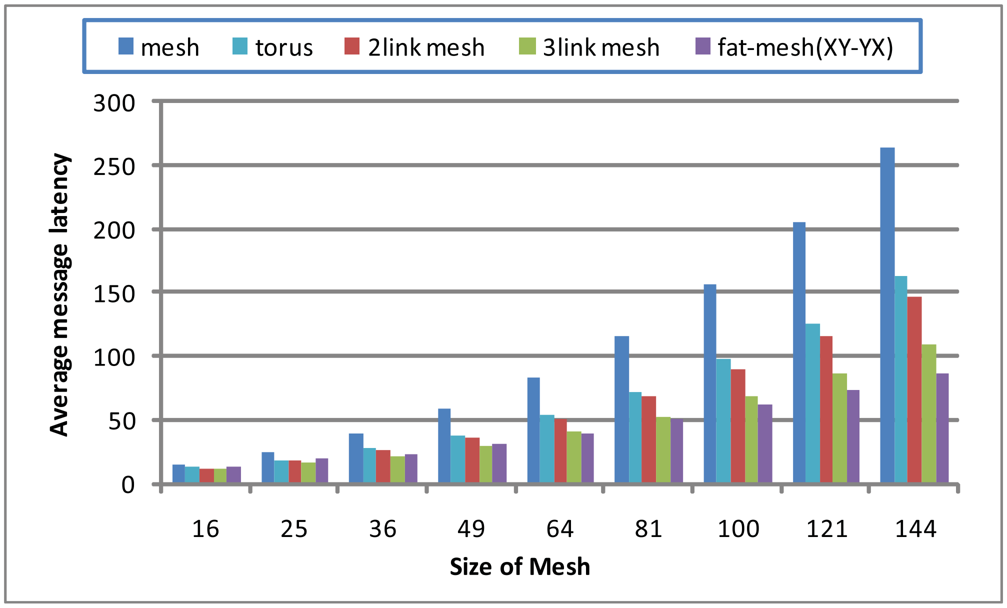


Figure 37: Average message latency during all-to-all communication with mesh size from 2×2 to 12×12 with different mesh and torus types including fat-mesh(XY-YX).

there are a lot of random, local communication inside it, thus the uniform two link mesh has the best performance. As these splash-2 benchmarks represents the typical communication patterns in parallel applications as shown in Table 4, we can conclude that the fat-mesh(XY-YX) does provide a performance advantage over a two-link uniform mesh and a considerable advantage of a single link mesh.

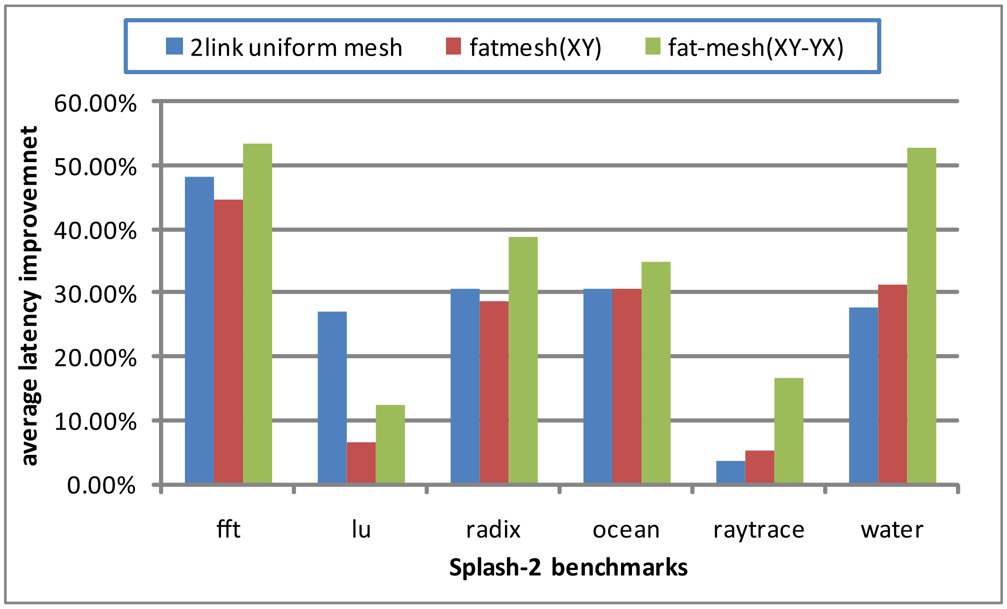


Figure 38: Average message latency improvement during all-to-all communication with six splash-2 benchmarks under different mesh types all using 8×8 size. All results are normalized by the normal one link uniform mesh's average message latency

5.0 HYBRID NETWORK WITH CIRCUIT SWITCHING

In this chapter, I present a hybrid network implemented in the simulator along with a compiler technique to facilitate the establishment of the hybrid network. As stated in Section 2.3, many high performance computing systems use packet-switched networks to interconnect system processors because they are simple to realize, easy to scale and fairly low-latency and high-bandwidth. As systems get larger, a scalable interconnect can assume a disproportionately large portion of the system cost when striving to meet the demands of low-latency and high-bandwidth. Although the packet-switched interconnections for large scale systems are worthwhile, circuit switching networks may have the potential of achieving higher efficiency than packet and wormhole networks at a relatively lower cost. However, the drawbacks of circuit switching cannot be ignored, such as the overhead required for the circuit establishment, reconfiguration and removal. The benefits of circuit switching can only outweigh its drawbacks when communication exhibits locality and when this locality is appropriately explored. As a result, a hybrid interconnection which consists of both packet switching and circuit switching is attracting more attention in the high performance computing area. The idea is for regular, predictable communications to use the circuit switching while some random, irregular or conflict communications still utilize the traditional packet switching network to avoid deadlocks or other unexpected situations.

5.1 HYBRID NETWORK IMPLEMENTATION

This thesis realized a multi-layer network system which comprises one packet switching plane and multiple circuit switching planes for the CMP system in the simulator. Based on the packet switching network module integrated into the SESC (described in Section 3.2.3), we implement

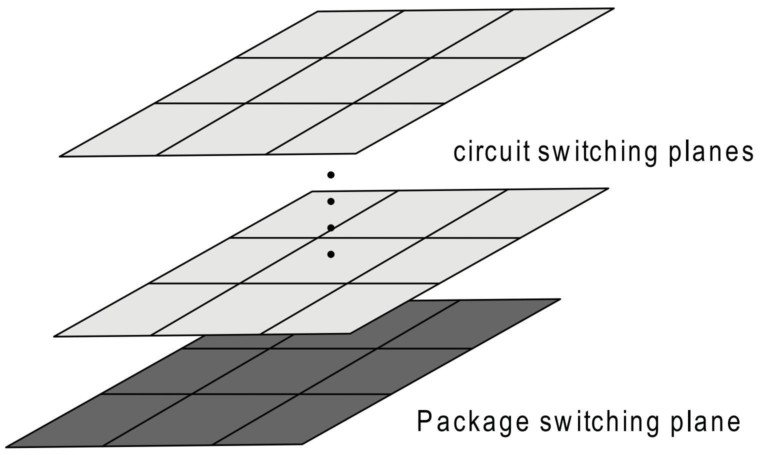


Figure 39: Hybrid model of independent circuit switching and packet switching plains.

additional circuit switching networks in the simulator. The differences are, routing table is always fixed in circuit switching, the circuit is either established or not between source-destination pair; there are no router setup time once the circuit is established; the inter-router transmission latency is very low due to the large bandwidth of the direct pipe provides.

The implemented circuit switching planes are independent from each other and also from the packet switching plane shown in Figure 39. Each circuit switching plane has its own router set and routing table. An arbitrator is modeled in the network adaptor in a way that for each message attempting to access the network, the adaptor will first check if there is any circuit switching plane which contains the direct source-destination circuit for the message. If there is one available, the message will be injected into this circuit plane. Only if there is no available circuit plane will the messages use the traditional packet switching planes. By putting the message on dedicated circuit links, we maximize the usage of the fast circuit switching, though the choice among all available circuit switching planes is totally random.

Theoretically, inside each plane, one router can have at most $N-1$ outgoing circuits connecting to other nodes, assuming that the total number of nodes in the network is N. In practice, this is not implementable because of the limited number of ports one router can have. Typically, one router can establish one single link at one time on a circuit switching plane. To achieve most efficient use of one plane, a permutation layer utilizes most of the nodes on a particular layer. A network consisting of N nodes with $N-1$ permutation layers acts like a fully-connected network. Figure 40 shows an example link configuration of 3 fast circuit planes in a 4 node network to form an all-to-all network.

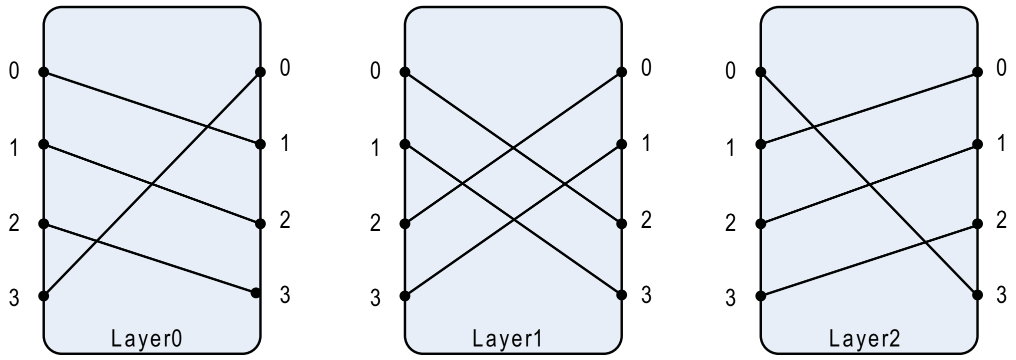


Figure 40: A 4 node network with 3 permutation planes form an fully-connected circuit network.

5.2 HYBRID NETWORK RESULTS

This section presents the performance results of applying multiple circuit switching planes. They are a rough approximation of how circuit switching can help improve overall benchmark performance in an ideal situation. We implemented an hybrid network extension consisting of one packet switching plane and multiple circuit switching planes described in Section 5. An arbitrator adaptor is modeled in each node that checks the availability of the network planes and assigns the message on first available plane.

Figure 41 shows the execution cycles with six Splash-2 benchmarks with normal meshes and hybrid networks. We constructed two forms of hybrid network. The dark bars show the result of applying 3 permutation layers in a 4 nodes system and the grey bars show the network consisting of 15 permutation planes in a 16 nodes system. Such hybrid networks already form an extreme situation that all communications will use the fast circuit switching as $N-1$ permutation layers make a direct circuit pipe between any possible source-destination pair in a N node mesh. It is surprising that the hybrid network with $N-1$ permutation doesn't show much performance improvement over normal mesh in term of cycle number. An average of 97.1% speedup executing in

4 nodes system and 92.9% speedup in 16 nodes system don't compensate the huge resource put to building the circuit switching plains and their energy consumption. The results demonstrate that the system bandwidth bottleneck doesn't reside in the network system. The communications experienced by the system are not heavy enough to show the bandwidth advantage of circuit switching. Though the final cycle does not depend on the network throughput very much, the circuit switching network do decrease the average message delay significantly.

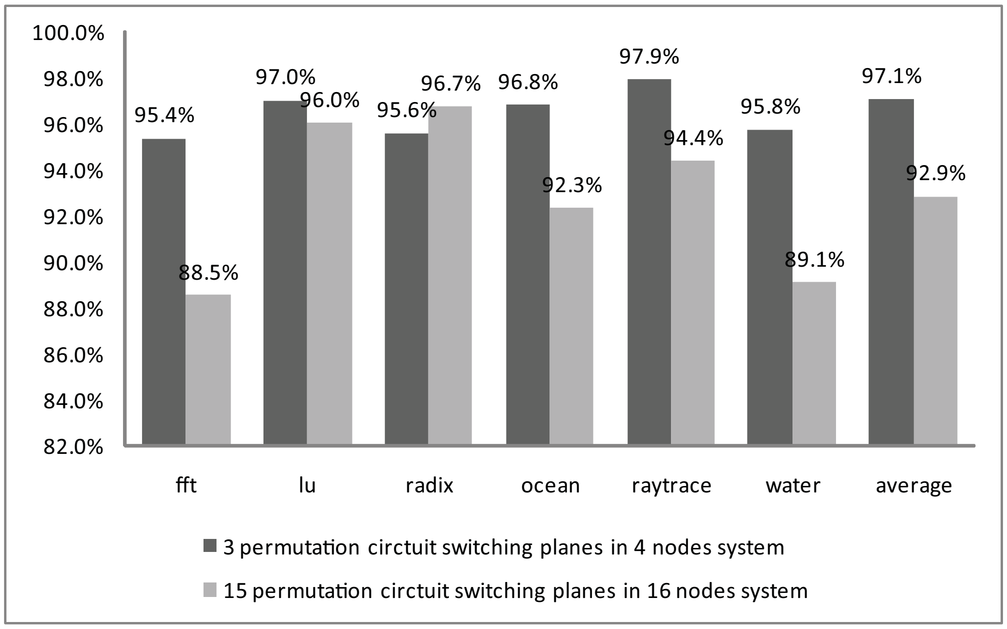


Figure 41: Comparison of execution cycles between mesh networks and hybrid networks with extreme permutation planes with 4 nodes and 16 nodes mesh size under 6 splash-2 benchmarks. Results are normalized by the normal mesh execution cycles.

Figure 42 shows the comparison of the average message latency between regular packet switching networks and hybrid networks. Using 3 permutation layers in 4 core system and 15 permutation layers 16 core system, the comparison is between the two extreme cases of pure packet switching plane and fully-connected circuit switching planes. We can see the message latency is greatly reduced by using circuit planes, from average 16 cycles in 16 core system and 6 cycles in 4 core system to almost 1 cycle in the systems with permutation layers. This demonstrates the huge bandwidth and fast exchange advantages of circuit switching. If using properly, great improvement can be achieved.

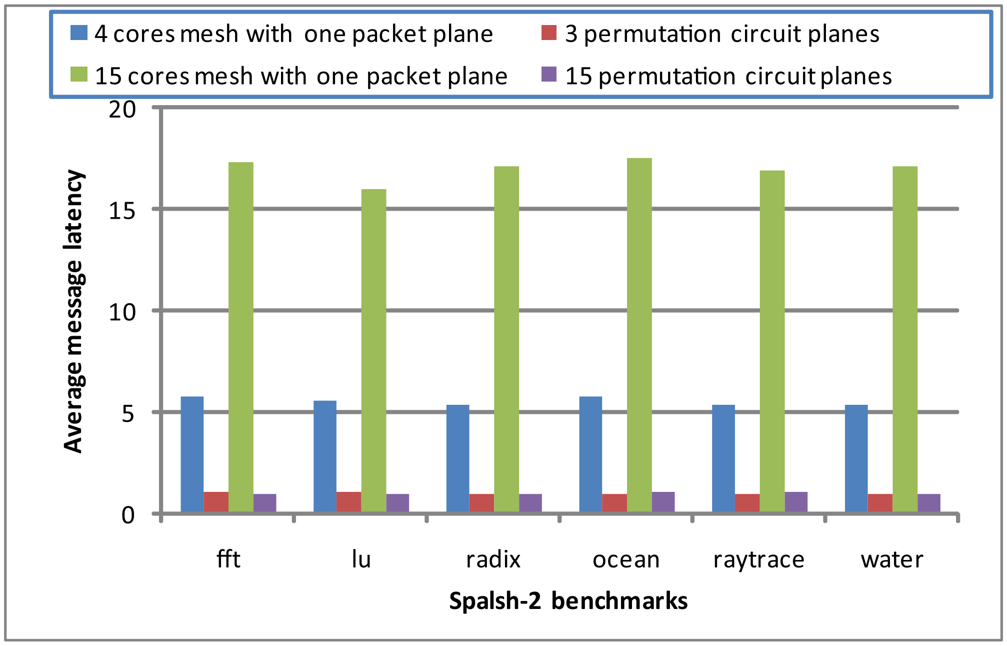


Figure 42: Comparison of average message delay in permutation circuit switching layers and packet switching network.

5.3 COMMUNICATION PATTERN ABSTRACTION

As mentioned earlier, even though circuit switching networks have the potential of achieving better performance than packet and wormhole networks at lower cost, the overhead associated with them is relatively high. The benefits of circuit switching can only outweigh its drawbacks when communication exhibits locality and when this locality is appropriately exploited. Thus, to effectively leverage circuit switching in the future, a method to determine the pattern of communication in the system is necessary.

Communication patterns abstraction using compiler techniques has been shown to be a promising approach for achieving efficient communications in the high performance computing domain [3, 54]. One of the key issues of this approach is the extraction of the communication pattern in a parallel application from its source code. In many cases the expression for a communication destination or volume will contain unresolved variables that cannot be solved at compile time. Thus, our approach focuses on manipulating symbolic expressions within the compiler to determine how these values are calculated.

In this section, based on a MPI parallel compiler my fellow researcher, Shuyi Shao developed in his PHD work [54], explores a symbolic expression analysis techniques to identify and represent the communication pattern at compile-time. In particular, symbolic analysis for loop structure is proposed as a complement for the comprehensive analysis.

5.3.1 Compiler-Based Analysis Infrastructure

Unlike traditional compiler passes which perform tasks like manipulating constants or reordering code for faster execution, our compiler technique is to analyze the communication operations within a parallel application. Because in many cases the expression for a communication destination or volume will contain unresolved variables, our approach manipulates symbolic expressions within the compiler to determine how these values are calculated. A compiled communication framework for parallel applications has been constructed. With the help of this compiler infrastructure, symbolic expressions can be created starting with simple blocks containing no control flow, then expanded to blocks containing conditionals and loops.

Our visualization and compiled communication techniques target MPI [40] parallel applications. MPI programs are written in Single-Program-Multiple-Data (SPMD) style. Each MPI process independently executes the same program on its private data. Communication will happen explicitly between MPI processes. Our technique is capable of extracting the applications communication pattern independent of various runs and our compiled communication performance improvement is not artificially tuned to a particular data-set but rather considers all possible paths.

Figure 43 shows the compilation flow of abstracting the MPI application's traffic pattern, which is part of a larger compilation framework that includes standard compiler techniques. The compiler is built within the SUIF research compiler infrastructure from Stanford University [63]. First the compiler identifies the MPI functions in the code and uses various analyses to construct a communication pattern for the application. Using in part symbolic expression analysis, the compiler infers and composes the communication pattern and according to particular interconnection

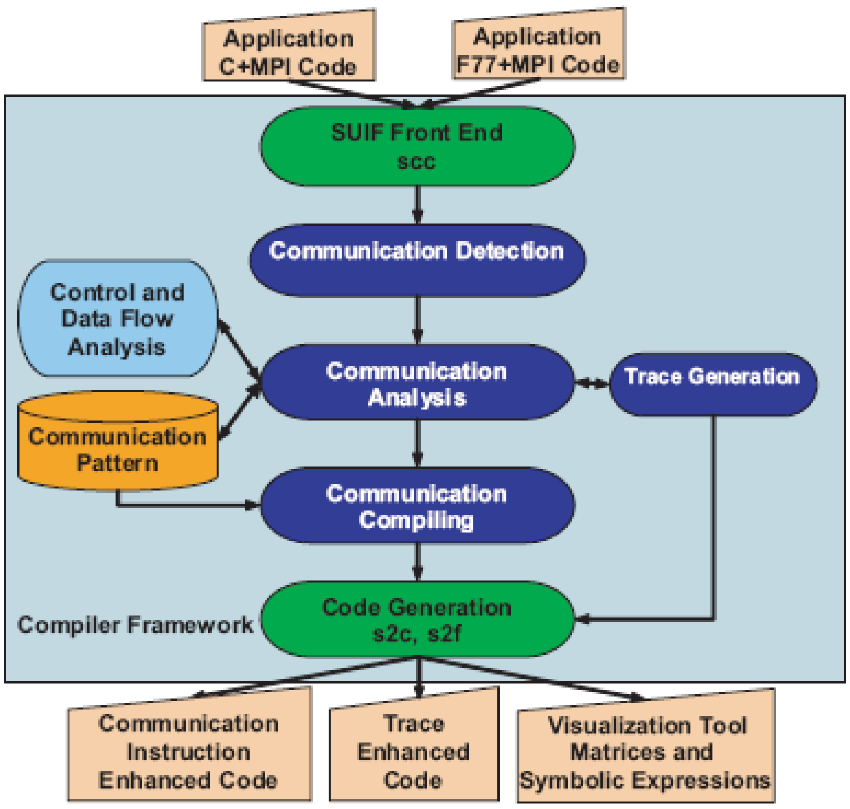


Figure 43: Experimental compilation flow.

network specifications, the communication pattern identified can be leveraged through network configuration instructions which can be inserted into the application source codes.

5.3.2 Symbolic Expression Analysis

To complete symbolic expression analysis we use a CDFG representation of the program execution [42]. CDFGs are essential for many types of compiler analysis. A control flow graph (CFG) is a directed graph representation of all possible traversal paths within a program during its execution. In the graph, each node represents a basic block. Each directed edge represents a possible control path (e.g. a branch or jump instruction). There are two special "pseudo-node"in a CFG the entry block, which is the unique entry point to the entire flow graph, and the exit block, which is the unique exit point leaving the flow graph. An example CFG is shown in Figure 44.

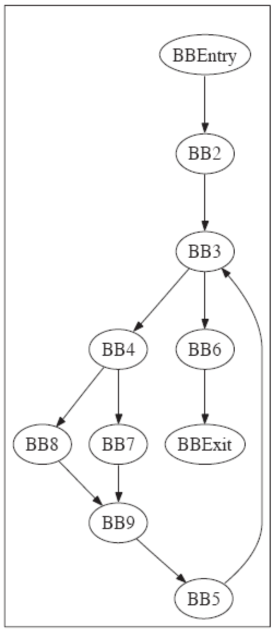


Figure 44: A CFG Example containing a loop.

Within each basic block is a data flow graph (DFG). A DFG, which is also a directed graph, represents the data dependencies within the code between control points. In a DFG, each node represents an operator (e.g. addition or logical shift) or an operand (e.g. a constant, a variable, or an array element). Each directed edge represents a data dependency that denotes the transfer of a value.

5.3.3 Loop Analysis

A large portion of MPI communication operations are embedded in loop structures. We see this characteristic in both benchmark applications such as the NAS parallel benchmarks, COMOPS, etc., as well as full blown parallel applications. This is not surprising as the loop structures in an application can often be classified as a phase of application in which communication configuration is persistent. This justifies that the communication information from loop analysis is a good candidate to configure the network. Additionally, for our symbolic expression analysis, the destination variables of MPI communication operations are also usually defined and calculated in loop structures.

The goal of our communication destination analysis is to acquire a list of symbolic expressions for destination variables in MPI calls for each loop iteration or/and a final symbolic expression for destination variable after the whole loop structure has finished executing. We have two kinds of loops in our CDFG: FOR loops, whose iteration number is statically deterministic, and DO loops, which terminate based on the condition following the loop body. WHILE loops are represented as DO loops embedded within an IF structure. Our target language constructs are formulated loops, e.g., FOR loops in C and DO loops in Fortran. The reason why we only focus on formulated loops is that we can determine their symbolic expressions through static loop analysis.

Note that loop structures form cycles in the CDFG. Normal graph traversal algorithms, i.e. DFS or BFS, can fall into infinite loops when cycles exist. To solve this problem, we take advantage of the information from the abstract syntax tree and treat all the CFG nodes (basic blocks) in a loop as an individual "super-node". The example CFG of Figure 44 has a loop inside, all the CFG nodes between BB3 and BB5 form a super-node. Our loop analysis targets each such communication-related super-node which contains information about that loop, including the upper bound, lower bound, loop index, step, and loop condition.

We also observe that the number of symbolic analysis target variables, such as destination and volume variables, is typically small in each benchmark. We also find that these variables are initialized once in the program and rarely changed later on. Thus, it makes sense to analyze each destination variable individually.

For each loop structure, we scan the loop body to see if it contains any definitions of communication destination variables. These variables fall into two classes: those that are dependent on loop index and those that are not. We handle each of these situations differently. If the variables are independent of loop index, our goal is to get a final symbolic expression after all loop iterations. In such case, we can leverage static symbolic loop solving functionality in Mathematica. Mathematica is a fully integrated environment for technical and scientific computing, especially strong in symbolic expression manipulation and calculation.

Mathematica provides an application programming interface (API) called Mathlink which allows users to communicate with the Mathematica kernel in other programming languages.

Our compiler feeds Mathematica via MathLink all of the loop information and the non-iterative symbolic expression generated for destination variables. Mathematica's Do loop command will automatically sort the loop information, iterate over the loop iterations, and return a symbolic expression for input variables after loop calculation. For example, consider the following code segment:

```
for(int i = 1; i<=3; i++)
    y = y^2 + i;
```

Our compiler feeds the following expression to Mathematica to represent the structure of the loop:

```
y = y0; Do[y = y^2 + i] {i,1,3,1}; y
```

where y0 represents the expression from the block preceding the loop. The result returned from Mathematica is:

```
3+ (2 + (1 +y0)^2)^2)^2
```

If the destination variables depend on the loop index, e.g. the destination is defined as an array $dest[i]$, each element of the array may correspond to a different communication destination. In this case, it is necessary to generate a symbolic expression for each loop iteration with the help of Mathematica. First, we generate symbolic expressions of the destination variables in the CDFG with the loop index unchanged. The initial value of the loop index and its symbolic expression as it increments between iterations is easy to obtain since the loop information is already recorded. We feed these expressions to Mathematica through MathLink for each iteration in loop order (e.g. N times, where $N = \frac{loop_{upbound} - loop_{lowbound}}{loop_{step}}$). As a result, Mathematica will return a list of symbolic expressions for the destination variables for each loop iteration.

5.3.4 CG Compiler Loop Analysis

This section shows how the loop analysis procedure can be applied to real-life benchmark CG. CG is a benchmark in NAS Parallel Benchmarks2.3 suite [2] which is designed to study the performance of parallel supercomputers at NASA Ames Research Center. CG computes largest eigenvalue of a large, sparse, symmetric positive definite matrix based on a Conjugate Gradient method. It implements unstructured grid computations and communications.

There are two communication patterns existing in CG. The first one is the communications between matrix transpose processors. Consider following code segment:

```
exch_proc = ( myId % nprows)* nprows + myId/nprows
```

exch_proc is the destination processor **myId** wants to communication, it's also the transposed destination of one matrix block owned by **myId**.

The other one is the neighbor processor communication, the code segment is as follow:

```
div = npcols;
for( i = 0; i < log2npcols; i++ ) {
    j = (myId + div/2) % div + myId /div * div
    reduce_proc[i] = proc_row*npcols + j
    div = div / 2
}
```

reduce_proc[i] is the neighbor destination processor **myId** wants to communicate. It's embedded in a FOR loop, which forms a good loop analysis candidate. The destination variable **reduce_proc[i]** depend on the loop index, it is necessary to generate a symbolic expression for each loop iteration with the help of Mathematica. The variable **div** and **npcols** are also symbolic expressions which turn out can be solved at compiler time using CDFG transversal.

The information the loop contains can be collected from CDFG graph, including loop index, lower bound, upper bound and step. The symbolic expression containing these information and the initial variables are feed to Mathematica in each loop iteration. For the first iteration, input to Mathematica:

```
div = 4; reduce_proc(0) = Floor[myId / 4] * 4 + Mod[ myId -
Floor[me/4]*4] + Floor[div /2] , div ] + (myId -
Floor[me/4]*4)/div *div
```

Floor[] is the Mathematica build-in function equivalent to the floor() function in C which returns the closet integer number smaller than the input parameter. Output from Mathematica will be,

```
reduce_proc(0) = myId + Mod[2+ myId - 4*Floor[myId/4], 4]
```

For the second iteration, input to Mathematica:

```
div = div/2; ; reduce_proc(0) = Floor[myId / 4] * 4 + Mod[ myId -
Floor[me/4]*4] + Floor[div /2] , div ] + (myId -
Floor[me/4]*4)/div *div
```

Output from Mathematica:

```
reduce_proc(0) = myId + Mod[1+ myId - 4*Floor[myId/4], 2]
```

The output expressions from Mathematica only contain **myId**, which is a runtime parameter in MPI program and will be resolved at load time.

We run our parallel compiler to apply symbolic expression analysis on MPI parallel applications and to identify the communication patterns and the generation of network configurations. The compiler also instruments the MPI applications with instructions to pre-establish circuits in a particular network configuration. We demonstrate that the benefits of using symbolic expression analysis by comparing the message delay using the compiler directed scheduling to the message delay using a runtime threshold based scheduling. In our group's previous paper [56], results showed up to 9.7 speedup of communication efficiency.

5.4 MULTI-CORE ARCHITECTURE IMPLICATION

Though the target applications we apply our compiler techniques on are Message Passing Interface (MPI) programs, which are more dominant in a distributed memory environment, we believe that these compiler techniques have the same benefits to shared memory applications and can be leveraged with proper modifications.

To utilize the compiler techniques in the shared memory programs such as the ones written with Pthread syntax, we need look for the memory access points in the program instead of the explicit message passing functions. The read or write operations of a memory location, either a variable or an array element, become possible cache accesses in the system, and need to be analyzed by the compiler.

Another challenge of applying the communication pattern abstraction technique in the DSC architecture is DSC's locality elimination property. As stated, the distributed shared organization tends to interleave the cache block at the regular granularity, either at block level or coarser levels. The resulting cache mapping policy usually destroys the natural traffic patterns inherited by the applications. This is also demonstrated by the marginal cycle reduction in adding additional circuit switching planes in the network. There needs a way to retain the locality of the application in the cache organization.

In distributed cache organization, locality is realized by the private cache. The private cache locally stores the data to reduce the communication and to reduce the cache capacity with sharing. In the distributed shared cache, a "Unique Private Cache" (UPC) concept is proposed in our research group to retain the locality of the application. UPC stores single copy of the data and assigns an owner to each data. The data is assigned to the node which accesses it with most times, which can be determined (at least partially) with the help of compiler. The system requires a directory to store the owner/mapping of the data. This directory can be further distributed at each node to avoid the access hotspot. At runtime, requestor can access the owner node for the needed data by looking up the directory.

In such multi-core architecture, we have a UPC system retaining the localities of the applications and a new compiler technique first determining the ownership of the data, and then applying the similar symbolic expression techniques to extract the communication pattern of the shared memory applications. It is hoped that these techniques can together better explore the utilization of circuit switching in multi-core architecture and generate performance improvement for future many-core system.

6.0 CONCLUSIONS

As the architecture community is starving for a fast yet functional multiprocessor simulator that meets the current multi-core to many-core transition, this thesis presents a parallel architecture simulator. It's based on a multiprocessor architectural simulator named SESC, which utilizes a MIPS instruction set emulator to fully model an out-of-order processor pipeline with branch prediction. A DSC system is modified from the original shared L2 cache. The DSC model is logically shared, which means the processors share one memory address space, but the cache bank is physically distributed at each processor node. The advantage of such a multi-core system is its ease of use for the programmer as it provides a transparent interface and convenient programming environment, while at the same time, keeping the same data locally to fully take advantage of computation capacity of multi-core system. In order to keep the data consistency among cores, a directory-based MSI cache coherency policy is fully implemented. A mesh network module is also re-implemented to interact with the DSC for scalable inter-processor communication. As these efforts complete a simulation infrastructure baseline, the thesis does not stop with the state-to-art architecture the simulator provides, two novel interconnections are thus proposed to meet the new NoC requirement. In particular,

1. A novel "fat-mesh" network similar to the idea of fat-tree. We designed and implemented two fat-mesh schemes using routing algorithms XY and XY-YX, respectively, and derived of the analytical models for both schemes.
2. A hybrid network consisting of a packet switching plane and multiple circuit switching planes along with a compiler aided symbolic expression analysis technique to extract the communication pattern of parallel applications with the help of Mathematica.

A non-uniform "fat-mesh" that attempts to alleviate the stresses on the interconnect was created similar to the idea of fat-tree. The links are systemically added at the connections experience heavy traffic. Expressions for the number of messages that traverse each link are described in two fat-mesh architectures with different routing algorithms during all-to-all personalized communication. This approximates the expected traffic pattern of popular CMP configurations. Based on this information the normalized link requirement is calculated to guide the construction of the corresponding fat-mesh structure in the simulator.

The results show that non-uniform fat-meshes offer performance using fewer resources than similar uniform fat-meshes while significantly outperforming meshes. The non-uniform fat-mesh also demonstrates performance improvements over uniform fat-meshes for the splash-2 benchmarks.

While the additional connections near the center do require additional area and resources, this is potentially offset by the reduction of links required in locations for which drivers to I/O pins may utilize the additional space. Therefore, according to the performance prior and potential overhead elimination, fat-mesh is a preferable interconnection structure in CMP design.

While there have been many modifications to mesh and torus interconnects and related architectures that studied approximately two decades ago, it is important to revisit these issues due to the new constraints of networks on chips. The hope is that the fat-mesh structure can provide some useful insights into mesh designs for many core systems, which will lead to further improvements and designs based on its theory.

The hybrid network with circuit switching acts as a preliminary model for future hybrid optical networks with its ultra-high throughput and minimal access latency. The electrical packet switching is used for irregular communications while the circuit switching can be established for regular long-lived, bulk traffic. while it still needs large adjustment including compiler aided communication pattern abstraction. the results demonstrate its promise.

7.0 FUTURE DIRECTIONS

One of the future directions of this work is parallelizing the simulator itself. Taking advantage of commercial available multi-core processors, we can speed up the simulation linearly if we can modify the simulator to a parallel version. There are already some works exploring this area. Wisconsin Wind Tunnel I [49] and II [43] are both efforts to explore key operations that underlie parallel, discrete-event, direct-execution simulation. they try to model loosely-coupled, distributed shared-memory systems and used direct-execution to model the individual processing elements. Chidester and George [10] present a distributed simulator for target CMPs based on the MPI designed to run on a host cluster of workstations. Several design space tradeoff are explored and good speedups are shown in dual-CPU workstations.

For our research, one initial step is to de-couple the processor module and cache/network module. Some initial experiments have already demonstrated that MPI is a good practice for separating the processor and the memory system. As shown in Figure 45, after starting the simulator, two

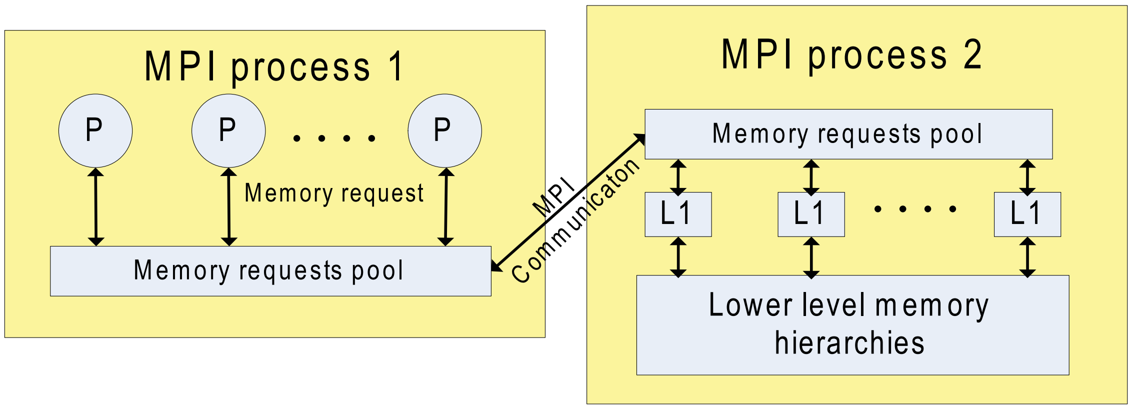


Figure 45: A possible processors and memory system decouple method .

MPI processes are created to run CPU and cache/network separately. The only interface between these two processes are various types of memory requests issued from processors to caches and acknowledgements back from caches, this gives a natural starting point to parallelize the simulator. The MPI process on the left is in charge of the core system which generates memory requests, the other process contains memory hierarchies which are driven by the received memory requests. Requests are wrapped into MPI messages and sent back and forth between two processes. As they run in different address spaces, synchronization and concurrency between them are critical. However, the ultimate goal of parallelizing the simulator is to de-couple the system to the next step such that each physical processor can run the simulation of one or more simulated cores simultaneously, the cache and network can be handled in another physical core or distributed to each physical core. We believe that the performance of simulation can be improved in proportional to degree of parallelization.

Another topic worth pursuing is traffic pattern recognition in the shared memory applications written with Pthread syntax. The communication pattern abstraction described in Section 5.3 targets MPI applications which utilize explicit point-to-point communications among processes. In Pthread programs, the multiprocessor interactions are transparent to the programmer, which means the programmer is not aware of the underlying architecture. The communications among threads are simply memory access points, including loads and stores of shared variables. This imposes a demand for compiler analysis techniques on the memory access pattern instead of the explicit send and receive function pair analysis. The initial research on this topic shows many of the memory access patterns can be still obtained using compiler analysis.

As stated in Section 2.2, when using a distributed shared cache, multiprocessor simulators such as Simics and SESC interleave the memory caching based on cache line size or some other regular address mapping. However, unless the application has been developed with this in mind, the interleaving will not match how the program is parallelized. Furthermore, the traffic pattern exhibited by the application will not show on such an organization of cache interleaving. One direction we are pursuing is, after recognizing the data allocation and traffic patterns of particular benchmarks using compiler analysis technique, trying to create a cache-data mapping organization which mimcs the pattern written in the benchmarks. The idea of UPC is described in Section 5.4. UPC tries to recognize the owner of the shared data, and fix it in the owner node. A directory is used to keep the mapping information. By doing this, we could faithfully model the behaviors of communications between the processors, and thus potentially improve performance based on the authentic pattern inherited from the applications.

BIBLIOGRAPHY

[1] T. Austin, E. Larson, , and D. Ernst, “Simplescalar: An infrastructure for computer system modeling,” *Computer*, vol. 35, no. 2, pp. 59–67, February 2002.

[2] D. Bailey, T. Harris, W. Sahpir, and R. van der Wijingaart, “The NAS parallel benchmarks 2.0,” Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Tech. Rep. NAS-95-020, December 1995.

[3] K. Barker, A. Benner, R. Hoare, A. Hoisie, A. Jones, D. Kerbyson, D. Li, R. Melhem, R. Rajamony, E. Schenfeld, S. Shao, C. Stunkel, and P. Walker, “On the feasibility of optical circuit switching for high performance computing systems,” in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, November 2005, p. 16.

[4] B. M. Beckmann and D. A. Wood, “Managing wire delay in large chip-multiprocessor caches,” in *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 319–330.

[5] D. Bhagavathi, H. Gurla, S. Olariu, J. L. Schwing, L. Wilson, and J. Zhang, “Time- and vlsi-optimal sorting on enhanced meshes,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 929–937, October 1998.

[6] R. Bhargava, L. Barnes, and B. Sander, “Amd. personal email communication.”

[7] F. Cappello and C. Germain, “Toward high communication performance through compiled communications on a circuit switched interconnection network,” in *Proc. of the Int. Symp. on High Performance Computer Architecture (HPCA)*, 1995, pp. 44–53.

[8] D. A. Carlson, “Modified-mesh connected parallel computers,” *IEEE Transactions on Computers*, vol. 37, no. 10, pp. 1315–1321, October 1988.

[9] K. F. Chen and E. H.-M. Sha, “The fat-stack and universal routing in interconnection networks,” *Journal of Parallel and Distributed Computing*, vol. 66, no. 5, pp. 705–715, May 2006.

[10] M. Chidester and A. George, “Parallel simulation of chip-multiprocessor architectures,” *ACM Transactions on Modeling and Computer Simulation*, vol. 12, no. 3, pp. 176–200, July 2002.

[11] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators," in *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 249–261.

[12] I. Corp., ".4 touchstone del x4 system description," 1991.

[13] H. G. Dietz and T. Mattox, "Compiler techniques for flat neighborhood networks," in *Proc. of 13th Int. Wrokshop on Languages and Compilers for Parallel Computing*, 2000.

[14] J. Donald and M. Martonosi, "An efficient, practical parallelization methodology for multicore architecture simulation," *IEEE Computer Architecture Letters*, vol. 5, no. 2, p. 14, July 2006.

[15] J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks: An Engineering Approach*. IEEE Computer Society Press, 1997.

[16] J. Emer, "Hasim talk at ramp retreat," 2007.

[17] M. Forsell, "A scalable high-performance computing solution for networks on chips," *IEEE Micro*, vol. 22, no. 5, pp. 46–55, September-October 2002.

[18] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing*, 2nd ed. Addison Wesley, 2003.

[19] R. I. Greenberg, "The fat-pyramid and universal parallel computation independent of wire delay," *IEEE Transactions on Computers*, vol. 43, no. 12, pp. 1358–1364, 1994. [Online]. Available: citeseer.ist.psu.edu/greenberg94fatpyramid.html

[20] R. I. Greenberg and C. E. Leiserso, "Randomized routing on fat-trees," *Advances in Computing Research*, vol. 5, pp. 345–374, 1989.

[21] Z. Guz, I. Keidar, A. Kolodny, and U. C. Weiser, "Nahalal: Cache organization for chip multiprocessors."

[22] M. R. Haghighat and C. D. Polychronopoulos, "Symbolic analysis for parallelizing compilers," *ACM Transactions on Programming Languages and Systems*, vol. 18, no. 4, pp. 477–518, July 1996.

[23] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, "Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 4, pp. 31–35, March 2004.

[24] T. J. Harris, "Shared memory with hidden latency on a family of mesh-like networks," Ph.D. dissertation, University of Edinburgh, Department of Computer Science, 1995.

[25] R. R. Hoare, Z. Ding, S. Tung, R. Melhem, and A. K. Jones, “A framework for the design, synthesis and cycle-accurate simulation of multiprocessor networks,” *Parallel and distributed computing*, August 2005.

[26] T.-C. Huang, U. Y. Ogras, and R. Marculescu, “Virtual channels planning for networks-on-chip,” in *Proc. of the International Symposium on Quality Electronic Design (ISQED)*, 2007.

[27] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas, G. Mehta, and J. Foster, “A vliw processor with hardware functions: Increasing performance while reducing power,” *IEEE Transactions on Circuits and Systems II*, 2006.

[28] S. Kamil, A. Pinar, D. Gunter, M. Lijewski, L. Oliker, and J. Shalf, “Reconfigurable hybrid interconnection for static and dynamic scientific applications,” in *ACM International Conference on Computing Frontiers*, 2007.

[29] H. Kariniemi and J. Nurmi, “Reusable xgft interconnect ip for network-on-chip implementations,” in *Proc. of the International Symposium on System-on-Chip (SoC)*, 2004, pp. 95–102.

[30] D. Kusic, R. Hoare, A. K. Jones, J. Fazekas, and J. Foster, “Extracting speedup from c-code with poor instruction-level parallelism,” in *IPDPS Workshop on Massively Parallel Processing (WMPP)*, 2005.

[31] A. T. Lawniczak, “Performance of data networks with random links,” in *Math. Computers Simulation*, 1999, pp. 101–117.

[32] F. T. Leighton, B. M. Maggs, A. G. Ranade, and S. B. Rao, “Randomized routing and sorting on fixed-connection networks,” *Journal of Algorithms*, vol. 17, pp. 157–205, 1994.

[33] C. E. Leiserson, “Fat-trees: Universal networks for hardware-efficient supercomputing,” *IEEE Transactions on Computers*, vol. 34, no. 10, pp. 892–901, 1985.

[34] V. Leppänen, “Studies on the realization of pram,” Ph.D. dissertation, University of Turku, Department of Computer Science, 1996.

[35] F. Li, C. Nicopoulos, T. Richardson, Y. Xie, V. Narayanan, and M. Kandemir, “Design and management of 3d chip multiprocessors using network-in-memory,” in *ISCA ’06: Proceedings of the 33rd annual international symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 130–141.

[36] J. Liang, A. Laffely, S. Srinivasan, and R. Tessier, “An architecture and compiler for scalable on-chip communication,” *IEEE Trans. on Very Large Scale Integration Systems (TVLSI)*, vol. 12, no. 4, pp. 711–726, July 2004.

[37] R. Lin, S. Olariu, J. L. Schwing, and B. f. Wang, “The mesh with hybrid buses: An efficient parallel architecture for digital geometry,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 3, pp. 266–280, March 1999.

[38] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, February 2002.

[39] N. Manjikian, "Parallel simulation of multiprocessor execution: Implementation and results of simplescalar," in *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software*, Nov 2001, p. 147151.

[40] *MPI: A Message-Passing Interface Standard*, Message Passing Interface Forum, June 1995.

[41] G. E. MOORE, "Cramming more components onto integrated circuits," *Electronics,*, vol. 38, no. 8, pp. 114–117, April 1965.

[42] S. Muchnick, *Advanced Compiler Design and Implemenatation.* Morgan Kaufmann, 1997.

[43] S. Mukherjee, "Wisconsin wind tunnel ii: A fast and portable parallel architecture simulator," 1997.

[44] T. Nesson and S. L. Johnsson, "Romm routing on mesh and torus networks," in *SPAA '95: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 1995, pp. 275–287.

[45] U. Y. Ogras, "Application-specific network-on-chip architecture customization via long-range link insertion," in *Proc. ICCAD*. IEEE, 2005, pp. 246–253.

[46] S. R. Ohring, M. Ibel, S. K. Das, and M. J. Kumar, "On generalized fat trees," in *Proc. of the Parallel Processing Symposium*, 1995, pp. 37–44.

[47] V. S. Pai, P. Ranganathan, , and S. V. Adve, "Rsim: An execution-driven simulator for ilp-based shared-memory multiprocessors and uniprocessors," in *In Proceedings of the Third Workshop on Computer Architecture Education, February 1997*, October 1997.

[48] R. S. Ramanujam and B. Lin, "Randomized partially-minimal routing on three-dimensional mesh networks," *IEEE Computer Architecture Letters*, vol. 7, 2008.

[49] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood, "The wisconsin wind tunnel: Virtual prototyping of parallel computers," in *In Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 1993, pp. 48–60.

[50] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, , and P. Montesinos, "Sesc simulator," January 2005, http://sesc.sourceforge.net.

[51] M. R. Samatham, "Augmented multiprocessor networks," US Patent 5134690, 1992.

[52] C. L. Seitz, W. C. Athas, C. M. Flaig, A. J. Martion, J. Seizovic, C. S. Steele, and W.-K. Su, "The architecture and programming of the ametek series 2010 multicomputer," in *Proceedings of the third conference on Hypercube concurrent computers and applications:*

Architecture, software, computer systems, and general issues. New York, NY, USA: ACM, 1988, pp. 33–37.

[53] D. Seo, A. Ali, W.-T. Lim, N. Rafique, and M. Thottethodi, "Near-optimal worst-case throughput routing for two-dimensional mesh networks," in *Proc. of the 32nd annual international symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 432 – 443.

[54] S. Shao, A. Jones, and R. Melhem, "A compiler-based communication analysis approach for multiprocessor systems," *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 10 pp.–, April 2006.

[55] S. Shao, A. K. Jones, and R. Melhem, "Compiler techniques for efficient communications in circuit switched networks for multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, 2008, in press.

[56] S. Shao, Y. Zhang, A. Jones, and R. Melhem, "Symbolic expression analysis for compiled communication," in *IEEE International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–8.

[57] D. Shires, L. Pollock, and S. Sprenkle, "Program flow graph construction for static analysis of mpi programs," in *Proc. of Int. Conf. on Parallel and Distributed Processing Techniques and Applications(PDPTA)*, June 1999.

[58] J. P. Singh, W. Weber, and A. Gupta, "Splash: Stanford parallel applications for shared-memory," *Computer Architecture News*, vol. 20, no. 1, pp. 20(1):5–44, Mar. 1992.

[59] Y.-J. Suh and K. G. Shin, "All-to-all personalized communication in multidimensional torus and mesh networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 1, pp. 38–59, January 2001.

[60] B. Towles and W. Dally, "Worst-case traffic for oblivious routing functions," 2002.

[61] P. Tu and D. Padua, "Gated ssa-based demand-driven symbolic analysis for parallelizing compilers," in *Proc. of SC*, 1995, pp. 414–423.

[62] J. E. Veenstra and R. J. Fowler, "MINT: A front end for efficient simulation of shared-memory multiprocessors," in *MASCOTS*, 1994, pp. 201–207.

[63] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarsinghe, J. M. Anderson, S. W. K. Tjiang, S. W. Liao, C. W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "Suif: An infrastructure for research on parallelizing and optimizing compilers," *ACM SIGPLAN Notices*, vol. 29, no. 12, pp. 31–37, December 1994.

[64] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "Smarts: accelerating microarchitecture simulation via rigorous statistical sampling," *SIGARCH Comput. Archit. News*, vol. 31, no. 2, pp. 84–97, 2003.

[65] ——, “An evaluation of stratified sampling of microarchitecture simulations.” *Workshop on Duplicating, Deconstructing and Debunking in conjunction with ISCA*, June 2004.

[66] Y. Yang and J. Wang, “Near-optimal all-to-all broadcast in multidimensional all-port meshes and tori,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 2, pp. 128–141, February 2002.

[67] S. G. Ziavras, “Rh: A versatile family of reduced hypercube interconnection networks,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 11, pp. 1210–1220, November 1994.