ABSTRACT

DHANOTIA, ABHISHEK. Scalable Software and Architecture for Network Routing Protocols. (Under the direction of Associate Professor Greg Byrd).

The scale and complexity of the internet has grown dramatically in recent years. While the network data plane processing has been keeping up with the ever increasing bandwidth requirements, there has not been considerable research towards scaling the network control plane. Routing protocols form a critical part of the network control plane and their processing requirements have grown significantly in the last decade. A critical issue is the increasing convergence time of routing protocols, which directly leads to higher data loss on route transitions. The serial nature of legacy code in routing protocol implementations has inhibited a shift to multicore processing in the control plane, even though there is much inherent parallelism. In this work, we investigate the use of multicore as the compute platform for routing applications using BGP, the ubiquitous protocol for routing in the Internet backbone, as a representative application.

In this work, we analyze the programmability and architectural aspects of running routing protocols on multicore processors. We develop a scalable multithreaded implementation for BGP protocol and evaluate its performance on several multicore configurations using a fully configurable multicore simulation environment based on the Simics virtual system simulator and the GEMS memory simulator. We implement several optimizations at software and architecture level and achieve a speedup of 6.5 times over the sequential implementation, which translates to a throughput of ~170K updates per second. Subsequently, we propose a generic architecture and parallelization methodology which could be applied to all routing protocol implementations so as to achieve significant performance improvement.
Scalable Software and Architecture for Network Routing Protocols

by
Abhishek Dhanotia

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

June, 2010

APPROVED BY:

Dr. Greg Byrd
Committee Chair

Dr. James Tuck

Dr. Khaled Harfoush
DEDICATION

To My Parents
BIOGRAPHY

Abhishek Dhanotia was born in December 1984 in Ratlam, India. He received the bachelor degree in Information and Communication Technology from Dhirubhai Ambani Institute of Information and Communication Technology (DAIICT), India in May, 2005.

After graduating from DAIICT, he held Hardware Design Engineer positions at Mindtree Consulting, Bangalore (2005-2006) and ST Microelectronics, Noida (2006-2008). Currently, Abhishek is a graduate student in Computer Engineering at North Carolina State University, under the guidance of Dr Greg Byrd. His research interests broadly lie in Computer Architecture and Networking. He is also affiliated with Center for Efficient, Scalable and Reliable computing (CESR) at NCSU.
ACKNOWLEDGEMENTS

First of all, I would like to thank my family for encouraging and motivating me in getting good education throughout my life.

It has been a privilege working with Dr Greg Byrd who has helped and supported me in numerous ways through my Masters. It has been a great learning experience working with him for the last two years.

I would like to thank Sabina Grover who worked with me throughout the project. I really appreciate all her help with coding, thesis writing and presentation. A lot of ideas came up in the brainstorming sessions that we had.

I would like to thank Niket Choudhary and Shivam Priyadarshhi who have been great friends for a long time. They have been with me through thick and thin and provided continuous encouragement and feedback throughout my Masters. I would also like to thank Salil Pant for his help in the initial phases of my work.
# TABLE OF CONTENTS

LIST OF TABLES ........................................................................................................ vii
LIST OF FIGURES ..................................................................................................... viii

1. Introduction ................................................................................................................. 1
   1.1 Contributions ......................................................................................................... 3
   1.2 Related Work ......................................................................................................... 4
   1.3 Organization ......................................................................................................... 5

2. Developing a Simulation Platform ............................................................................. 6
   2.1 Methodology ......................................................................................................... 6
   2.2 Determining a Network Topology ........................................................................ 7
   2.3 BGP++_conf - Generating Input files for NS2 ....................................................... 8
   2.4 Performing Network Simulation ........................................................................... 8
   2.5 Running Simulations on Multicore ...................................................................... 9
       2.5.1 Quagga Routing Software .......................................................................... 9
       2.5.2 BGP Simulation Client ............................................................................... 9
       2.5.3 BGP Daemon ............................................................................................... 10
       2.5.4 Distributed Network Simulation on Simics .................................................. 10

3. Quagga Software Architecture ............................................................................... 12
   3.1 BGP State Machine ............................................................................................ 12
   3.2 Common Libraries ............................................................................................... 14
   3.3 Critical Data Structures ...................................................................................... 15
   3.4 Packet Processing ............................................................................................... 16

4. Parallelizing the BGP Protocol Stack ..................................................................... 19
   4.1 Methodology ........................................................................................................ 19
   4.2 Task Scheduling Scheme .................................................................................... 19
       4.2.1 Methodology ............................................................................................... 19
       4.2.2 Dynamic Task Scheduling Scheme ............................................................. 20
       4.2.2.1 BGP thread library with synchronization support ................................ 22
   4.3 Synchronization ................................................................................................... 23
       4.3.1 Peer Lock ..................................................................................................... 23
4.3.2 Routing Table Locks ........................................................................................................... 24
4.3.3 Socket Descriptor Locks .................................................................................................... 24
4.4 Removing Select Logic ......................................................................................................... 25
4.5 Thread Based Streams .......................................................................................................... 27
4.6 Multiple Queues .................................................................................................................. 29

5. Results – Software Optimizations .......................................................................................... 31
   5.1 Performance of naïve DTS scheme ...................................................................................... 32
   5.2 Performance Improvement with Thread Based Streams ....................................................... 33
   5.3 Performance Improvement with Multiple Queues .............................................................. 34

6. Results – Architectural Optimizations ................................................................................... 36
   6.1 Varying Cache Sizes .......................................................................................................... 37
   6.3 Private Vs Shared L2 Cache ............................................................................................... 40
   6.4 Directory Vs Broadcast based Memory System ................................................................. 41
   6.5 Interconnection Network Behavior .................................................................................... 42
   6.6 Large Network Simulation Traces ....................................................................................... 44
   6.7 Overall Performance Speedup ............................................................................................ 46
   6.8 Performance Comparison with Peer Based Task scheduling Scheme ............................. 47

7. Conclusion and Future Work .................................................................................................. 49

References ................................................................................................................................. 51
LIST OF TABLES

Table 6.1: Multicore Simulation Parameters ........................................................................ 37
LIST OF FIGURES

Figure 2.1: BGP Simulation setup on NS2 ................................................................. 7
Figure 2.2: BGP Simulation Client ........................................................................... 11
Figure 3.1: BGP State Machine .............................................................................. 13
Figure 3.2: Queues Inside Quagga .......................................................................... 15
Figure 3.3: Packet exchange during initialization phase ............................................ 17
Figure 3.4: Quagga Software Architecture ............................................................... 18
Figure 4.1: Dynamic Task Scheduling .................................................................... 22
Figure 4.2 (a) Serialization due to File Descriptor Sets ........................................... 26
Figure 4.2 (b) Flow after removing File Descriptor Sets .......................................... 27
Figure 4.3: Thread Based Stream Optimization ........................................................ 28
Figure 5.1: Performance of the sequential implementation ...................................... 32
Figure 5.2: Performance of the multithreaded implementation with thread based stream optimization .......................................................... 33
Figure 5.3: Execution Time with different optimizations .......................................... 35
Figure 6.1: Generic Architecture of Multicore machine used for our simulations .... 36
Figure 6.2: Execution Time with varying L1 Size ..................................................... 38
Figure 6.3: Execution Time with varying L2 Size ..................................................... 39
Figure 6.4: Execution Time on Shared Vs Private L2 System ................................... 40
Figure 6.5: Execution Time on Broadcast Vs Directory based system .................... 42
Figure 6.6: Execution Time with varying link bandwidth ....................................... 43
Figure 6.7: Percentage Link Utilization with varying link bandwidth ..................... 44
Figure 6.8: Execution Times for 500 consecutive updates on 16 processors .......... 45
Figure 6.9: Overall Performance Speedup after all optimizations on different number of processors .......................................................... 46
Figure 6.10: Comparison between execution times for Peer Based Task Scheduling and Dynamic Task Scheduling Schemes .................................................. 48
Scalable Software and Architecture for Network Routing Protocols

1. Introduction

The past decade (1999-2009) has witnessed a tremendous increase in the size and scope of the internet [1]. The user base and the bandwidth demand per user have grown considerably. Consequently, the network backbone has scaled to support the increased usage. Routers are the main constituent of a network backbone and perform most of the intelligent functionality of the network. A router has two primary software components: Control Plane and Data Plane. Data plane functionality mostly involves forwarding traffic from an incoming interface to an outgoing interface. Control plane applications, on the other hand, primarily involve maintaining information about all the nodes in the internet. Routing protocols form a critical part of the network control plane and help in determining which outgoing interface the packets should take if they are destined for a particular network or IP address. In order to determine the correct outgoing interface, routers maintain reachability information usually in the form of routing tables.

Border Gateway Protocol (BGP) [2] is the de facto standard for routing in the Internet core. Since the Internet backbone consisting of Autonomous Systems (AS) performs a major chunk of data forwarding and is based on routing decisions made by the BGP protocol, it makes studying BGP all the more important. With the increased usage of the Internet, the data plane has scaled well with lot of new hardware and architecture solutions being proposed for increasing data forwarding bandwidth and throughput [3][4]. However, there has not been much work on scaling control plane applications and architectures.
The number of BGP routers and the routing table sizes within each router has grown dramatically requiring high amount of compute resources. During peak activity, the number of update messages processed by routers has been known to grow exponentially [5] compared to the updates that a router generally processes. With the current compute resources available for BGP packet processing, this may lead to scalability issues for BGP in the near future. Apart from scalability, convergence time is another key parameter which needs attention. With increasing routing table sizes, routers take more time to process a single packet, thereby increasing the turnaround time for a change to propagate across the network. This reduces the real time responsiveness of the network. With increased bandwidth and data traffic, this also means a greater number of data packets being dropped by the network. If processing time is large, it would take longer for the system to converge and a large number of intermediate messages may be produced. In the extreme case, the incoming TCP buffer may overflow and false information might get ingrained in the routing tables which may lead to loops and black holes in the system causing further data loss [5]. Increasing the processing power of the routers is the obvious solution for these problems.

Multicore processors are known to give great performance benefits for applications with high inherent data level parallelism. Several router data plane applications have already been ported to multicore machines in order to achieve scalability for high traffic and bandwidth requirements. Control plane applications also tend to exhibit significant data level parallelism which makes multicore processors an ideal choice of compute platform for these applications.

Running any application on multicore requires it to be written in a parallel fashion, so as to exploit the available processing power of the underlying machine. Most contemporary control plane applications are written sequentially, with almost no support for concurrent execution. The scale and complexity of control plane applications makes porting them to
multicore a challenging task. As most routing protocol suites carry over legacy code, and only incremental development is done when new features are introduced in the protocol, there is a lot of inertia to overcome when changing their software architecture to add support for multithreading. Hence, programmability of routing protocols for multicore processors is another important area where research needs to be done. Schemes or methodologies for multithreading the router software which require minimal changes to the software architecture would greatly help in speeding up the process of migrating the routing applications to multicore processors.

In this work, we show the benefits of using multicore processors in network routing protocols, taking BGP as a representative application. We also propose a task partitioning scheme for converting the legacy serial routing protocol code into a multithreaded one, point out bottlenecks in the system and suggest ways to mitigate them.

Subsequently, a detailed architectural analysis is done by studying the performance of this scheme for different parallel architecture paradigms including Private Vs Shared L2 cache, different cache sizes and coherence mechanisms, interconnect behavior and an optimized architecture configuration is proposed. We also evaluate the interconnect traffic behavior by studying the link utilization on different topologies.

Although, we do all our analysis using BGP, applying similar techniques to other inter-domain and intra-domain routing protocols, for instance OSPF and RIP, could give similar performance improvements.

1.1 Contributions

Specifically, the major contributions of our work are listed below,
- Develop a realistic BGP simulation environment which enables performance analysis at both the hardware and software architecture levels.
- Give a comprehensive analysis of the generic software architecture for routing protocols and propose several avenues to extract parallelism. We identify bottlenecks in the parallel implementation and suggest ways to mitigate them.
- A detailed analysis of the underlying architecture to find an optimized configuration.
- Develop a task partitioning scheme that gives a consistently good performance irrespective of the load across various peers.

1.2 Related Work

The use of distributed and multicore architectures in the data plane has been well researched. Thanks to the impressive performance gains, parallel processing has become the norm in the data plane and has found its way into modern day commercial routers.

However, the control plane has caught attention only recently, with a variety of papers talking about different applications in the control plane, including network management, security, routing protocols etc. Most research in routing protocols has been done on speeding up BGP, as it is the most important application with high computation requirements, taking over 60% of CPU utilization on routers in certain cases [6]. Several techniques have been applied to extract parallelism in BGP on multiprocessor systems. Klockar et al [7] developed a distributed router based on modularized BGP. Zhang et al [8] propose another BGP model where route computations are done on multiple agents. There has also been an effort to run BGP on multicore using the Thread Level Parallelism (TLP) approach [9] but it deals only with the computation kernels within the BGP implementation, and does not look at optimizations on the application level.
More recent work in this domain has been focused on speeding up different aspects of the protocol, but none of them provides a comprehensive mechanism to achieve overall performance improvement. Lei et al [10] develop a Threaded BGP implementation and combine it with a two-level trie based routing table structure for fast lookup in the routing tables, with which they show an overall speedup. However, they do not present a breakdown on how much speedup is achieved by multithreading alone, and fall short of doing a comprehensive system level analysis of the performance of this multithreaded scheme. Moreover, the scheme is not scalable, as it limits the number of threads to the number of peering sessions. This may lead to an under-utilized system. Furthermore, this doesn’t take into account the incoming traffic from different peers. If a peer is mostly idle, its corresponding thread will still be allocated the resources that could have been utilized by a more aggressive peer.

While the works mentioned above achieve speedup for a variety of multicore systems, none of them provides a scalable approach which can utilize the next generation of multicore processors. Moreover, no prior work provides a perspective on programmability issues involved in writing multithreaded routing protocols. Our work fills that void by providing a comprehensive approach towards developing next generation routing protocols which could benefit from the increased processing power provided by multicore machines.

1.3 Organization

This document is organized as follows. Section 2 discusses the simulation platform that is used for running the simulations. Section 3 describes the software architecture of Quagga BGP implementation that we use as the baseline code for developing the multithreaded implementation. Section 4 presents several software level optimizations done in order to improve performance. Sections 5 and 6 present the results with various software and architectural optimizations that are performed and section 7 concludes our work.
2. Developing a Simulation Platform

2.1 Methodology

BGP is run on the internet backbone and involves several thousand nodes (Autonomous Systems) communicating with each other. Hence, the number of nodes in the target simulation platform should be of a magnitude comparable to the actual internet in order to generate realistic processing requirements on any particular node.

Simulating a network with such a large number of nodes requires lot of computation power and time. It is impractical to run a detailed implementation of the BGP protocol along with the underlying hardware for all the nodes in such a large network. Hence, a trace driven methodology is chosen, wherein a network simulation is performed at a higher level of abstraction using the Network Simulator (NS2) with a BGP protocol agent attached to it, and a packet dump is generated at an arbitrary node in the network.

Subsequently, this node is simulated at a detailed level using Simics [11] and Gems [12] and packets are fed to this node from the generated packet dump. This enables us to have a network simulation with a large number of nodes, while also allowing us to do a detailed performance analysis for any node in the network using the packet traces.

Figure 2.1 shows the various steps that are performed in order to generate the BGP packet traces.
2.2 Determining a Network Topology

Several topology generators were evaluated in order to generate a realistic network topology on which simulation could be performed.

- **Boston University Representative Internet Topology Generator (BRITE)**
  BRITE [13] supports generating flat as well as hierarchical topologies and is flexible enabling to model links with different bandwidth and latency. It allows for each node in the network to have a different degree (number of connections to different peers) and provides several built in schemes for automatically generating variable degrees. It provides an intuitive GUI based interface and is hence easy to use and modify.

- **Georgia Tech Internetwork Topology Models (GT-ITM)**
  The GT-ITM topology generator [14] can be used to create flat random graphs and two types of hierarchical graphs, the N-level and transit-stub.
**INET Topology Generator**

INET [15] generates topology and connectivity information but does not model latency and bandwidth for the network links. It takes the number of nodes in the network as an input and generates random networks based on data from internet topologies.

BRITE was chosen for generating topologies, as it provided both the topology information as well as the link bandwidth and latency information.

### 2.3 BGP++_conf - Generating Input files for NS2

BRITE generates topology information in the form of a .brite file. Subsequently, a translator is developed which reads files in the .brite format and translates it into the input format of BGP++_conf [16].

BGP++_conf is a configuration utility from Georgia Tech for BGP++ (BGP protocol agent for NS2, described in next section). It takes as input simple configuration parameters and generates a '.tcl' file, describing the high-level simulation setup (e.g. topology, BGP routers, traffic agents), and a number of '.conf' files, specifying the configuration of individual BGP routers.

### 2.4 Performing Network Simulation

NS2 [17] is used for performing network simulations on the generated topologies. NS2 doesn’t support BGP protocol for simulation. Hence an external protocol simulator (agent) needs to be attached to NS2 for performing BGP communication between the network nodes. BGP++ [18] is such an agent from Georgia Tech which can be attached to NS2.
After attaching BGP++, simulations are performed on NS2, which allows packet dumps to be generated at all the nodes in the network. The dump files are generated in MRT format [19].

### 2.5 Running Simulations on Multicore

Once BGP traces are generated for a single node on the network, this node could be run on a multicore machine in order to analyze its performance. A BGP Simulation Client program was developed which reads BGP packets from the trace file and feeds it to the simulated node. The node in consideration runs the multithreaded BGP implementation on a multicore

#### 2.5.1 Quagga Routing Software

Quagga [20] is an advanced routing software package that provides a suite of TCP/IP based routing protocols. It implements OSPF, RIP and BGP daemons which run the complete protocol stacks for the routing protocols. This software suite is used for performing detailed analysis of the BGP protocol stack. OSPF and RIP are disabled for our study so that only the BGP daemon is run on the processor.

#### 2.5.2 BGP Simulation Client

A BGP simulation client is a C program which imitates the functionality of several BGP peers and maintains connectivity with the BGP daemon under study.

The client interacts with the TCP/IP stack of the host machine and initiates a BGP session (1 for each peer) with the BGP daemon. Once the TCP session is established, it reads the packet
dump file (in MRT format) which was generated by performing BGP network simulation and parses information from the file on a per peer basis. It then generates BGP packets based on the parsed data and sends them to the BGP Daemon. All the response packets from BGP Daemon are discarded. Once all the packets from the file are parsed and sent, the client terminates the TCP session with BGPD.

2.5.3 BGP Daemon

BGP Daemon is the node which is run on a simulated Multicore machine using SIMICS virtual system simulator and gems memory simulator. BGP Daemon runs a parallel version of the Quagga routing software which is described in Section 4. SIMICS provides flexibility to vary the number of cores simulated on a system. GEMS/Ruby memory model is also attached to the machine simulating BGPD for simulating the memory hierarchy.

2.5.4 Distributed Network Simulation on Simics

Simics provides support for performing network simulation through its integrated support for running a network between the simulated nodes. Two multicore machines are run within a single process on Simics, one of them running the multithreaded BGP daemon and the other running the simulation client. An Ethernet link is created between the machines, which is used for all packet communication. The multicore machines run Solaris 10 operating system with support of complete POSIX thread library and TCP/IP stack. All performance numbers are generated for the machine running BGP daemon. Performance numbers for the machine running simulation client are either disabled or ignored.
Figure 2.2 depicts the multicore simulation environment being used for running all the simulations. The simulation client runs on a separate Simics node and feeds packet to the BGP daemon through the network established between the two nodes. A separate TCP connection is maintained for each peering session. MRT files are loaded in the machine running the client before starting the simulation.

Figure 2.2: BGP Simulation Client
3. **Quagga Software Architecture**

This section presents the general software architecture of Quagga BGP which is used as the baseline code for our multithreaded implementation. Quagga [20] is a complete routing protocol suite supporting several intra-domain and inter-domain routing protocols. Quagga BGP daemon is implemented as a monolithic code with no support for multithreading or performing different tasks in parallel. A TCP connection is maintained with each active peer and packets are exchanged by implementing the standard BGP state machine for each peer (BGP neighbor) session. The state machines are run for each peer and are independent of each other. The state transitions for one peer session do not affect those in the other sessions. Following subsections describe further details about the software architecture.

### 3.1 BGP State Machine

Figure 3.1 shows the BGP state machine that is implemented in the protocol stack.
In order to make decisions in its interactions with other BGP peers, a BGP peer uses a simple finite state machine (FSM) that consists of six states: Idle, Connect, Active, OpenSent, OpenConfirm, and Established. A state variable is maintained for each state within the peer structure. The BGP protocol defines the messages that each peer should exchange in order to change the session from one state to another. Each peer session starts in the “Idle” mode when all resource initialization happens and a TCP connection request is sent to the peer. The second state is “Connect” where the router waits for the TCP connection to complete, transitioning to the "OpenSent" state if successful. In our simulations, the BGP daemon operates in the passive open mode where the daemon just waits for a connection request from the peer instead of actively initiating a connection. Hence, after transitioning to “OpenSent” the session immediately transfers to “Active” state where it listens to connection requests from peers. On a connection request, BGP Open and Keepalive messages are exchanged next, and upon successful receipt, the router is placed in the “Established” state.
Once established the router can now send/receive Keepalive, Update, and Notification messages to/from its peer.

### 3.2 Common Libraries

There are some common libraries which are used by all the routing protocols in the Quagga routing suite. The prominent ones among them are thread, stream and queue.

- **Threads in Quagga sequential code** refer to the functions that need to be performed on any state machine transition or timer expiry. Each thread holds a pointer to a function which gets executed when the thread is called. This is independent to the threads that we talk about in our multithreaded implementation. From here on, the library threads will be referred to as tasks.

- **Streams** are buffers that are maintained for each peer session which hold the incoming and outgoing packets. Two streams are maintained, one for the incoming packets from a peer session and the other for outgoing packets.

- The implementation maintains five different task queues namely event, ready, read, write and background. Tasks to be performed by BGP router are segregated into these queues. Read queue holds tasks scheduled to read data from the incoming TCP buffer. When there is data to be read, the task is moved from the read queue to the ready queue. Similar is the case for write queue. All BGP timers are held in the background queue and moved to ready queue when they expire. Event queue holds all the events generated by the state machine. Tasks are picked up from the event and ready queue and executed one at a time. The Event queue has priority over the ready queue. Figure 3.2 shows the flow of tasks between different queues in Quagga.
3.3 Critical Data Structures

Some of the critical data structures maintained inside the code are described below.

- **Routing Information Base (RIB)** – Stores active paths to all the prefixes which are received via BGP update messages from BGP peers. Paths which are filtered after applying different BGP policies are held in this table.
  - This table also maintains conceptual Adjacent Routing Information Base for incoming and outgoing entries (*Adj-RIB-in* and *Adj-RIB-out* respectively).
  - *Adj-RIB-in* stores the reachability information which is extracted from incoming update messages.
- **Adj-RIB-out** stores the reachability information that needs to be sent to the neighbors through outgoing updates.

- **Forwarding Information Base (FIB)** – Holds the best routes which should be used for reaching a particular network prefix. If a new entry is added in RIB which has a better route to the same prefix, then FIB is updated appropriately. This table is common across Quagga routing suite and is updated by other routing protocols as well and is used by the data plane for actual data forwarding.

### 3.4 Packet Processing

The BGP daemon initially starts in passive open mode listening for any connections. When a peer comes up, the initialization process happens with a sequence of BGP Open and KeepAlive messages being exchanged. Figure 3.3 shows the sequence of packets that are exchanged during the initialization phase.
After the initialization is complete, periodic updates are exchanged between the peers.

Figure 3.4 depicts Quagga software architecture along with the packet flow that happens at the ingress and egress ports of the BGP router. It also shows the structures which should be modified in a critical section when multithreading is implemented. Subsequent sections describe in detail how synchronization is achieved by accessing these data structures in a critical section.
Figure 3.4: Quagga Software Architecture
4. Parallelizing the BGP Protocol Stack

Most of the routing protocol implementations presently in use are sequential with limited or absolutely no parallel code. What this means is that these implementations do not exploit the inherent parallelism of the protocol, nor would they give any speedup when run on a Multicore machine. In our implementation, we try to come up with a multi-threaded implementation of the BGP protocol which exploits the inherent parallelism. We derive the base implementation of BGP from the Quagga routing software suite.

This section presents our scheme of multi-threading the Quagga BGP implementation. It also discusses various software and hardware level optimizations implemented to improve performance on multicore machines.

4.1 Methodology

The parallel routing protocol suite was developed by making modifications to the existing sequential Quagga implementation. The software architecture of sequential implementation was studied in detail and opportunities for exploiting parallelism were explored. Subsequently, a multithreaded code was developed. POSIX thread library was used to implemented all the multithreading and synchronization mechanisms.

4.2 Task Scheduling Scheme

4.2.1 Methodology

The sequential implementation described in the previous section involves reading tasks from the head of event and ready queues and executing the associated functions one at a time. When executing the functions, more tasks are enqueued in the event and ready queues and
the execution continues. However, the disadvantage of this approach is that all the tasks are executed sequentially even when they may be completely unrelated to each other. For instance, if two update messages are enqueued in the ready queue, they would be picked up one at a time and executed. However, they are completely independent and can be processed simultaneously if there are multiple threads. This is the fundamental approach that we follow when writing the multithreaded code. In a multithreaded implementation, there can be several ways in which tasks could be picked up from the ready queues and executed. An obvious approach could be to assign a peer to each thread and segregate tasks based on the peer to which they belong. This approach has already been implemented in a related work on Threaded BGP where each thread maintains operates on a separate peer session. However, a disadvantage of this scheme is that it may lead to improper load balancing if some peers are more aggressive than the others leading to some threads being idle and clogging up compute resources when they have nothing to do. This scheme also limits the number of threads spawned by assigning one thread for each peer. These disadvantages could be mitigated by several mechanisms by spawning threads and assigning tasks to them in a scalable and load balanced manner.

This is the approach that we follow when devising the Dynamic Task Scheduling scheme. By task scheduling scheme, we mean a particular mechanism in which threads are spawned and tasks are picked up from these queues and executes. In case there are access conflicts on the common data structures, the task scheduling scheme also defines the mechanism to avoid/mitigate such conflicts or executing such accesses in a critical section with appropriate locks so that only one thread has access to the data structure at a time.

**4.2.2 Dynamic Task Scheduling Scheme**

In Dynamic Task Scheduling Scheme (DTS), a pre-determined number of threads are spawned at the start of the router software. Common ready and event queues are maintained
as in the case of original code. Each thread scans for executable tasks in the ready and event queues and processes them irrespective of the peer which they belong to. A master thread is responsible for moving data from read, write and background queue to the ready queue. All the other threads just keep on scanning the ready and event queues and execute tasks as and when they arrive. This scheme thus decouples the number of threads from the number of peers and hence is scalable with the number of processors available on the multicore machine.

However, this scalability comes at a cost. Now that multiple threads could be operating on packets for the same peer, another level of synchronization needs to be maintained in order to keep the data coherent for a particular peer. As mentioned in section 3, two streams are maintained for each peer to store the incoming and outgoing packets. When multiple threads operate the same peer, these streams could potentially get corrupted and hence need to be accessed in a critical section. A peer lock is maintained inside the peer structure which restricts the peer data structures to be modified by a single thread at a time. Figure 4.1 below depicts the DTS scheme.
4.2.2.1 BGP thread library with synchronization support

As all the task queues (ready, event, read write etc) can be modified by multiple threads, it becomes necessary to access them in a critical section. Whenever a thread has to read or
write data to any of the task queues, it would have to first acquire a lock on the queue and then perform any operations.

The thread library defining these queues is common across all the routing protocols in the Quagga suite, hence a custom thread library is developed which has synchronization support for accessing all these queues in parallel. A POSIX mutex is associated with each queue and any thread accessing the queue has to first acquire the associated mutex variable.

### 4.3 Synchronization

Synchronization is required in the parallel implementation because multiple threads may be accessing the common data structures simultaneously thereby corrupting them. Several lock variables are defined in order to prevent access the associated data structures by multiple threads simultaneously.

#### 4.3.1 Peer Lock

A peer structure is maintained for holding information associated with a particular peer. This includes the socket descriptor for the peer, information about the current tasks that are pending execution in various queues, information about work queues, and pointers to the input and output streams. When a connection is established with this peer during the initialization phase, all these fields are populated with appropriate values.

When parallel threads execute tasks for a particular peer simultaneously, there is a possibility that the information stored in peer structures may get corrupted. Hence, the peer structure information needs to be accessed in a critical section. A peer lock variable is maintained in the peer structure which needs to be acquired before accessing any value inside the peer structure.
4.3.2 Routing Table Locks

Tasks in the BGP daemon access the local BGP routing table (RIB) as well as the Forwarding table (FIB) which is shared by all the routing protocols. Routing tables are organized in the form of a tree. Two lock variables, a traversal lock and a node lock, are maintained for each node within the tree in order to control access to these nodes and modifying the table. A traversal lock variable is to be acquired when a thread wants to traverse the tree to search for any prefix. A node lock is to be acquired when a thread needs to modify information stored within the node, which include the information about its parent and child nodes as well as the reachability information about this prefix.

4.3.3 Socket Descriptor Locks

The sequential implementation uses file descriptor sets (used by select () function) to peek into the TCP buffer and see if there is data pending on any socket. If there is data pending to be read for any peer, the select function would set the corresponding bit in the file descriptor set. When a thread executes the read task for the peer, it would reset this bit so that it could be reused by the select logic again. Hence, the file descriptors could be modified by each thread and hence need to be accessed in a critical section. A lock variable is introduced to preventing access to these file descriptor sets.

The subsequent sections describe the optimizations that are done to improve the performance of this scheme.
4.4 Removing Select Logic

As mentioned in the previous section, the sequential implementation maintains file descriptor sets to keep track of any new packets arriving on the incoming TCP buffer. However, a major disadvantage of this is that it combines this check for all the peers, thereby serializing the code. All the threads have to wait for this check in order to proceed with the processing of the incoming packets. This can be totally avoided as each peer has its own independent file descriptor and the thread handling this check can do so independently. Figure 4.2 (a) describes how the serialization happens and figure 4.2 (b) depicts how it has been replaced in the parallel implementation.
Figure 4.2 (a) Serialization due to File Descriptor Sets
4.5 Thread Based Streams

We introduce the concept of thread based streams to resolve the contention on peer streams when multiple threads process incoming packets for a particular peer. When streams are associated with a peer, threads processing the same peer have to acquire a lock to access...
these streams. Moreover, with this approach only one packet per peer can be read at a time. However, in realistic network scenarios, there is a possibility that multiple packets from a peer arrive at the same time requiring the need to process them in parallel. If streams are maintained on a per thread basis, it allows each thread to access its stream independently of the others. This allows multiple packets from the same peer to be read and processed in parallel.

There is an additional overhead requirement of storing these streams as the number of threads is greater than the number of peers. For instance, if we have 10 peers and 40 threads, we would require an additional storage of 30 streams (30 * 512 bytes/stream ~ 15KB). However, the overhead of storing additional streams is marginal and can be ignored given the performance benefits that it provides. Figure 4.3 shows the thread based stream optimization.
4.6 Multiple Queues

In the naïve implementation of DTS scheme, only one set of queues is maintained that is shared by all the threads in the BGP daemon which leads to contention on accessing these queues. As the queues are accessed frequently (every time a thread picks up a function to execute or schedules new functions to be executed), this becomes a severe performance bottleneck.

This contention can be potentially reduced if multiple queues are maintained. Each thread or a group of threads can pick up tasks from a different queue. Similarly, when adding new tasks to be executed, each thread accesses a different queue. Different schemes could be applied to add and pick data from multiple queues. A simple and effective choice is to add data to the queues in a round robin fashion. Separate round robin variables are maintained for each queue. Tasks are added to the queue corresponding to the round robin variable. Subsequently, the value of the variable is incremented.

With this mechanism, queue contention is greatly reduced as each thread now has to acquire locks just for modifying its corresponding ready and read queues as opposed to a single lock for the common queue. While picking up data, no lock needs to be acquired if a separate queue is maintained for each thread. In another optimization, the need for round robin variables and corresponding locks is also eliminated by associating a queue with each thread so that a thread always adds/picks packets from its own queue.

Another advantage of this scheme is that it ensures load balancing among threads. As packets are added in a round robin fashion, this allows each thread to process packets irrespective of the peer they come from. There is no storage overhead of having multiple queues as this scheme just segregates the tasks into compared to keeping them inside a common queue.
There are no additional structures that need to be maintained in order to implement this scheme.

Figure 4.4: Multiple Queues Optimization
5. Results – Software Optimizations

This section presents the results and analysis of simulations performed on the Dynamic Task Scheduling scheme. We first compare the results of the baseline sequential implementation with the naïve DTS scheme and subsequently discuss how different optimizations help in achieving better performance speedups. We also evaluate the performance of the underlying multicore architecture and study the impact of various architecture parameters on the performance of the multithreaded application.

As discussed in section 2, the run time of BGP Daemon could be divided into two phases, the initialization phase where connections are set up for each peer, and the update processing phase when incoming updates are processed and subsequently new paths are advertised to peers. When generating performance numbers, we just consider the update processing phase of the BGP simulation, as the initialization phase is run only once when the router is booted and is not very compute intensive compares to the update processing phase.

The performance numbers for this section are generated using a network topology involving 1000 nodes for processing 1000 updates messages. The update messages are sent to the node in consideration as fast as possible so as to evaluate the maximum update processing capability. We also run simulations for processing a much larger number of updates and show that processing time for 1000 updates is a representative number for comparing performance numbers. The node in consideration has 12 active peers with which it maintains connectivity and exchanges messages. Figure 5.1 gives the execution times for the sequential implementation on different number of processor cores. As the implementation is completely sequential, no speedup is achieved when adding more number of cores.
5.1 Performance of naïve DTS scheme

First, we evaluate the performance of the naïve task scheduling scheme. The naïve implementation has only one set of read, write, ready and event queues on which all threads operate. Each peer has its own stream buffer which means only one packet per peer can be read at one time. A common set of file descriptors are used for probing TCP buffers using the select command. This implementation does not include any of the optimizations that were discussed in section 4. The second last bar in Figure 5.2 shows the improvement in the execution time achieved with this scheme. Only the results for execution time on 16 processors are shown on the graph. On a lower number of processors, the lock contention among threads increases the execution time significantly and hence the numbers blow out of

Figure 5.1: Performance of the sequential implementation on different number of processor cores
range. Hence, in the subsequent optimizations, we focus on lock contention in order to improve performance on a lower number of processor cores as well.

![Figure 5.2: Performance of the multithreaded implementation with thread based stream optimization on different number of processor cores](image)

### 5.2 Performance Improvement with Thread Based Streams

The first improvement made over the naïve scheme was to associate streams with threads instead of peers as described in section 4.5. This enables multiple packets from a single peer to be processed in parallel. Also it decreases lock contention when multiple threads process packets from the peer simultaneously. As can be seen in the third set of bars in Figure 5.2, thread based stream optimization gives significant speedup when run on different number of cores. The speedup achieved is approximately 2.4 times over the sequential implementation.
A disadvantage of this scheme is that if there is a large number of threads per processor, the overhead of storing streams increases. As streams get operated upon very frequently, all the streams cannot be cached in L1 / L2 caches and data needs to be fetched from the main memory.

### 5.3 Performance Improvement with Multiple Queues

The next optimization was to create a set of read, write, event and ready queues per thread instead of having a common set for all threads, as explained in section 4.6. This distributed approach eliminates the contention among threads to access the common queues. With this approach, each thread picks up data from a different ready and event queue. Data can be added to the multiple ready and event queues by several mechanisms.

Two among such schemes were evaluated: one where tasks are allocated to queues in round robin fashion, and the other where an initial distribution is made across queues and all subsequent packets of the same peer are allocated to the same queue. The round robin scheme consistently performed better than the other one because of better load balancing at each thread. In the peer based scheme, when multiple packets from the same peer arrive, they get assigned to a single list when other lists may be empty. This reduces the load balancing and degrades the performance severely. We thus chose the round robin scheme of assigning tasks for all our experiments.

Figure 5.3 shows performance of the multithreaded implementation with multiple queues. As can be seen, this scheme significantly improves the performance over the last optimization. This gives a speedup of approximately 2.1 times over the last optimization and an overall speedup of 5.05 times over the sequential implementation for a multicore with 16 processors.
An anomaly in this scheme is observed for a 2 processor machine in which case the performance of this scheme degrades. No concrete reason could be attributed to this behavior but this may be due to the overhead of maintaining queues and the memory behavior. However, this can be ignored as most multicore machines today already have more than two cores.

![Figure 5.3: Execution Time with different optimizations on varying number of processor cores](image)

Figure 5.3: Execution Time with different optimizations on varying number of processor cores
6. Results – Architectural Optimizations

This section presents the architectural analysis done over the optimized multithreaded implementation described in Sections 4 and 5. All the results are performed on a system with MOSI coherence protocol and a private L2 cache associated with each processor. The interconnection network has a hierarchical switch topology unless specified otherwise. Table 6.1 shows the parameters and figure 6.1 shows the generic multicore architecture used for our simulations.

![Generic Architecture of Multicore machine used for our simulations](Image)

Figure 6.1: Generic Architecture of Multicore machine used for our simulations
Table 6.1: Multicore Simulation Parameters

<table>
<thead>
<tr>
<th>Multicore Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulated Processor</td>
</tr>
<tr>
<td>UltraSparc III running Solaris 10</td>
</tr>
<tr>
<td>L1 Cache</td>
</tr>
<tr>
<td>split I&amp;D, 64 KB 4-way set associative, 1 cycle access latency, 64-byte line</td>
</tr>
<tr>
<td>L2 Cache</td>
</tr>
<tr>
<td>4 MB per processor, 6 cycle access latency, 64-byte line</td>
</tr>
<tr>
<td>On-chip Network</td>
</tr>
<tr>
<td>Hierarchical switch, 13 cycle link latency, 4 virtual channels, 4 buffers per virtual channel</td>
</tr>
<tr>
<td>Coherence</td>
</tr>
<tr>
<td>MOSI at L2-cache level</td>
</tr>
<tr>
<td>Ethernet</td>
</tr>
<tr>
<td>Link latency – 1000 cycles</td>
</tr>
</tbody>
</table>

### 6.1 Varying Cache Sizes

We analyze the L1 and L2 cache behavior of the multithreaded implementation. Figure 6.2 shows the execution times for different L1 cache sizes. The application performance is very sensitive to the L1 cache size especially when the L1 cache is small. When the size is increased beyond 64 KB the performance saturates giving negligible speedup for subsequent doubling of the cache size.
The application behavior on varying L1 cache size falls along the expected behavior. However, an interesting point to note is the L2 cache behavior. The performance of the DTS Scheme remains almost inelastic to L2 Cache size variation as shown in Figure 6.3, where the cache size is increased from 64KB to 8MB per bank (1 bank per processor on a 16 processor system). The speedup achieved by adding a large on chip L2 cache is marginal when compared to the speedup seen by varying the L1 size. This behavior could be attributed to the fact that most data structures used in the implementation are allocated dynamically and hence do not exhibit any spatial locality. Moreover, all tables, lists and queues are organized as linked data structures and caches are known to perform poorly for such implementations. There are schemes available in literature to improve the cache performance of applications.
with linked data structures by using customized memory allocation routines so that elements in a single list are assigned consecutive memory addresses. However, we don’t evaluate such schemes in our implementation.

![Figure 6.3: Execution Time (ruby cycles x 10^6) with varying L2 Size](image)

Figure 6.3: Execution Time (ruby cycles x 10^6) with varying L2 Size

As the application does not require large L2 cache size, we perform an optimization on L2 cache size by keep it small and reducing the cache access latency to achieve performance improvement in memory accesses. This gives a further speedup of 2.9% over the optimized software implementation.
6.3 Private Vs Shared L2 Cache

We analyze the performance when the L2 cache is kept private to each core and when it is shared among all the processors in the system. The results are evaluated on a directory based system (described in section 6.4). The total L2 cache size of the system is kept constant. The Figure 6.4 shows the performance comparison with both the multicore systems. As can be seen, the performance of a system with private L2 is better than that of a shared L2. For small number of cores, the performance difference is significant and the gap shrinks for large number of cores. The lower performance of shared L2 can be attributed to the fact that the application shows little data sharing (discussed in later sections) and hence keeping the L2 private eliminates the need for maintaining cache coherence at L2 levels thereby increasing the performance.

![Figure 6.4: Execution Time (ruby cycles x 10^6) on Shared Vs Private L2 System](image)
6.4 Directory Vs Broadcast based Memory System

We evaluate the performance on a directory and a snoopy / broadcast based coherence protocol. In a directory-based system, the data being shared is placed in a common directory that maintains the coherence between caches. The directory keeps information where all the memory addresses are cached in the system and their state information. When an entry is changed, the directory either updates or invalidates the other caches with that entry. In a broadcast based or snoopy system, caches monitor address lines for accesses to memory locations that they have cached. When a write operation is observed to a location that a cache has a copy of, the cache controller invalidates its own copy of the snooped memory location.

Figure 6.5 shows the performance of the multithreaded application on a Directory and a Broadcast based memory system. A general behavior observed for this application is not much data sharing between the processors. This could be attributed to the fact that all threads have dedicated data structures on which they operate. The only shared structure is the routing table which is large and each thread is usually updating different parts of the table. Hence, not many updates or invalidations happen during the simulation.

In a broadcast based system, each cache sees all the memory requests, which is useful when there is lot of data sharing. In our case, as most of the messages on the interconnect pertain to getting data from the memory, all caches don’t need to see those messages, thereby limiting the amount of processing at the cache controllers. Hence, the directory based system performs better where each cache doesn’t need to see all the traffic on the interconnection network and limits the number of messages processed by the cache controller.
6.5 Interconnection Network Behavior

We analyze the behavior of the multithreaded application on different interconnection network configurations. Figure 6.6 shows the execution time with varying link bandwidth on a system with 16 processors. The performance of the application is very sensitive when the link bandwidths are small but the performance gain saturates quickly. This shows that the application is not very sensitive to interconnection network bandwidth and links with low bandwidth are sufficient for achieving optimal performance.
Figure 6.6: Execution Time (ruby cycles x 10^6) with varying link bandwidth (in bytes/cycle)

Figure 6.7 shows the percentage link utilization for different interconnect bandwidths. The network utilization steadily falls as the network bandwidth is increased. The application does not require large interconnect bandwidth and complicated topologies to optimize the traffic and network consumption. Hence a simple low latency network could be used. This further reduces the execution time by 6.5% on a 16 processor system giving an overall speedup of 9.35% over the optimized software implementation.
Figure 6.7: Percentage Link Utilization with varying link bandwidth (in bytes/cycle)

6.6 Large Network Simulation Traces

Simulations were run on larger trace sizes with each node advertizing four times the original number of IPV4 prefixes. This would lead to an increased routing table size and hence more time to traverse the routing tables. As the simulation progresses, the routing table sizes would grow and each update processing may take incrementally more time. However, the processing time remains almost constant as throughout the simulation. As can be seen in figure 6.8, the execution time for 500 updates remains approximately constant at about 3 million cycles.
The blue line in figure 6.8 shows the trend in update processing time as the simulation progresses. The line has a very small slope indicating that the processing times increase only slightly with larger routing tables. A reason for this may be the organization of the routing table. Since it is a binary tree, having four times the number of prefixes would increase the depth of the tree by two or three levels, increasing the traversal time marginally as each level just adds another iteration to the tree traversal code. The increase in memory required to store the routing tables also does not have much impact on the processing time.

![Figure 6.8: Execution Times for 500 consecutive updates on 16 processors](image)
6.7 Overall Performance Speedup

Finally, after all the above mentioned optimizations are applied, the best configuration is chosen and a speedup of 6.5 x is achieved with 16 processors as seen in figure 6.9.

Figure 6.9: Overall Performance Speedup after all optimizations on different number of processors
6.8 Performance Comparison with Peer Based Task scheduling Scheme

In another work on parallelizing BGP in our group, Peer Based Task Scheduling scheme is developed in which a thread is maintained for each peer the simulated BGP node is connected with. Whenever the BGP node receives connection request from a peer, a new thread is spawned. This thread maintains a TCP connection with the respective peer and runs the corresponding BGP state machine. Each thread maintains separate read, write, event, ready and background queues for the corresponding peer. When processing incoming and outgoing updates, it acquires lock on the routing tables so as to update them in a critical section. A master thread common among all peers is also maintained, which handles all the housekeeping functions that are not associated with any peer. These include periodically clearing routing tables, checking socket buffers for new data and maintaining file descriptor sets etc.

We compare the performance of Peer Based Task Scheduling scheme with the Dynamic Scheduling scheme along in Figure 6.10.
Figure 6.10: Comparison between execution times for Peer Based Task Scheduling and Dynamic Task Scheduling Schemes
7. Conclusion and Future Work

The key contributions of our work include providing comprehensive analysis of the generic software architecture of routing protocols and proposing ways on how it can be converted to a multithreaded application suitable for running on multicore processors. We also propose several hardware architecture level optimizations which can give performance improvements on multicore. We take Quagga BGP as a representative routing protocol implementation and show that peer based task scheduling scheme could give up to 6.5 times speedup when run on 16 processor cores.

Although we do all our implementation and analysis on a particular implementation of a specific routing protocol, the parallelization techniques and analysis that we propose are applicable to other protocol implementations. For instance, Quagga routing suite also supports intra domain routing protocols namely OSPF and RIP. Implementations for these protocols use the same thread, stream and queue libraries that are used by BGP. In our implementation, we propose alternative implementations for stream and queue libraries. Making similar optimizations to OSPF and RIP implementations can give significant speedups in their performance.

We view our work as a step towards the development of multithreaded routing protocols which could exploit the enormous computer power of multicore. Although we provide a detailed analysis for extracting speedup on BGP, there are several other avenues which could be explored. Some of these avenues are described herewith.

- Transactional Memory – Transactional memory allows simultaneous execution of critical sections with the flexibility of rolling back the result in case there are access conflicts. This makes them particularly suitable for modification to structures like
trees and linked lists where each thread usually modifies different part of the data structure. Hence, transactional memory could be used for controlling access to routing tables. This should enable multiple threads to access the table simultaneously thereby providing faster table lookups.

- Other Scheduling Schemes – As discussed, Peer Based Task Scheduling is an alternate scheme which is deployed in another work which gives significant speedup compared to the sequential implementation. Other task scheduling schemes could be explored which may pick up tasks from the ready queue by a different mechanism. This could lead to different access patterns on the queue and stream structures and may help in improving performance.

- Routing Table Organization – Several schemes have been proposed to achieve better access times on the routing table which could be deployed in conjunction with our scheme to further improve the performance.
References


