ABSTRACT

GROVER, SABINA. Using Multicore to Accelerate Network Routing Protocols. (Under the direction of Dr. Gregory T Byrd.)

The number of Internet users and connected devices has grown considerably over the past decade with approximately one in every three people on the globe an Internet user. The number of websites and web servers has also grown into the millions. With mobile computing becoming more pervasive, we will soon be exhausting all the available IPv4 addresses and moving into the IPv6 era. The Internet backbone is under tremendous performance pressure to keep up with this exponentially increasing usage. Network Processors form the heart of the physical Internet infrastructure, and improving their performance can go a great deal toward providing better connectivity, bandwidth and an overall better Internet user experience.

Border Gateway Protocol (BGP) has become a critical routing application, as it forms the primary protocol that is used for packet routing across different Autonomous Systems on the Internet backbone. Good performance of BGP on network processors directly translates to better convergence time for route changes on the Internet, leading to reduced data loss and better network reliability. As BGP is the ubiquitous routing protocol on the Internet core, analyzing its performance and exploring avenues for speeding it up can greatly help in improving the responsiveness and scalability of the Internet.

In the past, BGP routers have benefited from the increasing single thread performance of processors to keep up with its growing performance requirements. However, the microprocessor industry has now shifted its focus to multicore processors to maintain their performance growth according to Moore's law. Thus, it is imperative to change the BGP software architecture to reap the benefits provided by multicore processors.

In this thesis, we investigate the use of multicore as the compute platform for routing protocols, using BGP as a representative routing application. We propose a multithreaded BGP implementation called PBTS that focuses on improving the update processing functionality which forms the critical path of the BGP implementation, and gives a speedup of 5.5x on a multicore processor with 16 cores. Subsequently, we analyze the performance of both serial and parallel BGP implementations on a fully configurable multicore simulation environment based on the Simics virtual system simulator and the GEMS memory model. We identify several bottlenecks in the parallel BGP software and the underlying multicore architecture that limit performance and scalability. Finally, we propose a canonical multicore architecture that can mitigate or remove these bottlenecks and give better thread scaling for parallel BGP implementations. The analysis and proposed schemes in this work would greatly help in understanding the behavior of BGP, thereby assisting in the design and development of next generation network processors.

Using Multicore to Accelerate Network Routing Protocols

by
Sabina Grover

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2013

APPROVED BY:

_____	_____
Dr. Gregory Byrd Committee Chair	Dr. Yan Solihin
_____	_____
Dr. Khaled Harfoush	Dr. James Tuck

DEDICATION

To

My Parents, for showing me the path

And to my loving husband,

For choosing to walk beside me.

BIOGRAPHY

Sabina Grover was born in 1984 in the city of Ludhiana in Punjab, India to Rashmi and Dinesh Grover. She lived there throughout her schooling after which she moved to New Delhi for four years to pursue her B.E. in Instrumentation and Control Engineering at Netaji Subhas Institute of Technology (NSIT), Delhi in 2002.

Sabina then joined the Electrical and Computer Engineering at North Carolina State University as a Ph.D. student in 2006, where she also received her Masters Degree in Computer Engineering in 2010. Here she pursued her work on her Ph.D. under the guidance of Dr. Greg Byrd. Sabina's research interests primarily include computer and network processor architectures. While in graduate school, she also did two internships in the area of computer architecture at Intel Corporation where she will also be joining full time once she graduates.

ACKNOWLEDGEMENTS

Graduate school is a long journey with many ups and downs, through which a number of people have had tremendous impact on my life and the outcome of this dissertation. My first words of gratitude go to my loving husband, Abhishek who has been a great friend and soul mate through a good part of my life in graduate school. He not only helped me stay focused on the single most important goal of every graduate student, which is to graduate timely, but was also always around to discuss ideas, or to simply cheer me up. This thesis would simply not have been as much fun if things had been any different. He has been the ideal companion through the crucial last few months, never complaining when I was working at odd hours and always finding ways to make it easier for me. How will I ever make it up to him!

I have been blessed with a wonderful family that has been my biggest stalwart all through my life. My parents have been with me through every decision and change in my life, big or small, positive or negative, and have filled my life with endless love. They have always supported me in everything I've done, even if it meant sending their daughter to the other side of the globe to an unknown land to pursue her dreams. I have learnt the most important lessons in life from them and hope to be able to live life in their footsteps. My brother, Suvir has always been able to boost my morale and make me laugh. I wish I can be as good a sister to him as he is a brother to me.

My advisor, Dr. Greg Byrd, provided me with invaluable guidance at each step throughout my Ph.D. His insights and critical reasoning have gone a long way in shaping this work. He has made himself very approachable to his students and I have been able to just walk in to his office and find him willing to discuss. He is not only a good advisor, but also a very fine person. He really cares for his students. I have learnt a lot from him, not only in research, and am hoping that I am able to inculcate his work ethics, honesty, and fairness in my life. He has constantly been a source of inspiration throughout school and will be a role model for life.

Ms. Elaine Hardin at the ECE graduate office deserves a special mention for her tireless efforts in making the paper work a breeze throughout these years.

My cousin, Ankush and his wife, Brinda went to great lengths to ensure that my move to the US was as smooth as possible. Ankush flew down to NC to receive me at the airport and made sure I had everything I would need before he left. I inherited my first car from Brinda, after which I realized how much I was missing one in the first place.

Zainab Zaidi, my roommate of six years and closest friend, can brighten up my day by simply being around. If there is anything that I will miss about Raleigh, it is her company. Not only is she fun to be with, she goes to great lengths to make a positive difference to people around her. I have known her for more than ten years now, and it is a bond meant to last. She is the sister I never had. Veena Karthikeyan, my second roommate, and Zainab

continued to stay close to the university campus for four years after they started working so that we could still stay together, and this is something I'll forever be grateful for.

I also found a wonderful friend in Poulomi Pal, and I used to look forward to spending time with her both inside and outside our lab. When she graduated, being in the lab was suddenly not so much fun any more.

I met Ratna didi and Raghu Bhaiya for the first time when I came here, but it never felt that way. They provided me with a home away from home, especially when I would be missing my family the most after my first few trips to India. I'll always cherish their love and friendship.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1: Introduction

Routers are the primary constituent of the Internet backbone and perform most of the intelligent functionality of the network. The functionality of routers can be broadly classified into two categories: Control Plane and Data Plane. While the data plane is concerned with switching and forwarding packets from one interface to another, the control plane has the responsibility of identifying the correct interface to forward those packets on based on their destination address. The control plane achieves this by maintaining a map of the Internet called the *routing table,* which is pieced together by the connectivity information it receives from its neighbors. Each router runs Routing Protocols to assimilate this information and build a database of the routes available in the Internet, which it in turn advertises to its neighbors to keep each router's routing table coherent.

The Internet has grown tremendously over the last few decades, in terms of both size and usage. With the increased usage of the Internet, the router's data plane has scaled well, with several new hardware and architecture solutions being proposed for increasing data forwarding bandwidth and throughput [2, 12]. However, there has not been much work on scaling routing protocol software and router architecture to keep up with the increasing

Internet size. Physical Internet infrastructure is made up of Autonomous Systems, which is an aggregation of one or more routing prefixes that are maintained by an Internet Service Provider (ISP). Routing protocols that operate within an Autonomous System (AS) and maintain routing information for network prefixes inside a single AS are classified as Intra-domain routing protocols. Most commonly used Intra-domain routing protocols are Routing Information Protocol (RIP) [27] and Open Shortest Path First (OSPF) [39]. Inter-domain routing protocols are run between different Autonomous systems. Border Gateway Protocol (BGP) [52] is the de facto standard for inter-domain routing protocol that is used on the Internet core. Figure 1.1 represents a very generic view of different routing protocols in a network ('R' represents a router and 'ASn' represents an Autonomous System).

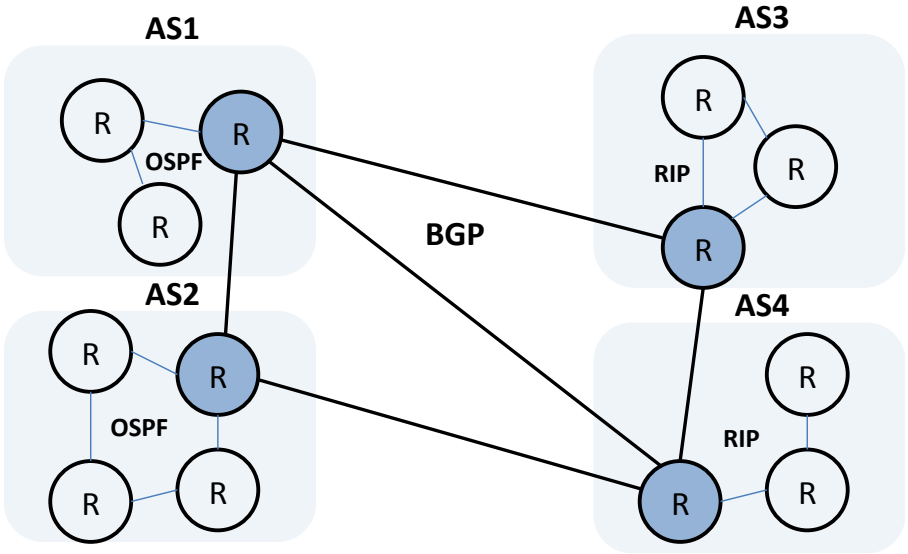

Figure 1.1: Inter-AS and Intra-AS Routing Protocols

As the routers on the Internet backbone perform a major chunk of data forwarding based on routing decisions made by the BGP protocol, it makes studying BGP all the more important. Over the years, researchers and industry practitioners working in the networking domain have made several efforts to study BGP deployment on the Internet, including traffic trends and BGP packet surges [30, 37], and to improve traffic engineering [21, 50, 51] by configuring the routers efficiently to control and improve routers performance. With the increasing number of BGP routers and Autonomous systems, efforts have also been made to change the protocol itself to make it more scalable, like adding support for confederations [57], route reflectors [4] and flap damping [58]. However, limited effort has been made to optimize the underlying router hardware itself so that it is able to run the BGP protocol more efficiently and scales well with the network size. In our work, we try to fill this gap by doing a detailed study of BGP from the computer architecture perspective with the ultimate goal to contribute towards development of scalable BGP software implementations and efficient network processor hardware.

1.1 Multicore for Routing Protocol Processing

Network control plane applications in general, and routing protocols in particular, offer plenty of opportunities where the inherent parallelism of the applications can be exploited to achieve better performance. The primary functionality of any routing application is to communicate with other routers on the network to exchange information about network

topology and how to reach a particular IP address on the network. Each router can have multiple network interfaces and can potentially receive packets from several neighboring routers simultaneously. For each incoming packet, the router applies different filters based on the configured policies, and subsequently performs a routing table update if the packet passes all the filters. Packets received from different interfaces are independent of each other and can be processed in any order, even in parallel with each other.

There has been a paradigm shift in the microprocessor industry with the focus having shifter from Single core superscalar processors to Multicore. Multicore processors are known to give great performance benefits for applications with high inherent data level parallelism. A wide variety of applications in different fields have benefitted from the transition to multicore. Even on network processors, several data plane applications have already been ported to multicore machines in order to achieve scalability for high traffic and bandwidth requirements. As network control plane applications also tend to exhibit significant data level parallelism, multicore processors are an ideal choice of compute platform for these applications. In this thesis, we investigate the applicability of Multicore towards improving the performance of routing protocols. Chapter 2 discusses the performance issues in the current BGP deployment over the Internet and shows that moving to multicore will be more of a necessity than a choice.

1.2 Analyzing BGP from a Computer Architecture Perspective

While there have been numerous efforts to improve BGP performance from a network protocol perspective, there has not been much work on understanding the BGP performance issues from a computer architecture and system perspective. In this work, we try to address this by undertaking a two-pronged approach of understanding the performance bottlenecks that exist in BGP software implementations, as well as understanding issues in generic CPUs when they run BGP applications.

As a first step towards analyzing BGP, we realized that there are not many tools or simulation software available in academia or industry that enable running BGP on a configurable multicore machine. To enable our work, we have devised a novel methodology to simulate BGP from a computer architecture standpoint. The BGP simulation environment we developed uses a two staged approach, where the first step involves generating BGP packet traces from a realistic network topology and subsequently performing detailed architectural simulations with these packet traces. More details on the simulation environment are discussed in Chapter 3.

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 in 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. Coming up with schemes 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 thesis, we delve deeper into an open source routing protocol implementation to understand its software architecture, and propose schemes to convert the legacy BGP software code into a multithreaded application, including implementation of synchronization constructs and ways to access routing tables in parallel. We do detailed performance analysis of the sequential and parallel BGP implementations on our simulation environment, and provide several optimizations to the parallel implementation that further improve the application performance and make it more scalable. More details on the parallel BGP implementations are discussed in Chapter 3, and its performance analysis and results are shown in Chapter 4. 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.

1.3 Canonical Multicore Architecture for Routing Protocols

The data plane of the network processors have used customized hardware implementations to improve their packet processing capability for a long time. Popular network processors like Cisco Quantum Flow and Intel IXP provide customized chip designs which can handle more than 100+ Gb/s packet processing bandwidth inside a single chip which is embedded with several application-specific cores. As an example, the Cisco ASR 1000 [13] consolidates 40 application-specific processing cores on a single piece of silicon to deliver wire-speed data path forwarding features and services. However, for the network control plane, most network processors still rely on general purpose processor architectures. This trend cannot continue in the future, with more and more processing requirements on the control plane. The scale and sheer number of network processors on the Internet backbone justifies having customized control plane hardware, too, which is capable of scaling with the data plane.

In our work, we identify several hardware bottlenecks that limit the performance of sequential and parallel BGP software. We show that packet parsing is a critical function of routing protocols that consumes a large fraction of the total CPU cycles, and propose Stream Hardware Assist (SHA) to significantly speed up this parsing functionality. We also describe that routing protocol software heavily relies on the performance of the underlying TCP/IP networking subsystem for its performance. Any optimizations that improve the latency and bandwidth of the network subsystem have a significant impact on a routing protocol's packet

throughput. Finally, we propose a canonical multicore architecture suitable for routing protocols that mitigates or eliminates the performance bottlenecks. The changes we propose can be combined with customizations that are required for network control plane applications other than routing protocols, in order to develop efficient Network Processor designs for the future. Further details about the hardware optimizations and canonical architecture are discussed in Chapter 5.

1.4 Contributions of our work

This research makes the following contributions.

- We develop a realistic BGP simulation environment that enables performance analysis at both the hardware and software architecture levels.
- We give a comprehensive analysis of the generic software architecture for BGP and propose avenues where parallelism could be extracted by multicore machines [15, 25].
- We propose a task partitioning scheme for BGP called Peer-Based Task Scheduling (PBTS) [25], show that it achieves significant performance speedup over the sequential implementation, and compare it with another task partitioning scheme developed as a parallel effort in our group [14].
- Based on performance studies, we propose and evaluate architectural mechanisms to improve routing protocol performance.

The simulation environment was co-developed with contributions from Abhishek Dhanotia [14].

Chapter 2: Background and Related Work

2.1 Internet Trends

The number of Internet users has grown more than five times in the last decade (2001-2011) [31], and now spans about one-third of world's population. The number of connected devices on the Internet is even larger and crossed the number of humans for the first time in 2012 [22]. Interestingly, the amount of data traffic on the Internet will reach more than a Zettabyte/year by 2016, which is of the same order of magnitude as the amount of data transferred by the eyes to the brain of the entire human race in a month [32]. To support these ever growing bandwidth and throughput requirements, the Internet backbone infrastructure has grown considerably.

The advent of mobile computing, especially the popularity of smart phones and tablets, has led to many more connected devices on the Internet. This has led to further aggravation in the rate at which available IPv4 addresses are getting exhausted. This trend is expected to grow in the future with the introduction of many more IPv6 based devices. Internet Service

Providers (ISPs) need to add more Autonomous systems to keep up with this growing trend of connected devices.

BGP software architecture has not changed much since it was first introduced in 1989 [52]. However, the number of connected Autonomous Systems and routers running using BGP has grown considerably since then. Figure 2.1a shows the growth in active BGP routing entries since its inception, and 2.1b shows the growth trend in the number of active Autonomous Systems in the last decade. A more important factor to consider is that the growth rate of BGP update messages on the Internet vastly exceeds the growth rate in the routing table size. Huston et al [30] discuss various trends in the number of update messages exchanged on the Internet through the year 2005, including update messages/day, BGP prefix table sizes, and BGP table size projections. They show that processing requirements will grow 4x in the next 3-5 years. These trends in BGP usage point to the fact that it is imperative to have faster BGP packet processing capability in network processors to keep up with the requirements.

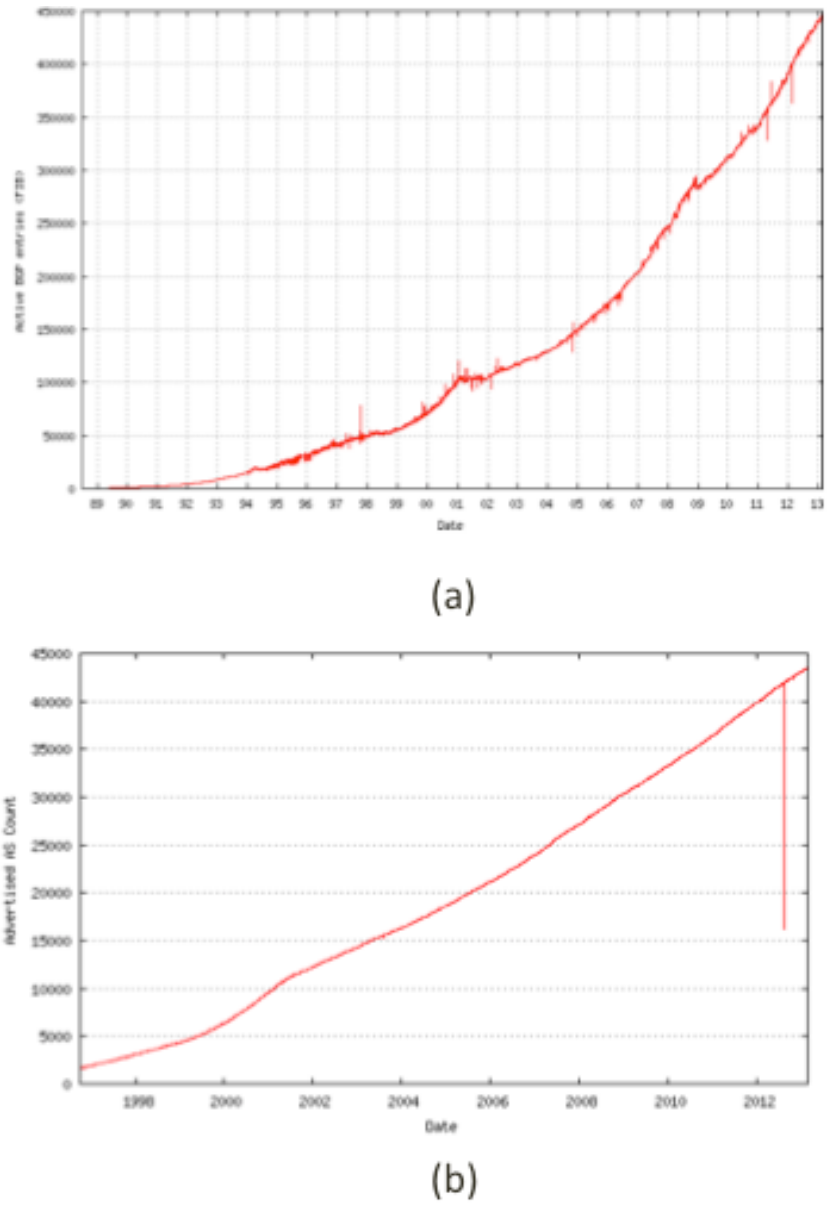

Figure 2.1: a) Number of Active BGP entries and b) Advertised AS assignments over the last decade. (Source: [24])

2.2 BGP Performance and Moore's Law

With the increasing BGP processing requirements, one might argue that it matches with the compute power of the latest processors, which is growing at the rate predicted by Moore's law (i.e. 2x every 1.5-2 years). However, the performance improvements in the latest generation of processors are being made possible by increasing the number of processing cores and not by improving the single-thread performance. In fact, over the last few years, there has been only marginal improvement in the single-thread performance as well as processor frequencies, primarily due to power constraints when running at high frequencies. (See Figure 2.2.) Multicore processors have a much better performance/power ratio, and hence they are able to keep up with the compute power as projected by Moore's law.

While the BGP software has benefitted from the increased single threaded performance in the past to keep up with its scaling requirements, it is no longer possible to keep up that trend with the processor technology shift towards multicore. Hence, it is imperative for the BGP software architecture to adopt multi-threading to reap the benefits of new processors.

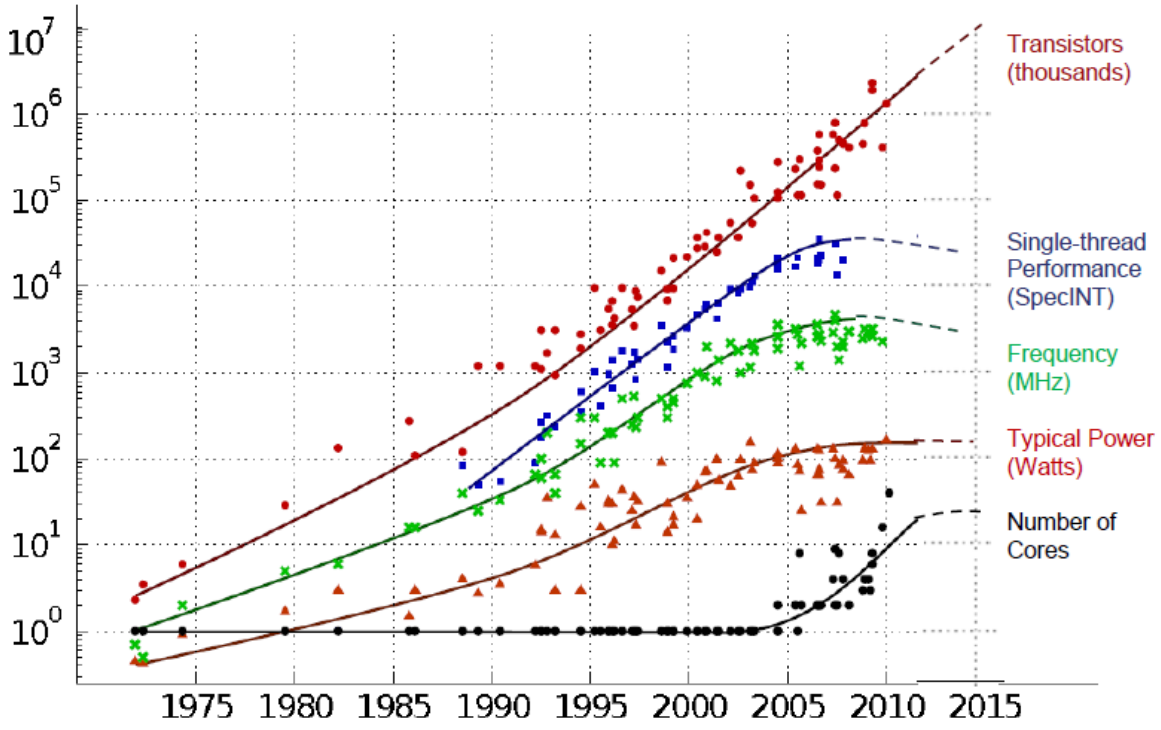

Figure 2.2: Trends in Processor Technology [1]

(Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten)

2.3 BGP traffic surge and worm attacks

An interesting BGP behavior is that the number of update messages received by a router varies significantly through the day. During peak rates, the number of update messages processed by routers has been known to grow exponentially and can reach up to 1000 times the daily average [29]. Thus, there can be very good days in the history of BGP routers when

the traffic load was really low at all times, and then there will be bad days when the traffic surges have lasted for few hours with traffic load being many times the average.

There have also been several cases in recent years when dedicated worm attacks have caused a sudden surge in the BGP update traffic, leading to the network routers becoming unresponsive. One instance of that was the slammer worm attack [37], when there was a 24x increase in update traffic between two small edge Autonomous Systems. BGP flap damping [58] is a scheme to prevent such attacks by filtering out changes on a particular network link/route if the changes happen at more than a certain frequency.

Due to this erratic traffic load behavior, it is important to note that when considering the compute power required by BGP, we should take the peak rates into account and not just go by the daily or monthly average rates. With the current compute resources available for BGP packet processing, this may lead to scalability issues for BGP in the near future when there will be cases where the router is not able to handle the peak traffic rates [30].

2.4 Convergence, Stability and Scalability

Some metrics that can be used to measure BGP router performance include packet throughput, update processing time and convergence time. Packet throughput pertains to the number of updates packets that a router can process per second. Update processing time is the latency from the time it takes for reading an update packet stored at the ingress port of the

router, parsing the packet for path vectors, applying filtering policies, performing a routing table lookup for the network prefix and finally updating the corresponding entry in the routing table with this path vector information.

As each BGP router only shares network information with only its peers, which subsequently forward this information to their peers and so on, it might take a long time before a network change propagates through the Internet and every router has the updated network information. The BGP steady state convergence time is measured as the time from a network change to the time when all routes are accepted, entries are made in the routing table and the ingress and egress queues of each router are empty with respect to that change. Higher convergence time means that if the link changes happen very frequently, and the BGP network is not able to propagate them fast enough, the network might not reach a steady state, leading to stability issues and reducing the real time responsiveness of the network.

Low convergence time is very important when scalability of the protocol and network is concerned. The number of route updates exchanged between routers grows exponentially, even when the network size scaling is linear. Another drawback of high convergence time is the resulting loss of packets when the routing tables don't have correct information of the network topology, and the routed packets may keep looping through the network.

2.5 Related Work

While there has been a plethora of work [18, 20, 28, 49] to speed up the network data plane using various parallelization techniques, the control plane has lacked such attention. Although the control plane has been evolving in terms of functionality and complexity, routing continues to be the most fundamental task of the control plane. Most research on 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 [3]. Most research on BGP in the last decade has been done to improve the protocol specification and configuration to achieve better scalability and reliability, as well as implementing better traffic engineering schemes to improve performance and packet throughput. Only very recently has the focus started shifting towards optimizing the network processor hardware itself to run routing protocols more efficiently. In this section, we discuss some recent efforts that focus on the characterizing and improving network processor hardware performance for networking applications and, more specifically, routing protocols.

2.5.1 Benchmarking Network Processors

There have been few different benchmarking suites for Network Processor performance. The most popular among them are CommBench [60], NetBench [47], NpBench [40], and Intel's NP Benchmarking methodology [38].

CommBench consists of eight different applications, which are classified as either header processing or packet processing applications. NetBench has a different way of classifying applications according to their applicability in the networking stack. They define three types of applications – IP-level, Application level and lower-level micro-workloads. Both Netbench and Commbench mostly focus on the data plane functionality of network processors and do not specifically deal with control plane. Network Processors Benchmark Frame work (NpBench) was the first benchmarking suite that clearly identified control plane as a separate set of applications in addition to the data plane. NpBench includes traffic management, QoS, and security related protocols as part of their control plane benchmark suite. Intel's benchmarking methodology defines workloads at the hardware level, micro level, function level and system level. It clearly identifies workloads that can be used to characterize different processor and memory performance features for both data and control plane functionality. As we can see, the benchmarking suites have started identifying the control plane applications as a critical component that determines the overall performance of a network processor.

2.5.2 Evaluating BGP performance on different hardware systems

A study by Wu et al. [61] compares the performance of a broad range of possible router implementations with different processor designs that run BGP: a general purpose uni-core machine (Pentium-3), a dual-core processor (Xeon), an embedded network processor (IXP

2400) and a commercial router (Cisco 3620). To evaluate the performance of these systems, they develop a benchmark that exercises different workload scenarios that represent different routing information exchanges that occur in a network.

Table 2.1: BGP Performance without Cross-Traffic (transactions per second)

	Pentium III	**Xeon**	**IXP2400**	**Cisco**
Scenario 1	185.2	2105.3	24.1	10.7
Scenario 2	312.5	2247.2	36.4	2492.9
Scenario 3	204.1	2898.6	26.7	10.4
Scenario 4	344.8	1941.7	43.5	2927.5
Scenario 5	1111.1	3389.8	85.7	10.9
Scenario 6	3636.4	10000.0	230.8	3332.3
Scenario 7	116.6	784.3	11.6	10.7
Scenario 8	118.7	673.4	14.9	2445.2

In table 2.1, the first four scenarios focus on the router's response in the BGP start-up phase and ending phase, where large numbers of announcement and withdrawal messages are processed. The other four scenarios test the router's response to incremental BGP updates. Scenarios 5 and 6 do not result in an update in the forwarding table, whereas scenarios 7 and 8 do. As we can see, Intel Xeon gives the best overall performance among the systems

evaluated because of having two cores and support for 2x hyperthreading, giving a total of 4 logical cores. This finding makes a strong case for using multicore machines in the control plane.

2.5.3 Multicore for BGP

Several techniques have been applied to extract parallelism in BGP on multiprocessor systems. Klockar et al. [36] developed a distributed router based on modularized BGP. Zhang et al. [62] propose another BGP model where route computations are done on multiple agents. A major drawback of these multiprocessor implementations is that they do not exploit shared memory (which is commonly available in multicore processors), and hence they incur a lot of communication overhead and provide only limited speedup.

There has been some interesting work that emphasizes the applicability of customized multicore and multiprocessor systems for improving control plane performance. Frank et al. [23] are working on a Wire Speed Processor (WSP) project for handling packet traffic at wire speed. WSP is a generic processor architecture that integrates 16 multithreaded processor cores, hardware accelerators and high speed I/O functions into a single chip. They show the benefits of using a customized network processor design that tightly couples several hardware features to get wire speed performance on several data and control plane applications.

Mittal [48] proposes optimizing the multicore architecture for networking applications. He proposes using hardware offload engines for packet classification and security features which do not come in the actual packet stream flow. He further proposes using a queue at the network interface, and then doing per-processing-element traffic management to utilize the several cores available in a multicore machine. He also proposes modifying the memory architecture to manage separate queues for different data paths.

Previous work most related to our research is done by Lei et al. [42], who devise a Thread Level Parallelism (TLP) based mechanism to exploit the inherent parallelism in BGP. However, they work on speeding up the computation kernels and do not deal with optimizing the complete BGP application. They use the min-cut algorithm, which is based on control flow and data dependence analysis, to partition BGP into 32, 64, 256 and 512 threads, allowing 4, 8, 12 and 16 threads to run in parallel on an eight-core Sun Fire 1000 server to achieve a 1.51-2.92x speedup. This work develops a scheme to parallelize BGP in hardware but falls short of getting significant speedup as it does not exploit data parallelism in BGP. In another work, Lei et al. [41] propose a mechanism to speed up routing table accesses. They 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. Also, no prior work on BGP 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.

Chapter 3: BGP Implementations

BGP is a variant of distance vector routing protocols, which exchanges routing information in the form of path vectors with all its peer routers. Each entry in the routing table consists of destination network, next router and the overall path to reach the destination network. There are several policies that a network operator may choose to configure, based on which networks routes can be added or filtered out. Each BGP router receives update packets with path information from its neighbor, which it adds to its routing table, and then modifies the packet before sending it out to other neighbors. The modified message is essentially appended with its own AS number as the next hop in the path vector. Unlike other network protocols, BGP uses TCP as the underlying transport protocol to ensure reliable communication with its neighbors. This eliminates the need for additional reliability and flow control features in the BGP protocol itself.

There are several commercial and open source implementations of BGP protocol that are used on the Internet. The most popular routing suites that implement BGP are Cisco IOS [8], XORP [26] and Quagga [33]. While the general software architecture of each protocol is the same as they implement the same BGP state machine, there are some differences in terms of

the BGP protocol related optimizations and features they implement. For all our BGP analysis, we choose the Quagga BGP implementation, as it is an easily available open source application and that can be compiled and run on Linux/Unix and it implements most of the BGP protocol features. Quagga routing software also implements other popular routing protocols such as RIP and OSPF, and uses some common libraries across the different protocols. Thus some of the improvements that we propose in the Quagga common libraries should also help speed up the other routing protocols. Similarly, although we use Quagga BGP implementation for all our performance analysis, we believe the improvements we propose are applicable to other BGP implementations as well including XORP and Cisco IOS, as they have similar software architectures.

3.1 Quagga BGP

Quagga [33] is a routing protocol suite that is a fork from the GNU Zebra project [34]. It is designed as a collection of separate processes (called daemons) for each routing protocol in the suite. There is a common Zebra daemon, which provides the configuration utility for all protocols in the suite. The following subsections describe further details about the Quagga BGP software architecture and implementation.

3.1.1 BGP State Machine

As mentioned earlier, BGP is a path vector protocol where each router exchanges path information about all the addresses it knows of with its neighbors. A connection is

maintained with each active peer, and packets are exchanged by implementing the standard BGP state machine as specified in the RFC [52]. 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 other sessions. Figure 3.1 shows the different stages in the BGP FSM.

BGP protocol allows five different types of messages that are exchanged between the routers at any point of time. BGP Open message is used to establish a new connection with a neighboring router. BGP update packets are exchanged most often and contain the one of multiple network prefixes along with their corresponding next hop and path vectors. Update packets are also used to withdraw any path vectors that a router might have communicated earlier but no longer has an active connection on that path. Keep Alive messages are exchanged when there have not been any active route updates before the KeepAlive timer expires (usually 30-60 seconds). A BGP notification message is used to inform the peer about any corrupt packets.

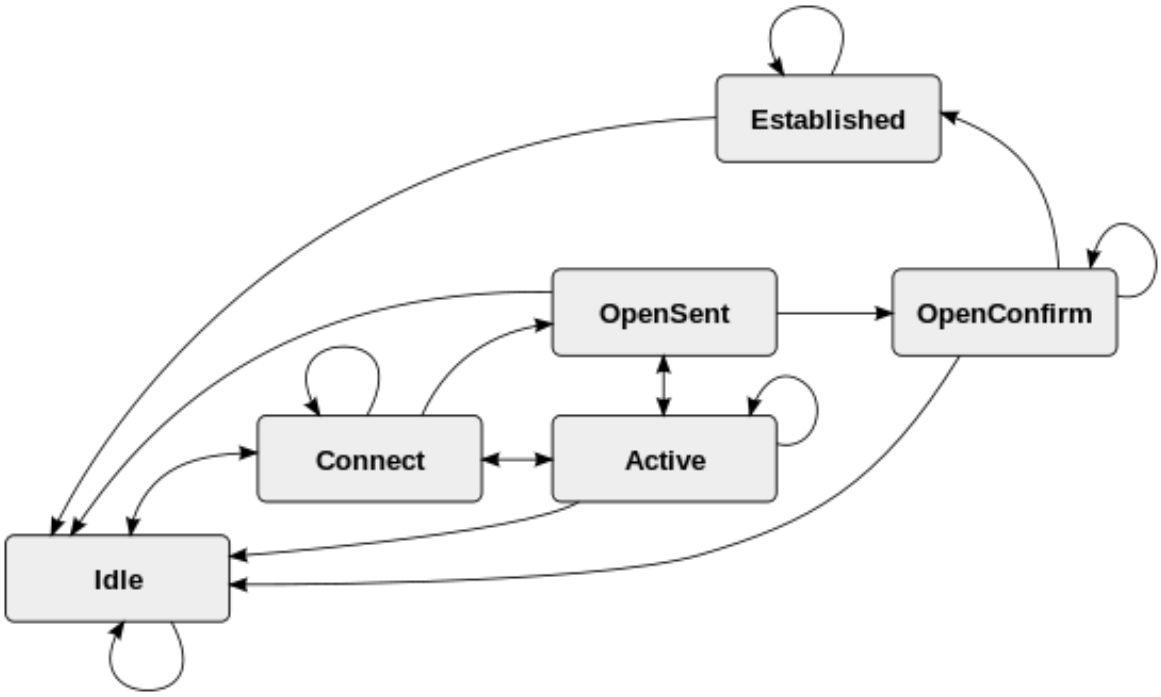

Figure 3.1: BGP FSM (Source: Wikipedia [59])

3.1.2 Connection Establishment

The BGP daemon always keeps a socket open in passive mode, listening for any connections. When a new router comes up, it tries to establish connection with all the neighboring routers that it is preconfigured for. After identifying the peer, a sequence of BGP Open and KeepAlive messages are exchanged in order to establish a connection. After successful connection establishment, the router gets a large number of updates from its peers to establish a network view of its own. Following this initial flooding of updates, the router

reaches a steady state where it only gets update messages for any network changes that are happening.

3.1.3 Routing and Lookup Tables

The routers exchange update messages with path vectors to share the routing information of different network addresses stored in their routing tables. The path vectors are called Network Layer Reachability Information (NLRI), and a router can receive and advertise multiple NLRI's simultaneously. On receiving an NLRI message, the router makes appropriate entries in the routing tables and advertises routes to its peers based on the policies it exchanged with the peer during the initialization phase.

There are a few different conceptual routing/lookup tables maintained inside each router. Each routing table entry consists of a destination network prefix, a path vector (a list of AS numbers), and the next hop AS information.

- Local Routing Information Base (RIB) – This is the master routing table that is maintained by the BGP router. It stores NLRI's for all known network prefixes.

- Adjacent Routing Information base – This is further broken down into Adj-RIB-In, which stores incoming NLRI's from a particular neighbor, and Adj-RIB-Out, which stores outgoing NLRI's. There are few different policy attributes that each incoming path vector should meet, before it is allowed entry into the Adj-RIB's.

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

When we mention conceptual lookup tables, it means that, in Quagga, there is only one physical entry maintained for each NLRI. The Local and ADJ RIB's store NLRI pointers that point to the same physical entry in the NLRI database. Figure 3.2 depicts the Quagga software architecture and shows the critical data structures.

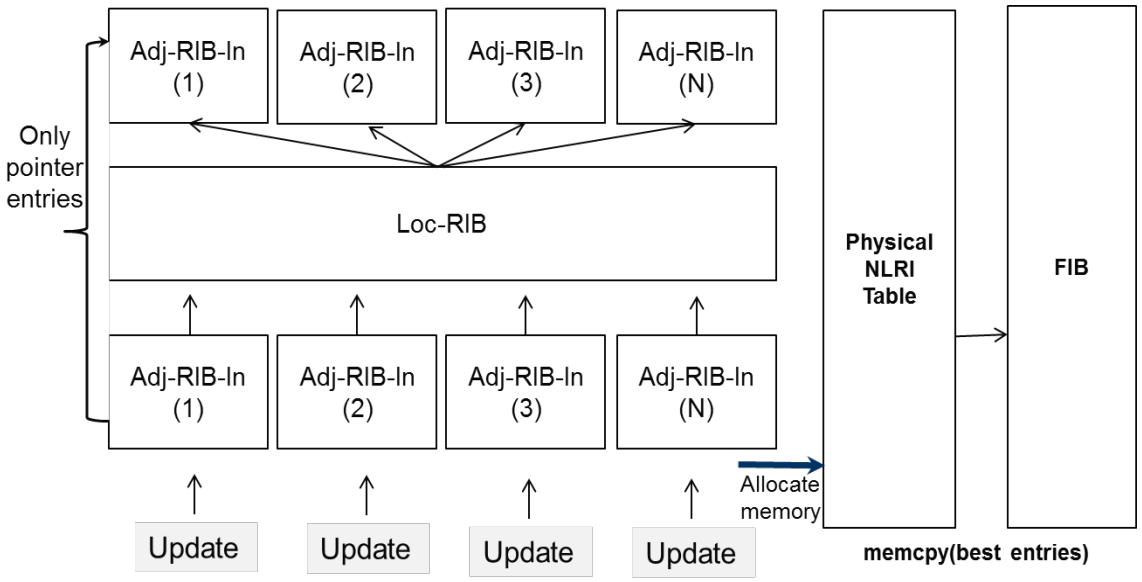

Figure 3.2: Routing and Lookup tables in Quagga

3.1.4 TCP communication and Stream Processing

Streams are byte buffers that are maintained for each peer session and hold the incoming and outgoing packets. In the BGP implementation, two streams buffers are maintained for each peer connection, one for the incoming packets from a peer session and the other for outgoing packets. Each buffer is sized to hold one maximum-sized BGP packet. When a new packet arrives from a neighbor on the corresponding TCP socket, it is read into the incoming stream buffer using TCP read. Subsequently, all packet parsing functionality is performed by reading data from this buffer. Similarly, any outgoing BGP packet is first composed in the outgoing stream buffer by appending different header and payload fields before it is pushed out using TCP write. Quagga routing suite provides a separate stream library API to handle these byte buffers, which are used by all the protocols in the routing suite. As we will see later, we propose some improvements to the stream buffering and processing functionality, which is done by making changes to the stream API.

3.1.5 Task Scheduling

Tasks in Quagga refer to the functions that need to be performed on any state transitions or timer expiry. Each task holds a pointer to a function that gets called when the corresponding event occurs or the associated timer expires. (Tasks are referred to as threads in the Quagga code but they are independent to the threads that we talk about in our multithreaded implementation.)

The implementation maintains five task queues: 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.

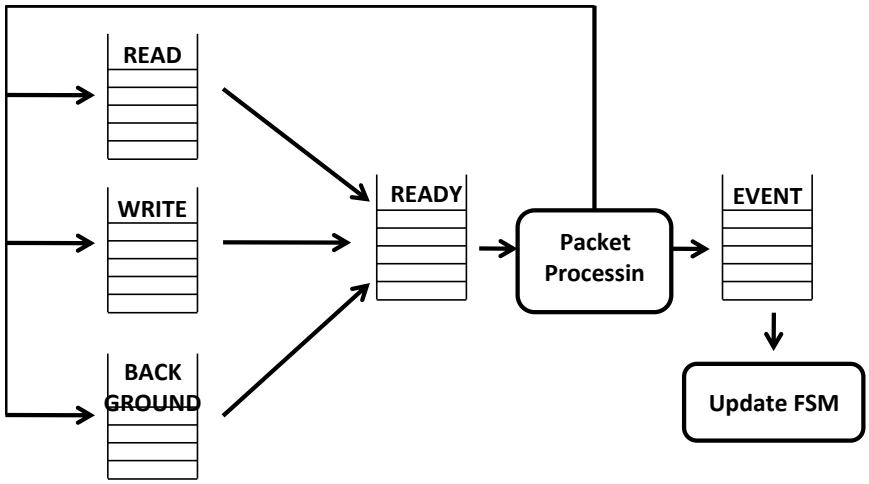

Figure 3.3: Task Scheduling in Quagga

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 queues and are executed one at a time. Event queue has priority over the ready queue. Figure 3.3 shows the flow of tasks between different queues in Quagga.

3.1.6 Packet Flow in Quagga BGP

After booting and completing the connection establishment with each of the configured peers (as described in section 3.1.3), the BGP router receives and sends route updates and withdraw messages. On receiving a route update, it is first moved to the Adj-in-RIB, from where each packet is filtered based on different BGP policies that are configured by the service provider. A packet that passes the filters goes through the processing phase, where its network prefix is looked up in the loc-RIB, and the NLRI entries are updated. Subsequently, the route advertisement functionality is performed, which determines which NLRI updates or withdraws need to be sent out to different peers. These packets are moved to the Adj-out-RIB. When the route advertisement timer for a particular peer connection expires, the corresponding ADJ-out-RIB is looked up and update/withdraw packets are composed and sent out. Figure 3.4 shows the overall packet flow through the software pipeline.

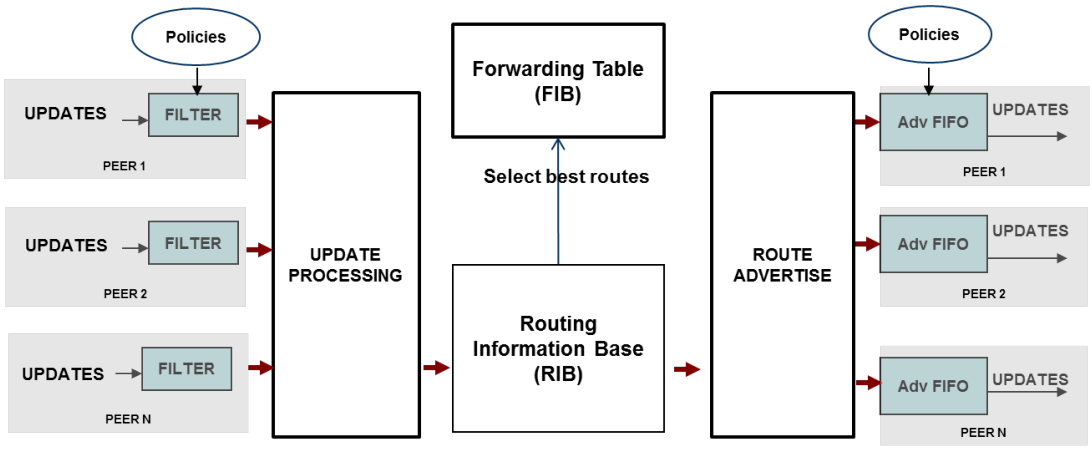

Figure 3.4: Packet Processing in Quagga

3.2 Peer Based Task Scheduling

Once the router has booted and established TCP connections with each neighboring node, each neighbor transfers a huge number of route updates, transferring all the relevant routing table information. On a real network, this would mean a router receiving updates corresponding to almost all the AS prefixes on the internet, which is around 10 MB of TCP data received within a few seconds at the router's ingress queue. Similarly, during packet surges, there are many update packets queued up at the incoming interface.

An interesting behavior of the BGP protocol is that data from all incoming peers is independent of each other. A separate TCP connection is maintained for each peer and the state transitions for one connection are independent from another. The first BGP parallelization scheme that we develop exploits this peer connection independence property to divide the work done by the router into multiple threads. As each thread operates on incoming data from a particular peer, we call this scheme Peer Based Task Scheduling (PBTS) [25].

3.2.1 Parallelization Approach

As the first step towards implementing PBTS, we identify the code sections from the sequential BGP implementation that perform the actual processing of incoming updates and form the most-executed code path of the application. In Quagga BGP, the unit of work is identified as a task (Section 3.1.5). For instance, reading a packet from the ingress and

processing it is classified as a _READ task, a state machine transition is identified as a _EVENT task, and all timers including housekeeping, route advertisements, and so forth, are identified as _TIMER tasks. Each task has an associated validating condition, which when true enables the task to be marked as _READY for execution. As an example, the validating condition for any timer is when the current wall time is greater than the timer expiry time. The validating condition for a _READ task is when the data becomes available at the network interface.

BGP Figure 3.5 shows the basic loop that the sequential BGP implementation iterates through, where it primarily does two things: a) Check the validating conditions of _READ, _EVENT and _TIMER tasks and flag them as _READY if the conditions are true, and b) execute associated functions for all _READY tasks.

```
while (task_fetch())     //Fetches tasks from ready queue
            task_call();

task_fetch() {
            Check for executable tasks in event queue;
            Check for executable tasks in read queue;

            select(FDSET); // 32bit FD-Set contains 1 bit
                              for each valid network connection

            for each bit in FDSET {
                If (bit is set)
                        mark the task as ready;
            }

            for each timer in timer_list {
                 If (current_time > timer_expiry_time)
                        mark the task as ready;
            }

            return a ready task;
}

thead_call()  {
            Execute the associated function;
            Update the connection FSM;
            if (timer_executed)
                        Reset timer to 0;
            if (read_executed)
                        schedule another read task;

}
```

Figure 3.5: Hotspot in BGP Sequential Code

PBTS scheme's primary focus is to modify this task scheduling functionality to enable execution of independent tasks in parallel. The POSIX thread library (pthreads) [19] have been used to implement all the multithreading and synchronization mechanisms. In PBTS, we associate each task with a peer and assign it to the corresponding thread for execution.

We split the main implementation loop into two segments: one is executed by the master thread and the other is executed by all other threads in PBTS, called peer threads. All threads

run on their dedicated cores. Figure 3.6 shows the code executed by the master thread and 3.7 shows the code executed by the peer threads.

```
while (task_fetch())    //Fetches tasks from ready queue
                task_call();

task_fetch() {

                //Listen to new connection requests
                BGP_listen(FD)  {
                                // Calls TCP_listen()
                }

                if (FD has data) {
                                // Mark the task as ready
                }


                for each timer in timer_list {
                     If (current_time > timer_expiry_time)
                                mark the task as ready;
                }

                return a ready task;
}

thead_call()  {

                if (new connection request) {
                        Exchange the connection establishment messages
                        Create a new pthread
                        Initiallize  peer structure and timers
                        Schedule a READ task in the _READ task queue

                }

                if (timer_ready) {
                        Execute associated function;
                        Reset timer to 0;
                }

}
```

Figure 3.6: PBTS Master Thread

```
while (task_fetch())    //Fetches tasks from ready queue
                task_call();

task_fetch() {
                Check for executable tasks in event queue;
                Check for executable tasks in read queue;

                read();   //Checks FD socket for new data

                If (FD has data)
                      Mark the read task as executable;

                for each timer in timer_list {
                     If (current_time > timer_expiry_time)
                                mark the timer task as ready;
                }

                return a ready task;
}

thead_call() {
                Execute the associated function;
                Update the connection FSM;
                if (timer_executed)
                                Reset timer to 0;
                if (read_executed)
                                schedule another read task;

}
```

Figure 3.7 PBTS Peer Threads

The thread definitions for PBTS implementation are as follows:

- Master-Thread - Common among all peers and handles
    - Execute all the housekeeping functions that are not associated with any peer

- Clearing routing tables,
- Checking socket buffers for new data and notifying peer threads, as well as maintaining file descriptor sets.
- Keep a TCP socket open in passive mode and listen for any new BGP connection requests. Whenever the router receives a connection request from a peer router, the master thread spawns a new peer thread.

- Peer-Thread - This thread maintains a TCP connection with the peer and runs its corresponding BGP state machine. This thread also maintains separate read, write, event, ready and background queues for the peer. Peer threads perform all the functionality related to a packet flow, including reading the packet from TCP stack into the stream buffer, parsing the header and payload information, applying different policies and finally doing a routing table lookup or modification if required. Whenever this thread encounters a task that does not belong to its own peer, it adds that task to the task queue of the master thread. The task assignment flow is shown in figure 3.8.

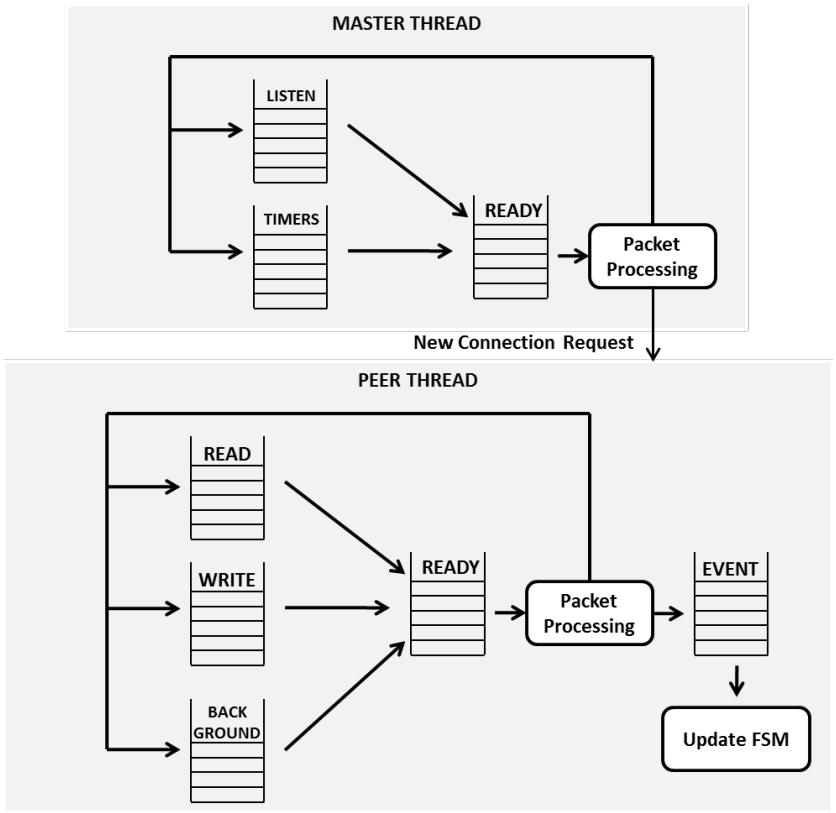

Figure 3.8: Task Assignment flow in PBTS

3.2.2 Synchronization

Lock variables are maintained in the multithreaded code to resolve contention among the threads for accessing key data structures. Primarily, there are three different types of lock variables that are maintained to prevent simultaneous access on the routing tables, peer specific byte buffers, and global file descriptor sets, respectively.

Routing Table Locks

Routing tables in Quagga are maintained as trees, with each prefix represented as a node in the tree. Two lock variables are defined per routing node: one for traversal when a thread just wants to read the tree without modifying any node, and the other for node modification when a new node needs to be added to the tree or contents of a node need to be modified. These two node specific lock variables enable maintaining a fine grain locking in the routing tables, allowing multiple threads to modify the tree as long as they are accessing different locations in the tree and do not interfere with each other.

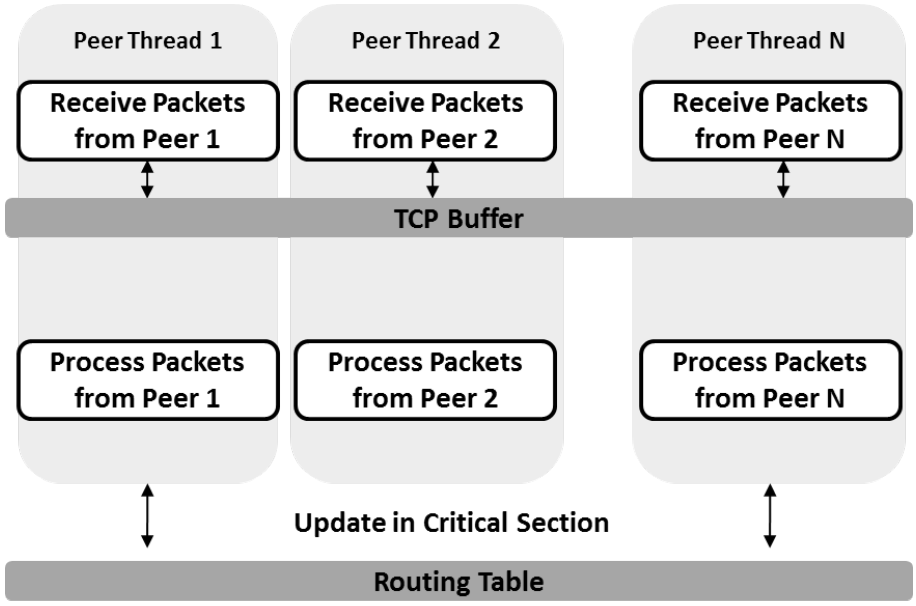

Figure 3.9: Packet flow in PBTS

File Descriptor Set lock

In order to read data from the underlying TCP buffer, independent sockets are maintained for each peering router. Select() function is used to wait for new data on the TCP buffer, and common read and write file descriptor sets are maintained to track new incoming packets. As the file descriptor sets are shared among different peers, the use of select logic necessitates modification of FD sets serially, to ensure that when select logic is called, no other threads is accessing the FD set. Hence a lock variable has to be acquired before accessing the FD set to know if there is new data for the particular peer on the TCP buffer.

When evaluating the performance with FDSET locks, we observed that there is significant contention on the lock variable, because each thread needs to access it before starting processing for a new packet. Hence an optimization is implemented to eliminate the need for this lock variable. Instead of maintaining a common file descriptor set (FDSET), separate file descriptors are maintained for each thread. This enables each thread to access the underlying TCP buffer independently of any other threads. Figure 3.10 depicts this optimization.

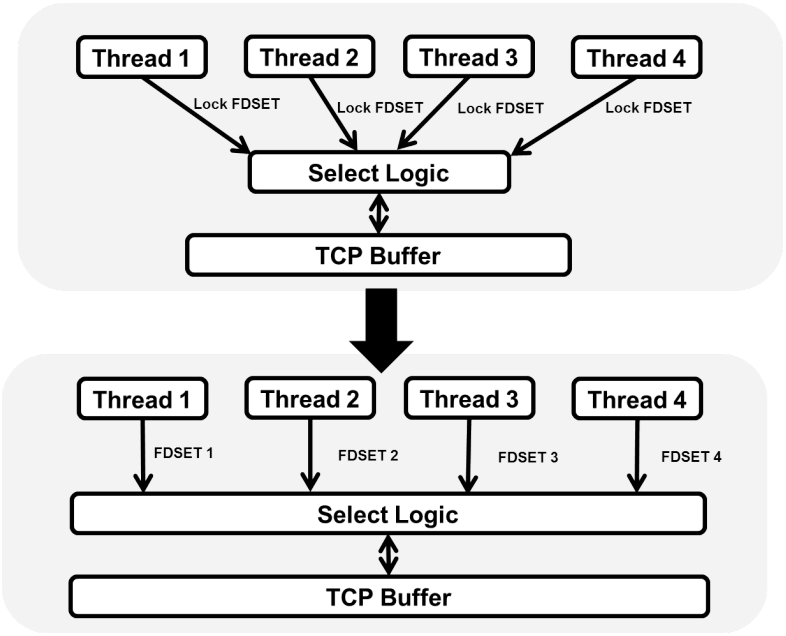

Figure 3.10: Eliminating common select logic

3.2.3 PBTS performance considerations

A major advantage of assigning threads on a per-peer basis is that all the data structures in the implementation are contention-free except for the routing table itself. As soon as the master thread establishes the connection and a read task is assigned, the threads becomes independent and there is no requirement to communicate with other threads. Moreover, this scheme provides an opportunity to associate a TCP flow and a separate TCP stack itself with each thread. This enables much lower latency when reading or writing packets to the network interface. More details about this are discussed in section 5.3.

However, a disadvantage of this scheme arises in cases when all the peer nodes do not communicate the same number of packets, and the incoming data flow is skewed towards only a few threads instead of being evenly distributed across all threads in the multicore processor. In these cases, the hardware is not efficiently utilized for packet processing. This disadvantage is mitigated by an alternate task scheduling mechanism called Dynamic Task Scheduling (DTS), which we discuss in the next section (3.3).

3.3 Dynamic Task Scheduling (DTS)

While the PBTS scheme spawns threads on each new connection request, DTS follows a different approach and spawns a fixed number of threads when the router boots up. The number of threads started is proportional to the number of cores on the multicore machine. This approach allows all the CPU cores to be utilized evenly for packet processing. In DTS, the master thread does the job of maintaining the _READ and _TIMER and _READY task queues for all peers. The actual functionality of executing the _READY tasks is moved to the other threads. Each thread has a separate _READY queue from where it picks up valid tasks and executes them. The master thread scans for new packets on the TCP buffer and performs timer management for all peer connections. When any _READ or _TIMER task becomes ready, it copies the ready task to _READY queue of a thread in a round robin fashion. This ensures load balancing on each thread.

As each thread can execute tasks belonging to any peer, this scheme thus decouples the number of threads from the number of peers. However, as 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 earlier, two streams are maintained for each peer to store the incoming and outgoing packets. When multiple threads operate on packets from 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 limits the peer data structures to be modified by a single thread at a time. Figure 3.11 shows packet flow in DTS and 3.12 shows the code being executed in the main loop of DTS. More details on several optimizations as well as detailed performance results for DTS are part of a separate work [14].

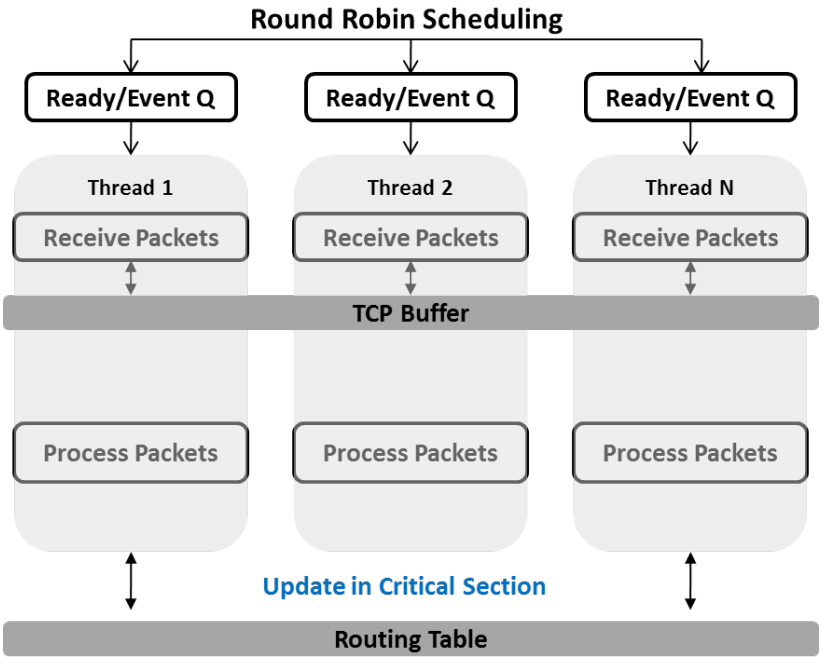

Figure 3.11: Dynamic Task Scheduling

MASTER THREAD

```
while (task_fetch())    //Fetches tasks from ready queue
                        task_call();

task_fetch() {
                        Check for executable tasks in event queue;
                        Check for executable tasks in read queue;

                        select(FDSET); // 32bit FD-Set contains 1 bit
                                              for each valid network connection

                        for each bit in FDSET {
                           If (bit is set)
                                              mark the task as ready;
                        }

                        for each timer in timer_list {
                             If (current_time > timer_expiry_time)
                                              mark the task as ready;
                        }

                        return a ready task;
}

thead_call()  {
                        thread_round_robin = (thread_round_robin +
1)%num_theads;
                        Add task to READY Queue [thread_round_robin);
                        if (timer_executed)
                                              Reset timer to 0;
                        if (read_executed)
                                              schedule another read task;

}
```

OTHER THREADS

```
While (1) {
         if (ready queue has data) {
                        Execute the associated function;
                        continue;
        }
else
                        wait();
}
```

Figure 3.12: Code for DTS Threads

Chapter 4: Performance Analysis

4.1 Simulation Environment

Current deployment of the Internet backbone has more than approximately 40,000 Autonomous Systems implementing BGP [24]. Therefore, a simulation environment to be used for evaluating performance of BGP should be representative of its actual deployment on the Internet, such that packets at each BGP node scale with the actual network topology and size. Since the focus of this work involves detailed computer architecture studies, we also need an environment that can run the complete BGP router code and model an actual multicore processor with shared memory. It should be able to run hardware threads and model threading overheads and data contention correctly. To the best of our knowledge, there is no single software or environment that supports doing detailed architectural simulations on a complete network topology, primarily because it is practically impossible given the size of the network that needs to be simulated. During the initial phase of our work, we developed a simulation environment that is both realistic and representative of the actual

BGP workloads, based on a two-step approach of generating functional traces and then using trace driven methodology to run detailed multicore simulations.

1. Generate a BGP packet trace at every autonomous system by running high-level network simulations with a sufficiently large number of nodes and a representative Internet topology. These simulations are functionally accurate but do not model any performance related issues.
2. Choose an arbitrary system from the above network simulation and run detailed architectural simulations (with appropriate CPU and Memory modeling) driven by packets from the trace generated in step 1.

Section 4.1.1 elaborates the packet trace generation methodology using network simulator, and section 4.1.2 details the GEMS [45] and Simics [44] based evaluation environment that we use for detailed multicore architecture simulations and analysis.

4.1.1 Generating BGP Traces

Network Topology

As the first step towards network simulation, we generate a network topology having 1000 Autonomous System nodes communicating with each other. We choose the BRITE Topology Generator [46], as it is allows different configuration options, including different link bandwidth and latencies and different degrees of connectivity at each node, and it provides a

convenient GUI-based interface for configuration. We generate a network with each node connected to 10-15 neighboring nodes.

Network Simulations

Once we have a representative network topology, the next step is to generate functionally accurate network simulations. We use Network Simulator (NS2) [54] for simulating the BGP network. NS2 requires a script that defines all the components in the network along with the connectivity information. It uses BGP++ as an agent for BGP simulations.

- BGP agent – NS2 does not have a built-in protocol simulator for BGP. BGP++ [17] is used as an NS2 agent that performs BGP simulation based on the given configuration. It is based on the same code used in Quagga routing software with modifications done in order to use the TCP/IP stack of NS2 for packet transmission and reception. BGP++ takes a configuration file as input, which contains the peer information and the prefixes to be advertised by a particular node, and generates a log for all the state machine and path information. It also generates a dump (in MRT format [6]) for all the packets exchanged. This tool was used to perform network simulation for several BGP nodes and generate corresponding packet dump files.

- BGP++ Configurator - BGP++ configurator [16] takes the network topology as an input and generates configuration files for BGP++.

Figure 4.1 shows steps involved in generating BGP packet trace using network simulation.

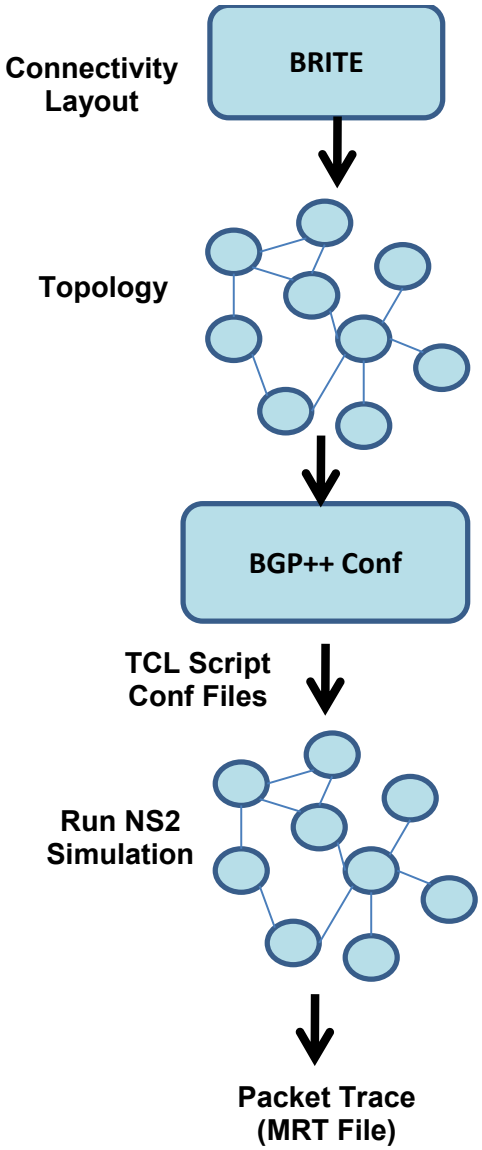

Figure 4.1: Network Simulation Setup for generating BGP packet trace.

4.1.2 Multicore Simulation Environment

Once BGP traces are generated, we choose a center node from the network that has 12 peer connections, and use the MRT-based packet trace collected at the ingress of this node. This packet trace file will be used to feed packets into the multicore model of the BGP processor. A Simics-based simulation environment is used to run the multicore network simulations. We initiate two virtual hardware systems in Simics, and connect them using a single Ethernet link. The first virtual machine runs the sequential or parallel version of the Quagga BGP daemon. The second virtual machine runs the BGP Simulation Client. This simulation client is a program we developed that reads BGP packets from the trace file and feeds it to the Quagga BGP daemon.

BGP Simulation Client

The BGP simulation client is a C program which imitates the functionality of several BGP peers and maintains TCP connectivity with the actual BGP daemon. The client interacts with the TCP/IP stack of the host machine and initiates a BGP session (one for each peer) with the BGP Daemon. Once the TCP session is established, it reads the packet dump file (the 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 its TCP session

with BGP Daemon. We have designed three different variants for this simulation client to run different types of experiments.

- The first version reads the packets from the packet trace MRT file in-order and sends them to the BGP daemon. This leads to a random order of packet dispatch on the different peer connections. In this case the BGP daemon receives packets in exactly the same order as what was generated using the network simulator.

- Second version of the client does a round robin read of the packets, picking one packet for each peer connection and then moving on to the next connection. This scheme allows a balanced packet load for each thread in the parallel BGP implementation. We further configure this version to allow sending N packets to each peer, instead of one, before moving onto the next connection.

- The third version of the simulation client is multi-threaded, where we spawn one thread for each peer and maintain one TCP connection per thread. Each thread reads packets corresponding to its peer and transmits them. This represents the most realistic scenario to what happens in real world, where each router operations independent of the others.

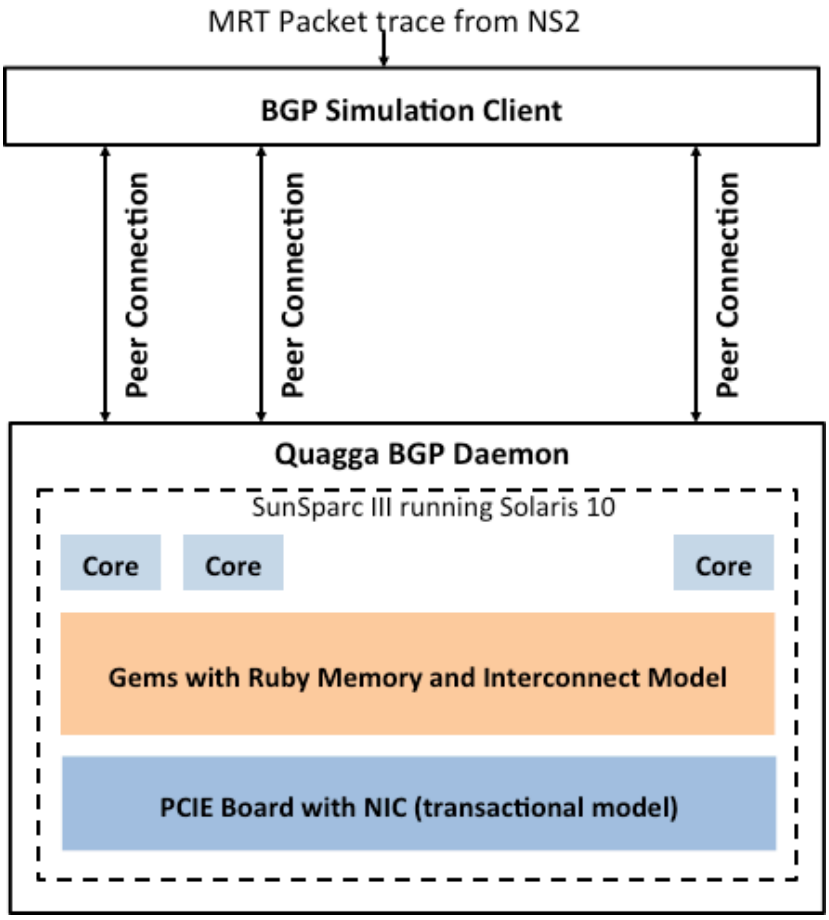

Figure 4.2: BGP Simulation Environment

BGP Daemon

BGP Daemon runs a parallel version of the Quagga routing software, which was described in the previous section. The virtual hardware system running the BGP daemon is set up to do detailed architectural analysis using Simics. It is configured to support different number of cores, and has a GEMS/Ruby memory model attached to it for simulating the memory

hierarchy. GEMS enables us to run different memory configurations including different cache sizes and coherency mechanisms. Figure 4.2 shows the connectivity between the BGP Daemon and the Simulation Client.

4.2 BGP Performance Analysis

In this section, we provide a detailed performance analysis of the sequential and multithreaded BGP implementations. We evaluate the performance under different multicore configurations and BGP network scenarios. All the simulations are performed on a system with MOSI coherence protocol and a private L2 cache. Processing times are measured in terms of Ruby cycles (Ruby is the memory and interconnect component of GEMS and a Ruby cycle corresponds to one cycle of the cache controller). Table 4.1 shows the generic configuration of the multicore processor used for all simulations unless specified otherwise. We use a generic compute server architecture for the processor model, since BGP routers still mostly run on general purpose architectures with little customization for control plane processing.

Table 4.1: Multicore Parameters

Processor	Ultrasparc III running Solaris 10
L1 Cache	Split I&D, 64 KB 4-way set associative, 1 cycle access latency, 64-byte line
L2 Cache	Private L2 cache per processor, (1 MB), 6 cycle access latency, 64-byte line
Interconnect	Hierarchical switch, 13 cycle link latency, 4 virtual channels, 4 buffers per virtual channel
Coherence	MOSI Protocol
Ethernet	Link latency – 1000 CPU cycles

4.2.1 Code modifications for Performance Analysis

In order to do detailed performance analysis and understand the performance impact of our parallel implementation under different scenarios, we make the following changes in the sequential and parallel implementations.

- Disable Timers – There are several timers that are run in the BGP protocol, such as KeepAlive timer that is triggered when there has been no communication with a peer for some amount of time. Since these timers are not performance critical and do not require much processing, we disable all such timers when running the performance simulations.

- Feeding packets as fast as possible – The primary metric to determine the effectiveness of the parallel BGP implementations is the throughput when processing route updates. Hence, we feed route update messages to the BGP daemon as fast as possible. To enable this, we ignore the timestamp values when reading packets from the MRT packet trace in the BGP simulation client, and allow the client to send the packets as fast as it can.

- Disabling Route Advertisements - Route advertisements to the peers happen periodically when the route advertisement timer expires. These operations are not in the critical path of the BGP router performance bottleneck, and so we disable the route advertisement timer in both sequential and parallel BGP implementations. The processing time that we report in the performance analysis sections is only for the

critical path which is the time from receiving an update at the ingress port of the router to the time is accesses or modifies the routing table.

- Trigger detailed architecture analysis after connection establishment – The connection establishment phase completes when the BGP simulation client has initiated connection requests for all the peering connections and the BGP daemon has responded with appropriate response messages. The detailed architectural simulator and performance stats collection is triggered only after all BGP connection establishment messages are complete.

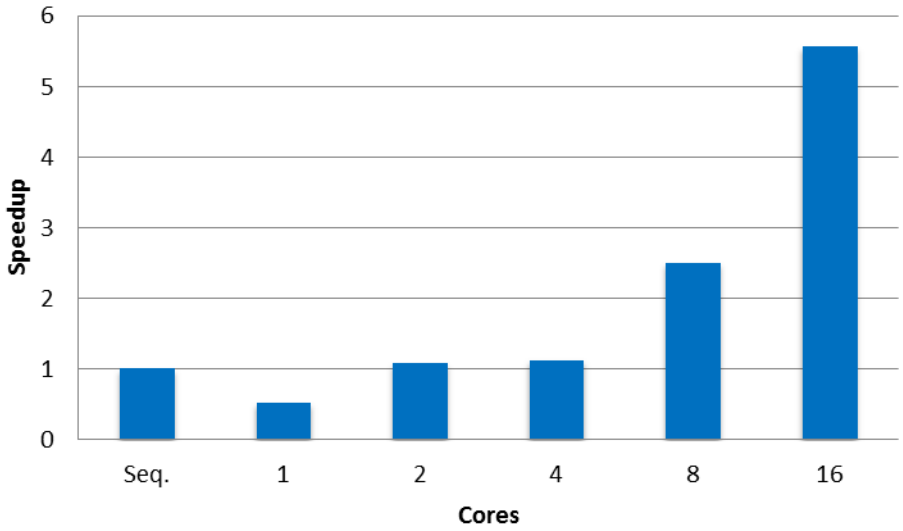

Figure 4.3: Speedup on different number of cores for PBTS

4.2.2 Parallel Application Speedup

Figure 4.3 shows the speedup achieved using the PBTS scheme on different numbers of cores. PBTS gives a speedup of over 5.5x when run on 16 cores. The best performance for both parallel implementations is achieved when one thread is running on each core. Increasing the threads/core ratio to more than one results in thread scheduling conflicts, leading to significant slowdown compared to threads/core ratio of one. The primary reason for this behavior is because the CPU utilization for each core is close to 100% even when a single thread is running on each core. Thus, there is not much scope for more utilization of system resources for achieving performance improvement. Having more than one thread per core also leads to cache pollution, thereby increasing the memory access times. An important point to note here is that in our simulations, the BGP node in consideration has 12 peers; therefore, only 12 threads are spawned by the system and 4 cores remain unused. Hence, the speedup numbers for 16 cores only use 12 cores for speeding up the application. We provide all the subsequent analysis with a total of 12 threads and the threads/core ratio equal to one for both DTS and PBTS. The performance of the multi-threaded code on one core is slower than the sequential code, as expected. The main reason for this slowdown is the thread scheduling and synchronization overhead. Further analysis on threading overheads is done in the next subsection.

4.2.3 Profiling Sequential and Parallel Implementations

To analyze performance, we divide the basic functionality of a BGP router into the following:

- Connection Establishment: This happens during the initialization phase when a router establishes connection with its peers. Since this happens only once when the router boots up, we don't analyze the time taken by the router in the initialization phase.
- TCP Polling: Time it takes for the router to probe the underlying TCP Buffer for new packets.
- Task Queue Handling: This involves moving tasks from Read and Write queue to ready queues when the router receive update messages and when it needs to advertise routers to peers.
- Packet Processing: This is the main part that involves the entire packet processing functionality of the router. This could be further divided into six functions:

  1, 2) Header Read and Header Parse for processing the header of the BGP packet. The read part involves reading the packet from TCP buffer.

  3, 4) Payload Read and Payload Parse involve reading and parsing the variable length payload of the packet. This part also includes applying different BGP policies to the packet before any changes are made to the routing table.

  5) Routing Table Update involves extracting network prefixes from the payload and performing route updates/withdrawals from the routing table.

6) Housekeeping covers other miscellaneous functionality including checking the peer status and validity of the packet before starting to process it.

Packet processing for BGP involves processing incoming packets and generating BGP route advertisement messages for peer routers. Since the route advertisements are done periodically when a timer expires, they do not form the critical path of packet processing. Hence, we provide a detailed analysis of only the incoming packet processing part. Figure 4.4 shows the time taken by different parts of the packet processing functionality for the sequential BGP implementation. Approximately 70% of the time is spent on the packet parsing and routing table updates. The TCP buffers reads (Header and Packet read) add up to 26% of the total time taken. Housekeeping overhead for the packet is almost negligible at 3%.

Time taken for routing table updates is dependent on the size of the routing table and this number could increase for larger routing tables. However, the routing table is organized as a binary tree and doubling the size of the table just increases the number of iterations in the search loop by one. Hence, unless there is a significant increase in the tables (more than 2-3 orders of magnitude), the percentage time of table update remains roughly the same. This is further elaborated through an experimentation performed in Section 4.2.5. Profiling the serial application clearly identifies packet parsing and table update functionality as the candidates that need to be sped up in order to achieve significant performance improvement.

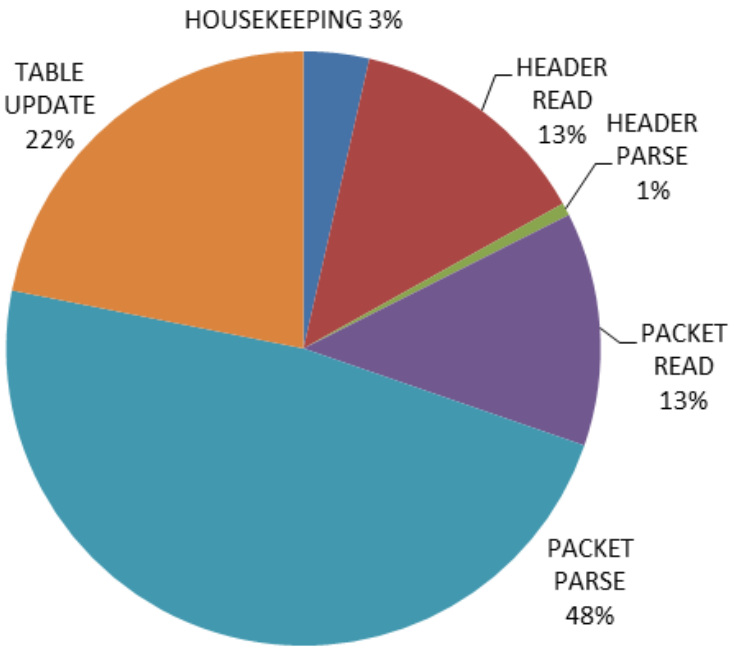

Figure 4.4: Time taken in different functions for Sequential Implementation

Figure 4.5 shows the speedup achieved for different packet processing functions for PBTS implementation. In accordance with Amdahl's law, the speedup is not linear, because the underlying TCP stack on which BGP is implemented is sequential. Thus, there is a constant part to the execution time that can be attributed to reading packets from the TCP buffer. Also, lock contention and task scheduling account for additional overhead, which leads to sub linear speedup.

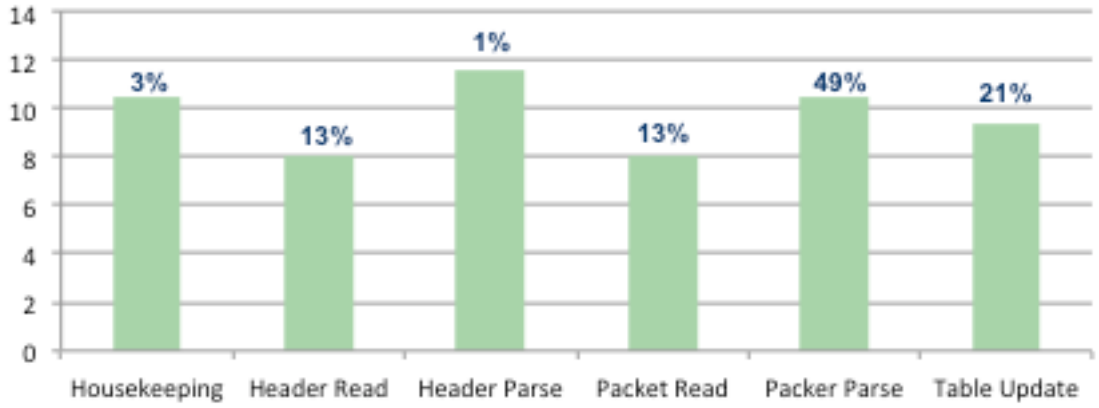

Figure 4.5: Speedup achieved for different functions using PBTS

Table 4.2 shows the average number of ruby cycles spent in each function when processing a single BGP update packet. The last column shows the speedup achieved on each function by the parallel implementations. The ideal speedup for any function is 12, because 12 threads may be performing the function simultaneously. As can be seen, the table update and packet parsing functions are sped up by 10.4 and 8.0 times respectively. The difference between ideal and actual speedup can be attributed to the synchronization overhead in the routing tables and other data structures.

Table 4.2: Average number of ruby cycles and speedup achieved per packet for different functions

Function	Sequential	PBTS	PBTS Speedup
Table Update	272.8	26.2	10.4
Packet Parse	1123.1	139.7	8.0
Header Parse	58.9	5.1	11.6
Packet Read	1087.2	135.3	8.0
Header Read	4176.6	397.7	10.5
House-keeping	1857.0	198.5	9.4

It is important to note that PBTS is designed to speed up the update processing in BGP, which is in the critical path towards improving packet throughput and convergence time. The update processing functionality is sped up by 9.5 times in PBTS. However, this speedup is not reflected in the overall execution cycles for PBTS, mainly due to the following reasons:

1. The Quagga BGP code collects stats and timestamps for each packet, which internally does a system call, causing the application thread to context switch and consuming significant number of CPU cycles.
2. In our simulations, we evaluate time for processing 1000 updates. When multiple threads are running they receive packets in an arbitrary order where some threads might start processing packets much earlier than others. Hence, the start and end

times for each thread don't match, causing a higher total run time. This is a simulation artifact and has nothing to do with any threading overhead in PBTS.

Figure 4.6 shows the parallelization overhead when single thread performance is considered. The overhead in header and packet read functions is due to accessing the TCP buffer. In the parsing and table update functions, overhead is due to the lock instructions.

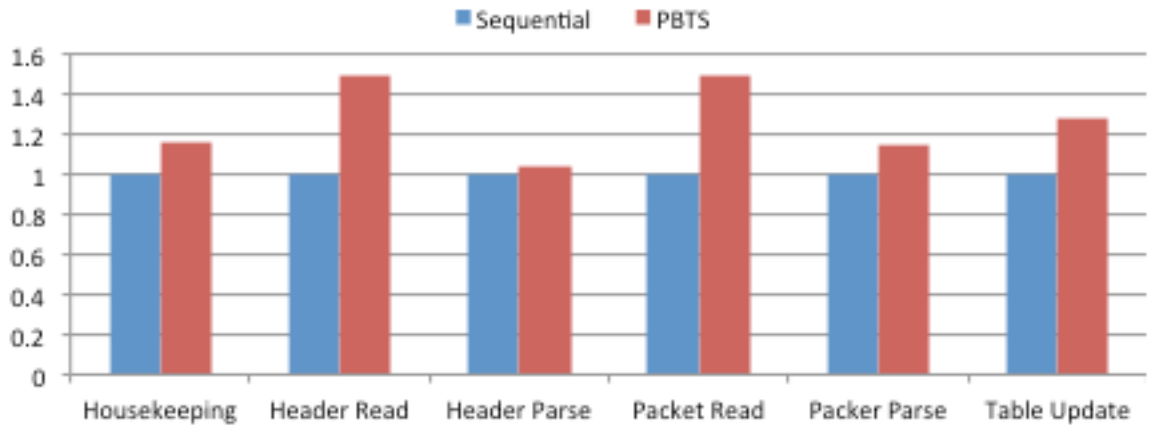

Figure 4.6: Parallelization overhead in single thread performance of PBTS

4.2.4 Traffic Behavior

PBTS performance depends on the activity factor from different peers. In this section, we provide an analysis of PBTS performance under different peer burstiness levels. What we mean by level of burstiness is the number of update messages received from a particular peer before the next set of messages is received from another peer. It can also be defined as the number of peers that are sending messages simultaneously. Higher burstiness level means, the higher the load on a particular core. Table 4.3 shows the different traffic mixes generated by the BGP simulation client.

Table 4.3 Different packet mixes generated by the BGP simulation client

Mix	Description
Best	All peers sending packets, 1 packet at a time.
Mix1	All peers sending packets, in a realistic order (generate from Network Simulation). Also called normal mix.
Mix2	4 peers sending packets at a time.
Mix3	2 Peers sending packets at a time.
Worst	Only 1 peer sending packet at a time.

Figure 4.7 shows the performance of PBTS under these different traffic mixes. As can be seen, the performance of PBTS degrades from best-case scenario when there is perfect load balance, to worst-case scenario when there is only one thread active at a time, thereby underutilizing the multicore system. As observed through our 1000 node BGP network simulations using NS2, the incoming packet traffic is a good mix of packets from different peers. Therefore, the worst-case scenario would be rarely observed and the performance does not need to be optimized for different burstiness levels. The speedup that we calculate for the parallel application is with Mix1, which is a realistic packet trace at an NS2 node.

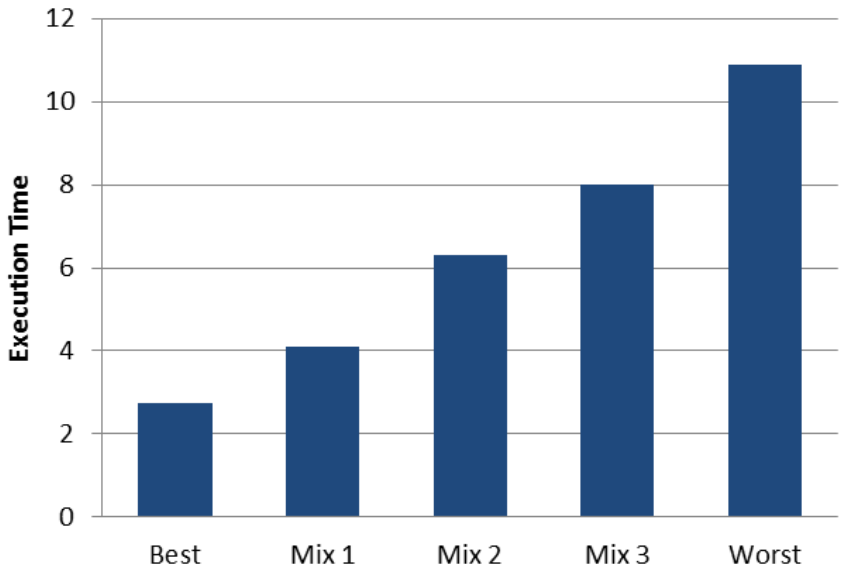

Figure 4.7: Execution Times (in million Ruby cycles) for different traffic patterns

4.2.5 Larger Traces and Routing Tables

We run some simulations on larger packet trace sizes from NS2 with each node advertising four times the original number of IPv4 prefixes, to understand the impact of routing table size on the packet processing times. Figure 4.8 shows the execution time for intervals of 500 updates, which remains almost constant as the simulation progresses. The reason for this lies in the way routing tables are organized in Quagga. Routing tables are structured as a tree, and we observed that having four times the number of prefixes increases the route traversal depth by only two or three levels in most cases. This has only a marginal impact on the total execution time in our simulations. In the real BGP deployment on the Internet, growing routing table size is not as big a problem when compared to the growth in the number of update messages being processed in the router [30]. In our work, we focus on speeding the routing table accesses by doing multiple accesses in parallel, but do not deal with the table lookup and traversal time by a single thread.

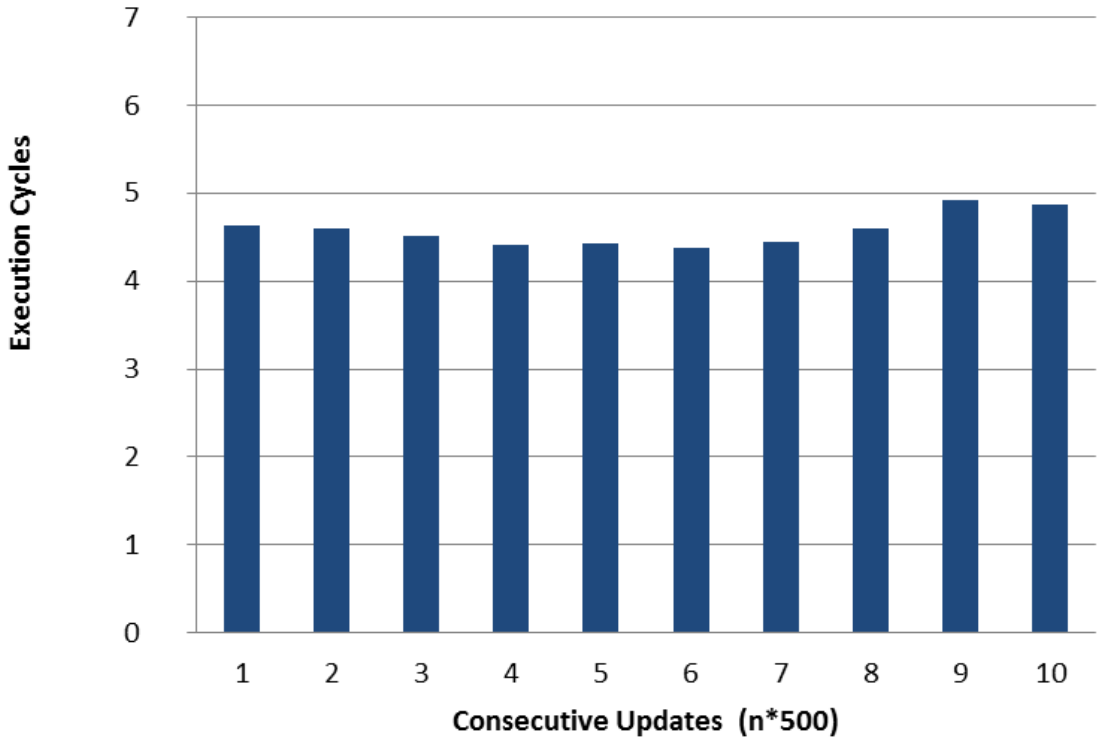

Figure 4.8: Execution Times for 500 consecutive updates on 16 cores

4.2.6 PBTS vs. DTS

Figure 4.9 shows a comparison of the performance of the PBTS and DTS schemes on different numbers of cores. As can be seen, DTS performs marginally better than PBTS. This can be attributed to the fact that DTS does a better job at load balancing because the task assignment to threads is independent of the peer. Hence, in cases where many packets arrive from the same peer, DTS would perform better in comparison to PBTS. On the other hand, for DTS to achieve better load balancing, it requires some complex optimizations and

overheads for maintaining thread based streams and dedicated task queues per thread, which result in execution overhead. Due to the pros and cons of each scheme, we see that the performance numbers of PBTS and DTS only differ marginally on 8 or 16 cores.

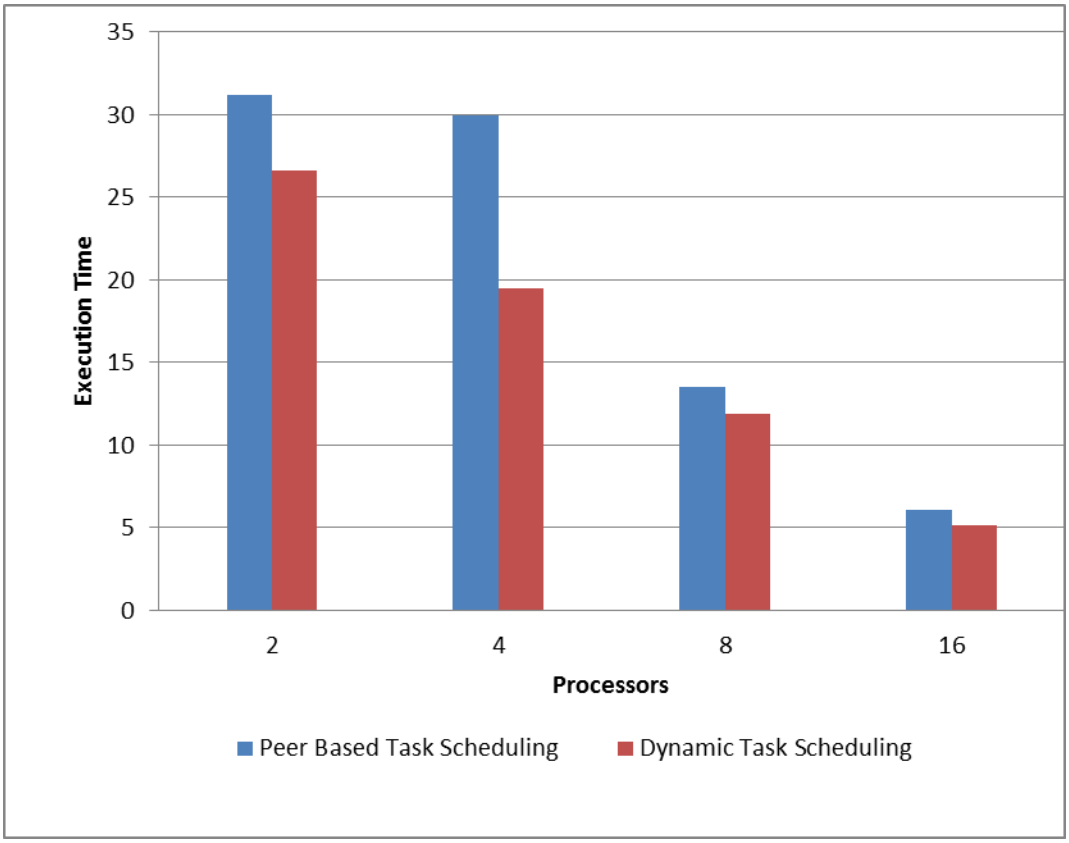

Figure 4.9: Comparison between execution times for Peer Based Task Scheduling and Dynamic Task Scheduling Schemes

Chapter 5: Multicore Architecture for BGP

This chapter dives into the performance of the underlying multicore processor to investigate possibilities for further speeding up the parallel BGP implementations. Our architectural analysis primarily focuses on three broad categories: a) Evaluating different configurations available in off-the-shelf multicore processes, such as different cache and memory sizes, etc., b) Customizing the multicore architecture for routing protocols by identifying new features that can be added to the processor to improve BGP performance, and c) Doing a literature survey of contemporary techniques that improve the performance of the network subsystem on multicore processors, including the TCP stack, and identifying relevant techniques which have the potential to remove network subsystem related bottlenecks in the parallel BGP implementations. Finally, we conclude that there are some basic architectural bottlenecks that limit the maximum speed up achieved by software alone using off the shelf multicore architectures. In order to get further performance improvements, the network processor architecture needs to be customized to better support network routing protocol workloads.

Section 5.1 evaluates impact of different memory configurations on sequential and parallel BGP performance. We propose adding a new feature in the architecture for improving the header and packet parsing functionality, which was identified as the biggest hotspot during the performance analysis in the previous chapter. The implementation details and evaluation of this new feature that we call 'Steam Hardware Assist' is discussed in Section 5.2. Section 5.3 provides a literature survey of several network subsystem and TCP optimization techniques, which can potentially speed up the header and packet read functions that form the second biggest hotspot in PBTS.

5.1 Memory Analysis

We analyze the memory behavior of BGP implementations to study the impact of larger cache sizes on the application performance. A general concern when running any application on multicore for performance improvement is that the speedup achieved may mostly be attributed to the increased memory size. To address this, we run the sequential application on varying cache sizes such that the single core system provides memory equivalent to the total memory provided by a multicore with 16 cores. Figure 5.1 shows the execution time of BGP with varying L1 and L2 cache sizes. In all three cases (sequential, PBTS, and DTS), the BGP application is not sensitive to L1 cache size variation. The working set size for the sequential BGP implementation is around 1 to 2MB for both the sequential and parallel implementations. As we can see from the L2 size variation charts, the performance of sequential BGP saturates at 2MB and at around 256K (per L2 cache bank) for PBTS and

DTS. As PBTS and DTS are running on 16 cores, the total L2 size thus amounts to 256K*16 = 4MB. The parallel implementations have a little larger working set, as expected, because of the additional data structures that were introduced in the PBTS and DTS for achieving performance speedup.

The L2 cache size required by BGP is significantly lower than the L2 cache size that is generally provided by most off the shelf multicore machines. The low working set size could be attributed to the fact that most data structures used in the implementation are allocated dynamically and therefore 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 data sets. This analysis concludes that the L2 cache area of the multicore processors could be traded off with adding some other on-chip hardware acceleration mechanisms, which may help to achieve better performance improvement, rather than having large L2 caches.

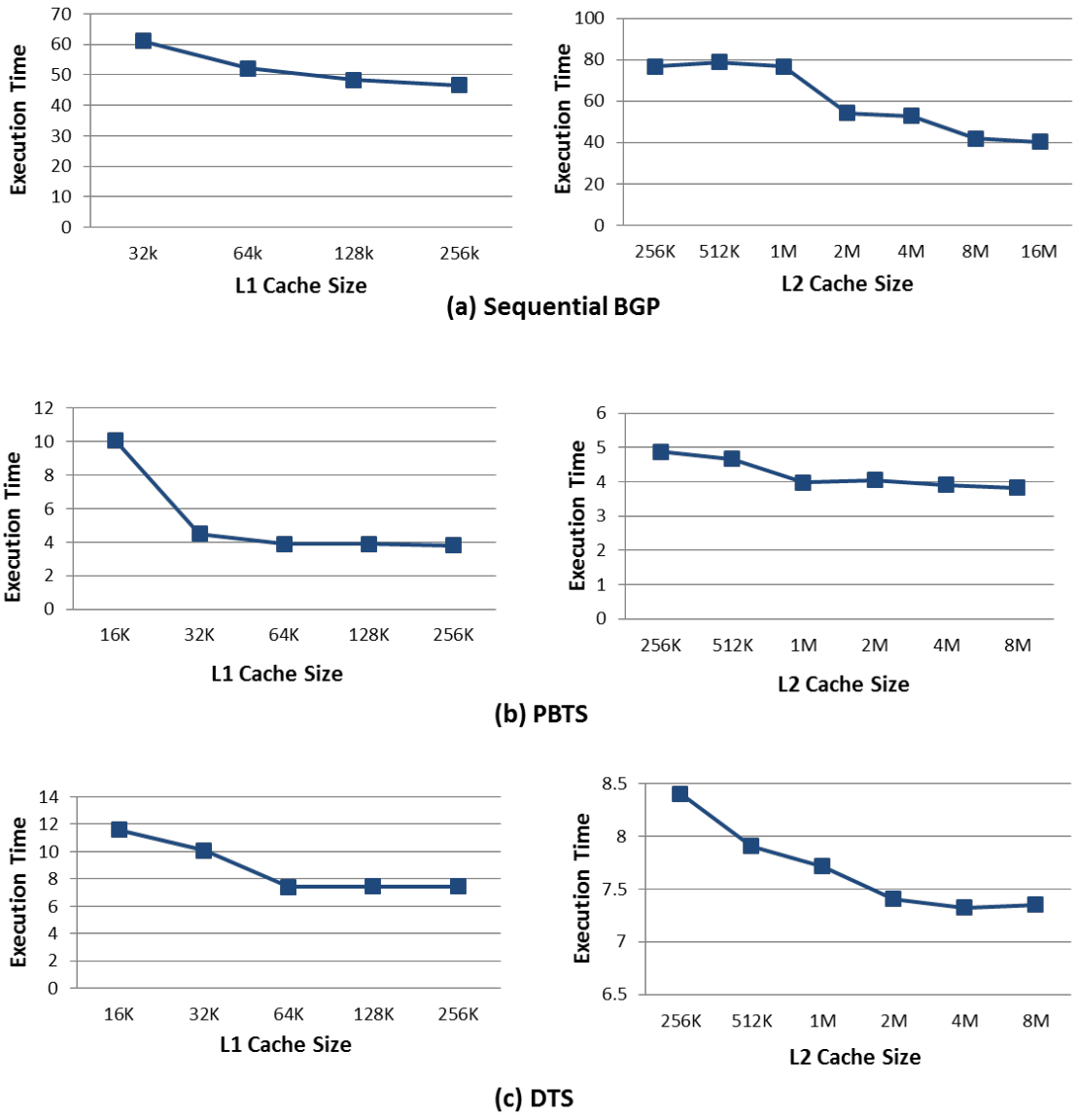

Figure 5.1: Execution time with varying L1 and L2 cache sizes for sequential implementation

5.2 Hardware Assist for Fast Stream Access

Packet and Header parsing forms a major part of the BGP execution time, taking as much as 47% of the execution time of the PBTS implementation. Inside packet parsing, most of the cycles are spent in reading data from the stream and processing it. In order to remove this bottleneck, we perform a limit study and propose a mechanism to speedup stream accesses called Stream Hardware Assist (SHA).

During the Header/Packet read and parse functions, the streams are written to and read from only once. In a processor with conventional memory hierarchy, when a thread tries to access stream data, it would cause a cache miss and request data from the memory. Subsequently, the data would be used once and evicted, leading to suboptimal performance of the L1 and L2 caches. On the other hand, if a dedicated hardware memory (working in parallel to an L1 cache) is assigned to each thread for exclusively storing the streams, the total number of stream access cycles can be significantly reduced. The latter scheme is the rationale behind SHA.

When a new packet arrives at the Ethernet interface, it can directly be copied to the SHA. Once the data is read, it can be preemptively evicted to make way for the next packet. This structure is different from a cache because caches are generally used for data that needs to be accessed repeatedly. In our case, stream data just needs to be read once and can be discarded after that. This structure is also different from a prefetch buffer [35], because in case of

prefetching, data is usually accessed from memory, whereas in case of stream hardware, the data can be directly accessed from the network interface card as discussed in the next subsection.

In the BGP implementations, the data structure used for storing ingress and egress packets are 1KB in size (to hold the maximum BGP packet size). Hence, the size for the dedicated stream buffer per thread should be 4KB. For such a small memory, we can safely assume single cycle access latency. We analyze the impact of stream buffers on PBTS and DTS performance by assigning single cycle latency for all stream accesses that happen from the packet parsing function. The simulations are run with normal latency for all other memory access instructions.

In order to estimate the performance of this scheme, we record all the stream accesses that are made by each thread in the parallel BGP implementation. This is achieved by instrumenting the BGP code to notify Simics each time a stream access is made. Subsequently, the memory accesses occurring within the instrumented code are replaced with stream hardware assist access latency. This hack does not prevent all the stream accesses from going to the memory system and hence does not include the performance improvement we would get by the freed up L1 and L2 cache space from stream addresses, which can be used to store other frequently-accessed data structures.

We analyze the impact of stream buffers on PBTS and DTS performance by assigning single cycle latency for all stream accesses that happen from the packet parsing function. The simulations are run with normal latency for all other memory access instructions. Stream accesses are 37.2% of the total packet parsing instructions and 17.4% of the overall instructions in the sequential implementation. (Packet parsing is 47% of the overall execution cycles.) Thus, speeding up stream access has a significant impact on overall execution time, as can be seen in table 5.1, which shows the speedup achieved by using hardware assist for fast stream accesses on PBTS and DTS. Stream hardware assist gives up to 5.85% speedup on PBTS and 6.67% on DTS.

To enable stream hardware assists in the real processors, support is required at the compiler and instruction set levels. The stream access instructions in the stream library need to be replaced with special instructions for accesses data from a different memory. As the stream accesses are made from a single stream library in the Quagga protocol suite, this requires minimal effort from the software perspective.

Table 5.1: Potential speedups using Stream Hardware Assist

Function	Speedup (PBTS)	Speedup (DTS)
Stream accesses	31%	38.5%
Packet parsing	13%	14.21%
Overall	5.85 %	6.67%

Hardware modeling of this technique involves two steps:

- Adding a cache-like data structure close to the core in parallel to the L1 cache, to which all the accesses to stream addresses are redirected.
- Modifying the TCP/IP stack to redirect all the incoming packets to the stream buffer. This can be achieved either by a) modifying the TCP functionality to copy data from the TCP buffer to Stream buffer when a read function is called from the BGP application, or b) Implementing a NIC with the capability to DMA the data directly into the stream buffers (discussed in the next section). The latter approach has less overhead and better performance compared to the former. Further discussion on placing the Stream Hardware Assist in hardware is done in Section 5.4.

Since the data in these structures should come directly from the NIC via DMA and not from the memory system, it would require changes to the NIC driver. The default NIC in Simics is a transaction-only model and doesn't support a full-fledged NIC driver, which is usually proprietary to the NIC provider. Although Simics does support manual NIC driver modeling, implementing that requires expertise beyond our scope.

5.3 Improving TCP Read performance

Almost 26% of execution cycles in PBTS and DTS are spent in TCP read that copies the data from TCP buffer to the stream data structure of each thread. The data is first copied from the network interface card to the TCP buffer and then from the TCP buffer to the input stream data structure. (To clarify, the stream data structure that we refer here pertains to the buffer that is allocated by the BGP software to store packets, and is different from the stream hardware assist that we proposed in the last section.). In this section, we dive deeper into this problem and discuss mechanisms to improve the header and packet read performance.

5.3.1 TCP Bottlenecks in Parallel BGP Implementation

The TCP bottlenecks in the PBTS scheme are twofold,

- The first issue is the high latency for transferring a packet from the NIC interface to the application buffer. The high latency comes from the fact that there are two sets of memory copies happening for the same data before it is used by the application. When an execution thread completes processing one packet, it wastes CPU cycles waiting for another packet to arrive, thus causing inefficiency.
- The second issue arises because only a single thread can access the TCP/IP stack at a time, creating a serialization point every time a TCP read or write is called from any thread. Moreover, the current simulation environment limits just using a single network interface for communicating with all the peer nodes. However, in a realistic scenario, a BGP router has multiple network interfaces to communicate with the

peers, so they can have incoming data from multiple interfaces at the same time requiring the TCP stack to be more efficient to handle multiple threads.

There has been a lot of research on optimizing TCP performance at the system level. Most research can be broadly classified in the following categories:

1) Reducing or completely eliminating the memory copy operations that cause high latency in the TCP stack. This also includes work on forwarding the data directly from the NIC to the application buffers.
2) Optimizing the TCP stack for multithreaded applications thereby removing any serialization when each OS thread/process is communicating with a different socket.
3) Alternate NIC implementations that address both of the above issues but with a different hardware implementation or moving the packet handling part of the NIC to software.

The following subsections discuss contemporary research work in each of these three areas and discuss how parallel BGP implementations can benefit from them to remove TCP related performance bottlenecks and improve overall performance.

5.3.2 Reducing TCP Read latency

Conventional TCP/IP communication incurs a high cost to copy data between kernel buffers and user process virtual memory. The primary reason for having a kernel space memory buffer is that the rate of outgoing TCP packets is different from the rate at which the

application generates the data, and this necessitates a need for introducing an intermediate buffer for rate matching. On the receiving side, two levels of copy allow OS to place the incoming data at a virtual address that is specified by the application. This situation has motivated development of techniques to reduce or eliminate data copying, making the TCP stack more efficient. Much work has been done to eliminate the double copies in the TCP/IP stack of different operating systems including Solaris [11] and Linux [56], as well as evaluation on OpenBSD [55]. To avoid the requirement of additional buffering in the kernel space, most zero-copy schemes involve complex interactions between the networking hardware, operating system and application software, including designing dedicated OS buffering schemes and application programming interfaces.

The most common zero-copy solution is based on page remapping and copy-on-write techniques [9, 10]. These schemes make use of variable sized network buffers called mbuf's that are used by conventional FreeBSD. Linked chains of mbufs are used to pass packet data between different levels of the TCP stack [10]. All packet data is transferred from the user space to the network driver – and the other way round – by reference, and making actual copies is avoided. Page remapping is done to change buffer access or address bindings without copying data to a new buffer. On the sender, the application may overwrite its send buffer before a packet is actually transmitted, in which case the kernel marks pages with pending transmits as copy-on-write and write access to the pending copy is disabled.

Chase et al. [10] do a detailed investigation of various schemes available in literature that improve TCP latency on sending and receiving hosts. They provide quantitative analysis of the potential improvements with these schemes on an experimental network setup. In addition to zero-copy, checksum offloading is another popular technique that helps reduce the TCP send and receive latencies. Since checksum is calculated on the entire TCP packet, including the header and payload, to detect packet corruption and is calculated on both sending and receiving sides. It requires all packet data to be loaded through the memory hierarchy to perform the checksum add operations. This introduces a high latency before the memory copy operation. Combining the checksum operation with another operation that touches all bytes of the packet can eliminate this cost. Since the packet passes through the DMA interface when being transferred between the NIC and host memory, checksum computation can be added to the DMA interface with minimal extra cost.

Our simulation environment uses the off-the-shelf Solaris OS stack and NIC hardware model. As the zero-copy schemes involve making modifications to the OS stack, as well as the NIC firmware, to support page-aligned packet boundaries among other things, we are not able to study the actual performance impact of these schemes on BGP implementations. As seen in performance analysis of sequential Quagga implementation, the BGP header and packet read functions take approximately ~1100 CPU cycles each to perform TCP reads. Using a zero-copy TCP scheme can potentially reduce the read latency overhead significantly and improve the performance of both sequential and parallel implementations.

5.3.3 TCP for Multithreaded Applications

Another TCP bottleneck that we encounter in PBTS is when multiple threads try to perform TCP reads at the same time. This issue arises because there is only a single TCP stack that each thread tries to access, and hence it becomes a serialization point. As shown in Figure 4.5, processor cycles spent performing the header read function increased by 90% in PBTS compared to the sequential implementation. Similarly, cycles spent in packet read function increased by almost 45%. This poses a big scalability issue when the number of threads is increased. There has been some work done on improving the performance of TCP for multi-threaded applications by instantiating multiple TCP stacks and adding packet classification features in the NIC. These schemes tend to give performance improvements when each TCP connection is assigned to only one thread. Since PBTS complies with this requirement, it can potentially benefit from these TCP optimizations.

Lemoine et al. [43] propose a new parallel network subsystem that processes incoming TCP protocol packets over a single interface in parallel. They implement a packet classifier in the NIC that interprets which core the incoming packet is destined for and puts that packet in that core's receive queue. There is one receive queue per core implemented in the NIC driver, from which the incoming packet is copied to the application buffer. Packets from one connection are processed only by one core, and there is no parallelism within a single connection, which means connection data can be made private, which reduces cache contention and need for locks, thus yielding better performance than packet-level parallelism.

To make their system robust against bursty incoming traffic, they use a Receive interrupt (RINT) coalescing scheme called NAPI [53] where an interrupt is generated for a group of packets instead of one-per-packet, and its processing overhead is reduced. Their software prototype is based on modifications to the Linux Kernel and the Myrinet NIC Firmware and drivers. (Myrinet is a LAN system designed as an interconnect for compute clusters [7].) Their implementation, which they call KNET, gives up to 34% speedup compared to NAPI (single network stack) in the best-case scenario, when MTU size is small and a large amount of data is being transferred.

5.3.4 Alternate NIC implementations

There has also been work that provides the benefits of both the schemes that we discussed in sections 5.3.2 and 5.3.3, by proposing alternative NIC implementations that address the high latency as well as non-scalability issues of TCP. Conventional NICs implement Layer-2 protocols and are not aware of the user application that will receive the incoming packets. The Layer-2 protocol copies the packet to the kernel and the upper layers of the protocol stack decipher the associated application. If, however, the NIC is enhanced to be aware of the designated core, it can potentially avoid multiple copies and expedite TCP reads on multicore. There are several proposals to enhance the NIC design with this feature.

Binkert et al. [5] propose simplified on-chip NIC designs that are integrated with the host CPU, with most of the functionality of the conventional NIC being handled by software.

Their first design, SINIC, proposes a NIC that is stripped down to its most basic components, which is a pair of FIFOs, and all other functionality, including the tasks of the DMA engine, are done by the device driver on a CPU core. Not only does their implementation give comparable performance to a conventional on-chip NIC, this also leads to more opportunities for software optimizations in packet processing.

This implementation makes way for another scheme for zero-copy by keeping the packet data in the receive FIFO until the packet header is processed. The packet can then be copied directly to the application buffer achieving true zero-copy. This idea can be used in conjunction with our implementation to speed up TCP reads. They further propose another NIC design call V-SINIC, which has multiple virtual FIFOs tied to a pair of physical FIFOs that lets packets to be processed in parallel. The V-SINIC implementation with zero-copy extensions implemented in the Linux 2.6 kernel provides bandwidth improvements of over 50% on an unmodified sockets-based receive-intensive micro-benchmark.

5.4 Canonical Multicore Architecture

This section provides an overview of how the hardware optimizations that we propose in the previous sections fit in to make a canonical architecture suitable for routing protocol workloads. We first do a brief walk through of how the packet flow happens in our baseline multicore implementation. Packets from different peers arrive at the network interface card of host machine running the BGP protocol. Subsequently, NIC interrupts the CPU, which runs memory mapped I/O (MMIO) to copy these packets from the NIC buffer to the kernel memory. Sequential or PBTS application then call TCP read, which copies the data to stream buffers that reside in user memory. Once the CPU core starts parsing packets and accessing some data from these stream buffers, data is copied to the L2 and L1 caches, one cache-line at a time. This flow is shown in figure 5.2a.

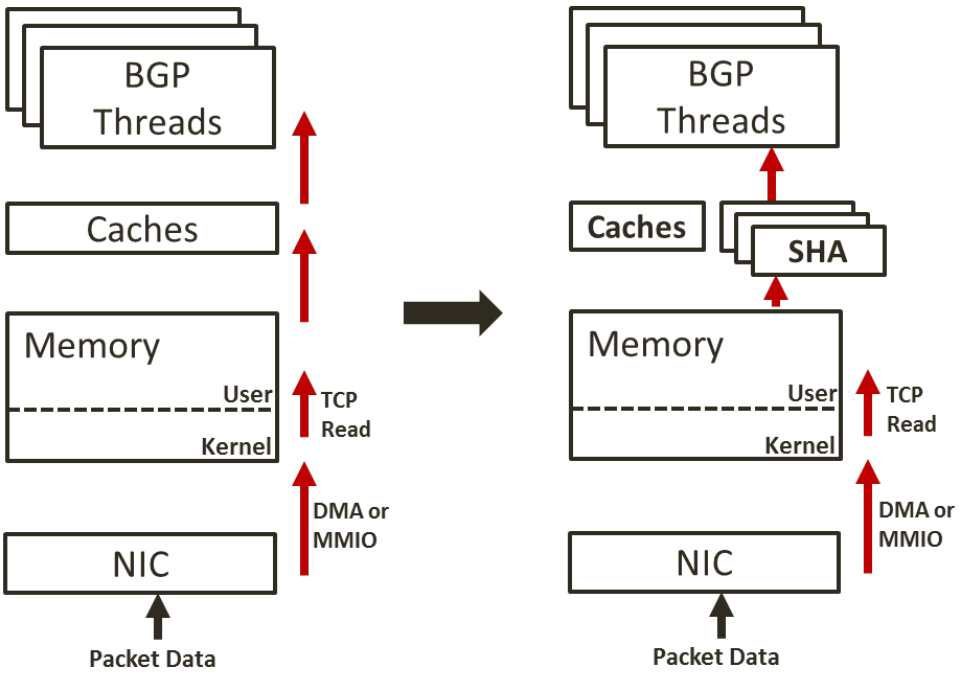

Figure 5.2: (a) Baseline architecture we used for sequential and parallel software analysis, (b) Introduced Stream Hardware Assist for faster packet processing

As we discussed in the memory analysis section, BGP threads do not reuse the stream buffer data once it is cached, and hence the most stream accesses miss in the L1 and L2 cache the first time they are fetched, causing high latency in reading packet data. To reduce this latency, we propose using smaller L2 caches, and introduce Stream Hardware Assist (SHA), which is dedicated for storing BGP packets for different threads. Figure 5.2b shows the placement of SHA in the system. In PBTS, each thread communicates with a different peer, and hence we propose assigning a separate SHA for each thread (or each core), which stores packets coming from only a particular TCP connection. Each SHA should be sized to store

one maximum sized BGP packet. While SHA's reduce the overall latency to access streams and speed up the packet and header parsing functionality, they still require data to be copied from the kernel space memory user space memory and subsequently to stream buffers, causing high memory copy overhead and latency.

This overhead can be eliminated if the data can be directly copied to the SHA from kernel memory or the NIC itself as shown in Figure 5.3. The green lines in figure (flow 2) show an optimization where a packet is copied from NIC to kernel space memory using the memory mapped IO, but instead of copying the data to user space using TCP read(), it copies the data directly into the relevant stream buffer. This can be achieved by using one of the zero copy TCP schemes we used discussed in section 5.3.1.

The red lines in figure show an optimization where data is directly copied from the NIC to SHA, and thus completely eliminates the need to have MMIO from the CPU to copy data into kernel space memory. Two schemes that we discussed in section 5.3 propose this kind of implementation. To enable this packet copy from NIC to SHA, the NIC should be able to classify packets for a particular TCP/BGP flow, and then copy the data to the corresponding SHA via memory mapped IO or DMA. Figure 5.4 shows the canonical architecture, which includes all the optimizations we discussed.

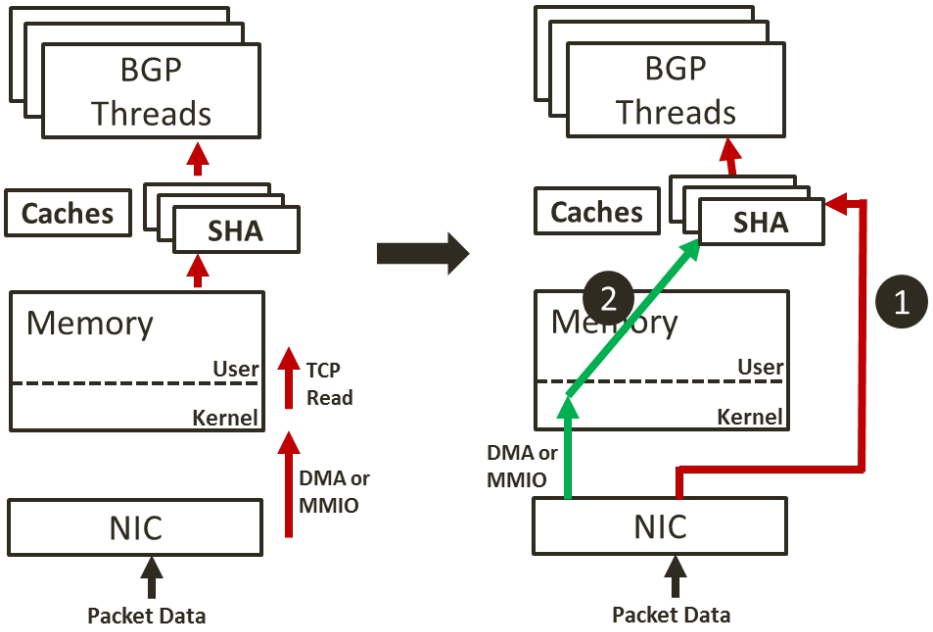

Figure 5.3: Use a V-SINIC like implementation to copy the data directly to the SHA

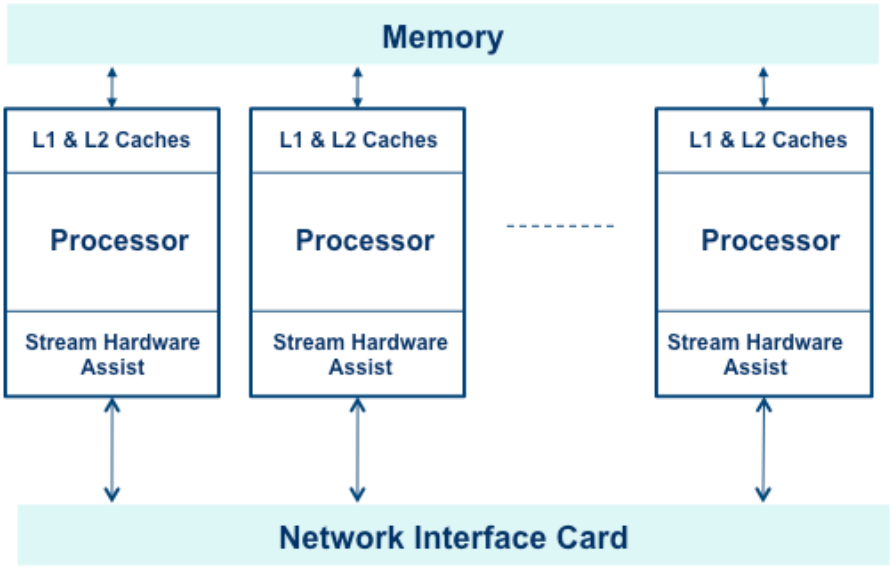

Figure 5.4: Canonical Architecture for Routing Protocols

Chapter 6: Summary

The key contribution of this work includes understanding the open source implementation of BGP protocol and developing a parallel BGP implementation based on the Quagga routing suite. We developed a realistic simulation environment capable of covering both the large network simulation as well as detailed architecture simulation aspects of our problem. Subsequently, we analyzed the parallel BGP implementation for performance and architectural bottlenecks and identified several optimizations that can improve parallel BGP performance on multicore processors. We also did a literature survey of a variety of techniques that improve the performance of the TCP/IP network stack that is used by BGP, and which takes up a significant portion of the BGP packet processing time. Further implementation and analysis of TCP/IP related issues forms a part of a broader research area which is not specific to BGP, Network control plane or Multicore as such and hence is beyond the scope of our work.

Finally, we propose a canonical architecture to improve the performance of packet parsing by using hardware based stream assists and adding hardware support to speedup up TCP accesses. These solutions can further help in a significant reduction of the overall packet

process time and improving throughput. 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.

We view our work as a step towards initiating BGP to the era of Multicore. A multithreaded BGP can benefit much more from the current and future generation of Microprocessors than the conventional implementations as there is no line of sight for a radical increase in the single thread performance in the near future. With a better BGP router performance, the BGP protocol will be able to cope in the world of high bandwidth applications and a huge number of connected devices, with better scalability and reliability.

References

[1] "Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten".

[2] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich and H. Wilkinson. The next generation of Intel IXP network processors. *Intel Technology Journal 6(3),* pp. 6-18. 2002.

[3] S. Agarwal, C. Chuah, S. Bhattacharyya and C. Diot. The impact of BGP dynamics on intra-domain traffic. Presented at ACM SIGMETRICS Performance Evaluation Review. 2004.

[4] T. Bates. BGP route reflection an alternative to full mesh IBGP. 1996.

[5] N. L. Binkert, A. G. Saidi and S. K. Reinhardt. Integrated network interfaces for high-bandwidth TCP/IP. *ACM Sigplan Notices 41(11),* pp. 315-324. 2006.

[6] L. Blunk, M. Karir and C. Labovitz. MRT routing information export format. 2009.

[7] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic and W. Su. Myrinet: A gigabit-per-second local area network. *Micro, IEEE 15(1),* pp. 29-36. 1995.

[8] V. Bollapragada, C. Murphy and R. White. *Inside Cisco IOS Software Architecture.* 2000.

[9] J. C. Brustoloni. Interoperation of copy avoidance in network and file I/O. Presented at INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE. 1999.

[10] J. S. Chase, A. J. Gallatin and K. G. Yocum. End system optimizations for high-speed TCP. *Communications Magazine, IEEE 39(4),* pp. 68-74. 2001.

[11] H. J. Chu. Zero-copy TCP in Solaris. Presented at USENIX Annual Technical Conference. 1996.

[12] Cisco Systems Inc., "The Cisco QuantumFlow Processor: Cisco's Next Generation Network Processor". 2008.

[13] Cisco Systems Inc. 1000 Series aggregation services routers.

[14] A. Dhanotia. Scalable software and architecture for network routing protocols. 2010.

[15] A. Dhanotia, S. Grover and G. Byrd. Analyzing and scaling parallelism for network routing protocols. IEEE International Symposium on Workload Characterization (IISWC). 2010.

[16] X. A. Dimitropoulos and G. F. Riley. Efficient large-scale BGP simulations. *Computer Networks 50(12),* pp. 2013-2027. 2006.

[17] X. A. Dimitropoulos and G. F. Riley. Large-scale simulation models of BGP. Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS). 2004.

[18] M. Dobrescu, N. Egi, K. Argyraki, B. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. ACM SOSP. 2009.

[19] U. Drepper and I. Molnar. The native POSIX thread library for linux. *White Paper, Red Hat* 2003.

[20] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, L. Mathy and P. Papadimitriou. Forwarding path architectures for multicore software routers. Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow. 2010.

[21] N. Feamster, J. Borkenhagen and J. Rexford. Controlling the impact of BGP policy changes on IP traffic. *AT&T Labs-Research, Tech.Rep.HA173000-011106-02TM* 2001.

[22] C. V. Forecast. Cisco visual networking index: Global mobile data traffic forecast update 2009–2014. *Cisco Public Information,* 2010.

[23] H. Franke, J. Xenidis, C. Basso, B. M. Bass, S. S. Woodward, J. D. Brown and C. L. Johnson. Introduction to the wire-speed processor and architecture. *IBM Journal of Research and Development 54(1),* pp. 3: 1-3: 11. 2010.

[24] Geoff Huston. "The 32-bit AS Number Report". *Http://Www.Potaroo.Net/Tools/asn32/*

[25] S. Grover, A. Dhanotia and G. T. Byrd. A Canonical Multicore Architecture for Network Routers. Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems. 2011.

[26] M. Handley, O. Hodson and E. Kohler. XORP: An open platform for network research. *ACM SIGCOMM Computer Communication Review 33(1),* pp. 53-57. 2003.

[27] C. L. Hedrick. Routing information protocol. 1988.

[28] X. Hu, X. Tang and B. Hua. High-performance IPv6 forwarding algorithm for multi-core and multithreaded network processor. Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2006.

[29] G. Huston. The BGP report for 2005. *Http://Www.Potaroo.Net/Ispcol/2006-06/Bgpupds.Pdf* . 2006.

[30] G. Huston and G. Armitage. Projecting future IPv4 router requirements from trends in dynamic BGP behaviour. Proc. of ATNAC. 2006.

[31] G. Huston, T. Bates and P. Smith. CIDR report. *Http://Www.Cidr-Report.Org.* 2010.

[32] C. V. N. Index. Hyperconnectivity and the approaching zettabyte era. *Cisco Systems, San Jose, CA* 2010.

[33] K. Ishiguro. Quagga, A routing software package for TCP/IP networks. 2006.

[34] K. Ishiguro. *Gnu Zebra–routing Software.* 2001.

[35] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. Proceedings of17th Annual International Symposium On Computer Architecture. 1990.

[36] T. Klockar and M. Hidell. Modularized bgp for decentralization in a distributed router. Winternet Grand Finale Workshop, Poster. 2005.

[37] M. Lad, X. Zhao, B. Zhang, D. Massey and L. Zhang. Analysis of BGP update surge during slammer worm attack. *Distributed Computing-IWDC 2003* pp. 833-835. 2003.

[38] S. Lakshmanamurthy, K. Liu, Y. Pun, L. Huston and U. Naik. Network processor performance analysis methodology. *Intel Technology Journal 6(3),* pp. 19-28. 2002.

[39] J. Le Roux, J. Vasseur, Y. Ikejiri and R. Zhang. OSPF protocol extensions for path computation element (PCE) discovery. 2008.

[40] B. K. Lee and L. K. John. NpBench: A benchmark suite for control plane and data plane applications for network processors. Proceedings of 21st International Conference on Computer Design. 2003.

[41] G. Lei, L. Mingche and G. Zhenghu. An improved parallel access technology on routing table for threaded BGP. 15th International Conference on Parallel and Distributed Systems (ICPADS). 2009.

[42] G. Lei, L. Mingche and G. Zhenghu. Exploiting the thread-level parallelism for BGP on multi-core. Communication Networks and Services Research Conference (CNSR). 2008.

[43] E. Lemoine, C. Pham and L. Lefèvre. Packet classification in the NIC for improved SMP-based internet servers. 2003.

[44] 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 35(2),* pp. 50-58. 2002.

[45] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News 33(4),* pp. 92-99. 2005.

[46] A. Medina, A. Lakhina, I. Matta and J. Byers. BRITE: An approach to universal topology generation. Proceedings of Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems. 2001.

[47] G. Memik, W. H. Mangione-Smith and W. Hu. Netbench: A benchmarking suite for network processors. IEEE/ACM International Conference on Computer Aided Design (ICCAD). 2001.

[48] Millind Mittal, "Optimizing Multicore for Networking Applications," *ECN,* 2009.

[49] Y. Qi, B. Xu, F. He, B. Yang, J. Yu and J. Li. Towards high-performance flow-level packet processing on multi-core network processors. Proceedings of the Third ACM/IEEE Symposium on Architecture for Networking and Communications Systems. 2007.

[50] B. Quoitin, C. Pelsser, O. Bonaventure and S. Uhlig. A performance evaluation of BGP-based traffic engineering. *International Journal of Network Management 15(3),* pp. 177-191. 2005.

[51] B. Quoitin, C. Pelsser, L. Swinnen, O. Bonaventure and S. Uhlig. Interdomain traffic engineering with BGP. *Communications Magazine, IEEE 41(5),* pp. 122-128. 2003.

[52] Y. Rekhter and T. Li. A border gateway protocol 4 (BGP-4). 1995.

[53] J. H. Salim, R. Olsson and A. Kuznetsov. Beyond softnet. Presented at Proceedings of the 5th Annual Linux Showcase & Conference. 2001.

[54] Network Simulator. *NS2.* 1989.

[55] K. Skevik, T. Plagemann, V. Goebel and P. Halvorsen. Evaluation of a zero-copy protocol implementation. Proceedings of Euromicro Conference 2001.

[56] L. Tianhua, Z. Hongfeng, C. Guiran and Z. Chuansheng. The design and implementation of zero-copy for linux. Eighth International Conference on Intelligent Systems Design and Applications (ISDA). 2008.

[57] P. Traina. Autonomous system confederations for BGP. 1996.

[58] C. Villamizar, R. Govindan and R. Chandra. BGP route flap damping. *ISI* 1998.

[59] Wikipedia. "Border Gateway Protocol". *Http://en.wikipedia.org/Wiki/Border_Gateway_Protocol*.

[60] T. Wolf and M. Franklin. CommBench-a telecommunications benchmark for network processors. International Symposium on Performance Analysis of Systems and Software (ISPASS). 2000.

[61] Q. Wu, Y. Liao, T. Wolf and L. Gao. Benchmarking BGP routers. Tenth International Symposium on Workload Characterization (IISWC). 2007.

[62] X. Zhang, P. Zhu and X. Lu. Fully-distributed and highly-parallelized implementation model of bgp4 based on clustered routers. *Networking-ICN 2005* pp. 433-441. 2005.