Applying Machine Learning to Identify NUMA End-System Bottlenecks for Network I/O

By

Harshvardhan Singh Chauhan
B.E. (NMAM Institute of Technology, Nitte) 2013

Thesis

Submitted in partial satisfaction of the requirements for the degree of

Master of Science

in

Computer Science

in the

Office of Graduate Studies

of the

University of California

Davis

Approved:

Dipak Ghosal, Chair

Matthew K Farrens, Co-Chair

Chen-Nee Chuah

Committee in Charge

2017

Dedicated to my parents, family and friends ...

without whom none of my success would be possible.

Also, to all researchers, who strive for the development of this world.

Contents

List of Figures v
List of Tables vi
Abstract vii
Acknowledgments viii

1 Introduction **1**
1.1 The Problem 1
1.2 Contributions 2
1.3 Thesis Outline 4

2 Background and Motivation **5**
2.1 The Hardware 5
2.2 The Operating System 6
2.3 The Workload 6
2.4 Machine Learning Approach 8

3 Related Work **10**

4 Experimental System Setup **15**
4.1 A Fully-Optimized System 15
4.2 Performance Characterization 18
4.3 Extensions to Previous Work 18
4.3.1 Exhaustive Tests 18
4.3.2 Kernel Introspection and Flamegraphs 18
4.3.3 Interrupt Coalescing 20

5 Machine Learning **22**
5.1 Efficient Data Analysis Approach 22
5.1.1 Application of K-means 22
5.1.2 Application of PCA 22

5.2 Multiple ANN Approach 23

6 Results **27**

6.1 Data Analysis Results 27

6.2 Machine Learning-based ANN Prediction Results 28

6.2.1 ANN-Frames: 29

6.2.2 ANN-`rx-usecs:` 30

6.2.3 ANN-Throughput and ANN-bytes/inst-ret(Efficiency): 31

7 Conclusion **34**

7.1 Limitation 35

7.2 Future Work 35

List of Figures

2.1 The Linux receive process for network I/O. 7

4.1 Block diagram of the end-system I/O architecture 16
4.2 A flamegraph of the system calls involved in a network transfer. 20
4.3 The efficiency (measured as bytes received per interrupt) for different settings of `rx-frames` in the same-core case. 21
4.4 The efficiency (measured as bytes received per interrupt) for different settings of `rx-usecs` in the same-core case. `rx-frames` had been hand-tuned to 47. 21

5.1 Core ML approach Flow Diagram 23
5.2 Working of ANN model 23
5.3 ANN predicting the frame size and usecs for given features 26
5.4 NN predicting Throughput and bytes/inst-ret(efficiency) for given features 26

6.1 Experiment architecture 27
6.2 Cluster-k value for different iterations 29
6.3 Kmeans after PCA 30
6.4 Training vs. Prediction graph for achieved throughput. 32

List of Tables

4.1 List of environmental parameters. 17

4.2 A table of the throughput performance for every possible combination of flow and application core binding (affinity). 19

6.1 Color Code Mapping (to be read with Figure6.3) 30

6.2 Predicted vs. hand-tuned rx-frames value 31

6.3 Predicted vs. hand-tuned `rx-usecs` values 31

6.4 Prediction vs. Actual data 32

Abstract

Applying Machine Learning to Identify NUMA End-System Bottlenecks for Network I/O

Performance bottlenecks across distributed nodes, such as in high performance computing grids or cloud computing, have raised concerns about the use of Non-Uniform Memory Access (NUMA) processors and high-speed commodity interconnects. Performance engineering studies have investigated this with varying degrees of success. However, with continuous evolution in end-system hardware, along with changes in the Linux networking stack, this study has become increasingly complex and difficult due to the many tightly-coupled performance tuning parameters involved. In response to this, we present the Networked End-System Characterization and Adaptive Tuning tool, or NESCAT, a partially automated performance engineering tool that uses machine learning to study high-speed network connectivity within end-systems. NESCAT exploits several novel techniques for systems performance engineering. These include using k-means clustering and Artificial Neural Networks (ANNs) to effectively learn and predict network throughput performance and resource utilization for end-system networks.

NESCAT is a unique tool, different from other previously designed applications. It is based on of machine learning and clustering techniques on NUMA core-binding cases. This work focuses on predicting optimal Network Interface Controller (NIC) parameters for performance predictability, which is a necessity for complex science applications. Through experiments, we are able to demonstrate the uniqueness of this technique by achieving high accuracy rates in predicted and actual performance metrics such as throughput, data rate efficiency, and frame rates. Our system is able to ingest large amounts of data to produce results within 2 hours for a machine with an 8-core end-systems. The root mean square error of the designed model is around 10^{-1} and thus predicts output efficiently when compared to live run data on an actual machine.

Acknowledgments

The completion of any work involves the efforts of many people. I have been lucky enough to have received a lot of help and support from many people during this work. So, with great gratitude, I take this opportunity to acknowledge all those people, whose guidance and encouragement helped me to emerge successful. Foremost, it is my privilege and honor to express sincere gratitude to my major advisor and guide Dr. Dipak Ghosal for his encouragement, motivation, knowledge and invaluable support towards my work and research. I would also like to thank my co-advisor Dr. Matthew K. Farrens for his support towards this work with immense knowledge and enthusiasm. Their continuous guidance, evaluation and mentoring helped me during the research process and achieve the completion of thesis.

I am greatly thankful to my committee member Dr. Chen-Nee Chuah for helping with timely evaluation, useful suggestions and feedback towards this work. I am greatly indebted to Nathan Hanford for helping with technical contents and mentoring me during thesis work. I am also grateful to all Professors whom I took courses with and all non-teaching staff of my department who extended their unlimited support.

I would also like to thank all my friends for their help and support. I would also like to thank my parents Dr. Pushpendra and Neera Chauhan, my sister Pavni and other family members for providing continuous encouragement and moral support.

Chapter 1

Introduction

1.1 The Problem

Complex data-intensive applications are becoming increasingly frequent in scientific discovery [1]. These applications rely on network middleware that supports high-speed and high-capacity data delivery across distributed resources. For communication across high-performance computing grids and cloud computing architectures, engineers have focused on improving application performance bottlenecks by tweaking computing workloads across the distributed nodes. For example, HPC(High-performance computing) scheduling algorithms and cloud load balancers developed via SLURM (Simple Linux Utility Resource Management) have demonstrated improvement in climate modeling application [2].

The grid clusters and chip makers have adopted Non-Uniform Memory Access (NUMA) multi-core processors to allow more computations per given frequency. This along with the use of various flavors of Linux operating systems (OSes) has considerably reduced the throughput predictability required by science applications. Assessing hardware profiles based on application code, especially in distributed environments, is increasingly important for designing successful applications. To cater to this effort, this thesis explores using machine learning (ML) algorithms to learn resource usage patterns across science applications and distributed computing architectures. The aim is to produce benchmark results for hardware and application performances on the system.

In terms of architecture, parallel and distributed computing systems like HPC and the cloud still rely on a full TCP/IP protocol stack for data communications [3]. However, based on the end-to-end principle [4], TCP Offload Engines (TOE) have reduced much of the TCP processing overhead. Additional On-Line Transaction Processing (OLTP) has impacted throughput predictability for communication networks. In order to improve throughput and latencies, vendors have turned to Non-Uniform Memory Access (NUMA) models to achieve higher network resource density. NUMA hardware provides physical addresses different access times, but is engaged via "dice rolling" to pick cores. In particular, the assignment of cores to flows is either performed randomly or via obscure hash functions, or based on some arbitrarily-chosen metrics. Our previous work on NUMA end-systems demonstrated wide variation in the achieved throughput performance depending on which core is assigned to the incoming flow. The results demonstrated that affinity (core-binding) serves as a key performance indicator along with memory controller and on-chip features that leads to system bottlenecks when NIC drivers respond to interrupts [5, 6]. Previous work on tuning parameters showed that interrupt coalescing parameters are critical to throughput performance as well [7]. However, studying performance benchmarks for bottlenecks and identifying the root cause of performance degradation becomes increasingly complex with multiple parameters and performance data. Additionally, the changes to the Linux kernel networking stack impact the temporal validity of the data. This thesis incorporate all of these factors and shows how performance engineering practices are impacted.

1.2 Contributions

This thesis presents the use of ML approaches to study performance patterns across multiple resource usage and application data sets. The model is used to predict throughput performances for various data-intensive experiments. The ML approach involves studying the performance in three stages. First, characteristics of the computing clusters involved per experiment are studied to produce an elbow curve. The elbow curve [8] is an established clustering technique for studying the variance among the number of k clusters used

and the performance received until no enhancement is detected. Second, the K-means clustering technique is used to create k clusters and find optimal multi-variable centroids across clusters [9][10][11]. Third, the data is then brought together to predict throughput and bytes/instruction-retired (a rough efficiency metric) by processing the data via artificial neural network models.

This thesis presents NESCAT, a portable, automated tuning methodology for high speed data transfer nodes (DTNs) and high speed 40 Gbps LAN systems. The main contributions of this works are as follows:

- An efficient yet exhaustive collection methodology for detailed end-system performance characterization across all possible NUMA boundaries.
- An automated scheme for detecting core affinity cases which effectively represent NUMA boundaries.
- An automated method for accurately predicting throughput and network processing efficiency, given flow and application core affinities.
- An adaptive and automated method for adjusting interrupt coalescing parameters for performance predictability and network stack processing efficiency.

The evaluation of results is done by generating large amounts of data and partitioning it into training, testing and validation sets. The test data is unexposed to the trained model. The features of the testing model are passed to the Artificial Neural Network (ANN) show in Figure 6.1, and the predicted values of throughput and bytes/instruction-retired are compared against actual values generated by the system during the experiments. The graph of sum of square error mean and the RMSE(Root Mean Square Error) value is considered as a measure of model accuracy. The root mean square error value for throughput and bytes/instruction-retired results is obtained around 10^{-1} on average. This high degree of accuracy allows NESCAT to tune coalescing parameters with a high degree of accuracy as well. Given the performance evolution of the Linux network stack along with efficient ML algorithms, this work is especially timely.

1.3 Thesis Outline

The rest of this thesis has been organized as follows. Chapter 2 discusses the background and motivation of this work. Chapter 3 discusses the related works both in the area of end-system optimization for high-speed networking as well as ML. Chapter 4 describes the experimental setup and monitoring tools for data collection. Chapter 4.1 describes the performance characterization of NUMA end-system. Chapter 5 discusses the ML methodology and the different algorithms used in this research. Chapter 6 discusses the results, and, finally, Chapter 7 gives the conclusion and outlines future work.

Chapter 2

Background and Motivation

2.1 The Hardware

While network and general interconnect speeds continue to grow to 400 Gbps and beyond [12], CPU clock speeds have stopped increasing to higher amount, thus creating a speed gap. Consequently, chip makers have been moving to NUMA multicore architectures. Furthermore, features which used to reside off-chip, such as the north bridge[1] are now integrated onto the processor die. Figure 4.1 shows a contemporary Intel micro-architecture with a Quick Path Interconnect (QPI) linking the north bridges on two sockets in a dual-socket system.

These micro-architectures are commonly being deployed in supercomputers, clouds, and clusters with variable performance results [13]. Understandably, with these deployed in scale-out environments, the amount of unpredictability increases resulting in severe performance unpredictability in HPC jobs [14]. In cloud environments, research is focused on batch job completion performances and OLTP, but over-provisioning of datacenters delivering Service-Level Agreements (SLAs) is still not studied, due to most of the utilization data being proprietary and confidential.

[1]The north bridge is a PCI bus controller, and additionally a memory controller on some systems.

2.2 The Operating System

Linux has become the standard operating system for an incredibly diverse variety of applications running on different hardware architectures, ranging from light bulbs to supercomputers. As a result, Linux kernels have become the subject of much speculation and development. Inherently a network-ready OS, Linux owes much of its popularity to its robust and feature-rich network stack.

2.3 The Workload

TCP/IP is the building block of the Internet. Many architects liken TCP/IP to the C programming language, popular due to its ubiquity. TCP/IP is a deceptively complex distributed set of processes, all working in a provably optimal and correct manner to provide a unique set of protocol properties [4]. Foremost of those properties is survivability. TCP/IP makes a minimal set of assumptions about the reliability of the intervening network between a sender and a receiver, yet it still provides reliable, connection-oriented behavior in a wide variety of network conditions. While TCP does not perform optimally or even well in all conditions, the protocol still performs correctly. This has been a key reason for its widespread deployment.

Using TCP/IP, network operators are able to easily scale workloads, using minimal assumptions about the intervening network. Application developers are also able to learn and utilize a single protocol stack across many platforms. However, the TCP survivability is not always free, making assumptions about network and adding complexity to the end system protocol stack. From a systems and architecture standpoint, this complexity can become difficult to manage. In addition, TCP/IP protocol processing is a memory copy intensive workload due to the fact that information must first be copied to the kernel, and then be copied to the application which consumes the data. Figure 2.1 illustrates the Linux TCP/IP network receive process. There have been many solutions proposed to ameliorate or even eliminate these copies. However, many of these solutions begin by making assumptions about the intervening network or the system state, but the application is usually unaware of either. In practice, TCP can invoke even more memory

copies due to a variety of possible page fault situations. There have been many efforts to ameliorate the effects of the resource-intensive network workload which are discussed in detail in Chapter 3.

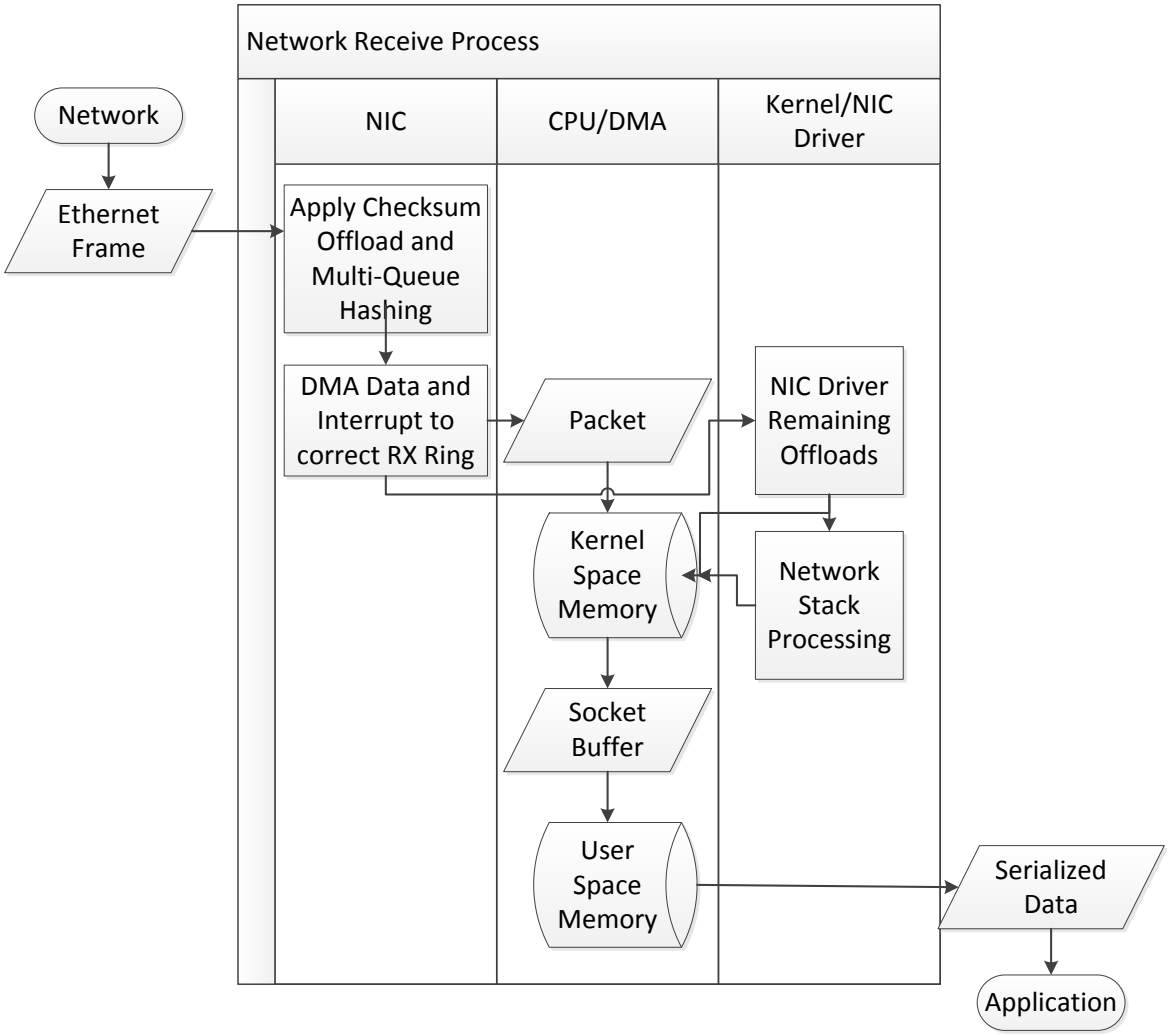

Figure 2.1. The Linux receive process for network I/O.

Interrupt coalescing is a technology with efficacy instrumental to this study. Out of dozens of the most-tuned parameters, this study determined that interrupt coalescing is by far the most important and perhaps the least understood by systems architects when it comes to end system network processing performance and efficiency. While there are many different parameters which can be set to control interrupt coalescing, there

are two essential parameters: `rx-frames`, the number of frames the NIC must receive before an interrupt is generated and `rx-usecs`, the time to wait, in microseconds, before interrupting the kernel's network receive routine[2]. Both of these parameters allow the user to change the 1:1 ratio between a packet and an interrupt through the use of a buffer or a timer, respectively. If the values are set too high, it can have detrimental effects on the latency of packet processing in the end-system. When set too low, receiving cores can be overwhelmed with interrupt processing. While the mechanism of either waiting for a timeout or a certain number of packets to arrive is simple, the way it interacts with a high-performance end-system is complex. For example, high clock-rate processors are better at handling interrupts quickly, but perhaps efficiency is still a concern. Furthermore, in this work we explore the points of diminishing return with regard to interrupt coalescing.

2.4 Machine Learning Approach

The data which constitutes of values obtained from different system parameters during execution of data extraction scripts is mainly nonlinear in nature. The goal is to tune the parameters which have categorical values as well as numeric values, so that when tuned, the system performs better than previously. As the dimensionality of data is high and non-linear in our case, methods like Naive Bayes is unfeasible. Since this is a multiclass problem, SVM (Support Vector Machine) has no standard way of dealing with this problem as it is more suited for binary classification problems. Also, logistic regression is ideally a binary classifier which does not suit the problem well. Due to the large number of dimensions, the current approach also uses PCA (Principle Component Analysis). In addition, a part of the current solution involves clustering which is a type of unsupervised learning technique. Thus, two models which stands out are Decision Trees (also known as Tree Ensembles) and Neural Networks. Of these, the technique that is adopted is based on factors like amount of data, training time, memory, over-fitting etc.

The decision tree based models subdivide the feature space into different regions and groups them who have similar labels and values. This is based on whether or not the problem is a classification problem or a regression problem. Decision tree based models

[2]Versions of these parameters also exist for transmitting data.

can be implemented for both supervised and unsupervised learning and they are more applicable when there are non-linear relationships in the data. Furthermore, it also tackles outliers quite well. The current work is a part of a more ambitious goal of end-system tuning for network I/O. This is discussed in Chapter 7.2. To achieve the overall goal of optimizing network I/O, the work will require multiple approaches based on machine learning's supervised, unsupervised and reinforcement learning techniques. One of the models which can be used in all three different ML techniques is neural networks.

The machine learning (ML) based Artificial Neural Network (ANN) can be used to analyze large amounts of data consisting of important end-system parameters that impacts the network I/O throughput and efficiency. The initial data analysis part using ML-based analysis and clustering algorithms helps us prune data with similar characteristics and collect effective data. The main reason for using ML and designing an ANN is to predict the values of throughput and efficiency. ANNs can compute huge amounts of complex data which have non-linear relationships and predict results quite accurately when compared to values extracted from live runs on a machine. It was felt that the presence of a model which can take in huge amounts of tightly coupled data in the context of high-speed networking and help users tune a variety of inter-related parameters was not available. Also to motivate others to come up with such models, we decided to create a baseline model based on ML.

Chapter 3

Related Work

Hardware architects have demonstrated that CPUs have been starved for data from memory hierarchies [15]. The processor is allowed more access to inter-memory bandwidth post the development of scale-out commodity computing, where multi-chassis clusters are increasingly common. However, whether or not memory controllers and "north-bridge" (high-speed I/O controller) features can realize throughput improvement, is yet to be tested. Additionally, other factors, such as the memory latency, processor speed gap, cache hierarchy bottlenecks and miss rates, continue to starve processors for data within many workloads.

In modern hardware architectures, north-bridge features have been moved directly onto the processor die, given the large amounts of real-estate opened by shrinking semiconductor fabrication processes. This has led to obvious benefits of reducing latency due to interconnect, but produced drawbacks on PCI-Express devices allocated to physical sockets in a multi-socket system. In order to allow sockets to scale out on the same motherboard, chipmakers have introduced high-bandwidth interconnects between the physical sockets. In this manner, applications running on socket 1 retain access to an I/O resource directly connected to socket 0. However, this method introduces additional workloads on on-chip features already tasked with handling overall memory accesses. Figure 4.1 shows this on a contemporary Intel microarchitecture.

There have been many architectural advancements which must be taken into account when thoroughly characterizing the TCP/IP network stack processing of an end-system.

Contemporary NICs support multiple receive and transmit descriptor queues. A NIC can send different packets to different queues in order to distribute processing among CPUs, by applying a filter to each packet that assigns it to one of a small number of logical flows. Packets for each flow are steered to a separate receive queue, which in turn can be processed by separate CPUs. This mechanism is generally known as Receive-side Scaling (RSS). The goal of RSS (as well as other scaling techniques) is to allow performance to increase uniformly. Receive Packet Steering (RPS) is essentially a simplified RSS, done in software (i.e., what the NIC does with the on-board RSS tables and interrupts and DMA queues is done in RPS using extra linked-lists in the host memory). Since it is in software, it is necessarily called later in the datapath - thus, while RSS selects the queue and hence the CPU that will run the hardware interrupt handler, RPS selects the CPU which will perform protocol processing above the interrupt handler. Another development in RSS has been RFS, which adds the key feature of migrating flows to the cores upon which the application already resides, using a pair of indirection tables. A special algorithm is used to ensure that migrations result in minimal packet loss or reordering.

However, RFS [16] chooses cores such that they share the lowest possible level of the cache structure[1]. For example, when a given core (e.g. core A) is selected to do the protocol/interrupt processing, the core that shares the L2 cache with core A should execute the corresponding user-level application. Doing so will lead to fewer context switches, improved cache performance, and ultimately higher overall throughput.

`irqbalance` scatters interrupts across cores based on the load statistics. In one sense, irqbalance is a variant of round-robin scheduling. Foong et al [17] and Narayanswamy et al [18] have analyzed the effect of processor affinity on networking performance of multicore systems. They demonstrated the limitation of `irqbalance` and the benefit of RSS.

There has been a good deal of attention paid to the use of interrupts in high-speed network flows [19]. Message-Signaled Interrupts (MSI-X) has provided a method for reducing the overhead of large numbers of interrupts on the processor core handling the network flow by combining the interrupt and the data into a single packet sent over the

[1]In this document we consider the L1 cache to be at a lower level (closer to the core) than the L2 cache, L2 lower than L3, etc.

PCI bus.

Pause frames [20] allow Ethernet to assist TCP with flow control, to avoid a multiplicative decrease in window size when only temporary buffering at the router or switch is necessary. However, there have been some implementation issues concerning deadlock with pause frames which have led network operators to disable them in production networks [21].

Jumbo Frames are simply Ethernet frames that are larger than the 1500-byte limit that has been most common in the past. In most cases, starting with Gigabit Ethernet, frame sizes are 9000 bytes. This allows for better protocol efficiency by increasing the ratio of payload to header size for a frame. Although Ethernet speeds have now increased to 40 and 100 Gbps, this standard 9000-byte frame size has remained the same. The reason for this is the various segmentation offloads. Large/Generic Segment Offload (LSO/GSO) and Large/Generic Recieve Offload (LRO/GRO) work in conjunction with Ethernet implementations in contemporary routers and switches to send and receive very large frames in a single TCP flow for which the only limiting factor is the negotiation of transfer rates and error rates.

With the advent of techniques like Direct Cache Access (DCA) [22], processor affinity has become very important. DCA-enabled drivers on DCA-capable systems can achieve much higher performance than systems which must undergo the throughput and latency bottlenecks of primary memory. However, this also creates additional performance unpredictability if cores are chosen arbitrarily and parameters are left untuned.

The work on the lines of reinforcement learning is done in CAPES [23]. They have tried to tune the congestion window parameter to increase the throughput of a Lustre 2.4 system. Using DNN (Deep Neural Network), based on the Q-learning method, they tune the unsupervised storage performance. The work on detection of bottlenecks on the fly is not published. The work does not cover the process of portability i.e. mapping of models on different environments. It falls short on suggestions about tuning of multiple parameters at a time. There are insufficient details about tuning of multiple parameters at a time. Also other kinds of performance outputs like efficiency, latency, energy etc. are

yet to be covered.

To tackle the problem of bottleneck detection in computer system, some studies have adopted belief networks [24] which are a type of Bayesian network i.e., probabilistic graphical model. The bottleneck is based on over-consumption of some hardware resource resulting in delay of completing workload. The technique is based on a functional model which consists of interaction between application workloads, the Windows OS and the system hardware. Uncertainty in workloads, predictions and counter values of Windows performance monitoring tool are characterized with Gaussian distributions. As an analysis, performance monitor values are used to find the best assignment of workload.

DeepRM tool studies the processes to improve performance of resource management in systems and networking [25]. This tool learns and manages resources directly based on previous experiences. This method competes against the traditional heuristics like Shortest-Job-First. It is a multi-resource cluster scheduler that operates in an online setting where jobs arrive dynamically. DeepRM learns to optimize various objectives and uses a gradient reinforcement learning algorithm on Deep Neural Network model. The work is mainly on the multi-resource cluster scheduling problem.

The study in [26] proposes automatically diagnosing previously encountered performance anomalies in HPC systems. Their machine learning framework is more related to learning of anomaly characteristics and the type of anomaly present in data which is collected along the lines of resources usage and performance counter data. The work uses two environments namely HPC cluster and public cloud computing platform to showcase the results. However, this study does not address how to tune the network related parameters and predicting the performance metrics such as throughput and latency.

The study in [27] is related to predicting time frames of node failures in a HPC system using the F-score. They assume that features of certain time frames before a critical event can serve as indicator of a failure. They train a model with critical and healthy time frames. The underlying problem is a classification problem and is not related to tuning of parameters and performance prediction.

Some work in the area of visual recognition tasks has been carried out in Project Adam

[28]. It is a design and implementation of a distributed system comprised of commodity server machines which is trained to achieve very high performance, scaling and classification accuracy on visual reorganization tasks. They exploit asynchrony throughout the system to improve performance and demonstrate that it improves accuracy of the trained model. The study discusses methods to reduce the number of memory copies to optimize the memory system to achieve high performance. A review of the latest work in applying ML in networks for workflow is given in [29]. Finally, a study in the area of multicore architectures to optimize running time and energy efficiency is reported in [30]. The study adopts a state-of-the-art statistical ML techniques using a 7-point and 27-point stencil code on two multicore architectures to optimize running time and energy efficiency.

Chapter 4

Experimental System Setup

4.1 A Fully-Optimized System

To replicate the ESnet's 100G testbed [31] for experimental purpose, two identical systems were used to establish a testbed at UC Davis. These systems used Intel Xeon 2637v3 processors, chosen due to high clock rate, coupled with the availability of thermal slack of 4 cores residing on a single package. Two processors were run on each end-system, in order to simulate a high-performance Data Transfer Node (DTN) residing within a commodity cluster [32]. During characterization, the processor clock speeds were set to 3.5 GHz using the `acpi` driver and the `userspace` governor, as opposed to the default `intel_pstates` configuration. This was done to stabilize the core behavior during repeated tests. Table 4.1 outlines the architectural and systems parameters.

The systems under test make use of PCI-Express Generation 3 connected to Intel Xeon 2637v3 processors which are also codenamed as Haswell. There are two quad-core processors per end-system[1]. Since the release of Sandy Bridge processors, each socket is directly connected to its own PCI-Express bus, such that certain PCI-Express slots are directly physically linked to single sockets. This limitation is overcome with the addition of low-level inter-socket communication provided by the Quick Path Interconnect (QPI). This architecture is seen in Figure 4.1.

In this work, Intel's Performance Counter Monitor (PCM) is used for the collection of

[1]Herein, the four-core packages will be referred to as "sockets" and individual multi-instruction multi-data (MIMD) cores be referred to as "cores".

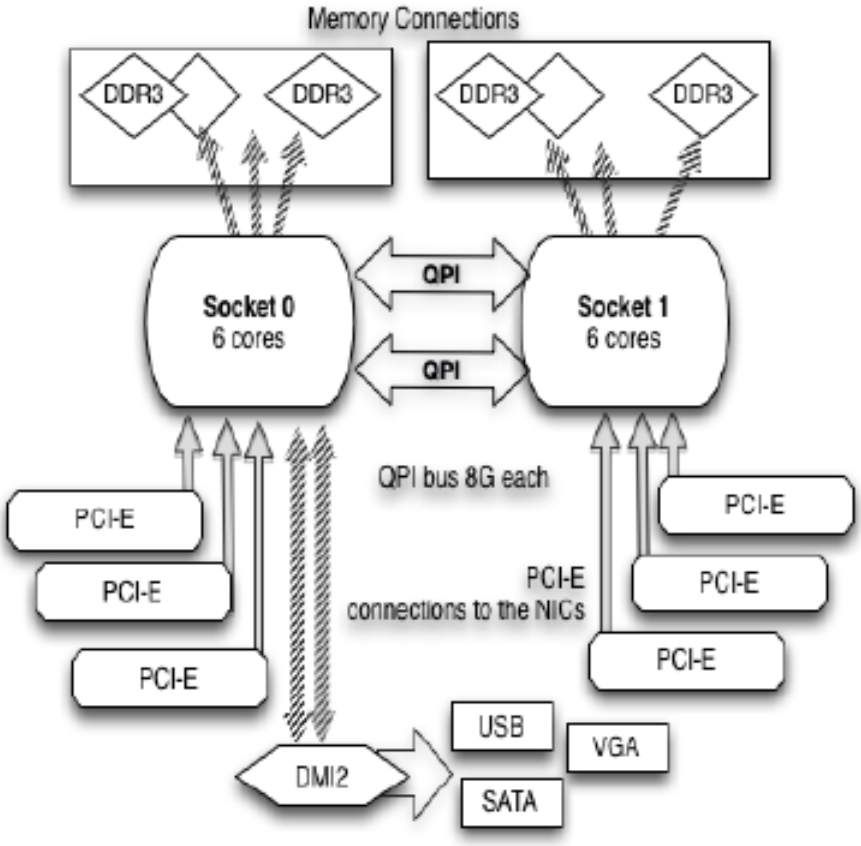

Figure 4.1. Block diagram of the end-system I/O architecture

(PMU) hardware counter information. The main reason for this is that the OProfile does not have `uncore` counters for the QPI. A data collection module was developed to easily accommodate new hardware counter monitoring tools or the PMU counters available in the `/proc` filesystem.

A scenario of back-to-back connected systems with 95ms RTT fiber loop is created for tests. While loop testing is important to test TCP's protocol semantics over long distances, the goal was to avoid the slow start phase and bandwidth limitations (due to noise rates) of TCP over long distances as much as possible. The research involves placing stress and analyzing the performance efficiency of the receiver end-system.

Both of the experiment systems were running Ubuntu Linux 16.04 with the 4.4 kernel.

Table 4.1. List of environmental parameters.

Component	**Value**
Processors	Dual Intel Xeon 2637v3 @ 3.5 GHz
NIC	Mellanox ConnectX-3 EN 40 Gbps
Operating System	Ubuntu Linux 16.04 4.4.0-97-generic
TCP CA Algorithm	HTCP
Linux TC QDisc	fq_codel
Hardware Counter Monitor	Intel Performance Counter Monitor
Workload Generator	iperf3
MTU	9000
Hyper-Threading	Off

The use of one of the latest Linux kernels assures that latest kernel advancements in networking design are employed. The benchmark application used to generate the TCP flows was iperf3. Again, to ensure that the stress was placed on the end-system, the transfers were performed in zero-copy mode, which makes use of the TCP sendfile system call to avoid unnecessary copies into and out of memory on the sending system. The focus of these experiments is on the end-system overheads involved in Linux network stack processing, not the semantics of the competing TCP flows. This makes using a single stream a better choice for uncovering bottlenecks. GridFTP [33, 34] is used to qualitatively compare the network stack processing implications of iperf3 against a GridFTP flow from memory to `/dev/null`, and obtained reasonably similar results. Since any intensive data-transfer workload will still have to use the same calls to the network stack, and the application is not the bottleneck in either case, it is reasonable to assume similar performance.

4.2 Performance Characterization

Every advancement charged with eliminating the end-system bottleneck has met with caveats. This is a product of the complexity of end-system network processing and the fact that many working proposed solutions included unexpected second-order effects. Indeed, the systems code in the Linux TCP/IP network stack still includes "magic constants" determined by quick performance characterization. This an observation rather than a criticism of Linux, as these constants must be used in order to accommodate new features of genuine utility.

One of the first analyses of end-system bottlenecks pertains to systems and architectural performance introspection from [35]. This was the initial inspiration for the continued work on the topic. The effect of NUMA architectures on the end-system bottleneck was roughly characterized in [36] and an additional possible solution using file chunking across multiple TCP streams was presented in [37]. However, after briefly investigating performance bottlenecks against ESnet's 100G testbed [31], it became apparent that a much more thorough analysis was needed for different NUMA architectures combined with different NIC hardware.

4.3 Extensions to Previous Work

4.3.1 Exhaustive Tests

This work expands the work of [36] by creating a system capable of exhaustively testing a receiving system for all possible combinations of network flow and application core binding (also referred to as "affinity") [5]. Table 4.2 shows a summary of the results of an exhaustive test for an 8-core NUMA receiver. The numbers in the body of the matrix represent the achieved throughput in gigabits per second. The diagonal of the table, starting with socket 0 core 1 on both axes, represents all the cases where the network flow and receiving application are bound to the same core.

4.3.2 Kernel Introspection and Flamegraphs

Broadly, Table 4.2 visually illustrates that throughput varies dramatically with the placement of the flow and application threads in a NUMA system. Immediately, results

Flow	Socket 1	Core 4	39	39	39	39	39	39	39	26
		Core 3	39	39	37	39	39	39	26	39
		Core 2	39	39	39	39	39	26	39	39
		Core 1	37	39	39	39	26	39	39	39
	Socket 0	Core 4	39	39	39	28	32	32	32	32
		Core 3	39	39	28	39	32	32	32	32
		Core 2	39	28	39	39	32	32	32	32
		Core 1	28	39	39	39	32	32	32	32
			Core 1	Core 2	Core 3	Core 4	Core 1	Core 2	Core 3	Core 4
			Socket 0				Socket 1			
			Application							

Table 4.2. A table of the throughput performance for every possible combination of flow and application core binding (affinity).

like this could be used to avoid certain core combinations if the Linux kernel made the `rps_sock_flow_table` available for editing. This would allow a core combination "blacklist" to be implemented directly on top of the RFS implementation, utilizing its ability to migrate flows and applications without loss. While this would constitute a reasonable solution on its own, in a completely untuned system over a quarter of the affinity combinations would have to be blacklisted. Rather than simply implementing the engineering solution, a continuous search for the end-system bottleneck using kernel introspection in OProfile was done. However, even analyzing the top 50 system calls over a variety of tests proved to be a difficult task. Figure 4.2 demonstrates the system call complexity for just a single experiment using a

Flamegraph [38]. In this Flamegraph, the length of a segment represents the percentage of time a counter interrupt was triggered during that call. The vertical position represents the callgraph hierarchy. The horizontal position is simply alphabetical order. Finding any correlation between hardware performance counters across different experimental scenarios also proved to be extremely difficult, as many different input parameters were tightly coupled, making simple correlations impossible [6]. Due to the difficulty in correlating these system calls with architectural bottlenecks while retaining genuine callgraph output, introspection has been replaced with detailed hardware analysis in the data collection, but kernel and driver introspection remain in the data collection plans in the near future.

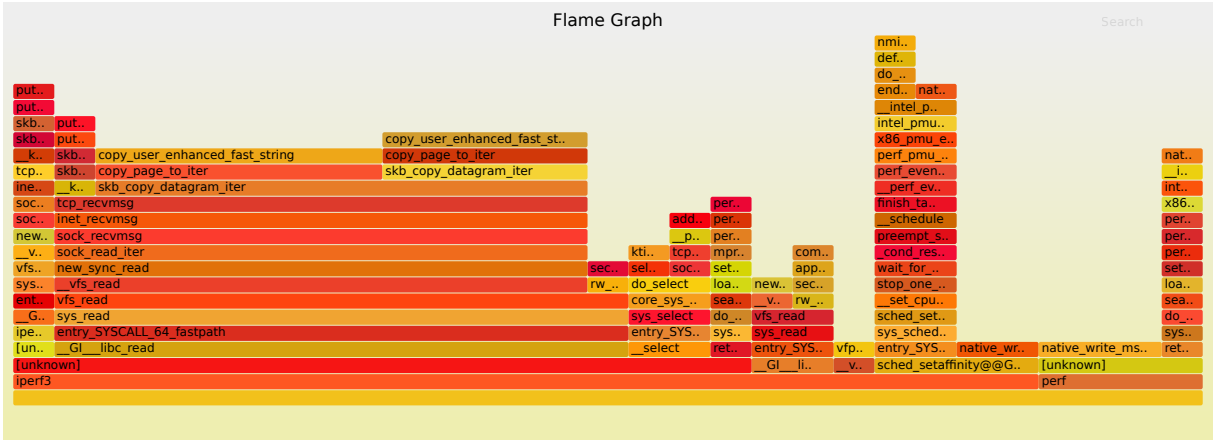

Figure 4.2. A flamegraph of the system calls involved in a network transfer.

4.3.3 Interrupt Coalescing

Some performance counters were also collected from the `/proc` filesystem itself. For example, there was usage of `/proc/interrupts` in order to gather the amount of interrupts generated by the system's receiving NIC. This enabled the gathering of an important metric on the performance of coalescing: the number of bytes received during a fixed-length transfer divided by the number of interrupts raised by the NIC. In this manner, there was a possibility to track whether or not coalescing was behaving as expected. Figures 4.3 and 4.4 shows that the lowest possible frames or microseconds parameter which creates the highest number of bytes/interrupt would be the value used, even if it doesn't quite maximize the efficiency of the transfer. This is due to the fact that these variables must be set as low as possible in order to have the minimum possible impact on the latency of the end-system. Finally, there is one more variable to consider: `pkt-rate-low`, or the packet rate threshold above which the previously set coalescing parameters take effect. Below this threshold, coalescing can be used for many NICs, but it is standard practice to leave coalescing off in order to be sensitive to latency-critical, low-throughput applications.

One interesting point about Figures 4.3 and 4.4 is that they demonstrate a repetitive relationship between the parameters and efficiency. For the rx-frames parameter, this relationship is most likely due to the need to batch precisely the number of packets that fit into the least number of DMA transfers that can be completed within the requisite

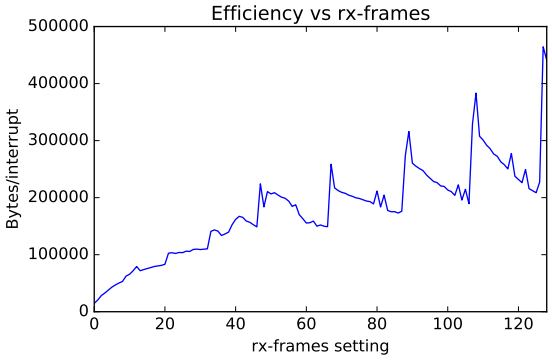

Figure 4.3. The efficiency (measured as bytes received per interrupt) for different settings of `rx-frames` in the same-core case.

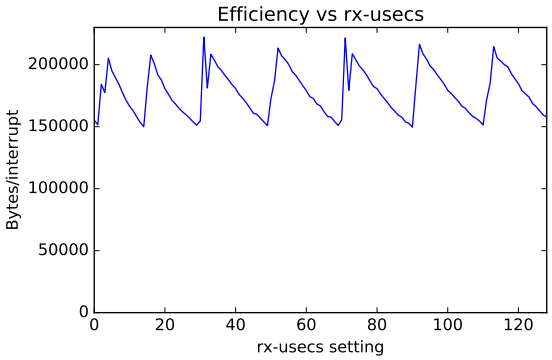

Figure 4.4. The efficiency (measured as bytes received per interrupt) for different settings of `rx-usecs` in the same-core case. `rx-frames` had been hand-tuned to 47.

number of clock cycles. The relationship between `rx-usecs` and `rx-frames` is similar since the timer can compete with the interrupt during execution. The first efficiency peak with the lowest setting of `rx-usecs` is the best result when concerned with latency implications and that should be picked while setting `rx-usecs`. The common tuning knowledge of setting the `rx-usecs` parameter to less than one quarter of the frames parameter held in our preliminary analysis, so NESCAT also follows this rule by default, and does not test higher values of `rx-usecs`.

Chapter 5

Machine Learning

5.1 Efficient Data Analysis Approach

5.1.1 Application of K-means

The process of manual end-system tuning for network stack processing is tedious and repetitive. Usually this involves processing large amounts of collected data to distill results such as throughput and bytes/instruction-retired (a rough measure of efficiency). Then this data, combined with parameter settings, must be used to predict tuning settings. Due to recent advancements in the speed, efficiency, and development of machine learning (ML) algorithms, an ML model can be constructed to do this. The data collected represents non-linear patterns and so in this work an Artificial Neural Network (ANN) is designed to predict throughput and efficiency to inform users about parameter settings to use to achieve predictably good performance. The core ML design flow-graph can be seen in Figure 5.1. On the initial data set, the elbow-curve is drawn to find the possible number of clusters k. Then, the process of iterating through multiple K-means is conducted to find cluster errors and optimal k-values

5.1.2 Application of PCA

The k-value serves as an input to the PCA procedure in order to reduce the number of dimensions to reflect the number of significant parameters. The ML dimension reduction algorithm is evaluated using principle component analysis in python's $scikit-learn$ packages. After reducing the dimension, to only represent significant parameters, the

$k - means$ algorithm is applied to the elbow-curve process. This is discussed in more detail in Chapter 6.2.3.

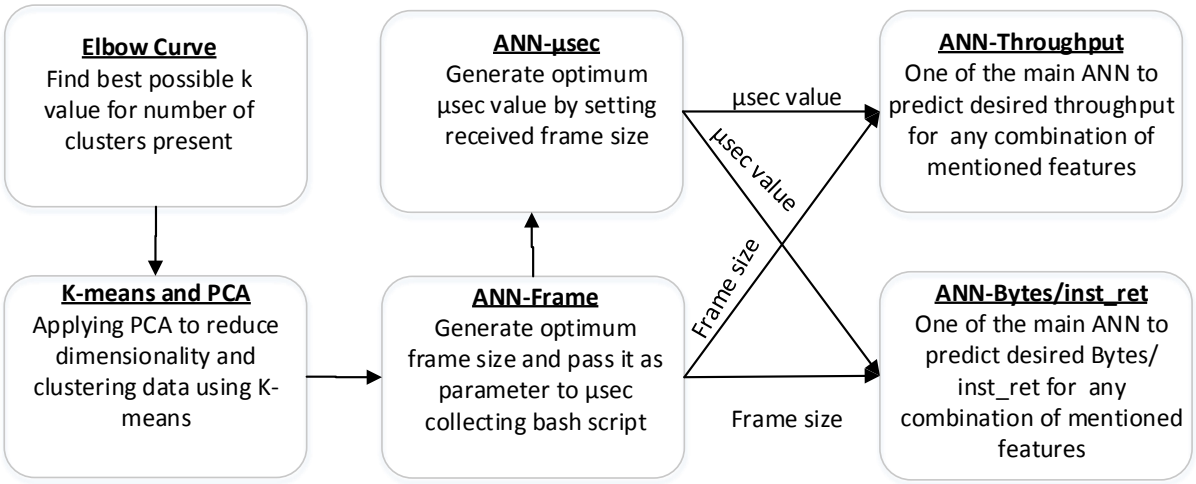

Figure 5.1. Core ML approach Flow Diagram

5.2 Multiple ANN Approach

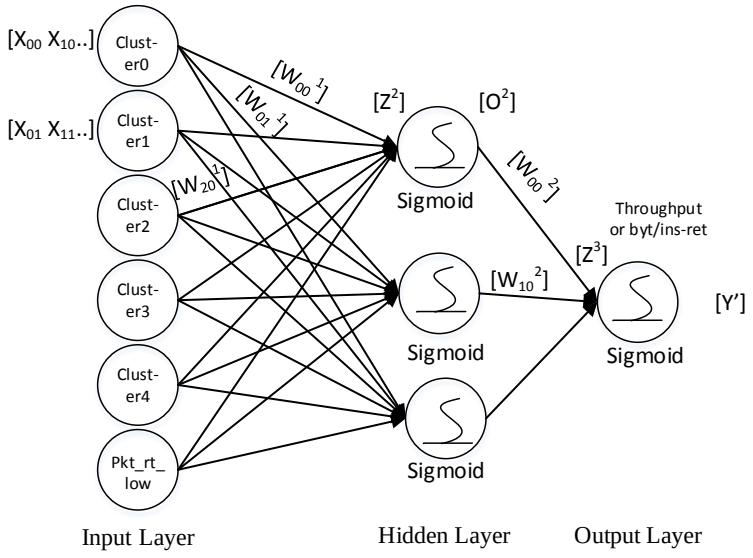

Figure 5.2. Working of ANN model

The Figure 5.2 gives a better overview of working of neural networks. The input layer is a matrix X, which is computed with weight matrix W^1 for the second layer Z^2. The

equation used is:

$$Z^2 = XW^2$$

The sigmoid activation function applied is f(Z^2) and the output matrix O^2

$$O^2 = f(Z^2)$$

The dot product of output matrix O^2 and weight matrix W^2 is computed and it generates the input matrix (Z^3)

$$Z^3 = O^2.W^2$$

The sigmoid activation function $f(Z^3)$ is applied to the output layer neuron and the throughput $Y^{'}$ is predicted.

The training of network considers a cost function C, sum of square error mean, to measure error. This is defined by

$$C = \sum \frac{1}{2}(Y - Y^{'})^2$$

To achieve a minimum cost function, we focused on reducing each synapse (weight) value using batch gradient descent method. Partial derivatives are calculated to find rate of error change i.e., C with respect to weights W^1 and W^2. Finally, the equations will be:

$$\frac{\partial J}{\partial W^2} = -(Y - Y^{'})f^{'}(Z^3)\Delta^3$$

where

$$\Delta^3 = \frac{\partial Z^3}{\partial W^2}$$

and

$$\frac{\partial J}{\partial W^1} = X^T\Delta^2$$

where

$$\Delta^2 = \Delta^3(W^2)^T f^{'}(Z^2)$$

The cost function is chosen to handle non-convex loss functions.

In conjunction with batch gradient descent, a back-propagation algorithm is used to back-propagate the error to each weight. The back-propagation error is fed back to the

network denoted by $\Delta 3$ and $\Delta 2$, where numbers denote errors of their respective layers. The larger the weight value the larger activation and the higher gradient value. Each gradient value drags the gradient descent in a certain direction with the final path being cumulative. In a nutshell, subtracting gradient values from weight reduces the cost.

To overcome the limitations of gradient descent and yield better solutions under realistic time constraints, we used the BFGS (Broyden-Fletcher-Goldfarb-Shanno) optimizer method to estimate second order derivatives. Sourced from the *scipy* optimize package [39], the trainer function trains input data on a set of features. A callback function also keeps track of the cost value while training. Upon training, the initial random weights are replaced with trained values.

The issue of over-fitting is resolved by using the validation data. This is shown in Figure 6.4. A comparison between training and validation data sets helps to identify the number of iterations required to train the model post over-fitting. Also a regularization parameter λ is used to tune relative costs. We leveraged the relative ease of collecting additional validation data in order to validate our results as we proceeded. Another ANN predicts the output - `bytes/instructions-retired` functions, similar to how throughput is predicted.

In order to begin the process of predicting throughput and efficiency given flow and application core bindings, the input features need to be fed into the ANN-Frames, shown in Figure 5.3. The ANN has 7 input features (CSwitch can also be avoided and similar results have been obtained with 6 inputs, but here it is used for better variation), one hidden layer of 3 neurons and an output layer to predict frame values. All ANNs work on similar strategies as explained in 5.2. Some concepts of how ANN works were taken and studied from online resources [40]. Figure 5.3 shows data is collected from the predicted frame values. Then the model uses 7 input features in input layer (again 6 inputs can be used by avoiding Cswitch and similar results are achieved), one hidden layer with three neurons and one output layer to predict μsecs, the performance metric discussed in Chapter 2.4.

The ANN model, shown in Figure 5.4, uses 6 input features and 1 hidden layer to

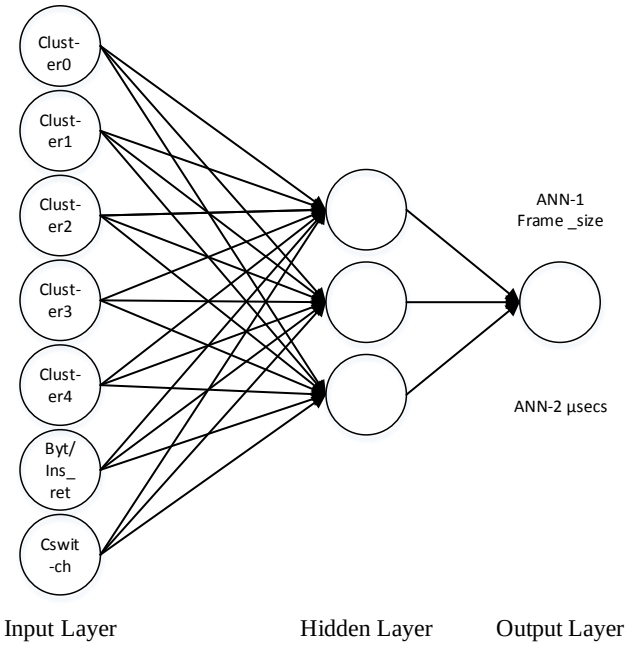

Figure 5.3. ANN predicting the frame size and usecs for given features

predict throughput.

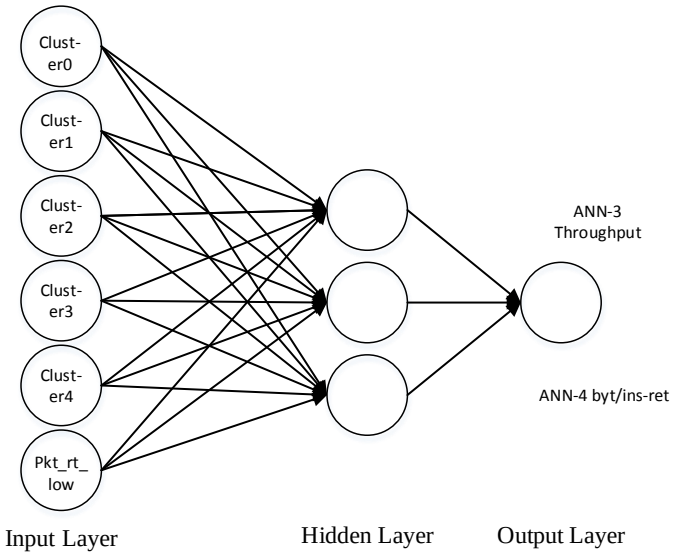

Figure 5.4. NN predicting Throughput and bytes/inst-ret(efficiency) for given features

Chapter 6

Results

The complete architecture of experimentation is shown in Figure 6.1. The results generated are divided in two main parts: analysis and prediction.

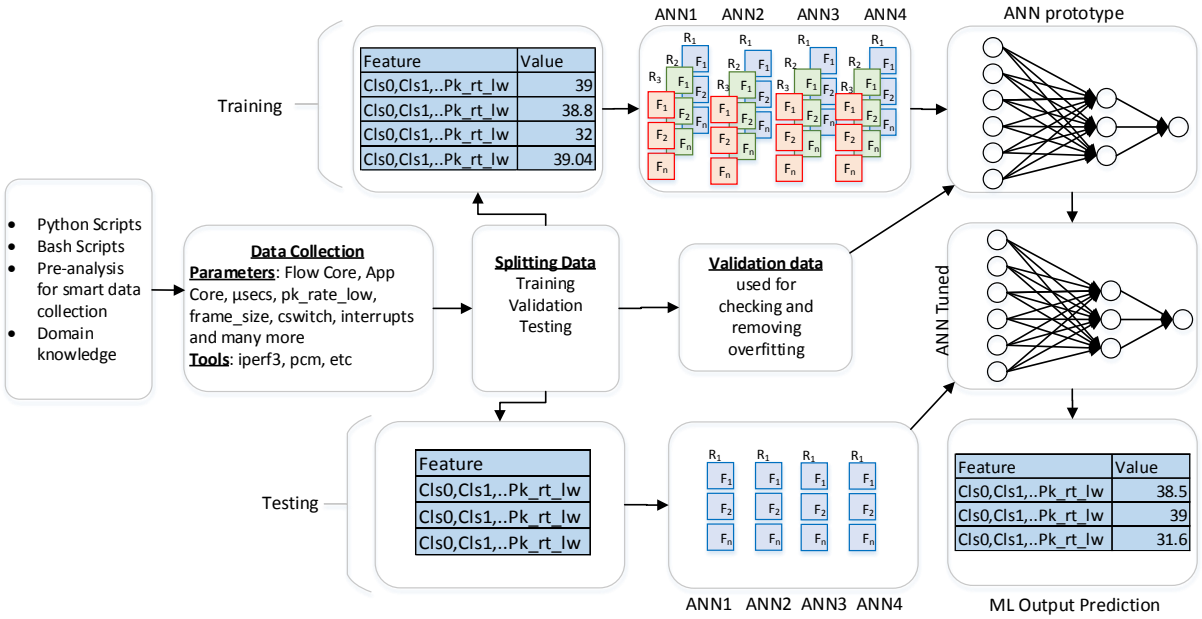

Figure 6.1. Experiment architecture

6.1 Data Analysis Results

Figure 6.1 shows how data is collected, post execution of python and bash scripts. Using the initial data set, `data_collection_1`, the goal is to identify the number of clusters present and eventually reduce or prune core combinations of similar properties. This

process is called efficient data analysis. The elbow method, PCA and k-means help us in this process. The `data_collection_1` has parameters like FlowCore, ApplicationCore, Context Switch, L3Hit, L2Hit, receive bytes and instructions retired. We convert receive bytes to Gbps and refer to it as *throughput*. Also we divide receive bytes by the number of instructions retired during that instance and refer to it as $efficiency$ or `bytes per instruction_retired`. The data is preprocessed before any analysis or before applying any ML algorithms. The general techniques used to process the data are scaling, normalization and one-bit-hot encoding. The general rule used for all parameters during processing of both `data_collection_1` and `ML_data_collection_2` are:

- Scaling: For the features which have a definite scale, eg. throughput that ranges from 0-40.
- Normalizing: All the features which have no definite scale and can have different values. We use MinMaxScalar from $scikit-learn$ preprocessing library [41].
- One-bit-hot encoding: Used for FlowCore and ApplicationCore number. The process makes sure that features keep their identity as numbers and do not introduce a bias while learning.

6.2 Machine Learning-based ANN Prediction Results

Data containing important parameters from `data_collection_1` are analyzed and multiple iterations of k-means are executed with k values from 0 to 10 to generate an elbow curve (see Figure 6.2). The figure shows how the optimum value of k is 5 as the cluster error, or the the distance between one cluster and another, becomes 0. We now process the input vector containing the parameters from `data_collection_1` and reduce its dimensionality to 2 using principal component analysis. Using the k value reported by the elbow-curve, a new input vector of reduced dimension is created and passed to the k-means clustering algorithm, assigning labels to clusters as shown in Figure 6.3. The list of core combination are chosen based on the centroid, calculated via using k-means. The core, or the nearest centroid in its respective cluster, is chosen. The core combination chosen for further `ML_data_collection_2` is listed below with the color code, which

represents all the Flow and Application core combinations in a particular cluster. Note: Use Table 6.1 as reference to read Figure 6.3 which shows all clusters.

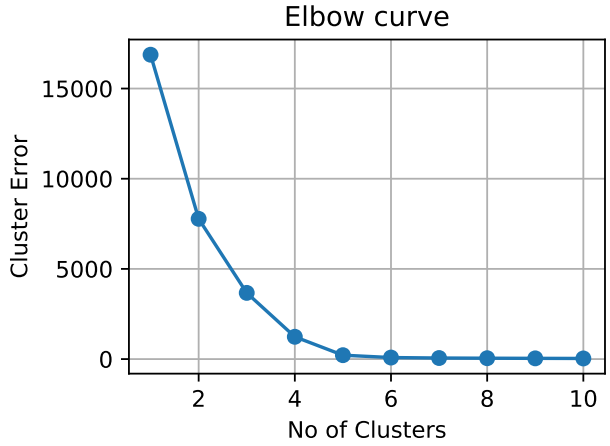

Figure 6.2. Cluster-k value for different iterations

After pruning the data to represent similar characteristics in terms of throughput and efficiency, `ML_data_collection_2` uses `iperf3`, `pcm.x`, and the `/proc` file system to collect important parameters such as FlowCore, ApplicationCore, receive-bytes, `rx-usecs`, `packet-rate-low`, `instructions_retired`, frame size, etc. The `ML_data_collection_2` is divided into three data extraction processes namely `ML_frames_data_collection_2.1`, `ML_usecs_data_collection_2.2` and `ML_pkt_rt_low_data_collection_2.3`. These 3 scripts collect important parameters with few common parameters against selected cores shown in Table 6.1. The data is preprocessed with the aforementioned methods and partitioned into three sets: Training, Testing and the Validation. Figure 6.1 shows that once the training data is collected, which constitutes multiple runs R1, R2, R3 and many more depending on amount of iterations, it is fed to respective ANN-models in the order: ANN1, ANN2, ANN3 and ANN4. The output data is validated using unexposed testing data and respective ANN (Figure 6.1).

6.2.1 ANN-Frames:

The `ML_frames_data_collection_2.1` is extracted. The value of packet-rate-low is fixed to 0 and `rx-usecs` to a high value (i.e 512), extracting different `rx-frame` values.

AppCore,FlowCore	Color Code	Cluster Number	Combination meaning
4,5	Orange	1	Different_Socket_Different_Core
0,4	red	3	Same_Socket_Different_Core
1,6	Blue	0	Different_Socket_Different_Core
5,7	pink	4	Same_Socket_Different_Core
0,0	green	2	Same_Socket_Same_Core

Table 6.1. Color Code Mapping (to be read with Figure6.3)

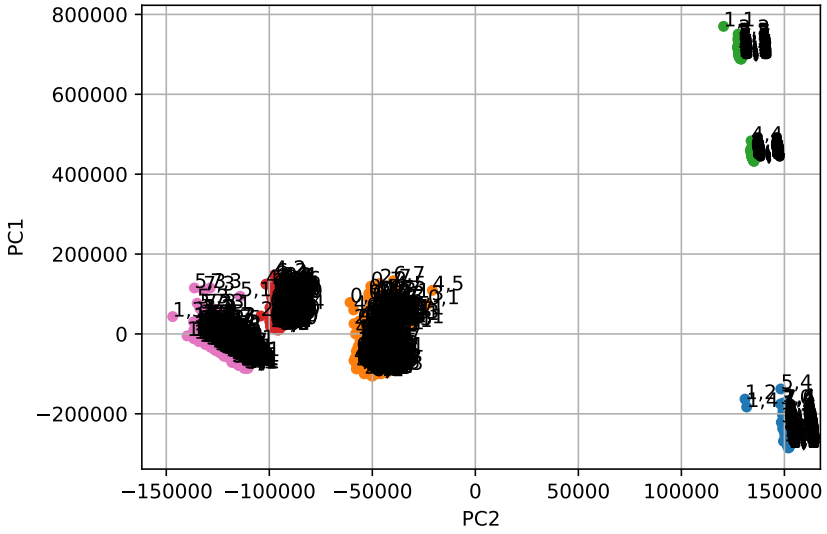

Figure 6.3. Kmeans after PCA

Here the `ML_frames_data_collection_2.1` data is fed to ANN-1. Table 6.2 shows the prediction results, where the frame-size is plotted against the highest bytes/instruction-retired(efficiency), passed to ANN-2 model.

6.2.2 ANN-`rx-usecs`:

Now, during `ML_usecs_data_collection_2.2`, the packet-rate-low is again set to 0 and selected frame-size from the previous model is set to collect data for different values of `rx-usecs` in the range 0-13. The predicted values of `rx-usecs` is shown in Table 6.3. We select the `rx-usecs` value against the highest bytes/instruction-retired (efficiency) obtained from ML prediction and data analysis.

Case	ML Model Learnt Value	Actual machine Value
Cluster	Frame	Frame
0	36	36
1	36	37
2	26	27
3	54	55
4	27	28

Table 6.2. Predicted vs. hand-tuned rx-frames value

Case	ML Model Learnt	Actual machine Value
Cluster	usec	usec
0	7	7
1	8	7
2	7	8
3	8	9
4	9	8

Table 6.3. Predicted vs. hand-tuned `rx-usecs` values

6.2.3 ANN-Throughput and ANN-bytes/inst-ret(Efficiency):

Both selected (tuned) values of `rx-frames` and `rx-usecs` are used and the final script for `ML_pkt_rt_low_data_collection_2.3` is executed for different values of `pkt-rate-low`. Figure 6.4 shows how validation data helps in avoiding over-fitting for throughput prediction. It can be seen that over-fitting may occur post 125 iterations and hence training is done accordingly. Similarly, validation data is used for prediction of `bytes per instructions_retired`. The RMSE value of 10^{-1} for the designed model shows that the prediction of *throughput* and `bytes per instructions_retired` is of high accuracy for unexposed testing data. Testing data is data extracted from live runs on an actual machine. The predicted values of *throughput* and `bytes per instruction_retired` by the ANN-3 model and ANN-4 model respectively on unexposed testing data is shown in Table 6.4.

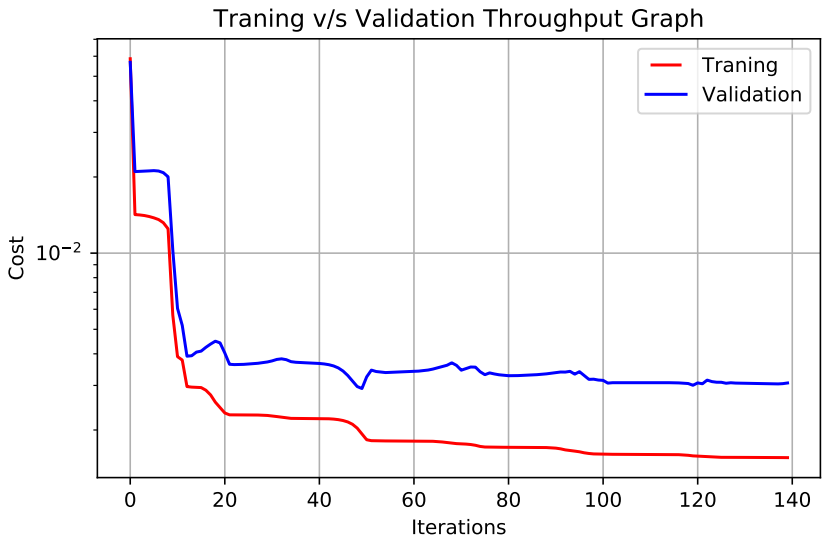

Figure 6.4. Training vs. Prediction graph for achieved throughput.

Thus, the overall summary of `data_collection_2` of important parameters needs to be tuned to fix `pkt-rate-low` to 0 and `rx-usecs` to a high value (e.g. 512). Additionally, data is extracted via `ML_frames_data_collection_2.1` for different values of frames. Then with this collected data, the frame-size value is selected against the highest bytes/instruction-retired(efficiency) obtained from the ML prediction and data analysis.

Cases	ML Model Learnt Value		Actual machine Value	
Cluster	Throughput	Bytes/Inst_ret	Throughput	Bytes/Inst_ret
0	31.96	2066345.67	32.93	2017785.452
1	39.31	1932287.241	39.02	1935329.628
2	38.25	2257874.84	38.33,	2218625.807
3	39.31	1781183.77	39.0	1863459.416
4	39.07	1752674.038	39.04	1781409.302

Table 6.4. Prediction vs. Actual data

During the `ML_usecs_data_collection_2.2`, the frame size is selected and data is collected for different values of `rx-usecs`. Similarly, we select `rx-usecs` value against the highest `bytes per instruction_retired` obtained from ML prediction and data analysis. Both the selected (tuned) values of `rx-frames` and `rx-usecs` are used and

the final script for `ML_pkt_rt_low_data_collection_2.3` is executed for different values of `pkt-rate-low`. The final prediction by ANN-Throughput for *throughput* and ANN-Bytes/Instruction-retired for `bytes per instructions_retired` is done by using `ML_pkt_rt_low_data_collection_2.3`.

One of the critical takeaways is that we were successfully able to predict the range for the `pkt-rate-low` coalescing parameter, which impacts the throughput for the same core same socket case (diagonal case). As seen in Table 4.2, the same core same socket (diagonal) entry had throughput in the range of 26-28 Gbps and our model was able to learn the best range which can predict the value up to 38.7 Gbps and help the user tune parameters accordingly.

Both ANN models learn from different values of $pkt - rate - low$ and correspondingly predict *throughput* and bytes per instruction_retired (efficiency). This helps users understand and find values of *throughput* and `bytes per instructions_retired` for current settings. To have high output values, one can test with different parameter values, which are tuned by supplying parameters as features to the ANN-models and get the predicted output. Thus NESCAT tool which is built upon earlier discussed methods, will help tune a parameter to a particular value or range for which the best output values can be achieved.

Chapter 7

Conclusion

In this paper we introduced NESCAT, a ML-based end-system characterization and tuning tool. In the process, we presented a scheme for exhaustive automated end-system characterization for high-speed network workloads. Furthermore, we introduced the notion of using ML methods to prune input data in order to automatically discover and represent NUMA boundaries on a physical end-system. We also applied Artificial Neural Networks (ANNs) to systematically tune core affinity and interrupt coalescing parameters. The result is more predictable and overall better performance for an end-system facing several internal throughput bottlenecks.

In addition to taking an introspective look into the sources of affinity-driven bottlenecks, we presented an overview of the benefits and drawbacks of interrupt coalescing parameter settings, and provide insights into the effects of different interrupt coalescing settings on each other and end-system network processing efficiency.

NESCAT leverages ML techniques including Artificial Neural Networks to both interpret vast amounts of performance characterization data and provide tuning parameters that increase the performance efficiency and predictability. It can be concluded from the ML data analysis and from all designed ANNs that the RMSE value is low enough that the prediction of $throughput$ and $bytes/instructions-retired$ is accurate when compared with actual data extracted from live runs on a machine. Thus the results generated in this work establish a baseline and it can be concluded that the tuning of parameters via a ML model is a good choice in practical scenarios.

7.1 Limitation

NESCAT currently needs some human intervention between consecutive scripts in order to set system variables and pass parameters between scripts. We are engaged in the process of fully automating NESCAT. In the near future, the application will be expanded to detect system variables and automatically set tuning parameters. Currently, NESCAT is only capable of explicitly setting coalescing parameters. In the future, we plan to expand this functionality to a core affinity blacklist as well as tuning additional end-system parameters.

7.2 Future Work

It can be seen that ML provides unique opportunities in the fields of end-system tuning and overall macro-architecture performance engineering. As a part of the future work, further work can be done in creating a more robust ML model which can learn from an even larger number of input features towards finding the best possible output values important for data transmission in the area of high-speed interconnects and HPC. Also, this model could be tested on an HPC testbed.

With the current research in predicting important parameters like throughput and efficiency, the work can be extend for prediction of latency and even other categories like tail latency, energy and others. Some of the important areas which are either unexplored or partially unexplored in the related work chapter is identifying the critical parameters on a fly i.e. online learning. As a part of futuristic goal, online learning will help in the dynamic process of identification of important system parameters, auto-tune them and predict the best output for each of performance parameters. This approach will be an alternative to throwing in a huge amount of data to the system and then making the model learn. It would be better to create a model which trains in batches based on importance of parameters. The step-1 of this model will understand and rank the critical parameters. It will collect the data of parameters rank wise and correspondingly train itself in that same order. This part of training will be on the lines of batch gradient descent where learning can be done on the fly as new data arrives. The criteria to achieve best output

for different performance parameter is known and hence the model could be trained as when to stop on a parameter and jump to another. The model should then proceed to the direction where multiple system parameters need to be tuned after completion of step-1. This model will be able to function in different kinds of systems; hence it will be portable and scalable in nature.

References

[1] H. Newman, I. Legrand, A. Mughal, R. Voicu, D. Kcira, and J. Bunn, "High speed scientific data transfers using software defined networking," in *Innovating the Network for Data Intensive Science (INDIS), SC*, 2015.

[2] A. B. Yoo, M. A. Jette, and M. Grondona, *SLURM: Simple Linux Utility for Resource Management*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. [Online]. Available: https://doi.org/10.1007/10968987_3

[3] O. Yildiz, M. Dorier, S. Ibrahim, R. Ross, and G. Antoniu, "On the root causes of cross-application i/o interference in hpc storage systems," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 750–759.

[4] D. Clark, "The design philosophy of the darpa internet protocols," in *Symposium Proceedings on Communications Architectures and Protocols*, ser. SIGCOMM '88. New York, NY, USA: ACM, 1988, pp. 106–114. [Online]. Available: http://doi.acm.org/10.1145/52324.52336

[5] N. Hanford, V. Ahuja, M. Balman, M. K. Farrens, D. Ghosal, E. Pouyoul, and B. Tierney, "Characterizing the impact of end-system affinities on the end-to-end performance of high-speed flows," in *Proceedings of the Third International Workshop on Network-Aware Data Management*, ser. NDM '13. New York, NY, USA: ACM, 2013, pp. 1:1–1:10. [Online]. Available: http://doi.acm.org/10.1145/2534695.2534697

[6] N. Hanford, V. Ahuja, M. Farrens, D. Ghosal, M. Balman, E. Pouyoul, and B. Tierney, "Analysis of the effect of core affinity on high-throughput flows," in *Proceedings of the Fourth International Workshop on Network-Aware Data Management*, ser. NDM '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 9–15. [Online]. Available: http://dx.doi.org/10.1109/NDM.2014.10

[7] ESnet, "Linux tuning, http://fasterdata.es.net/host-tuning/linux."

[8] R. L. Thorndike, "Who belongs in the family?" *Psychometrika*, vol. 18, no. 4, pp. 267–276, Dec 1953. [Online]. Available: https://doi.org/10.1007/BF02289263

[9] S. Lloyd, "Least squares quantization in pcm." Marray Hill, USA: Bell Telephone Laboratories Paper, 1957.

[10] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. 5th Berkeley Symposium*, 1967, pp. 281–297.

[11] J. Hartigan and M. Wang, "A k-means clustering algorithm," in *Applied Statistics*, 1979, pp. 100–108.

[12] J. Wei, Q. Cheng, R. V. Penty, I. H. White, and D. G. Cunningham, "400 gigabit ethernet using advanced modulation formats: Performance, complexity, and power dissipation," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 182–189, Feb 2015.

[13] B. Acun, P. Miller, and L. V. Kale, "Variation among processors under turbo boost in hpc systems," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: ACM, 2016, pp. 6:1–6:12. [Online]. Available: http://doi.acm.org/10.1145/2925426.2926289

[14] X. Yang, J. Jenkins, M. Mubarak, R. B. Ross, and Z. Lan, "Watch out for the bully!: Job interference study on dragonfly network," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 64:1–64:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014990

[15] S. A. McKee, "Reflections on the memory wall," in *Proceedings of the 1st Conference on Computing Frontiers*, ser. CF '04. New York, NY, USA: ACM, 2004, pp. 162–. [Online]. Available: http://doi.acm.org/10.1145/977091.977115

[16] T. Herbert, "rfs: receive flow steering, september 2010," http://lwn.net/Articles/381955/.

[17] A. Foong, J. Fung, D. Newell, S. Abraham, P. Irelan, and A. Lopez-Estrada, "Architectural characterization of processor affinity in network processing," in *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*. IEEE, 2005, pp. 207–218.

[18] G. Narayanaswamy, P. Balaji, and W. Feng, "Impact of network sharing in multicore architectures," in *Computer Communications and Networks, 2008. ICCCN'08. Proceedings of 17th International Conference on*. IEEE, 2008, pp. 1–6.

[19] X. Chang, J. Muppala, Z. Han, and J. Liu, "Analysis of interrupt coalescing schemes for receive-livelock problem in gigabit ethernet network hosts," in *Communications, 2008. ICC '08. IEEE International Conference on*, May 2008, pp. 1835–1839.

[20] B. Weller and S. Simon, "Closed loop method and apparatus for throttling the transmit rate of an ethernet media access controller," Aug. 26 2008, uS Patent 7,417,949. [Online]. Available: http://www.google.com/patents/US7417949

[21] S. Hu, Y. Zhu, P. Cheng, C. Guo, K. Tan, J. Padhye, and K. Chen, "Deadlocks in datacenter networks: Why do they form, and how to avoid them," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, ser. HotNets '16. New York, NY, USA: ACM, 2016, pp. 92–98. [Online]. Available: http://doi.acm.org/10.1145/3005745.3005760

[22] W. Su, L. Zhang, D. Tang, and X. Gao, "Using direct cache access combined with integrated nic architecture to accelerate network processing," in *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, June 2012, pp. 509–515.

[23] K. C. e. a. Oceane Bel, "Capes: Computer automated performance enhancement system," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[24] J. S. Breese and R. Blake, "Automating computer bottleneck detection with belief nets," in *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, ser. UAI'95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 36–45. [Online]. Available: http://dl.acm.org/citation.cfm?id=2074158.2074163

[25] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, ser. HotNets '16. New York, NY, USA: ACM, 2016, pp. 50–56. [Online]. Available: http://doi.acm.org/10.1145/3005745.3005750

[26] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. Brandt, V. J. Leung, M. Egele, and A. K. Coskun, *Diagnosing Performance Variations in HPC Applications Using Machine Learning*. Cham: Springer International Publishing, 2017, pp. 355–373. [Online]. Available: https://doi.org/10.1007/978-3-319-58667-0_19

[27] J. Klinkenberg, C. Terboven, S. Lankes, and M. S. Mller, "Data mining-based analysis of hpc center operations," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2017, pp. 766–773.

[28] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 571–582. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi

[29] M. Wang, Y. Cui, X. Wang, S. Xiao, and J. Jiang, "Machine learning for networking: Workflow, advances and opportunities," *CoRR*, vol. abs/1709.08339, 2017. [Online]. Available: http://arxiv.org/abs/1709.08339

[30] A. Ganapathi, K. Datta, A. Fox, and D. Patterson, "A case for machine learning to optimize multicore performance," in *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, ser. HotPar'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855591.1855592

[31] E. S. Network, "Esnet, http://www.es.net/."

[32] E. Dart, L. Rotman, B. Tierney, M. Hester, and J. Zurawski, "The science dmz: A network design pattern for data-intensive science," in *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2013, pp. 1–10.

[33] T. Ito, H. Ohsaki, and M. Imase, "Gridftp-apt: Automatic parallelism tuning mechanism for data transfer protocol gridftp," 2006.

[34] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, and S. Tuecke, "Gridftp: Protocol extensions to ftp for the grid," *Global Grid ForumGFD-RP*, vol. 20, 2003.

[35] V. Ahuja, A. Banerjee, M. Farrens, D. Ghosal, and G. Serazzi, "Introspective end-system modeling to optimize the transfer time of rate based protocols," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, ser. HPDC '11. New York, NY, USA: ACM, 2011, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/1996130.1996140

[36] V. Ahuja, M. Farrens, and D. Ghosal, "Cache-aware affinitization on commodity multicores for high-speed network flows," in *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*. ACM, 2012, pp. 39–48.

[37] V. Ahuja, D. Ghosal, and M. Farrens, "Minimizing the data transfer time using multicore end-system aware flow bifurcation," in *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*. IEEE, 2012, pp. 595–602.

[38] B. Gregg, "The flame graph," *Commun. ACM*, vol. 59, no. 6, pp. 48–57, May 2016. [Online]. Available: http://doi.acm.org/10.1145/2909476

[39] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: A structure for efficient numerical computation," *Computing in Science Engineering*, vol. 13, no. 2, pp. 22–30, March 2011.

[40] S. Welch, http://www.welchlabs.com/.

[41] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Nov. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1953048.2078195

Harshvardhan Singh Chauhan
December 2017
Master of Science

Applying Machine Learning to Identify NUMA End-System Bottlenecks for Network I/O

Abstract

Performance bottlenecks across distributed nodes, such as in high performance computing grids or cloud computing, have raised concerns about the use of Non-Uniform Memory Access (NUMA) processors and high-speed commodity interconnects. Performance engineering studies have investigated this with varying degrees of success. However, with continuous evolution in end-system hardware, along with changes in the Linux networking stack, this study has become increasingly complex and difficult due to the many tightly-coupled performance tuning parameters involved. In response to this, we present the Networked End-System Characterization and Adaptive Tuning tool, or NESCAT, a partially automated performance engineering tool that uses machine learning to study high-speed network connectivity within end-systems. NESCAT exploits several novel techniques for systems performance engineering. These include using k-means clustering and Artificial Neural Networks (ANNs) to effectively learn and predict network throughput performance and resource utilization for end-system networks.

NESCAT is a unique tool, different from other previously designed applications. It is based on of machine learning and clustering techniques on NUMA core-binding cases. This work focuses on predicting optimal Network Interface Controller (NIC) parameters for performance predictability, which is a necessity for complex science applications. Through experiments, we are able to demonstrate the uniqueness of this technique by achieving high accuracy rates in predicted and actual performance metrics such as throughput, data rate efficiency, and frame rates. Our system is able to ingest large amounts of data to produce results within 2 hours for a machine with an 8-core end-systems. The root mean square error of the designed model is around 10^{-1} and thus predicts output efficiently when compared to live run data on an actual machine.