Methods and Applications of Synthetic Data Generation

A Dissertation by
Jason William Anderson
December 2021

Submitted to the graduate faculty of the
School of Computing
in partial fulfillment of the requirements
for the Dissertation and
subsequent Ph.D. in Computer Science

Approved by:
Dr. Amy Apon, Committee Chair
Dr. Ken Kennedy, Committee Co-chair
Dr. Jim Martin
Dr. Kuang-Ching Wang

Abstract

The advent of data mining and machine learning has highlighted the value of large and varied sources of data, while increasing the demand to make data accessible for academic research and the development of digital infrastructure. However, in many cases the sharing of large collected datasets carries a risk of exposing sensitive personal or proprietary information. An alternative approach is to create synthetic data that is similar to the original data but has values that are not obtained by direct measurement. Ideally, synthetic data captures the structural and statistical characteristics of the original data without revealing personal or proprietary information contained in the original data set.

In this dissertation, we use examples from original research to show that, using appropriate models and input parameters, synthetic data that mimics the characteristics of real data can be generated with sufficient rate and quality to address the volume, structural complexity, and statistical variation requirements of research and development of digital information processing systems.

First, we present a progression of research studies using a variety of tools to generate synthetic network traffic patterns, enabling us to observe relationships between network latency and communication pattern benchmarks at all levels of the network stack, and to show that variation in latency between nodes of a distributed HPC application can lead to performance degradation.

We then present a framework for synthesizing large scale IoT data with complex structural characteristics in a scalable extraction and synthesis framework, and demonstrate the use of generated data in the benchmarking of IoT middleware.

Finally, we detail research on synthetic image generation for deep learning models using 3D modeling. We find that synthetic images can be an effective technique for augmenting limited sets of real training data, and in use cases that benefit from incremental training or model specialization, we find that pretraining on synthetic images provided a usable base model for transfer learning.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Data has been described as "the new oil of the digital economy" [190]. The collection, storage, transportation, analysis, and presentation of data have become a fundamental aspect of our world and the digital systems that surround us. From the mobile devices that collect data about our lives, to the networks transporting data to server farms for deep learning and filtration and analysis, and to the myriad of ways that data is used to enrich our lives, data is all around us. The digital computing systems that underlie these processes have emerged as a pinnacle of human engineering.

While there are many ways that valuable business information can be extracted from the vast quantities of data generated by our society, there is also an increasing demand for accessible data for academic research and the development of digital infrastructure. Systems designed for the collection, storage, transportation, analysis, and presentation of digital data typically require large amounts of that data somewhere in their development cycle. For example, fundamental research in network communication patterns of complex systems requires the simulation or observation of those systems. Databases and middleware systems need vast amounts of data to prove their performance and stability under extreme access and load conditions. Artificial intelligence and, in particular, deep learning models require large amounts of relevant data to extract patterns for later application.

> "The ingestion and integration of raw data, the extraction of valuable business information from the raw data, and planning for infrastructure capacity and analytic capability for analyzing this data are new challenges in the era of big data [119]."

The availability of quality data for research and the development of digital systems can,

in many cases, be problematic. Data that is collected by industrial or commercial entities may contain information that is proprietary and confidential to their business. The sharing of data belonging to government and medical institutions could risk exposing confidential personal records, even when identifying values are obfuscated or omitted. In some cases, the storage of very large sets of reproducible measurements may be costly or impractical. In other cases, the necessary data might simply not exist in the required scale and variation.

Synthetic data, i.e. data that has been created rather than collected, can be used to address the issues that arise with the collection and use of real data. Ideally, synthetic data sets have characteristics that are similar to the original data but have values that are not obtained by direct measurement. Examples to address the previously described used cases include the generation of synthetic network traffic for network communication research rather than relying on access to real systems; creating structurally and statistically similar data to test middleware systems rather than risking exposure of potentially sensitive personal information; and generating varied artificial images useful for training deep learning models to avoid costly labelling of real images. In many cases it can be less costly in time and risk to create the volumes of data needed for research and in the development of digital systems.

> "Synthetic data will accelerate the creation of complex and layered learning analytics infrastructure and help to address the ethical and privacy risks involved during service development [36]."

Other important use cases for synthetic data include allowing research on enterprise data collections to be conducted by third parties while protecting sensitive information, enabling researchers to evaluate experimental methodologies prior to granting access to sensitive data, modeling of rare events that are impractical to capture in observed measurements, and simulation of environments to efficiently train artificial intelligence systems, such as training autonomous vehicles without the risk and expense of real driving conditions.

The creation, validation, and use of synthetic data is not without challenges. Data privacy is an important issue that carries sizeable monetary, legal, and ethical risk if the anonymization measures are improper or insufficient. In some cases it can be difficult or computationally intractable to capture the aggregated structural and statistical characteristics of the original data when there are complex interdependencies between values or record groups. Data architects must consider

the tradeoffs between the level of statistical accuracy of the generated data, the need to maintain anonymity of the original data in the generated set, and the computational complexity and run time of the data generation process.

This work will explore a sample of these use cases to represent the much larger applicability of synthetic data generation. Different aspects and challenges of using synthetic data will be examined, such as information privacy, performant generation techniques, and potential shortcomings. It is the goal of this work to examine how synthetic data plays a role in current and future digital information systems, to explain contributions made over the course of the research, and to describe how further research in this area is warranted as new use cases emerge.

1.1 Thesis Statement

Using appropriate models and input parameters, synthetic data that mimics the characteristics of real data can be generated with sufficient rate and quality to address the volume, structural complexity, and statistical variation requirements of research and development of digital information processing systems.

1.2 Contributions and Thesis Organization

This dissertation provides examples drawn from original research that support the thesis statement, and is organized as follows. Chapter 2 will give background information on methods, applications, and challenges of synthetic data generation. Chapter 3 will explore techniques and applications of synthetic benchmark data related to high performance computer networking. Chapter 4 will present a scalable framework for generating complex synthetic data and using generator output patterns to benchmark cloud messaging middleware systems. Chapter 5 details the contributions made toward synthetic image generation for deep learning models. Finally, Chapter 6 will summarize our contributions and present the conclusions we have drawn.

Chapter 2

Background

Synthetic data is a very broad term. It can be defined simply as data that is *created* rather than *observed*, or to quote the McGraw-Hill Dictionary of Scientific and Technical Terms [131], "any production data applicable to a given situation that is not obtained by direct measurement." We may reason that the qualifier of "applicable to a given situation" restricts our scope to data that embodies some characteristics and mimics the form of real observed data. However, as it would require many more pages to adequately discuss all of the topics that could fit that description, we will focus instead on a narrower definition that fits the common understanding in the field of computer science for the purposes of this dissertation:

> Synthetic data is generated digital data that is wholly or partly disjoint from real data, is an ordered series of states or unordered set of records that each conform to a particular structure and syntax, is able to be generated at scale, and exhibits features relevant to the intended application that mimic real data in statistical distribution and interdependency.

While more restrictive, most intuitive examples of synthetic data will fall under this definition. The umbrella of synthetic data encompasses data that is generated and stored, such as collections of tabular data, documents, and images, as well as data that is algorithmically generated on demand, such as computer simulations and physical modeling.

To understand and discuss both real and synthetic data in more precise terms, we will abide by the following definitions in this work:

- *record* - an individual set of labelled values, also technically known as a singular *data*[1], such as a structured document or a *row* in tabular data
- *data set* - a collection of records, commonly used interchangeably with *data* in the plural form
- *attribute* - the values of a dataset sharing a common label, akin to a *column* in tabular data
- *dimensionality* - the number of different attributes in a data set
- *feature* - a joining of one or more attributes or subfeatures of a dataset. As it can consist of a single attribute, *feature* is often used in place of *attribute* in machine learning contexts. In this work, we will prefer *feature* and be specific where appropriate.

2.1 Feature Selection and Extraction

Feature selection is the process of identifying a set of data attributes that form the important features of a dataset. Feature selection is commonly used in machine learning to limit input dimensionality, remove irrelevant or redundant attributes, and filter out noisy data, with the goal of improving generalization capacity, learning speed, or reducing model complexity [107]. The dimensionality reduction of feature selection has become even more important in recent years as the world becomes saturated with a broad variety of data sources, collectively known as "big data" [44]. Feature selection algorithms are commonly evaluated on their stability, or the mean probability of models trained on feature-selected subsets of the input data to agree in their predictions [175, 102].

Feature selection may be complemented by *feature extraction* [80], which creates new features as functions of other features, where features found through selection map directly to existing attributes. Feature extraction is most commonly employed in data that has relatively few samples compared to the number of attributes. Examples of domains where feature extraction is useful include the analysis of handwriting, which may include thousands of distinct features with very few samples, and convolutional neural networks, which iteratively aggregate features with each encoding layer.

In the context of synthetic data, feature selection techniques are used to identify attribute dependencies that should be modeled and reproduced in the output data set. Additionally, high

[1]We will **not** be using *data* as a singular noun.

dimensionality in real-world data can be computationally intensive to reproduce in generated synthetic analogues as the potential dependencies grow factorially. Limiting the selection of modeled attributes to those that are important to the synthetic data use case can dramatically improve performance and quality.

Extracted features in the real dataset can introduce other challenges to generating quality data if the features they derive from are also reproduced, as this introduces additional feature relationships that should be retained in the output. This could be the case when attributes forming an extracted feature also form the basis for other features.

2.2 Applications

The many use cases for synthetic data are widely varied, and become difficult to separate from the uses of data in general. Rather than attempting to cover examples of such a broad scope, we instead present a loosely classified selection of applications where synthetic data can be of particular value.

2.2.1 Privacy

Ensuring that confidential data sources are protected is an important component to data-driven research. Agencies that collect confidential data typically strive to release data that protects the confidentiality of the subject's identities, retains properties that make it informative for analysis, and is straightforward for secondary analysts to use [145]. Simple perturbation methods such as aggregation, recoding, record-swapping, adding random noise, and omission of sensitive values are typically not satisfactory; statistical analysis and clever data mining techniques combining multiple data sources have been used to reverse these techniques when there is enough peripheral information to make inferences [167, 123, 124]. Statistical disclosure control (SDC) is a body of techniques used to ensure that individual data subjects are not identifiable from records made available to researchers.

In one of the fundamental works in this field, Rubin [151] proposed using multiple imputation, i.e., averaging multiple samples from a distribution for each value, to generate completely synthetic microdata with statistical properties similar to the original dataset. Fully synthetic data offered a means to release data that did not represent real individuals, and therefore protected the data sources.

Research on the privacy afforded by synthetic microdata [142, 141, 13, 54, 46] led to refinements of principles-based SDC techniques and measurements of data privacy quality, such as inferential disclosure probability to describe complete datasets and differential privacy ratio to measure the database query risk [65]. A body of literature was developed exploring the nuances and risks of generating partially synthetic datasets [143, 144, 139]. A formal notion of *plausible deniability* emerged as a quality measure of differentially private synthetic data generated with computationally tractable methods [39].

In many cases, care must be taken to ensure that SDC techniques do not anonymize data in ways that alter the interdependence of features. For example, in synthesizing electronic medical records for researchers studying disease outbreak detection, Buczak et al. [48] had to carefully preserve the relationship between patient backgrounds and care that was received, so that care patterns would be present in the synthetic records.

2.2.2 Data Science Research

The meta aspects of synthetic data can be valuable from a research perspective; as an example, consider this dissertation. Fundamental research on synthetic data enables the tools for applying that research in the many applicable fields. In [43], the authors use high dimensional synthetic datasets to evaluate feature selection techniques without the interference of distractors such as noise, attribute interaction, and irrelevant or redundant features. Burgard et al. [50] and Raab et al. [139] use privileged access to longitudinal population data to develop new mechanisms for generating high quality synthetic social science datasets.

2.2.3 Machine Learning

In some analysis applications, the frequency of records that exemplify a particular feature value need to be normalized to some degree, such as when modeling rare events for training machine learning systems. Gaber et al. [72] describe using rare event normalization in generated mobile payment log files to train a system to detect fraudulent transactions.

In other use cases, datasets simply need to be of sufficient size to ensure proper functioning of the target application. This is particularly important in the field of deep learning, where having a large and varied set of examples is necessary to train a model that generalizes well with new data.

Data augmentation has long been used to multiply the size of training datasets while increasing variation [155]. The current state of the art is for data scientists to employ simple augmentation techniques such as randomly cropping, warping, and channel shifting images to multiply small training datasets.

Fully synthetic datasets can be used to train machine learning models with varying degrees of success, depending on how well the generated domain maps to the real domain. Several researchers have had success in training text recognition models with rendered text images in natural scenes [97, 79], using tools to simulate a variety of fonts, shading, coloring, distortion, and noise values. 3D modeling of synthetic images has been used to train robot arm controllers to detect objects in piles of similar objects [49]. Others have used generated images of humans randomly placed in a camera's field of view to train models to estimate the size of crowds [67, 182]. In Chapter 5 we present results on the effectiveness of training with fully synthetic images and how training with mixed real and synthetic datasets can improve image segmentation models.

2.2.4 Third Party Analysis

A combination of business and technical reasons makes data analysis difficult from within an enterprise computing environment, particularly if the data does not fit a common model with widely available tools. This is especially true in the case of proprietary connected measurement devices such as industrial sensors, where data can be complex, varied, and inconsistent. The current state-of-the-art is that there is no common, comprehensive solution for data wrangling problems as each company has its own unique sets of data with unique attributes. Enterprise computing budgets are typically devoted to supporting Service Level Agreements and providing stable data operations [51], leaving little budget and computing resources for experimental research and development on captured data.

One solution for experimental research is to utilize academic partners or commercial cloud providers who are equipped with large scale computational resources. However, privacy constraints and security policies often bar the transfer of data to third parties. Synthetic data presents itself as a solution in this scenario, where a sufficient anonymization process can satisfy the restrictions placed on data outsourcing.

Data stewards can also opt to share synthetic data as tool for researchers to gather preliminary findings and validate their experiment design. An example comes from the area of patient-derived health information, where privacy restrictions require controlled access. Benaim et al. [34]

detail their methods for data anonymization to allow researchers to perform initial analysis before requesting access by an institutional review board, and find that using synthetic data is a powerful tool in shaping research hypotheses and estimating analyses without risking patient privacy.

In recent years, tools aimed at simplifying and generalizing the process of generating high quality anonymized synthetic datasets have emerged. These tools target a variety of data formats and use cases such as tabular data [129] and relational databases [132].

2.2.5 Benchmarking

Benchmarks typically generate data at some degree of scale, potentially with a configurable output rate, and are intended for evaluating data flow and processing of digital systems. For structured data, well-known sources of synthetic data are the two industrial general-purpose database benchmarks, TPC-C and TPC-H [2]. These benchmarks can generate arbitrary amounts of data. The generated data are inserted into flat tables with a number of predefined columns. Similar general-purpose benchmarks, which also generate synthetic data at scale, are BigBench [73] and YCSB [59].

Data generators can also be distributed, i.e., running in parallel on separate machines. A potential application is to multiply the potential output rate, such as the network packets generated by processes in a distributed denial of service attack. Another common use case is in distributed system benchmarking, such as in high performance computing environments. We describe the use of distributed synthetic benchmarks such as NPB and LAMMPS to observe network performance in Chapter 3.

2.3 Methods of Generation

Synthetic data generation can almost universally be separated into two phases: *kernel extraction* and *synthesis*. Broadly speaking, kernel extraction is the task of analyzing real data or simulation requirements to reduce the task to a set of algorithms and parameters. Synthesis is the invoking of the kernel algorithms to produce output based on the derived parameters. What follows is a representative sample of different generation mechanisms, classified by their application.

2.3.1 Structured Data

For tabular or structured data, where a single schema describes the constraints on a dataset, synthetic data generation is well studied and can be somewhat generalized. Many approaches exist for identifying and modeling value distributions and feature interdependencies, however. The simplest methods may simply fit independent statistical distributions to each feature and generate data by *imputation*, i.e., sampling values of the synthetic record from the related distribution. To address the problem of increased noise, one may use *multiple imputation*, or the aggregation of multiple synthetic datasets generated from the same kernel [151].

Relationships between features may be modeled in a number of ways. Synthetic reconstruction and combinatorial optimization are related approaches that apply conditional probability to impute from different distributions based on feature relations [89]. Increases in computational power led to the feasibility of more sophisticated machine learning techniques of feature relationship preservation, such as using support vector machines [64] and random forests [52] to generate categorical values in datasets where relationships between variables with many possible values are difficult to capture with standard parametric tools.

Other forms of structured data include images and time series measurements, where feature relationships can be described spatially within a record. Technically, any image transformation method can be classified as generating new synthetic image data, but one particularly useful application of image synthesis is the augmentation of training images in a deep learning pipeline. Image augmentation techniques range from basic image manipulations such as filters and geometrical transformations to sophisticated methods like adversarial training and neural style transfer [155]. Repeated presentations of training images with randomized augmentation parameters have the effect of expanding the source dataset with a high degree of variation.

2.3.1.1 Generative Adversarial Networks

Extracted feature relationships can be learned and used to inform particular transformations of the image to create new synthetic images. An interesting application of this technique is in Generative Adversarial Networks (GANs) [140], which have shown considerable promise as mechanisms for relationship preservation in generating anonymized synthetic datasets, particularly with structured data such as tabular records and images.

Park et al. [130] presented methods for generalizing tabular data with GANs using convolutional neural networks (CNNs) [130], which was built upon in [187] using recombinant CNNs to focus on preserving marginal distributions. Alzantot et al. [17] explored modeling phone vibration sensor time-series data using Long-Short-Term-Memory (LSTM) based generator and discriminator, though their preliminary work does not yet incorporate adversarial training.

GANs have been shown to be effective in generating differentially private synthetic data [29] and have applications in a variety of use cases, such as anonymizing private medical records [56, 31]. GAN models and trained weights, in essence, form a portable and transferable dataset generator which has applications in preserving data privacy [188].

GANs have applications in synthetic image generation, whether as part of a deep learning pipeline [155] or in creating a dataset for other purposes. Zhu et al. [194] use GANs as a form of augmentation, and demonstrate image-to-image translation models that can convert horses to zebras in the input. Other applications include using semantic label maps to generate photo-realistic image segments [183]. Shrivastava et al. [156] use a large dataset of synthetic human eye images as training input vectors to a GAN rather than random inputs, and achieved significant improvements in the model's resulting detection accuracy.

One issue common to synthesizing data with GANs is known as *mode collapse*, where the generator focuses on a few examples known to trick the discriminator. Synthetic records generated from a GAN in such a condition will tend to exhibit strong resemblance to those few examples, rather than capturing the statistical distributions of the training data.

Bayesian GANs [154] attempt to represent the posterior distribution over the generator and discriminator parameters by sampling the distribution at each training step, rather than finding the most likely parameter vectors. Their resulting models are able to recover complex multi modal distributions where standard GAN approaches fail. Further work has shown Bayesian GANs to exhibit output diversity comparable to more traditional techniques [29].

2.3.2 Semi-structured Data

The synthesis of semi-structured documents such as JSON or XML can be challenging, particularly when documents exhibit complex nested structures that depend on values within the document. One approach is to treat document structures as templates, and generate documents according to template example frequency. In Chapter 4.1, we detail our work on a platform for

scalable generation of synthetic semi-structured document data using the Hadoop platform.

Rule-based procedural generation is another possibility, especially to recreate complex and plausible data structures. The extracted kernel might be condensed to a set of schema segments with rules for reconstruction, along with the value distributions. The authors of [177] created a gene regulatory network generator that selects subnetworks from real examples and assembles new networks based on modeled interaction kinetics. The resulting topologies more closely approximate real regulatory networks than those created with alternative approaches such as random graph models.

2.3.3 Benchmarks

The underlying observation characteristics for a simulated benchmark might be modeled by an algorithm with rules for conditional generation based on external factors, such as the system clock or network interaction with peers. Examples include simple network traffic generators such as iPerf [10], where output is generated with content and frequency determined by application parameters. A more complex benchmark application is the NAS Parallel Benchmark suite [32], a distributed application that generates network traffic patterns modeled on common parallel algorithms used in high performance computing problems.

2.3.4 Simulations

Related to procedural generation is simulation, where the data generator mimics the relevant aspects of a real-world process to create an ordered series of outputs. One particularly interesting type of simulation with respect to synthetic data is in the creation of images for use in training deep learning applications. In this case, extraction refers to creating the 3D models and rules for scene simulation and rendering, and synthesis refers to the process of rendering images with conditions drawn from the input parameters. We present our original research on the viability of synthetic image data in deep learning applications in Chapter 5.

2.4 Measures of Quality

The quality of synthetic data and the generation processes can be measured in a number of ways.

Synthetic data can be measured in terms of *utility*, i.e., the similarity of the synthetic dataset to the training data or underlying population from which it was derived [28]. The *general utility* of a dataset compares the statistical distributions of the synthetic and the real, and can be expressed in terms of the propensity score mean-squared error (pMSE). The *specific utility* of a dataset measures the similarity of the results of analyses conducted with both the synthetic and real data, which can be expressed with measures such as confidence interval overlap [159]. These measures may not agree; it is possible to capture the distributions of the real data in the synthetic, and yet fail to capture feature relationships important to analysis tools.

Another measure of the similarity between real and synthetic data is to compare the performance differences in a simulation or system that uses the data. This measure is most useful when comparing datasets that have complex structures or dependencies between values or records, and depends on those complexities having significant effect on the system performance. For example, a collection of complex structured synthetic documents populating a document-oriented database such as MongoDB [55] should give search query benchmark times that are statistically similar to an identical database populated with the same number of real documents.

Chapter 3

Synthetic Data in Network Performance Testing

The high performance computing (HPC) community has a long and rich history of the study of the effect of network latency on parallel application performance [115, 87]. HPC applications can be complex and almost ubiquitously scaled across many interconnected compute nodes. It is well known that high performance execution of parallel applications demands that computational nodes be interconnected with low-latency networks.

Researchers use benchmarks such as the NASA Parallel Benchmark (NPB) [32] and Large-scale Atomic/Molecular Massively Parallel Simulator (LAAMPS) [137] that emulate the most common communication patterns of HPC applications to study the behavior of network traffic in complex systems. Administrators of such systems rely on similar benchmarks to validate systems and find areas for improvement. These benchmarks, which produce vast quantities of synthetic data from relatively simple algorithms, are a prime example of the applicability of synthetic data generation for performance testing.

In this chapter, we present our findings using synthetic communication patterns to expose underlying network characteristics and analyze the effects of a latency variation in several environments. First, we use simple low-level benchmarks to measure the effects of software switches on latency variation in Section 3.1. We expand on that research to include virtual machines and higher-level abstract benchmarks in Section 3.2. Next, we use those benchmarks to conduct a thor-

ough examination of latency variation in a low-latency HPC environment in Section 3.3. Finally, we broaden the research to measure latency variation in cloud-based computing environments in Section 3.4 and explore its effect on high-level benchmarks that simulate a real HPC application.

The resulting body of original research exemplifies several types of benchmark data generation, from low-level empty packets to the high-level complex patterns of a real HPC application at scale. This work has been condensed from its original published form to focus on the aspects relevant to this dissertation.

3.1 Measuring Long-tailed Latency Distributions with Software Switches

Our work began with a collaborative effort to understand how lightweight operating system-level virtualization (i.e., *containers*) could be used to isolate software environments in NFV. Over the course of this work, we observed that some supporting technologies such as software switches seemed to introduce higher packet delay variation.

In traditional enterprise and telecommunications networks, network services such as routing, intrusion detection systems, and firewalls are typically performed by specialized hardware appliances situated in the data plane. With NFV, these services are instead implemented as software applications that run in virtual environments on standardized general purpose computing hardware. This gives huge benefits in flexibility and scalability, at the cost of lower operating efficiency through virtualization and the use of general purpose hardware. It also opens up network services to management and orchestration techniques that allow for unprecedented control, and removes much of the complexity and specialized knowledge required with traditional systems.

One of the primary challenges of using virtual machines (VMs) in NFV has been the significant performance and efficiency costs of hardware virtualization [104]. Furthermore, network I/O, which is of critical importance to NFV, can also suffer in many configurations.

Containers are a relatively new technology which allows applications to run as sandboxed user-space instances on the host machine with isolation similar to hardware virtualization. Container packaging and deployment platforms such as Docker[1] simplify using containers to run virtualized services by packaging applications with a customized view of their runtime environment. Since

[1] http://www.docker.com

applications in containers run on the host OS without hardware indirection, they can run more efficiently than their VM-based counterparts [161] and allow higher application density on a host [70].

One challenge that is not sufficiently addressed by containers is network I/O. Containers are typically used to provide isolation for services that communicate using one or more network sockets bound to a port on the host. Traffic is handled by the host's network stack using a software switch such as the Linux bridge, which can incur performance cost and variation. While many services typically deployed in containers are not bounded by network performance, most use cases for NFV have strict requirements for network throughput and delay [116] that can be difficult to guarantee with traditional Linux networking.

Our research is the first in a line of investigations to better understand the networking issues in operating system level virtualization. It is the goal of this work to identify and quantify the factors that influence the packet delay and throughput of container-based applications and virtual machines, in the context of NFV service chains where VNF instances may exist on the same or multiple hosts on a network. We describe and report on controlled experiments devised to isolate these factors, and finally identify goals of future research in this area.

3.1.1 Methodology

Since chains of VNFs are meant to replace fast, high throughput hardware middleboxes, the maximum throughput, latency cost, and delay variation of the service chain is of primary importance. Throughput can be addressed to an extent by horizontal scaling, but there will always be a minimum delay cost of the chain even with minimal load. Previous work has found that virtualization can introduce throughput instability and abnormal delay variations to the network traffic [181].

Experiments were conducted on bare metal instances in CloudLab [147]. Each physical machine was a Dell C8220 server, with dual Intel Xeon E5-2660v2 10-core 2.20Ghz CPUs, 256GB ECC RAM, and Intel 82599SE 2-port 10Gbe network interface controller (NIC). In each experiment, machines were co-located on the same rack and connected by two networks: a 1Gbps control network, and a 10Gbps experiment network linked to a Dell Force10 S6000 switch.

Machines ran Ubuntu 14.04 LTS with the 3.13.0-57 low latency Linux kernel. In all experiments, task pinning and kernel scheduling exclusion were used to ensure that kernel threads, hardware interrupt servicing threads, and user threads were run on separate cores within the same

NUMA node. While this approach explicitly disallows L1 and L2 cache reuse between threads, the lesser degree of context switching allows us to obtain more reproducible results.

In our experiment configurations, interfaces are added to containers using network namespaces. *SR-IOV* VFs are created by the OS after setting the `num_vfs` parameter of the NIC. *OVS* and *bridge* use virtual Ethernet device pairs assigned to the switch and the container. These device pairs, as well as *macvlan* subinterfaces, are created with the `ip` command. Interfaces are then moved into the container's network namespace, similar to the *direct* assignment of a physical interface to a container.

Packet send and receive timestamps were recorded for measurements of jitter and lateness, a method validated by [162] and [185], and collected using tcpdump[2]. Measurements of latency were conducted using Netperf 2.6.0[3]. Ethernet frame sizes of 64, 128, 256, 512, 1024, 1280, and 1514 bytes were chosen according to the standard network device benchmarking methodology established by RFC 2544 [47].

3.1.2 Results

To compare performance, we evaluated each of the networking mechanisms for connecting processes in three environments – *Docker containers*, *Xen virtual machines*, and *running natively on the host* – and using three metrics: *latency*, *jitter*, and *efficiency*. In each test, UDP packets were sent from a client running on an external host to a server running on the virtualization host.

3.1.2.1 Latency

We first evaluated the networking technologies using latency to compare the costs of their different software stacks. Figure 3.1 illustrates the results. We found that for each packet size, device virtualization mechanisms such as macvlan and SR-IOV incurred less processing delay than software switches. On average, the Linux bridge and OVS increased latency 3.2% and 4.9% over the native network stack, respectively, while macvlan and SR-IOV increased the latency by 1.1% and 1.0%.

We then compared latency cost of sending packets to Docker containers, Xen virtual machines, and host native processes. The results in Figure 3.2 show that with each networking tech-

[2]http://www.tcpdump.org
[3]http://www.netperf.org

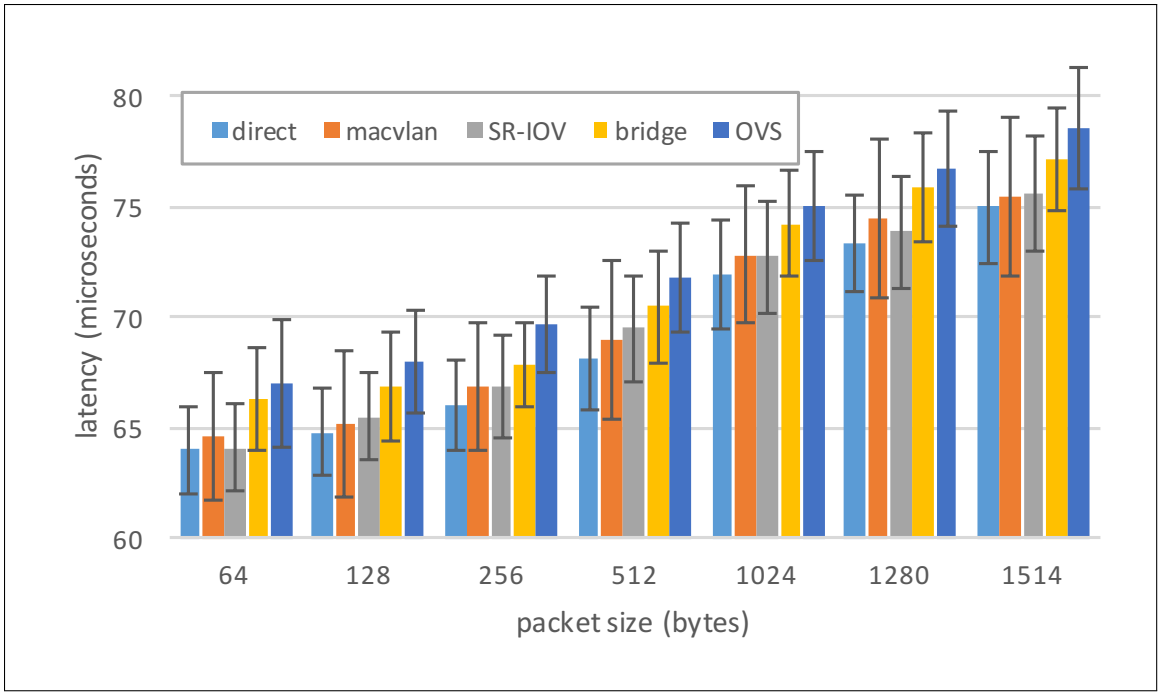


Figure 3.1: Latency and standard deviation of various-sized Ethernet packets from an external host to processes in Docker containers, using different networking technologies.

nology, containerized applications carry an additional latency cost (2.6%-16.1%) compared to native applications, but less of a penalty than the equivalent Xen VMs (53.9%-92.3%). Our findings of higher mean latency with Xen agree with those of other researchers [185].

3.1.2.2 Jitter

Next, we measured the delay variation (i.e., *jitter*) of packets sent from an external host to a receiver on the virtualization host using each of the network technologies. Ethernet frames of 64 bytes were generated at a constant bitrate of 130Mb/s, and samples were taken excluding the head and tail of the stream.

As maximum delay is also an important metric in NFV as late packets influence packet loss ratio[68], we also measured the lateness of packets relative to the minimum observed delay time. Figure 3.3 illustrates the results, which are detailed in Tables 3.1 and 3.2. In our experiments, we found macvlan to have the most stable jitter and lateness, while the Linux bridge and OVS experienced frequent delays in the milliseconds, despite a low mean variation. Included are results of a Xen VM receiving packets through OVS, which experienced significantly higher variation than

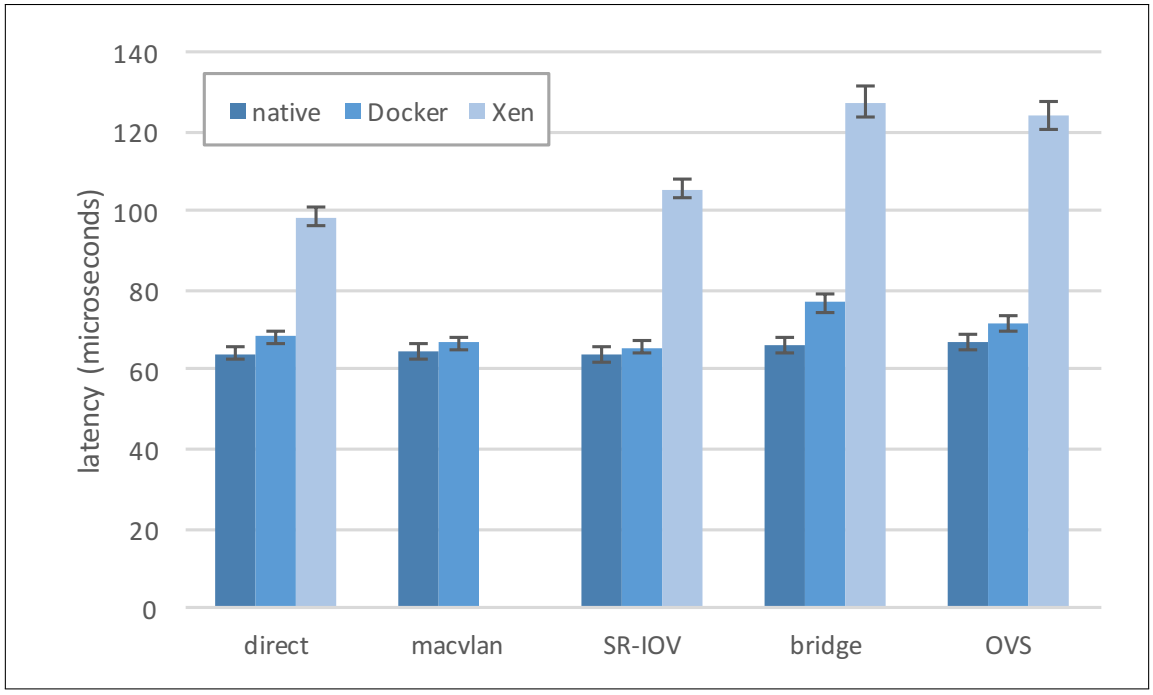


Figure 3.2: Latency and standard deviation of 64-byte Ethernet packets from an external host to processes running natively, in Docker containers, and in Xen virtual machines.

a native process served by OVS, along with a many packets (0.61%) arriving out of order.

Table 3.1: packet delay variation (μsec)

	direct	macvlan	SR-IOV	bridge	OVS	Xen+OVS
n	50000					
min	0.17	2.57	3.00	1.60	1.35	1.42
max	7.50	10.25	27.97	20.71	24.47	18644.02
$\bar{x}$	0.87	6.79	7.96	6.76	6.96	265.36
s	0.69	0.55	1.87	3.43	3.71	1233.03

3.1.2.3 Efficiency

While latency is an important metric, it is not necessarily an indicator of the throughput of the system. We measured the efficiency of each networking mechanism by sending a controlled-rate stream of 64-byte packets (˜64Kpps) from an external host to a receiver within a Docker container, while observing the CPU usage as reported by the system. Figure 3.4 shows the results according to user, hardware interrupt, and other kernel threads. We found that SR-IOV had nearly the same computational efficiency as direct assignment of the interface, while macvlan, the Linux bridge, and

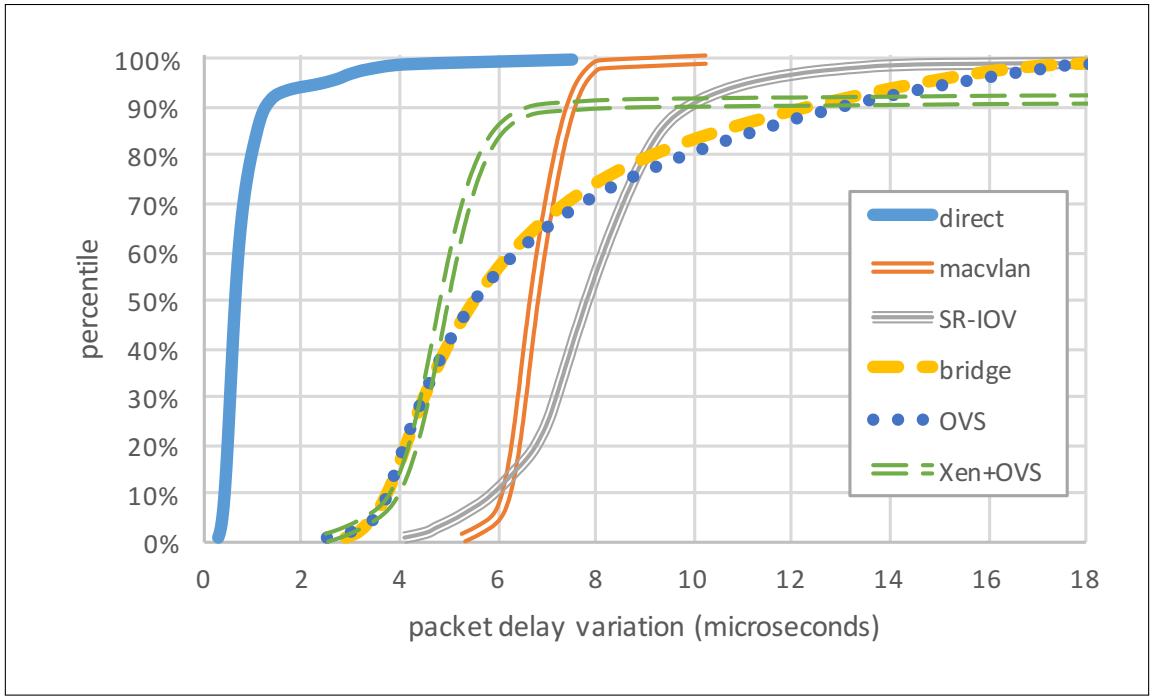


Figure 3.3: Packet delay variation (jitter) of 64-byte Ethernet packets received by native processes, using different network technologies. Measurements using Xen virtual machines and OVS are included for comparison.

Table 3.2: packet lateness (μsec)

	direct	macvlan	SR-IOV	bridge	OVS	Xen+OVS
n	50000					
max	214	114	391	3326	9727	72867
$\bar{x}$	41.40	12.22	53.83	1685.29	6068.86	782.85
s	40.49	5.88	43.23	1181.90	2289.98	6015.96

OVS increased overhead per packet substantially (11.2%, 53.4%, 26.6% respectively). As SR-IOV does not involve the CPU in packet switching, the low overhead is expected.

3.1.3 Summary

In this study we used generated network packets to measure the effects of software switches and virtualization on aspects of network performance. Our results support our hypothesis that OS-level virtualization can offer network performance benefits compared to hardware virtualization. In every case, Docker containers have lower latency cost and lower variability than equivalent Xen VMs running the same software.

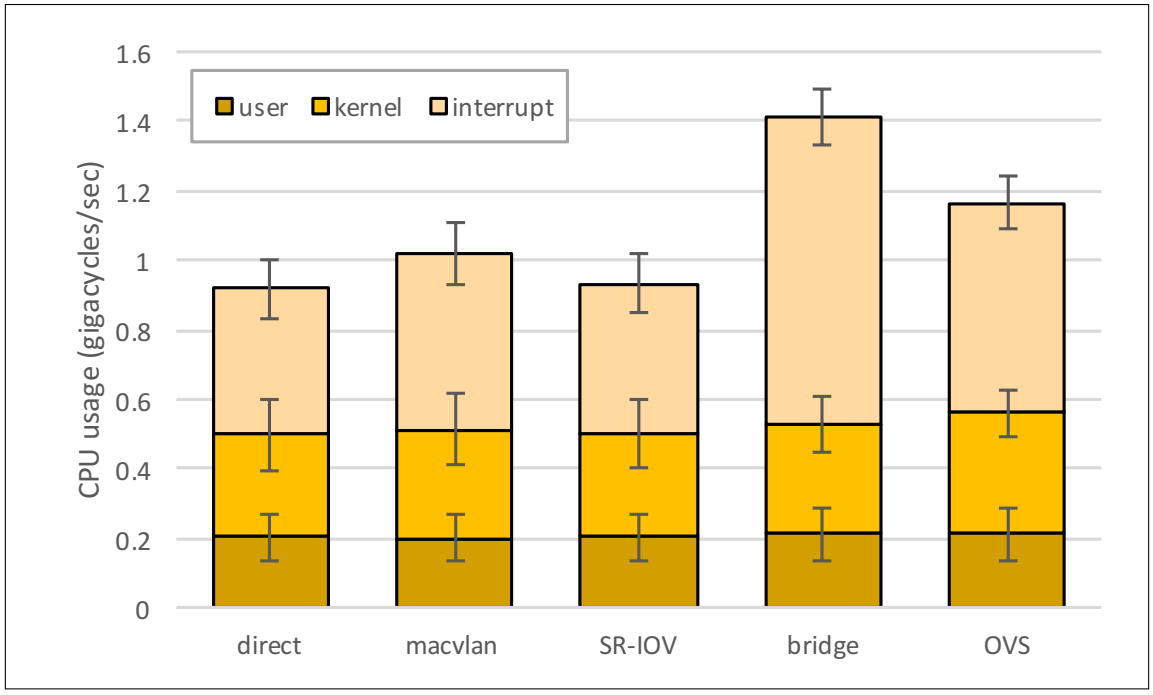


Figure 3.4: Computational efficiency and standard deviation of networking mechanisms forwarding 64-byte Ethernet packets from an external host to a Docker container, classified into user processes, kernel processes, and interrupt servicing.

In the case of ingress and egress of packets over a physical network interface, both macvlan and SR-IOV show lower mean latency and more predictable variation than the Linux bridge and OVS. Furthermore, in both the bridge and OVS we observe latency spikes in the milliseconds that could affect packet deadlines.

We also concluded that further study was needed on the effects of latency and jitter on packet throughput and predictable delays in a VNF service chain. We suggested exploring how packet train dispersion [98] is affected by factors such as chaining VNFs with multiple stages of buffering and transmission burst compression due to virtualization [185].

3.2 Measuring the Performance Impact of Software Switches in HPC

Recent work had shown that lightweight virtualization like Docker containers could be used in HPC to package applications with their runtime environments [83]. In many respects, applications in containers perform similarly to native applications [186, 152].

Building on the observation that software switches were associated with increased latency variation, we hypothesized that these increases could affect the performance of HPC applications in an environment where software switching was used. This latency variation may have an impact on the performance of some HPC workloads, especially those dependent on synchronization between processes [115].

In this work, we measure the latency characteristics of messages to and from Docker containers, and then compare those measurements to the performance of real-world applications. Our specific goals are to:

- Measure the changes in mean and variation of latency with Docker containers
- Study how this affects the synchronization time of MPI processes
- Measure the impact these factors have on real-world applications such as the NAS Parallel Benchmark (NPB)

3.2.1 Methodology

Typical Docker applications use the Linux bridge or Open vSwitch to direct traffic to and from applications in containers. To understand how applications are affected by both the software bridge and the container itself, we established four environments to conduct each benchmark:

- **native** — native application with normal access to the network
- **bridged** — native application using a veth interface attached to a Linux bridge
- **direct** — Docker application with the network interface directly assigned to the container
- **docker** — Docker application using a veth interface attached to a Linux bridge

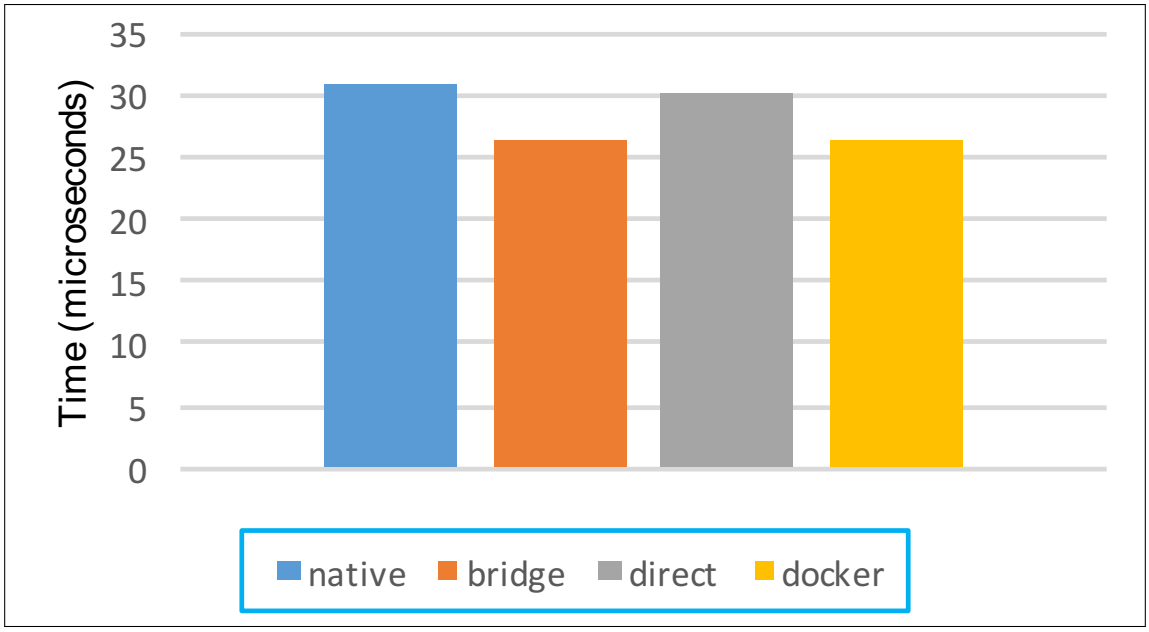


Figure 3.5: Mean round trip time of MPI pingpong test.

Experiments were conducted on 6 Cloudlab machines at the Wisconsin site, each consisting of a Cisco C220 M4 rack server with two Intel ES-2660 v3 10-core CPUs at 2.60 GHz, 160GB ECC memory, and a dual-port Intel X520 10Gb NIC connected to a shared Cisco Nexus C3172PQ top-of-rack switch. Machines were running Ubuntu 14.04 LTS, kernel version 3.13.0-68-generic, using Docker 1.11.2 and OpenMPI 1.6.5 to run NAS Parallel Benchmark for MPI 3.3.

In all experiments, we used custom benchmarks using the MPI framework to generate synthetic network traffic, and used the `tcpdump` tool to capture packet send and receive timestamps for analysis.

3.2.2 Results

3.2.2.1 Microbenchmarks

We measured the mean latency imposed by the test environments by conducting a ping-pong test between two MPI processes on separate nodes. We placed the receive side of the test in the test environment, while the sender ran natively to avoid doubling the effect. As shown in Figure 3.5, native had the highest mean latency, while tests using extra software layers of the Linux bridge and veth interface (bridge and docker) had lower means than direct interface access.

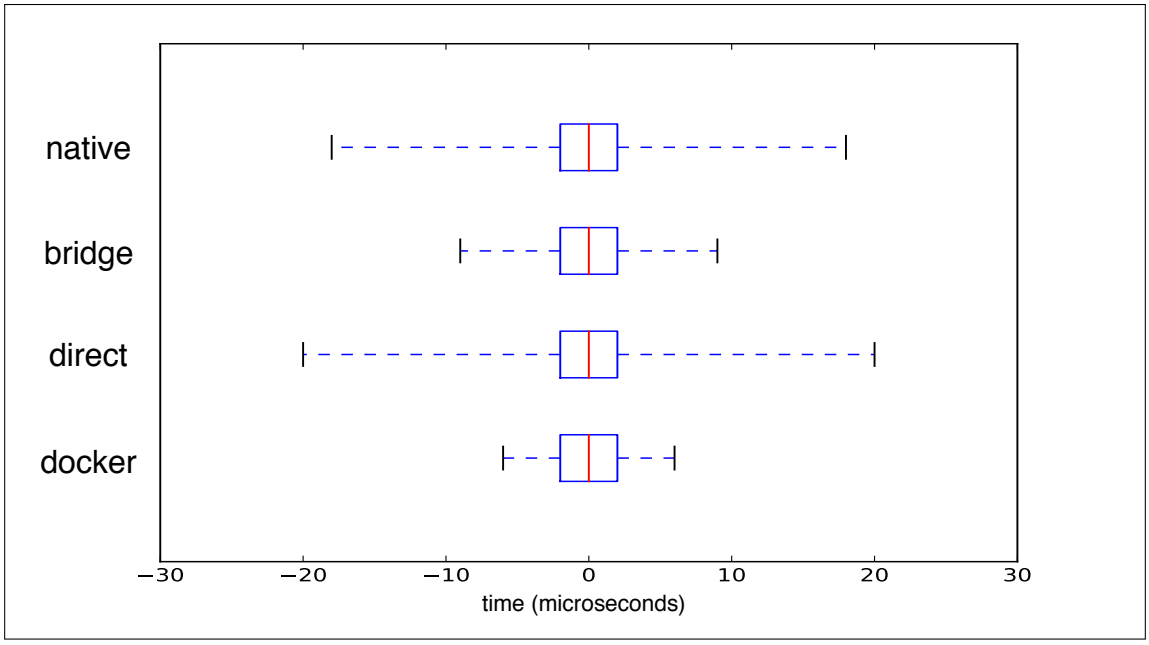


Figure 3.6: 1st, 25th, 50th, 75th, 99th percentiles of send-side gap variation.

To measure the latency variation, we send a constant rate stream of messages between MPI processes on separate nodes. To help determine where variation occurs, we further divide the tests into send-side and receive-side variation. We then calculate and report this inter-message spacing in Figures 3.6 and 3.7.

On the sending side, using the Linux bridge seemed to decrease latency variance. On the receiving side, however, the opposite seems to be true; the Linux bridge increased the latency variance. However, a Docker container had the effect of decreased variance in both network configurations.

3.2.2.2 Synchronization

For these tests, we measured the time required for 8 MPI processes on separate nodes to synchronize to an MPI_Barrier() call. Unlike the microbenchmarks, all nodes were under the testing environment. We report the mean and distribution of times in Figures 3.8 and 3.9.

As opposed to the microbenchmarks, native and direct had lower means and variation, while measurements in environments using the Linux bridge were more variable.

We also observe that the environments using containers have a higher mean synchronization time, despite the lower receive-side latency variation and equivalent mean and send-side latency variation of the microbenchmarks.

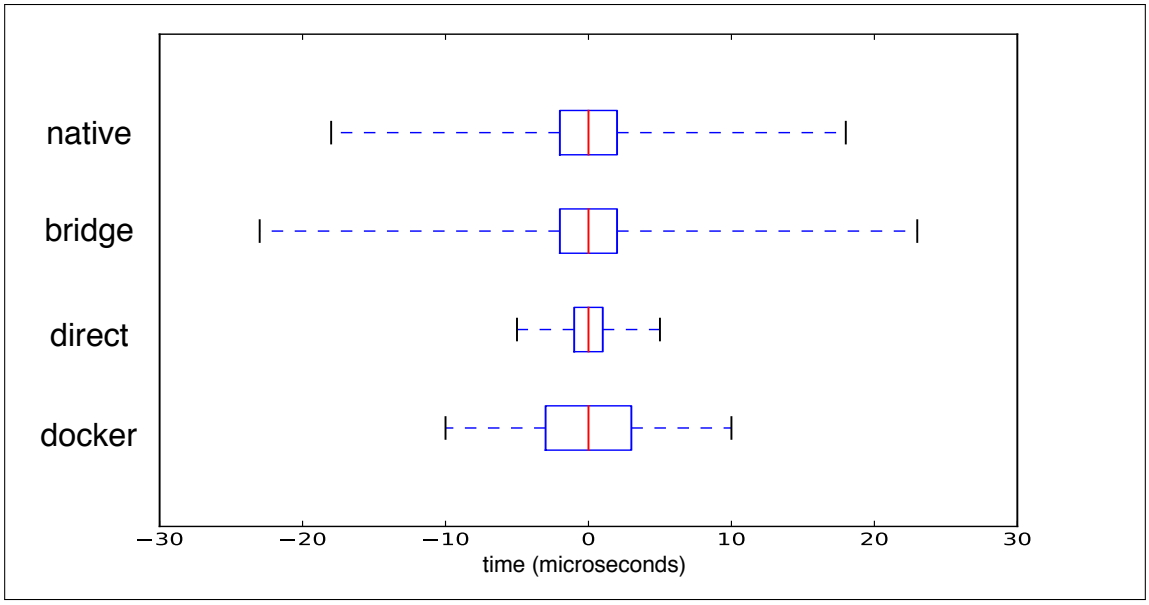


Figure 3.7: 1st, 25th, 50th, 75th, 99th percentiles of receive-side gap variation.

3.2.2.3 Application Benchmarks

We ran the NAS Parallel Benchmark with seven of the nine benchmarks compiled with problem sizes appropriate for a 160 core cluster. First, we measured the total number of packets that were transmitted between nodes for each program, then ran 30 iterations of each benchmark with processes distributed evenly across eight nodes. To estimate how dependent each benchmark is on communications, we compare each benchmark's mean packet size and transmission rate as compared to native run times in Figure 3.10. To estimate the impact of each environment on runtime, we compare each test's min, max, mean, and difference from native mean in Figures 3.11 and 3.12.

As with the synchronization tests, bridge and docker added a significant amount to the mean and variation on BT, CG, LU, and SP. We observe that tests with lower packet transmission rates are less affected by the bridge and veth interface.

3.2.3 Summary

In this work, we used custom microbenchmarks using MPI primitives to generate low-level network traffic patterns. Using those tools, we observed the effects of container virtualization and software switches on measures of latency variation, distributed synchronization, and internode

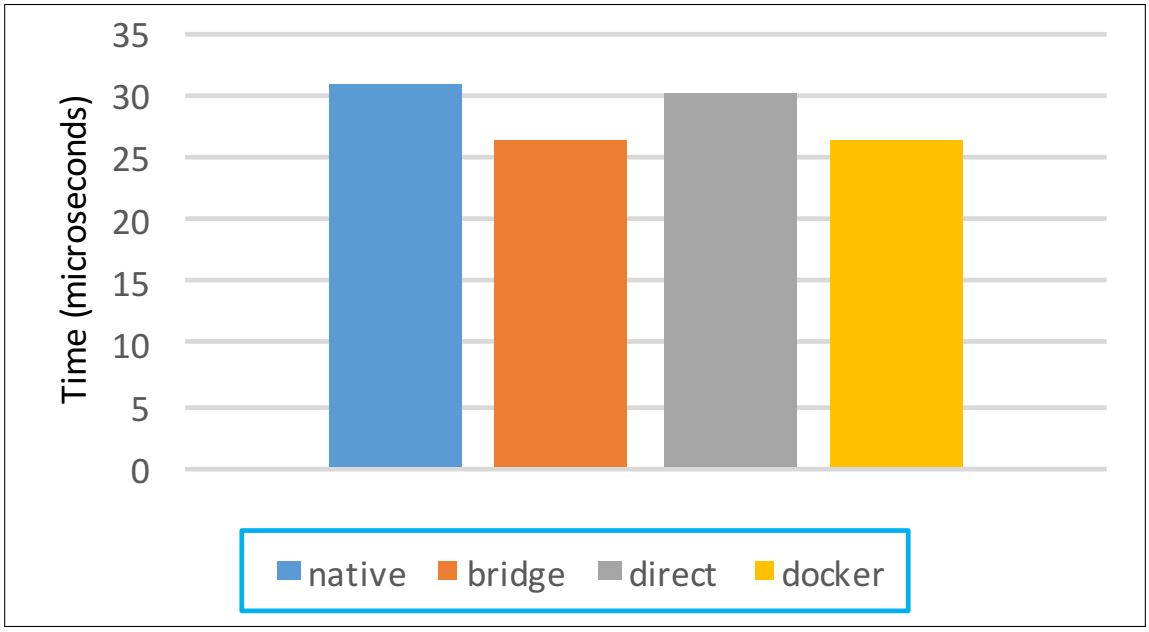


Figure 3.8: Mean time to synchronize 8 nodes.

throughput. We then used the NAS Parallel Benchmarks to generate distributed traffic patterns common to HPC applications and characterize the higher-level performance impact.

Although we observed in the microbenchmarks that the Linux bridge and veth interface lowered the mean and sender-side variation of message latency between MPI applications, we saw that synchronization time increased for the same test environments. Furthermore, we see that application benchmarks that send many packets per second are more affected by the extra software switch than those with direct access to the network interface.

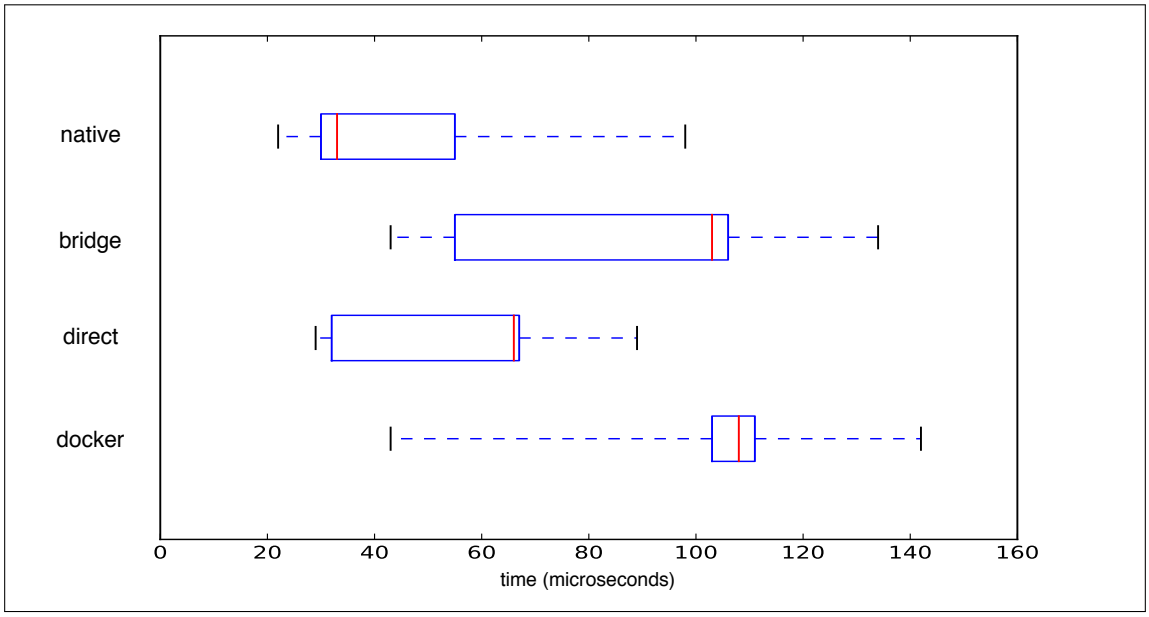


Figure 3.9: 1st, 25th, 50th, 75th, 99th percentiles of synchronization time variation.

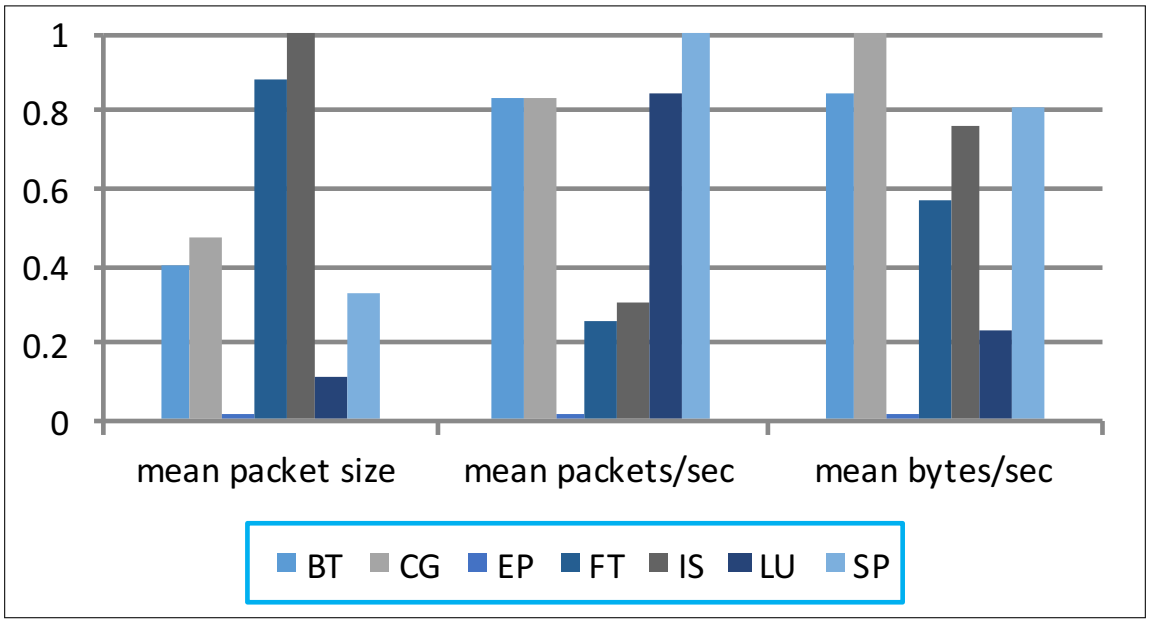


Figure 3.10: Normalized comparison of data transfer characteristics.

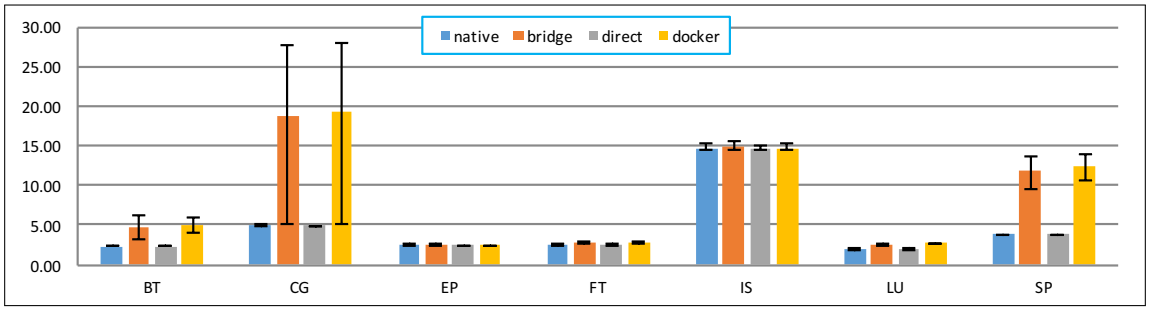


Figure 3.11: Minimum, mean, and maximum runtimes of NPB benchmarks.

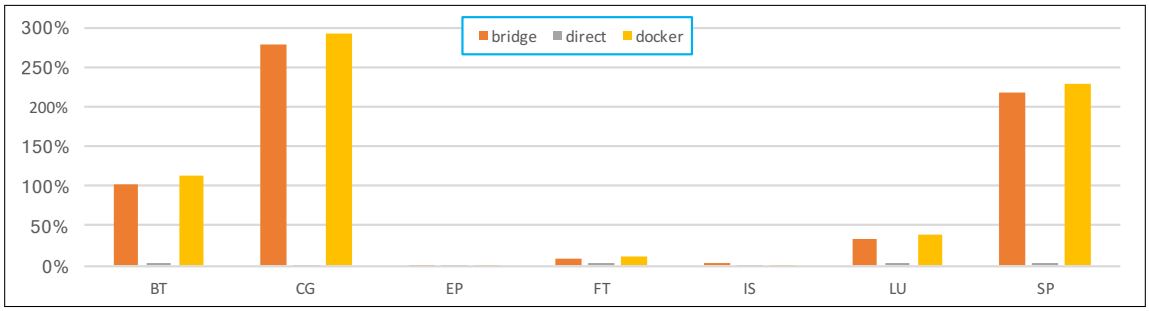


Figure 3.12: Percent difference from native for NPB benchmarks.

3.3 Characterizing Latency Variation as a Factor in HPC Performance

We had seen that latency variation, mean latency, and performance degradation in HPC benchmarks were all correlated with virtualization supporting technologies that introduced software switches. However, we were not entirely sure what was affecting the HPC benchmarks - mean latency, or latency variation. In our next work, we set out to find the answer.

To address one shortcoming of our previous work, we decided to use low latency networking to have results comparable to a modern HPC cluster. There has been extensive research on the development of low-latency, highly performing networks for HPC [136, 42, 40], and research has demonstrated the considerable effect of average network latency on the performance of HPC applications [153, 95].

Because compute nodes are increasingly multithreaded, network resources are under increased contention creating competition for these resources and increasing the variation in network communication time. In this work, we demonstrate that network latency variation by itself can have a significant effect on HPC workload runtime. That is, with equal mean latency, a higher variation in the network latency can result in significantly lower HPC application performance.

Our work confirms prior research that has focused on the packet and library level. At the packet level, increasing of mean network latency affects performance, but point-to-point communication is less affected by the variation in network latency. At the library level, latency variation affects the runtime of collective operations, particular those that involve most or all nodes in the computation [87, 84].

We present new results at the application level. We characterize the negative impact that latency variation has to the performance of classes of communication-intensive applications. The decrease in performance ranges up to 3.5 times slower for LU Decomposition [32], for example. We show that for communication-intensive HPC applications, changes in performance are more highly correlated with changes of variation in network latency than with changes of mean network latency alone. These results have implications for the design of HPC applications that must execute in a highly shared environment, say, using commercial cloud resources or in a multi-tenant environment, and suggest that implementation of mechanisms to control network variation latency may lead to better overall application and system performance than efforts to reduce average network latency

alone. Our main contributions are:

- A design and implementation of a configurable latency injector for many Mellanox and QLogic InfiniBand cards;
- Characterization of the distributions of latency in network performance for InfiniBand network in an HPC environment;
- Presentation of an experimental methodology using synthetically-generated latency to demonstrate the effects of latency variation on HPC workloads;
- Statistically significant evidence that latency variation is more highly correlated with HPC application performance than latency mean alone.

3.3.1 Background and Related Work

Prior research has reported how congestion-induced latency variation can have significant effects on application performance [38]. This is straightforward to observe, for example, in the `MPI_Barrier` routine. With a barrier routine, no process can continue until all of the processes entering the barrier have completed the synchronization step. Thus, the time to complete the barrier call is determined by the process that takes the longest time to enter and complete the barrier [87]. In networks with high latency variation, the time to synchronize to a barrier can be considerable [61].

Alizadeh et al [15] explored the effects of high bandwidth consumption on network latency, and found that increased competition for buffers in Ethernet switches could lead to long-tailed latency distributions with measurements as high as 1000 times the median. Our work is partly inspired by theirs, as we questioned whether the latency variation would also impact HPC workloads using MPI and zero-copy networks like InfiniBand.

One of the earliest papers to examine effects of variation in network protocols was the done by Zhang et al in [192]. In their work, they observed that performance of TCP based streams could experience increased latency if ACK packets were clustered in two way, TCP, communication. They proposed some adjustments to the TCP congestion control protocol to reduce clustering. Unlike their work on the latency in the network, our work focuses on the impact of this variation on the runtime of applications supported by the network. Additionally our paper has the added contribution of

quantifying the effects of the latency distribution.

There have been many studies of the effect of congestion on network latency. [37, 38] and [101] observed that increased bandwidth contention on links between nodes in a large cluster resulted in larger mean latencies and significant impact on HPC workloads.

One of the earliest was conducted by Jacobson [96]. These studies essentially show that congestion introduces additional latency and latency varation increase when congestion is present as the there is additional contention for the hardware. Our paper introduces the additional contribution that latency variation can have negative effects in and of itself, and the impacts of latency variation can be more significant than just considering latency.

3.3.2 Methodology

In this section we describe the methods used to create a controlled test environment for later experiments. Measurements of low-latency networks are fine-grained and sensitive to perturbations by other systems. Our goal is to minimize or eliminate noise in our testbed and to collect high quality measurements so that we ensure that relationships between independent and dependent variables can be correctly characterized.

3.3.2.1 Hardware Configuration

We ran our experiments on Cloudlab c8220 nodes [147]. These nodes are equipped as outlined in Table 3.4. This particular hardware was chosen to provide enough cores to support our MPI experiment configurations with one core per process, and adequate memory per process for each benchmark. To allow addition of an artificial latency generator in the network device driver, we chose hardware with QLogic infiniBand cards which have a open source user-space driver.

We validated our experiments on Cloudlab c6320 nodes. These nodes share the characteristics we identify as ideal for our experiments. These nodes are equipped as outlined in Table 3.3. Results were similar and produced identical conclusions between the two hardware types so the c6320 results have been omitted.

Cloudlab provides a means of specifying a desired network topology that it constructs using software defined networking. However, since this topology is imposed on the existing physical topology, there are artifacts in the underlying topology in latency measurements, such as nodes being in physically different racks having higher latency. We accounted for this variation by testing the

Table 3.3: Cloudlab c6320 Nodes

Hardware	Description
CPU	Two Intel E5-2683 v3 14-core CPUs at 2.00 GHz (Haswell)
RAM	256GB ECC Memory
Disk	Two 1 TB 7.2K RPM 3G SATA HDDs
NIC	Dual-port Intel 10Gbe NIC (X520)
NIC	QLogic QLE 7340 40 Gb/s InfiniBand HCA (PCIe v3.0, 8 lanes)

Table 3.4: Cloudlab c8220 Nodes

Hardware	Description
CPU	Two Intel E5-2660 v2 10-core CPUs at 2.20 GHz (Ivy Bridge)
RAM	256GB ECC Memory (16x 16 GB DDR4 1600MT/s dual rank RDIMMs
Disk	Two 1 TB 7.2K RPM 3G SATA HDDs
NIC	Dual-port Intel 10Gbe NIC (PCIe v3.0, 8 lanes)
NIC	QLogic QLE 7340 40 Gb/s InfiniBand HCA (PCIe v3.0, 8 lanes)

latency between all nodes. If a node was found to have statistically higher latency than its neighbors (to the level of $\alpha = 0.05$), that node was removed from testing and another was selected.

3.3.2.2 Software stack

Given the sensitive nature of $< 10\mu s$ latency measurements and awareness of the impact of OS noise on parallel computer performance [135], the software configuration was carefully tuned to minimize noise and eliminate extraneous variables. To avoid introducing traffic over the processor interconnect in a dual-socket system, we designed our experiments to use the cores of a single CPU package with the shortest electrical distance to the network interface over the PCIe bus. We also required that core "0" was not used for experiment processes, as the operating system always assigns certain critical tasks to that core. As our MPI experiments required 8 processes per node, the hardware was required to have more than 8 physical cores per CPU package.

We ran the experiments on a patched Ubuntu 16.04. To prevent OS scheduling of tasks on the same cores as our experiment processes, we controlled the CPU affinity of tasks by isolating physical cores 2-9 on CPU 0 of each node with the kernel flag `isolcpus=4-27`, and then forcing MPI to distribute processes only among those cores. This required a minor change to the Linux kernel to prevent scheduling kernel tasks on the isolated cores, a known issue which detailed at [62].

Other OS configuration included disabling hyperthreading and disabling CPU low power states to prevent clock speed throttling. We achieved this using the kernel commandline options `processor.max_cstate=1` and `intel_idle.max_cstate=0`.

We compiled our experimental codes using gcc version 5.4.0. When flags were not provided by the benchmark code, we used the flags `-O3 -march=native`. When build flags were provided, we use the provided flags.

We choose OpenMPI 1.10.2 from the Ubuntu repositories as our MPI implementation. We choose OpenMPI because of its performance on InfiniBand interconnects.

3.3.2.3 Overview of the Packet Level

In this section, we provide an overview of the codes that we ran to examine the packet level performance. The purpose of these tests are to capture the performance of the network interfaces with the minimal amount of overhead. We choose codes from the `perftest` package, which is part of the Open Fabrics Enterprise Distribution (OFED) [16], which are well–established for testing InfiniBand performance at the packet level. In particular, we used `ib_write_lat` with Reliable Connection (RC) transport protocol. This tool uses raw InfiniBand commands (called verbs) to measure the time to send remote write commands with delivery confirmation, and forms a low level pingpong test.

We did not consider other tests from OFED such as `atomic` or `send` operations because they introduce additional operations above and beyond what `ib_write_lat` does, and have higher latency and latency variation because they require additional CPU assistance to complete.

Our preliminary experiments indicated use of a software MTU of 2048 bytes as the most efficient configuration without message fragmentation.

3.3.2.4 Overview of the Library level

In this section we provide an overview of the codes we used to access the library layer. The purpose of these tests is to capture the performance in a more realistic scenario where a well–established abstraction such as MPI is used. We used two codes: `ping-pong` and `barrier`. `ping-pong` times a long sequence of `MPI_Send` and `MPI_Recv` calls. `ping-pong` is roughly equivalent to the packet level test except it is preformed at the MPI library level instead of the packet level. `barrier` times collective communications by calling a long sequence of `MPI_Barrier` calls. It tests the efficiency of collective communication. In each of tests, the time to complete each operation was considered the runtime.

3.3.2.5 Overview of the Application level

At the application level we chose the NASA Advanced Supercomputing (NAS) Parallel Benchmarks (NPB). The NPB consist of a series of benchmarks designed to test many features of a high performance computing cluster including its network based on problems seen in computational fluid dynamics. It consists of benchmarks: Integer Sort (IS), Embarrassingly Parallel (EP), Conjugate Gradiant (CG), MultiGrid (MG), 3D Fast Fourier Transform (FT). Of these benchmarks, three of them have high communication: CG, MG, and FT [32]. NPB also includes three pseudo applications: a block tri-diagonal solver (BT), a scalar penta-diagonal solver (SP), and a lower-upper Gauss-Seidel solver (LU). These tests stress the network interconnect and provide a model of latency variation similar to real-world network conditions.

To examine the effects of increased latency variation on real applications, we executed the NAS Parallel Benchmark across the eight nodes in our cluster. We ran five of the nine included tests, detailed in Table 3.5. NPB problem sizes (A-F) were chosen to ensure a long enough runtime for repeatable results, and the number of processors was chosen to fit each test's particular requirements while being evenly divisible by our eight nodes. In Table 3.5, `size` corresponds to the size we configured for our cluster, `procs` corresponds to the number of processes used at that size, `Mpackets` corresponds to the number of millions of packets sent across the network, `GB` corresponds to the number of gigabytes of traffic generated.

Table 3.5: NAS Parallel Benchmark Tests

Name		size	procs	Mpkts	GB
Conjugate Gradient	CG	C	64	9.30	3.92
3D fast Fourier Transform	FT	C	64	22.34	9.76
Integer Sort	IS	C	64	2.77	1.22
Lower-Upper Gause-Seidel	LU	B	64	4.75	0.62
Multi-Grid	MG	C	64	1.47	0.55

3.3.3 Workload Measurement and Characterization

In this section we describe our methodology for measuring and characterizing the effects of network resource contention, or (i.e., congestion). Our goal is to measure the effect of congestion on latency at the packet level, so that we can create a latency model for controlled emulation of congested network resources.

Algorithm 1 Congestion Simulator

```
clock_gettime(CLOCK_MONOTONIC, &last_time);
nsec_delay ← 0;
while !stopping do
    if nsec_bucket ≥ nsec_delay then
        MPI_Send(...);
        nsec_bucket -= nsec_delay;
        nsec_delay = random_from_exponential();
    end if
    clock_gettime(CLOCK_MONOTONIC, &cur_time);
    diff_time ← cur_time - last_time;
    last_time ← cur_time;
    nsec_bucket += diff_time
end while
```

Latency variation, sometimes also referred to as jitter, is measured at the packet level. in each experiment we perturb the distribution by introducing congestion or synthetic latency. We compute the variation by taking the standard deviation of the individual packet delivery times. We then use the measurements we take at the packet level and apply them to the library and application level.

We measure the effect of these perturbations to latency variation on the runtime of the application. We use "wall clock time" of the application as measured using the `time` command.

3.3.3.1 Characterization Procedure

For our later experiments, it is essential that we have accurate models of latency variation on InfiniBand hardware to configure our latency injector to match various levels of congestion. In this section, we describe the procedure for capturing the packet arrival latencies and modeling them with a latency distribution.

To introduce network congestion, we created an MPI network load generator to send data between pairs of compute nodes, saturating the one-way bandwidth between the network interfaces. We then controlled the level of congestion by altering the proportion of time that the sending node was transmitting. By having each node run the application twice in both sending and receiving mode, we were able to saturate a fraction of the node's maximum transmission rate. For each measurement of congested performance, we ran the load generator on all involved nodes, as illustrated in Figure 3.13.

There are a few key items to observe from the this pseudo code. First, we used the POSIX

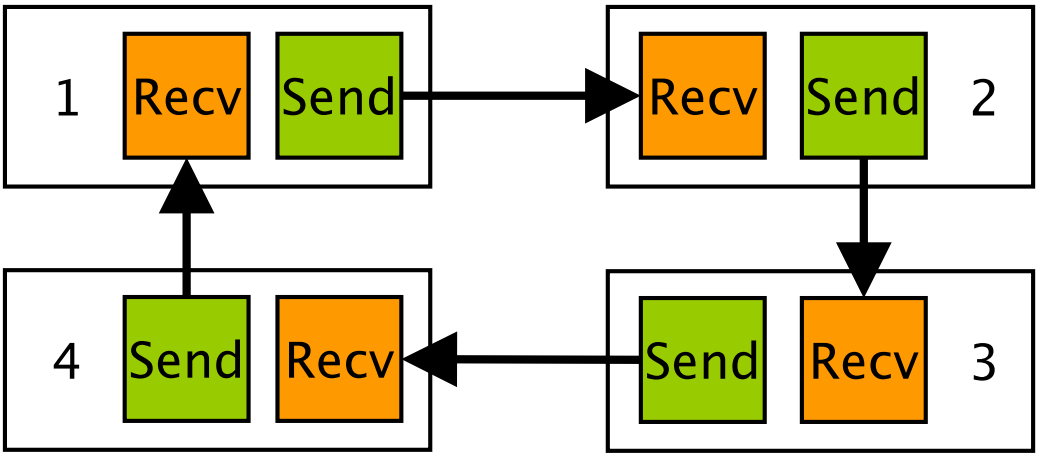


Figure 3.13: Send/receive pairs of background load generating processes on four compute nodes. Each pair saturates a controllable fraction of the one-way bandwidth between two compute nodes.

interface `clock_gettime` to measure time. It has two important features: `clock_gettime` is the highest performing and most precise clock available on most POSIX systems, and it does not require a trap into the kernel to measure the time as `gettimeofday` and other interfaces do. On the x86_64 hardware we used, it is implemented using a read of a timing register on the processor.

The sending application, detailed in Algorithm 1, is based on the "leaky bucket" rate control mechanism first described in [174]. To simulate the transmission characteristics of applications competing for network resources, we sampled the delays between messages from an exponential distribution, which models the long-tailed and highly variable inter-message gaps in large flow background traffic observed in [15]. Samples were provided by a Mersenne Twister 19937 pseudo-random number generator, which provides high quality entropy for its performance [118].

3.3.3.2 Characterization Results

Figures 3.14 and 3.15 illustrate the effects of background load on network packet latency, expressed as a percentage of the network interface's bandwidth. As the simulated network load increases from 0% to 100%, latency mean and variation increase as the sending applications competes for the network device queues.

There are two conditions to notice about the characterization results. First, for congestion

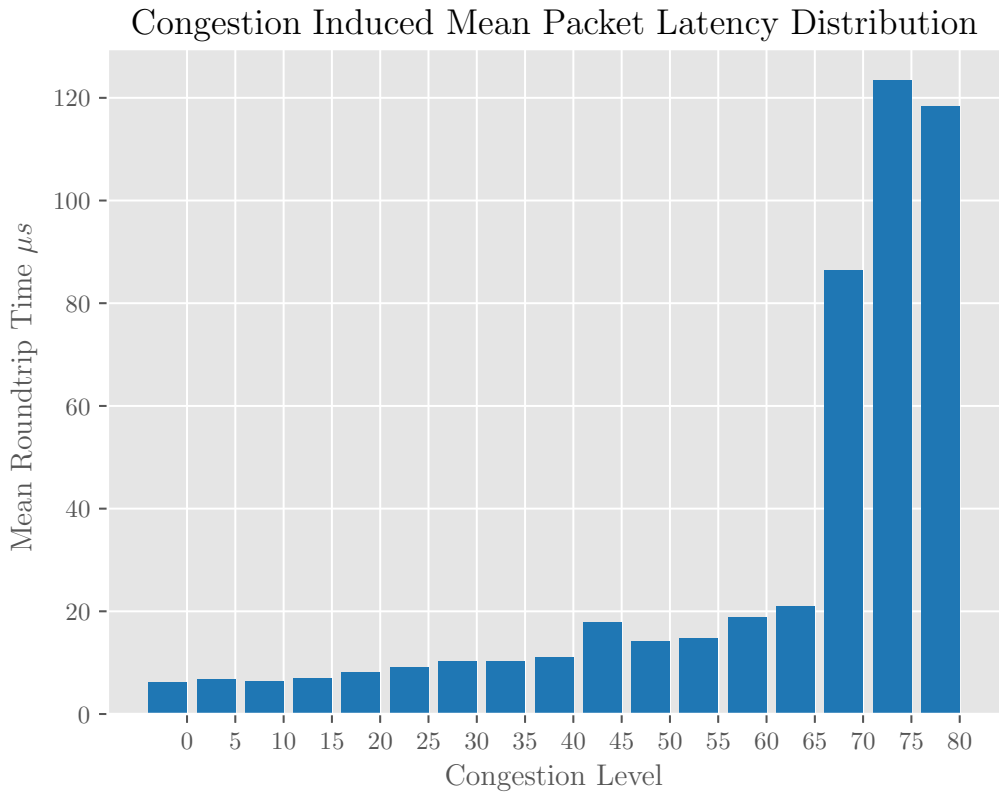


Figure 3.14: As congestion increases, so does the mean latency

above 80%, the latency mean and standard deviation become highly chaotic. For that reason we restrict the remainder of the measurement studies to simulated congestions below 80%. We leave studies of this region for future work. Secondly, we observe that our results of increasing mean and standard deviation of latency are consistent with the existing work on congestion. While the particular distribution collected is hardware and software dependent, the general shape center, and spread are consistent across hardware. We use these distributions to generate corresponding latency distributions to use with our injector.

3.3.3.3 Modeling the Existing Distribution

In this section, we describe the methodology we used to choose the distributions that will be used in our injector. First, we considered the distribution measurements we measured in Section 3.3.3. For an example distribution, refer to Figure 3.16. We observe that the distribution

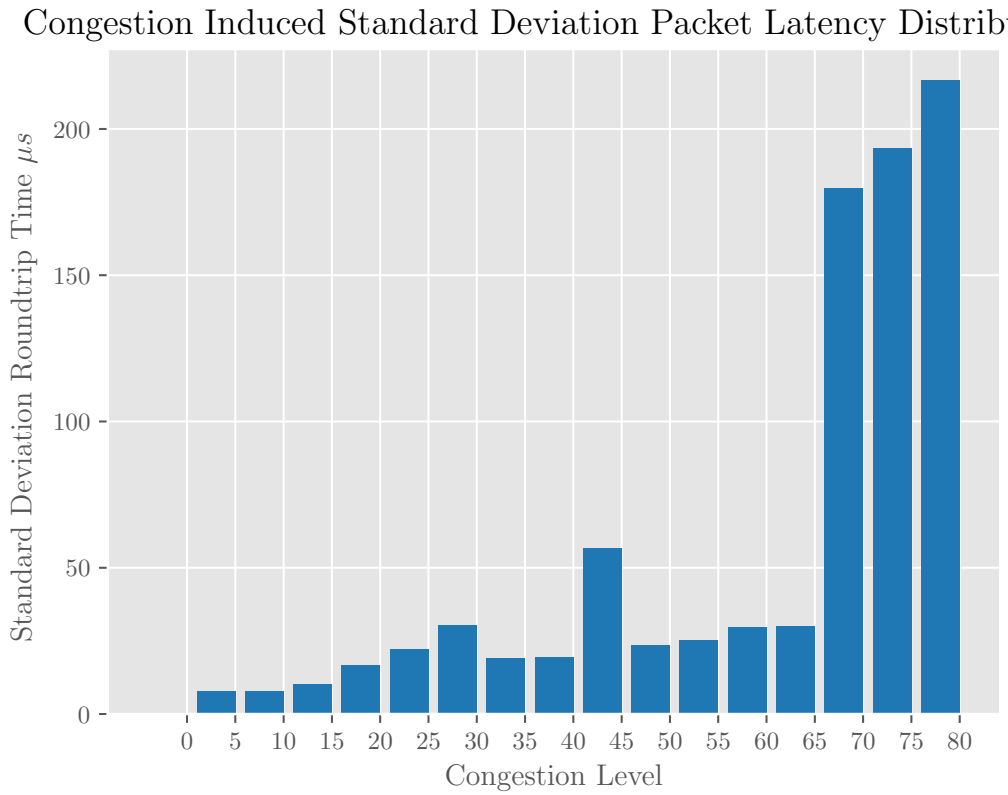


Figure 3.15: As congestion increases, so does the latency variation

is skew-right. Preliminary curve fitting showed that it is best modeled by a log–normal function. The dominant mode is at approximately $6.1\mu s$ with a minor mode of $7.2\mu s$. When plotted against cumulative packet count, the higher latency values are correlated with harmonics of the CPU and PCIe bus frequencies occurring approximately every 45 to $50\mu s$.

We fit five distributions to the data: a uniform distribution and four lognormal distributions with increasing shape parameters. The uniform distribution models only the increasing of the mean of the latency without increasing the standard deviation. The lognormal distributions increase the spread of the distribution without perturbing the overall shape or the mean. The increasing spreads leads us to evaluate the claim that increases in latency standard deviation is more highly correlated with application runtime than latency mean.

The uniform distribution only has one parameter: mean. We fit this parameter by binary searching for each mean from the congestion distribution.

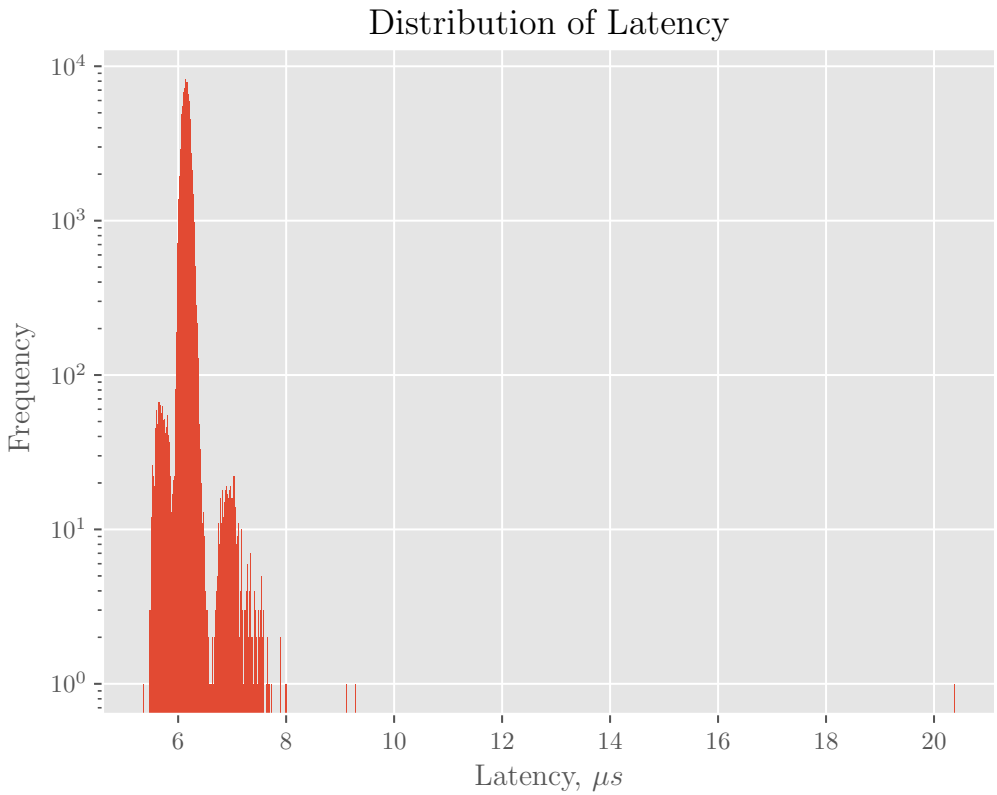


Figure 3.16: Distribution of latency of packets is tight, and highly skew right

The lognormal distributions has three parameters: shape, scale, and location. We estimated the shape parameter by using SciPy's `lognormal.fit` method for the corresponding level of congestion. We varied the scale parameter as one of our independent variables. We used binary search to search for the mean from the corresponding congestion distribution.

Finally, we wrote a generator that uses the distributions from the statistical functions included in SciPy 0.16.1 to generate the distribution files. The distribution files generated along with the distribution file generator will be released upon publication.

3.3.4 Experimental Suite 1: Latency Variation, apart from Congestion, causes Application Degradation

In this section, we use the results of our workload characterization to show that even without congestion or contention, latency variation is sufficient to cause application degradation. We will

accomplish this by simulating latencies along the distributions caused by congestion and contention. By injecting the latencies, no network resources will become constrained during these tests.

We will use the same tests that we utilized in workload characterization, except instead of running a background process to induce load, we will utilize the latency injector. As mentioned in workload characterization, we will consider two distributions: one where we increase only the mean (uniform), and one where we increase the mean and standard deviation (lognorm). The means of these distributions were chosen to correlate with the means of the distributions caused by actual congestion.

Ideally, we would just increase the standard deviation of the distribution. However, by virtue of increasing the standard deviation of a distribution with a strong lower bound, we will also raise its mean. Therefore, we must also consider a distribution where we just increase the mean but leave the spread unperturbed.

3.3.4.1 Packet Level

At the packet level, we have results that are as expected. In Figure 3.17 we observe that both distributions have very similar mean values. This validates the choice of distributions computed from the measured congestion distributions and also validates the functionality of the latency injector.

We also observe in Figure 3.18, some of the differences between these distributions. Here we see that the median latency is significantly higher for the uniform distribution at higher simulated latencies than the lognorm distribution. This suggests as we observe that there are a few very large latencies that were measured in the lognorm distribution that were not present in the uniform distribution. We finally observe that there is not much difference between the uniform distribution mean and median.

When we examined the results from the packet and library levels, we determined that their runtime distributions were not normally distributed. We confirmed our suspicions using the Kolmogorov-Smirnov test for Normality [128] (p=0.0, $\alpha = .05$)[4]. As such we cannot use parametric statistical methods to evaluate the distributions at these levels. Instead, we rely on the nonparametric Mann Whiteny U test which tests the null hypothesis that neither sample stochastically dominates the other. Nonparametric tests do not require the values to be sampled from a normal distribution.

[4]due to limited precision of the hardware the result rounds to 0

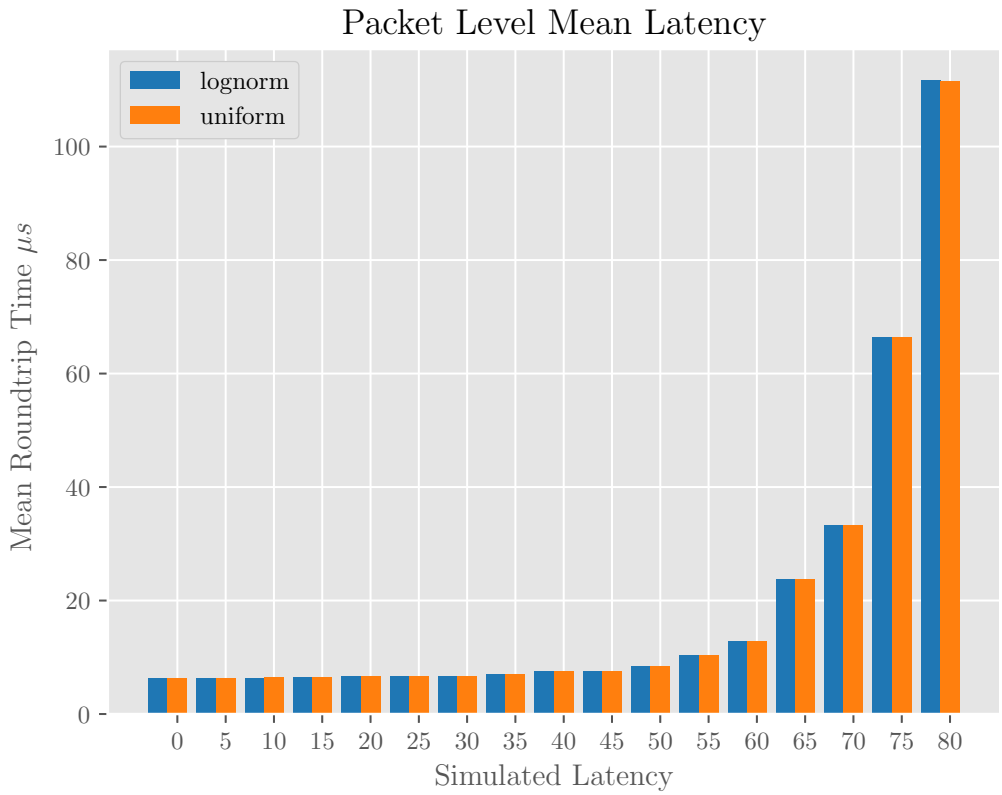


Figure 3.17: Packet level test with matched means. Both distributions have equivalent means at each level of simulated latency. This validates the choice of distributions from the workload classification.

3.3.4.2 Library Level

At the library level, we see a slightly more interesting picture. First, we consider the results from the `pingpong` test. We observe in Figures 3.19 and 3.20 that the results have the same shape, centers, and spreads as the packet level with a slightly higher latency. This is consistent with a slight amount of overhead introduced by the library level.

Secondly, we consider the results from the `barrier` test. As opposed to the packet level test, we see that the mean latencies diverge at higher synthetic latencies (Figure 3.21). We also observe that the lognorm distribution results in higher mean values of latency. This is to be expected as the lognorm distribution produces a small number of extremely large latencies. The mean, being sensitive to these extremes, is pulled to the largest values. The median displays a shape and spread that is similar to that of the `pingpong` test (Figure 3.22). This is consistent with the robustness of

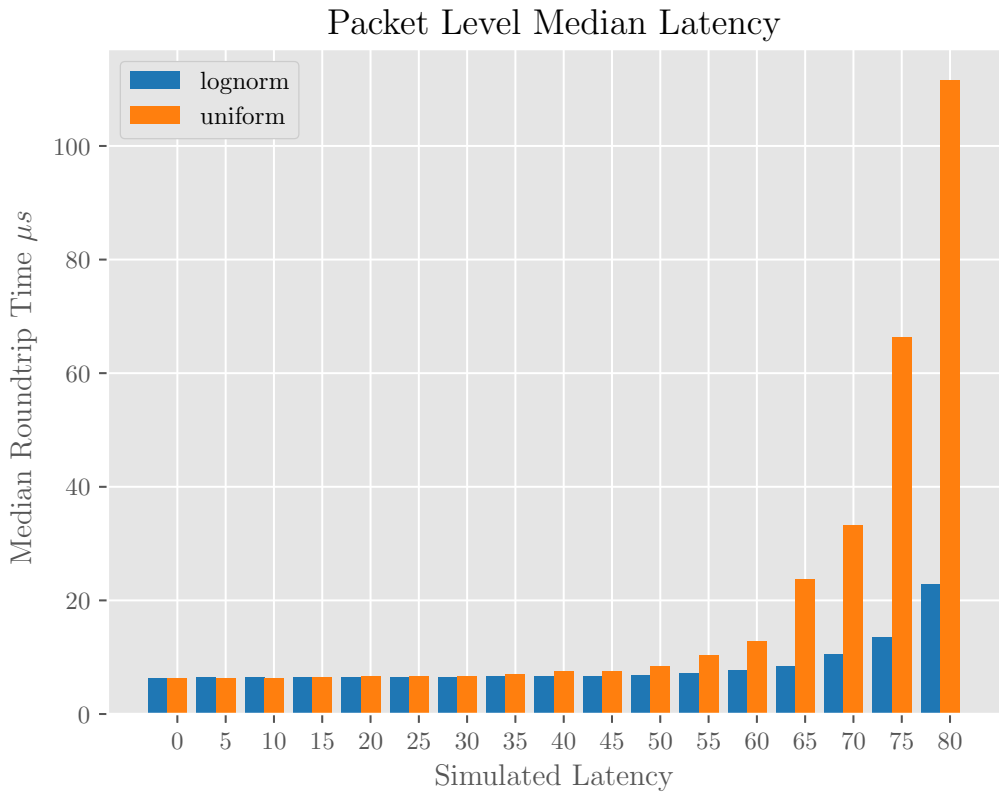


Figure 3.18: Packet level test with medians. Observe that lognorm has a lower median at higher simulated latencies, but matched mean. This indicates there are a small number of large latencies at higher simulated latency levels.

the median to a few extreme values. The vast majority of the latencies in the lognormal are shorter than the mean resulting in a smaller median latency than uniform distribution.

3.3.4.3 Application Level

Unlike the packet and library level, the application runtimes follow normal distributions. As such we are justified in using traditional statistical parametric methods to evaluate these results. We show only the results for the mean, there are no substantive differences between the mean and median for these results. This can be explained in part by the central limit theorem which states that the sum of independent random variables tends towards normality when the sample is suitably large. Secondly, unlike the previous levels, we are measuring total application run time as opposed to a particular message delivery time. Both of these factors drive the overall distribution towards

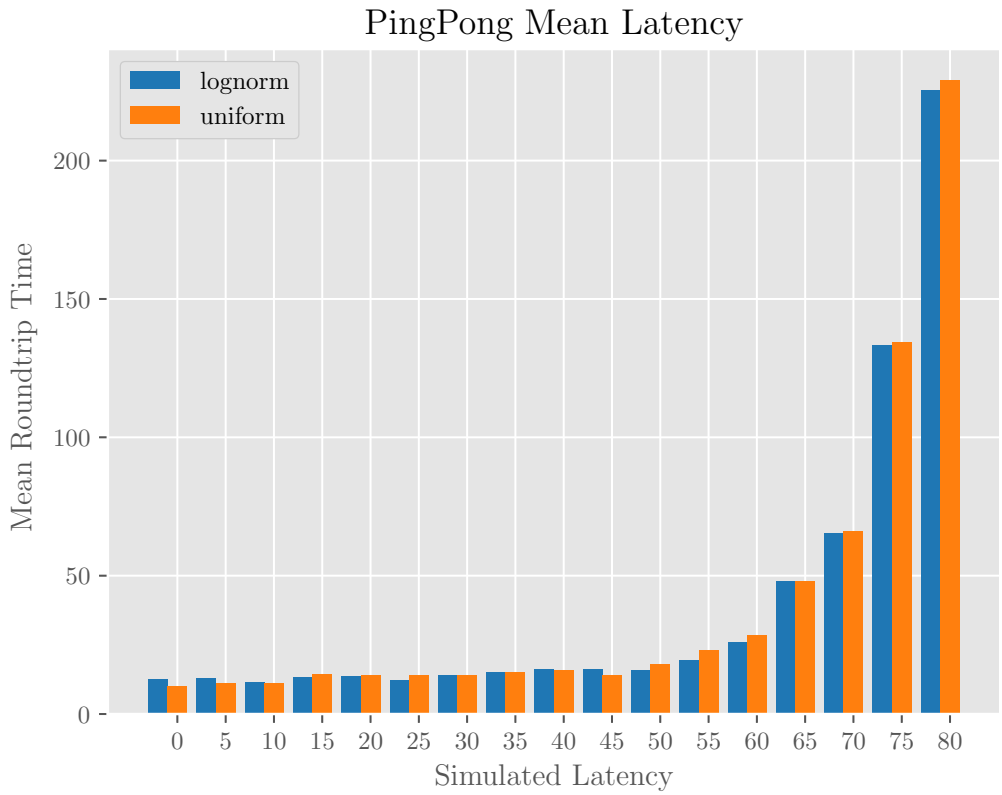


Figure 3.19: The results from the verb level carry over to the library level. The mean is slightly higher which represents higher than the packet level tests which represents overhead at the library level.

normality and, in effect, the mean towards the median.

There are some interesting results at this level. First, observe in Figure 3.24 that the majority of the applications are unaffected by the increases in latency mean at higher synthetic latencies. The principal exception is the LU solver code which roughly doubles at higher latency values. This can be explained by the communication–intensive nature of the LU solver.

Then, observe in Figure 3.23 that several of the applications increase run times at higher levels of variation in synthetic latency. The LU code roughly increases its runtime by a factor of 3.5. The communication intensive CG, FT, and MG codes also show increases in runtime that are not visible in Figure 3.24. This suggests that the latency variation has a significant effect on the runtime of applications beyond that of just latency.

In Experimental Suite I we have demonstrated that latency variation is sufficient to cause

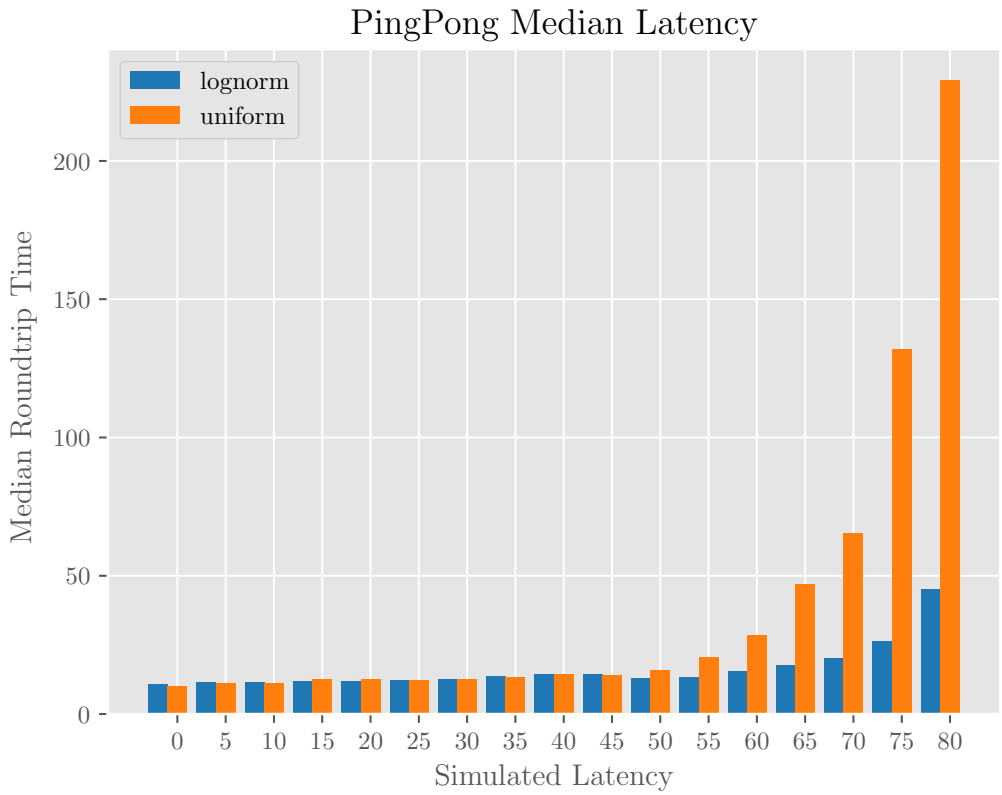


Figure 3.20: The results from the verb level carry over to the library level. The median is also slightly higher which represents higher than the packet level tests which represents overhead at the library level.

application degradation even absent the affects of contention or congestion. We traced these effects from the packet level up to the application level.

3.3.5 Experimental Suite 2: Latency Variation is More Highly Correlated with Application Degradation than Latency Mean

In this section we examine the claim that latency variation better explains application degradation than latency mean. We analyze the results from the previous section. We focus on the application layer by analyzing the results from the NPB tests. For each test from NPB, we plot its application runtime against latency mean for the injected distribution. We assess the strength of a linear relationship between runtime and mean using Pearson's Coefficient of Correlation. Similarly, for each test from NPB, we plot its application runtime against latency standard deviation for the

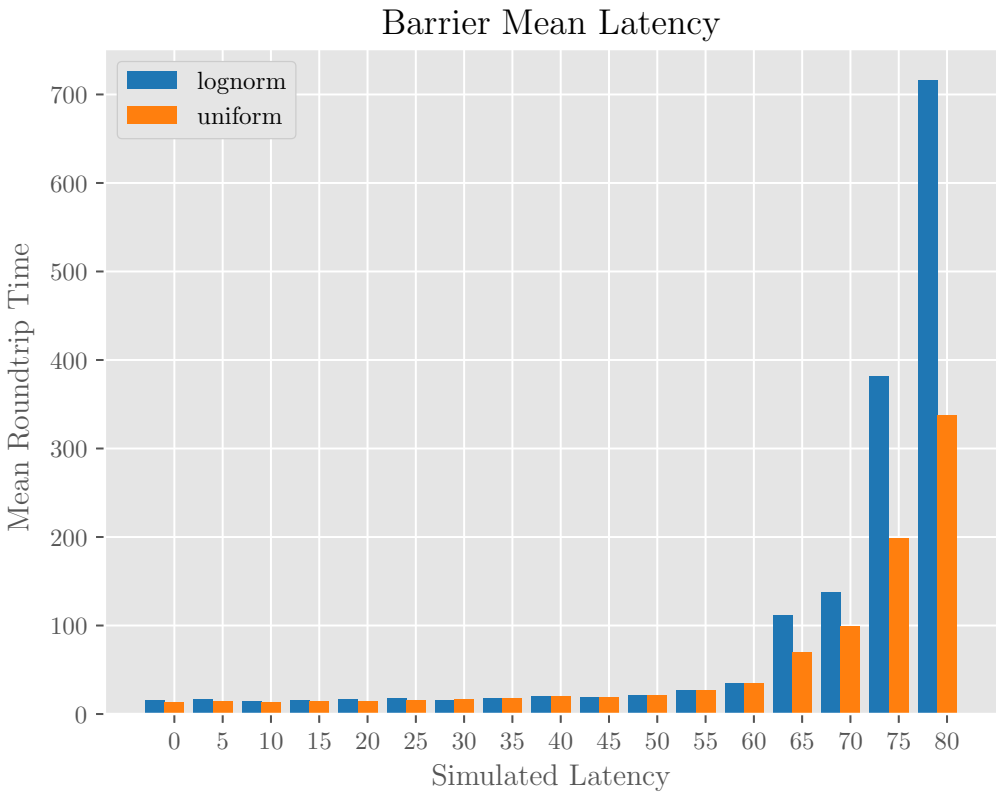


Figure 3.21: Unlike the packet level test, the means diverge. This is to be expected as barriers are as long as longest synchronization.

injected distribution. Again, we assess the strength of a linear relationship between runtime and standard deviation using Pearson's Coefficient of Correlation. We carefully choose the underlying distributions to inject so as to hold the mean constant for subset of experiments that correspond to a given simulated congestion level. We vary the standard deviation of the latency in each subset of experiments by adjusting the scale parameter of the injected distribution.

Figure 3.25 shows the relationship between the mean latency and application runtimes. We observe that a linear relationship is a plausible model for explaining application runtimes with respect to latency mean. Similarly, Figure 3.26 shows the relationship between the latency standard deviation and application runtimes. Again, a linear relationship is a plausible model for explaining application runtimes with respect to latency standard deviation. We further observe that of the five applications tested, the performance of the LU Decomposition application suffers the most radical effects of changes in latency. In the worst case in with a latency standard deviation above 400, the

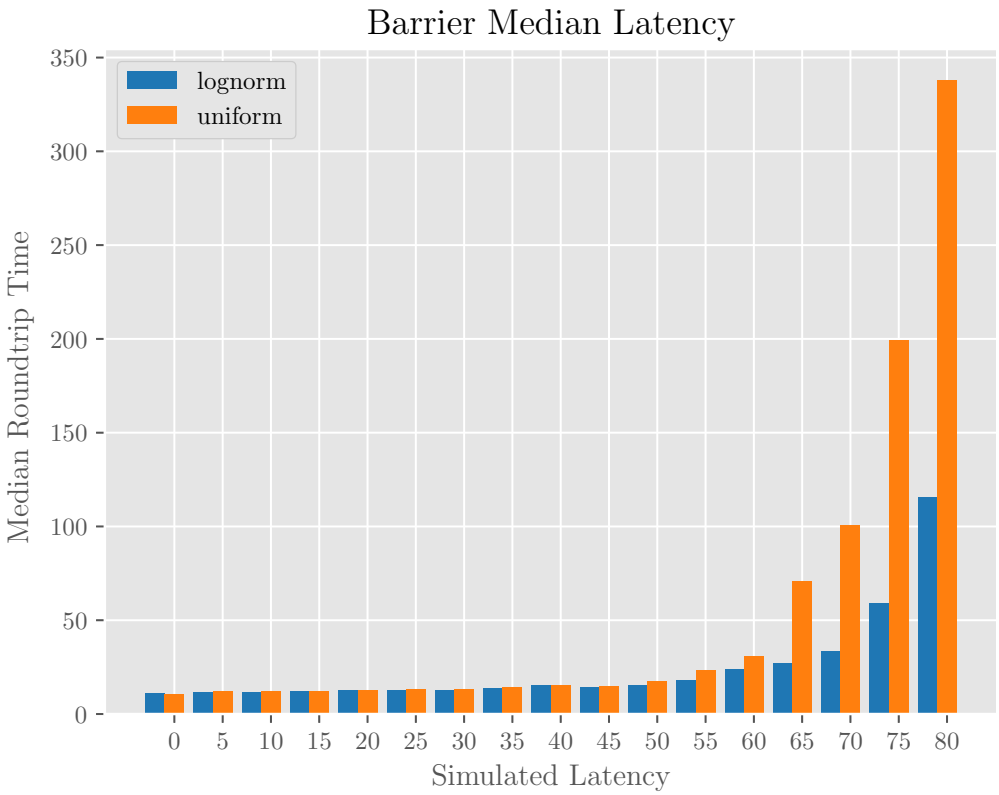


Figure 3.22: The median results are similar to ping pong with slightly higher latency. This is consistent with the medians robustness to extreme values.

application runs 3.5 times slower than when the standard deviation near zero.

We compute Pearson's correlation coefficient for each test. These results are summarized in Table 3.6. Most experts agree that a sample correlation coefficient (usually labeled r) above 0.7 is evidence of a linear relation, and a correlation coefficient above 0.9 is evidence of a strong linear relationship [105]. As shown in Table 3.6, the correlation coefficient, r_{mean}, that tests the strength of the linear relationship between the FT application runtimes and latency mean is $r_{mean} = 0.70$. The correlation coefficient, r_{std}, that tests the strength of the linear relationship between the FT application runtimes and latency standard deviation is $r_{std} = 0.89$. The correlation coefficients for four other NPB applications are also shown in Table 3.6.

The values in the column labeled "Significant?" are calculated using the Fisher z trans-

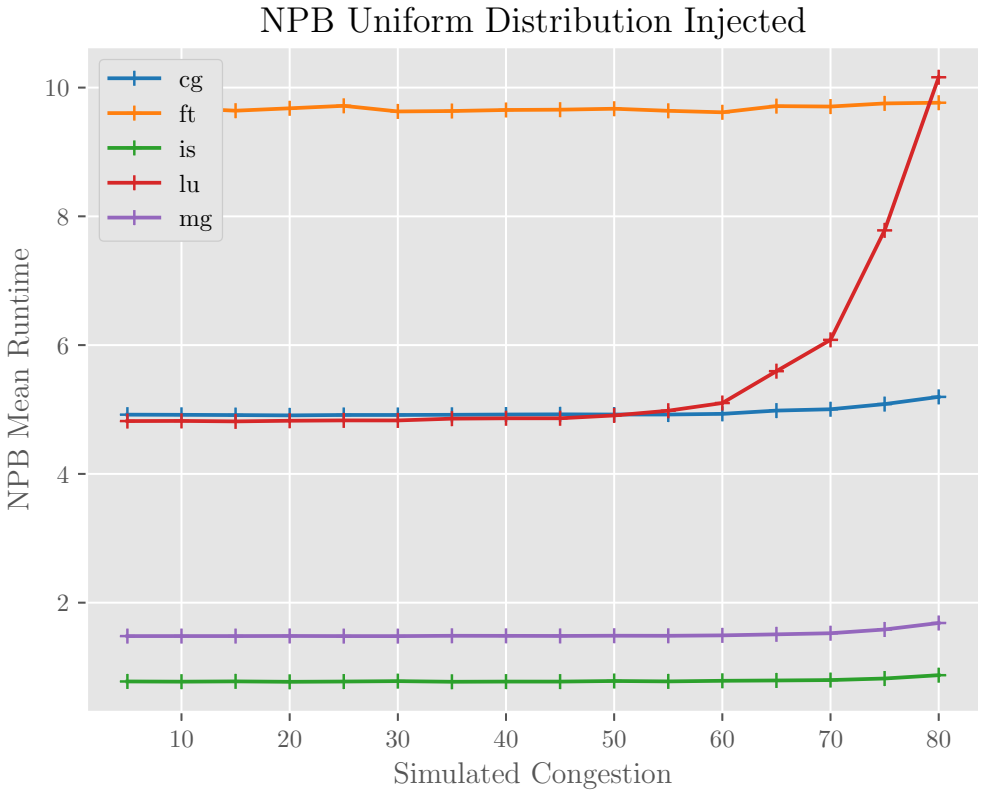


Figure 3.23: The increased mean results in slightly higher runtimes in LU only.

form [105], as:

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{z} e^{\frac{-t^2}{2}} \, dt$$

where,

$$z = \frac{arctanh(r_{mean}) - arctanh(r_{std})}{\sqrt{\frac{2}{n-3}}}$$

The very small values (close to zero) in the column labeled "Significant?" indicate that the difference between r_{mean} and r_{std} is highly statistically significant (to the level of $\alpha = 0.05$) for all applications tested. This means that the network latency variation is more highly correlated with application runtimes than network latency mean values. Pearson's coefficient is not considered to be robust to outliers. However, the residuals between the line of best fit and observed values are small. Thus, outliers are not a primary factor in the correlation coefficients.

Finally, we examine the performance impact of latency variation on application runtime. We

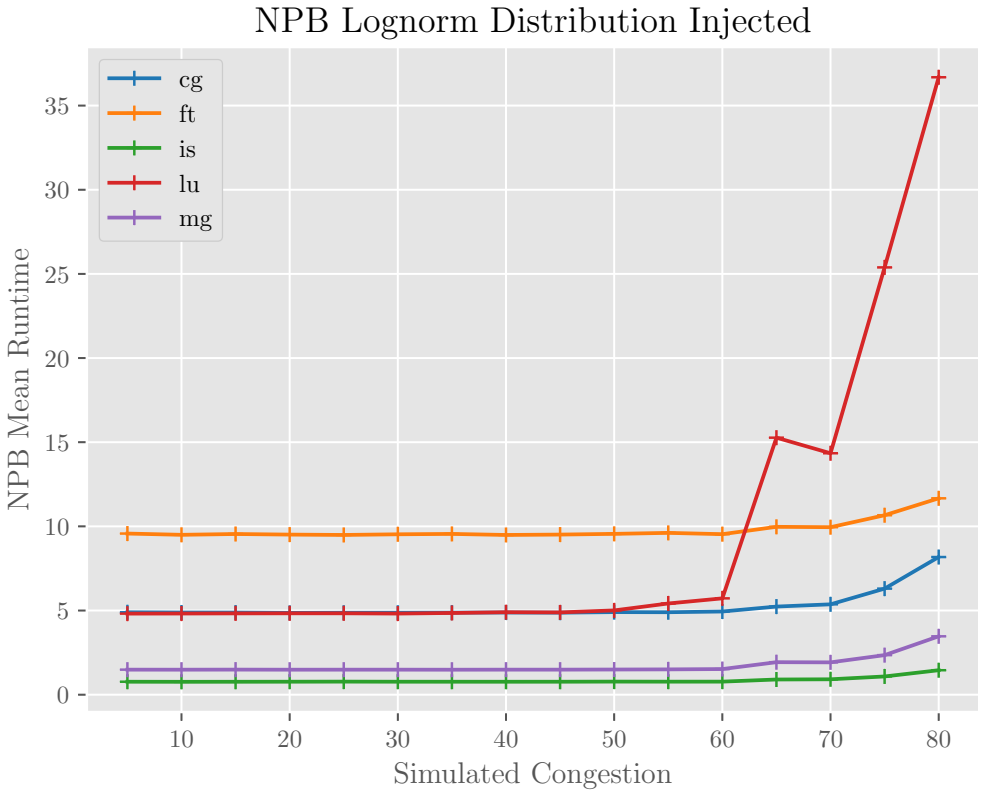


Figure 3.24: The increased spread results in significantly greater runtime for LU, but also greater run times for the communication intensive CG, FT, and MG.

examine two sets of experiments. In each set the mean network latency is fixed and there are several values for the network latency variation. We choose experiment sets with network mean latency that correspond to a congestion level of 20% and a congestions level of 70%, which are typically-observed means in cluster networks. In Figure 3.27, we see that for a fixed mean corresponding to a congestion of 20%, changes in the standard deviation of network latency appear to have a limited effect. However, in Figure 3.28, we see that for a fixed mean corresponding to a network congestion of 70%, increases in standard deviation cause a substantial increase to application runtimes. In particular, LU shows a nearly 25% increase in runtime with the larger latency variation, while CG and MG show more modest increases in runtime of 5% to 10%. Together, these charts suggests future work to study at which level of latency mean that latency variation begins to have a substantial effect on application runtime.

Based on the results of Experimental Suite II, we observe that the mean latency is strictly

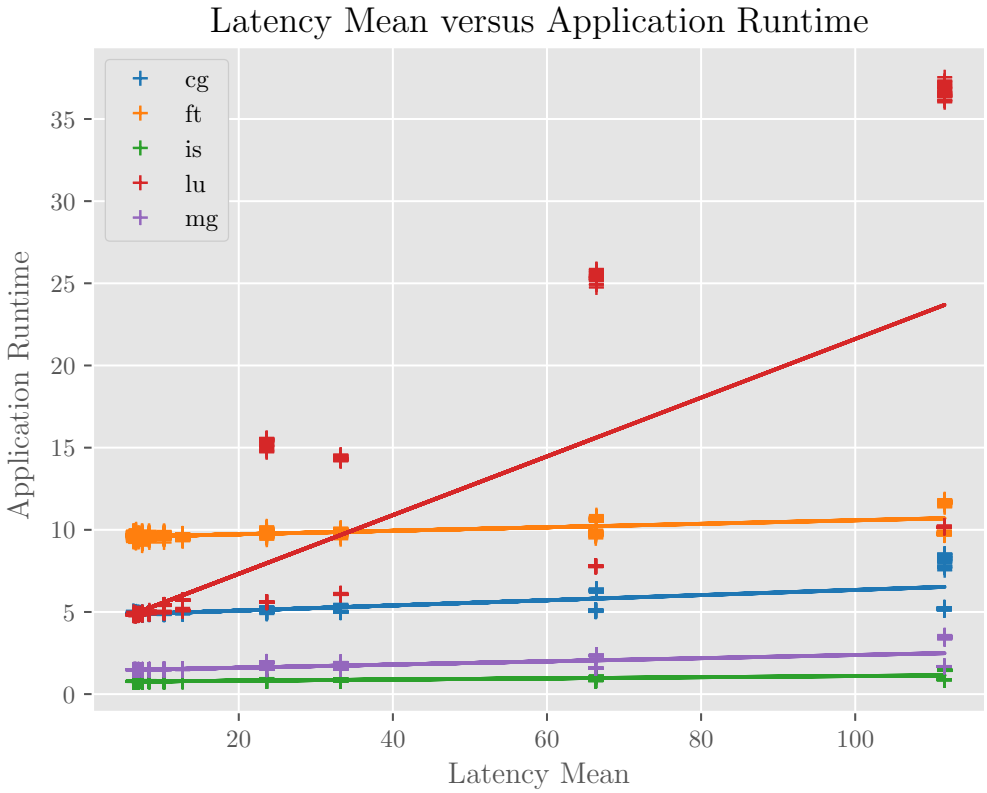


Figure 3.25: Mean runtime vs increased latency mean. Random scatter above and below the line of best fit indicates suitability of a linear model. High and low spreads about the line of best fit correspond to different standard deviations.

less correlated with application runtime than latency standard deviation for all tests we considered. We calculated the significance of the difference in the correlation coefficient results using Fisher's Z transformation. We find the results to be statistically significant, (to the level of $\alpha = 0.05$) indicating that the measurements that are this different are unlikely to occur by chance. The key result is that latency variation is more highly correlated with application runtime than latency mean.

3.3.6 Summary

We used several tools to generate synthetic network traffic in patterns related to HPC applications. First, we used low-level benchmarks from the OFED `perftest` package to gauge performance with minimal software-induced noise. Then, we employed MPI primitives to measure the relationship between packet-level performance and distributed communication mechanisms. Finally,

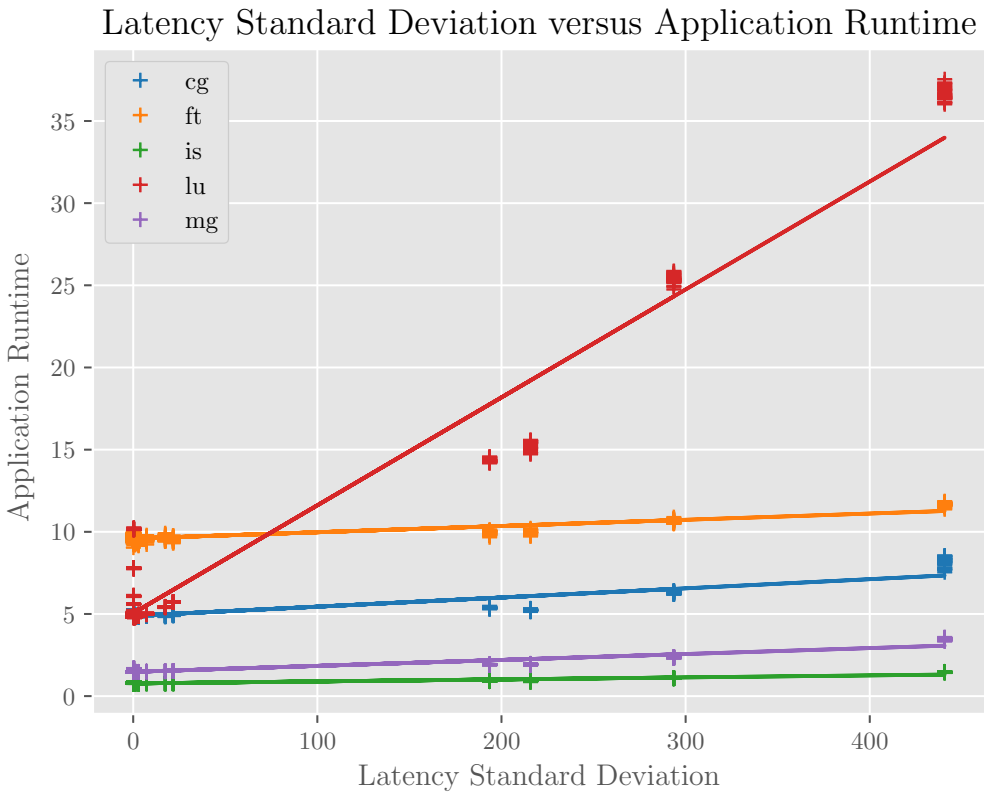


Figure 3.26: Mean runtime vs increased latency variation. Tight fitting about the line of best fit indicates the strength of model. Unlike the mean, variations above and below the line are not correlated with different means.

we measured the cumulative effect of induced packet-level latency on the synthetic traffic patterns and overall performance of the NAS Parallel Benchmarks.

We found statistically significant evidence that latency variation is more highly correlated with HPC application performance than latency mean alone. This result is somewhat surprising, since studies that focus on mean latency alone have shown the strong impacts of low message latency to applications in an HPC environment. We believe that these results may be important to consider in the design of scalable HPC systems and applications. As multitenancy becomes increasingly common in dedicated HPC systems and as HPC applications are more commonly run on cloud architectures, competition for network resources could lead to increased levels of network latency variation. We have demonstrated that serious degradation of application performance can result from high variation in latency.

Table 3.6: Correlation Results Rounded to the Nearest .01

Test	r_{mean}	r_{std}	Significant?
FT	0.70	.89	$p = 7.89 \times 10^{-33}$
CG	0.73	.91	$p = 4.94 \times 10^{-39}$
IS	0.76	.93	$p = 8.89 \times 10^{-50}$
MG	0.73	.95	$p = 1.31 \times 10^{-93}$
LU	0.76	.97	$p = 5.24 \times 10^{-148}$

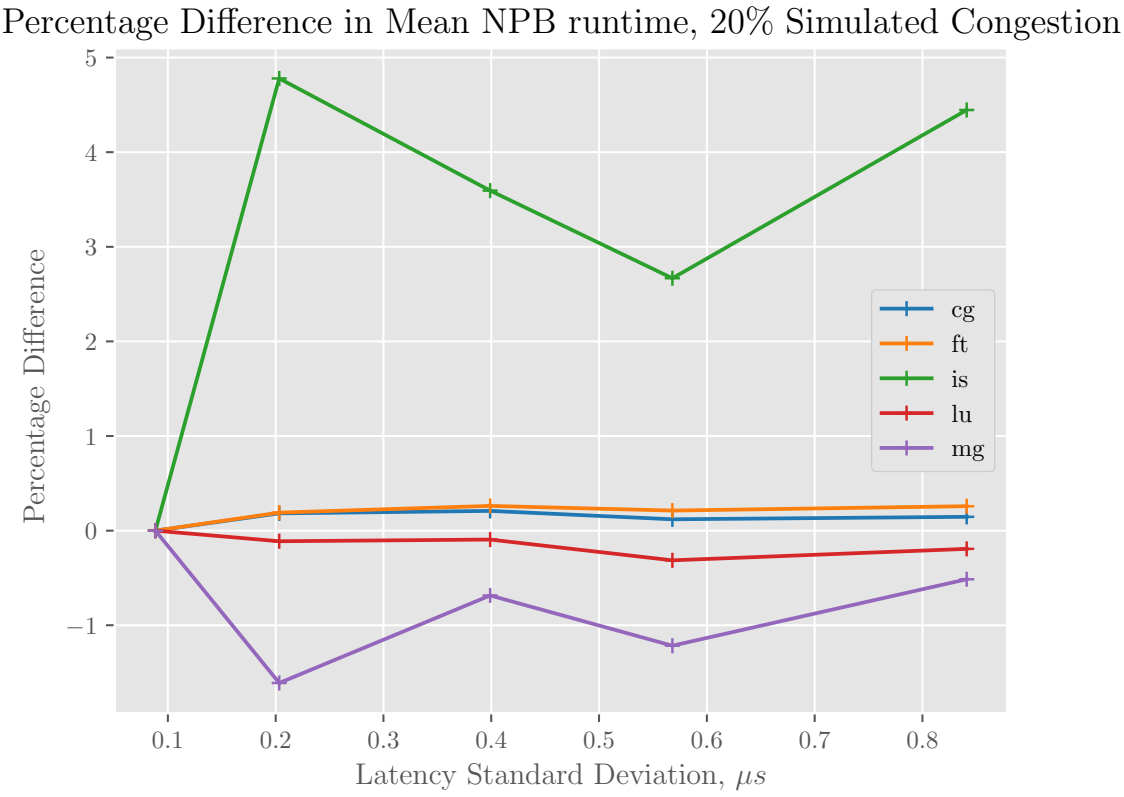


Figure 3.27: At the 20% level of network congestion, increases in standard deviation have a limited effect to application runtimes.

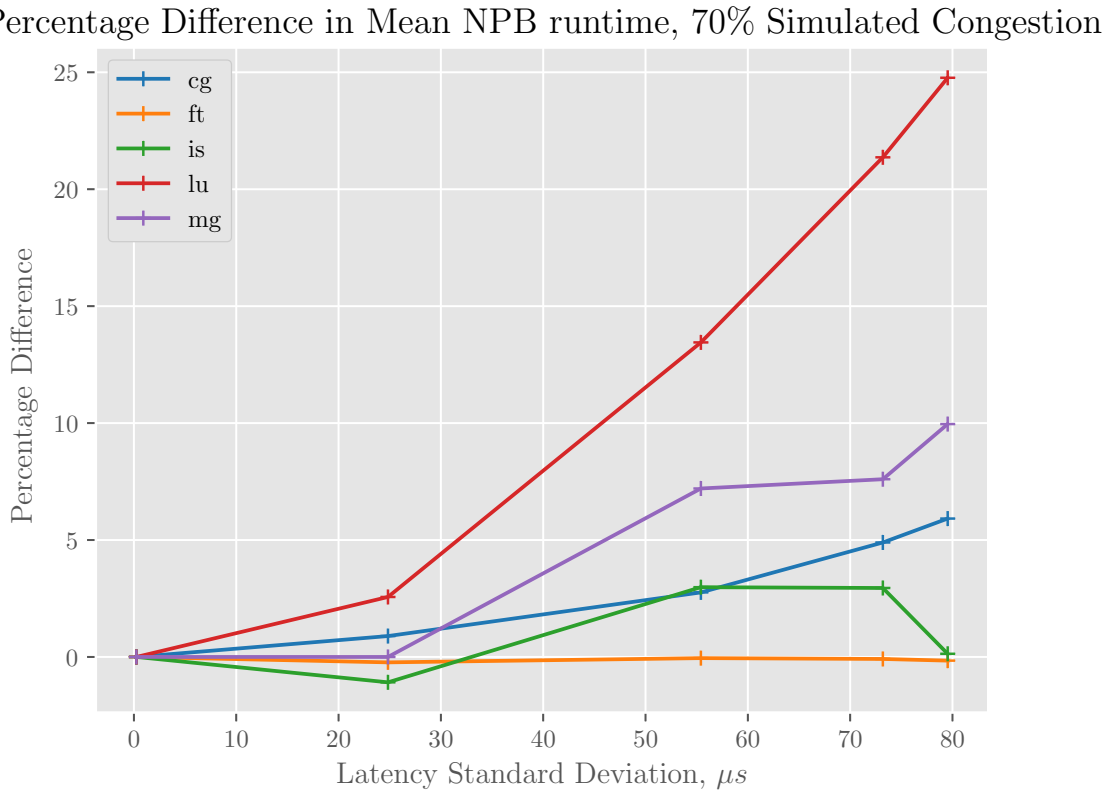


Figure 3.28: At the 70% level of network congestion, increases in standard deviation correspond to significant increases in application runtime.

3.4 Measuring Latency Variation in Cloud HPC Systems

Our prior work measures and characterizes the impact that the variation in network latency also has to application performance [176]. Even when the mean network latency is the same across compute nodes, higher variation in latency leads to a degradation in parallel application performance. This effect is significant enough that some environments that have networks with lower mean network latency but higher network latency variation have application performance that is worse than an environment with higher mean latency but lower network latency variation. We note that while we can demonstrate that latency variation in HPC systems correlates with HPC performance degradation, the small latency variation in a modern supercomputer with low-latency networking technologies such as InfiniBand generally makes this degradation minimal.

However, the commercial cloud has network performance characteristics that are much different than locally provisioned HPC clusters. The commercial cloud could be an abundantly available resource for research computing, but several challenges exist for execution of parallel applications in the cloud. Some challenges have been overcome in recent years. For example, the overhead associated with the virtualization environment of the commercial cloud was a significant factor in performance degradation, but the virtual machine implementations today provide only minimal impact to computation performance [82, 95]. However, the performance of network messaging between cloud compute nodes can still be significant factor in poor performance of parallel applications [82, 69].

Executing HPC applications on the cloud is a complex task. Due to the number of possible configurations and options to choose from, a user has many options on how to execute their HPC application. However, not all of these combinations will yield good performance and some configurations may negatively impact the performance of HPC applications. One of these configuration parameters that can be varied is the network configuration and placement of the different instances within the cloud.

Our contributions in this work are to provide a thorough measurement study of the point-to-point latency characteristics of instances in two clouds, Amazon Web Services (AWS) and Google Cloud Platform (GCP). We characterize performance across a range of node and network configurations. Utilizing the NASA Parallel Benchmark (NPB) [32] suite and Large-scale Atomic/Molecular Massively Parallel Simulator (LAAMPS) [137] benchmark, we evaluate the performance of parallel messaging and application execution with different node and network configuration options. We

test across both AWS and GCP and compare performance across the two clouds. Our contributions are, finally, to give insight to the most appropriate configurations for effective parallel application execution in these commercial clouds.

3.4.1 Background and Related Work

Cloud computing is the delivery of different types of on-demand computing resources through the Internet that typically utilize a pay-as-you-go-model [23, 77, 120, 91]. Although the services, terminology, and hardware can vary between different cloud providers, there are common characteristics that are similar across clouds. The construction and design of a cloud provider is similar to a traditional HPC environment as a cloud is simply a large collection of computing resources controlled by software. However, unlike HPC environments a typical cloud is designed to be a general purpose resource and is not highly optimized for parallel workloads as is a traditional HPC environment.

For this research, we focus on two specific cloud providers: Amazon Web Services (AWS) and Google Cloud Platform (GCP). We focus on these providers as they offer a large number of hardware combinations and services and are among the top three leading cloud providers in the 2018 Gartner Magic Quadrant [158].

3.4.1.1 Compute Resources

The services that are available vary between different cloud providers. Along with the variations in services, each cloud provider also differs in both terminology and the combinations of hardware that are made available to users. However, there are some terms and concepts that apply to both clouds. Both AWS and GCP refer to each server as an *instance*, referring to the traditional allocation of a *virtual machine (VM)* that is managed by a *hypervisor*. The hypervisor manages access to the underlying computing resources and hardware contained within the cloud. AWS and GCP also both utilize the term *vCPU* when referring to the compute power of an instance, which is a hyperthread of an available CPU hardware platform [22, 74].

AWS provides "Enhanced Networking Capabilities" through Single Root I/O Virtualization or SR-IOV. SR-IOV is a network virtualization technique that provides higher I/O performance and lower CPU utilization compared to traditional implementations for supported AWS instance types [18].

Another feature of AWS is the availability of "bare-metal" instances. These instance types

provide direct access to the compute and memory resources of the underlying server. Bare metal instances are built on AWS's Nitro system which is a collection of AWS-built hardware offload and server protection components that come together to securely provide high performance networking and storage resources to EC2 instances. Utilizing these instances it is possible to show the differences between VM based instances and the "bare-metal" instances.

3.4.1.2 Instance Placement and Networking

Both AWS and GCP refer to the specific geographic location where users can provision resources as a *region*. Within these regions, there are multiple datacenters that are divided up into *zones*. Each of the zones within a region are isolated from each other and are connected by a low-latency network [21, 75]. Although the terms for these zones differ slightly between the two clouds, Availability Zone (AZ) for AWS and Zone for GCP, they are equivalent in meaning. We will be using the term Availability Zone (AZ) when referring to both AWS and GCP in this paper for simplicity.

AWS and GCP have differing mechanisms to govern network performance between instances. In AWS the instances have a network performance category such as Low, Moderate, or High that refer to throughput limits from 5 Gigabit up to 25 Gigabit [19], and are assigned by the instance type's intended use. Throughput limits for GCP, on the other hand, are similar for all instance configurations and are simply given as 2Gb/s per vCPU and capped at 16Gb/s [76]. Even with these specifications, the network performance of instances in both AWS and GCP varies due to other factors such as the overall utilization of the network and other competing instances that may be located on the same underlying hardware.

AWS EC2 has a unique feature called a *placement group*, which is available only for certain instance types and allows clients to broadly specify how the instances are placed on the underlying hardware. A *clustered* placement group ensures that EC2 instances are able to communicate with all other instances within the placement group at the full line rate of 10 Gb/s flows and 25 Gb/s aggregate without any slowing due to over-subscription, and is most suitable for instances that require either low latency or high throughput. A *spread* placement group ensures that instances that placed on distinct underlying hardware to help ensure high availability [20].

3.4.2 Experiment Design

AWS Latency in the experimental environment is configured indirectly through the selection of the architecture of the parallel system in the cloud. In AWS, these architectural choices include the number of Availability Zones and the decision to use Placement Groups.

We have identified a number of network configuration use cases that we study in this paper including: single AZ, multi-AZ, single AZ with clustered placement groups, and single AZ with spread placement groups. The first two use cases apply to both AWS and GCP while the third and the fourth use case apply directly only to AWS as they utilize the AWS specific placement group construct.

The first use case that we will be exploring is the single AZ. This use case was chosen because it is the most likely scenario for a user who is attempting to execute their workload in the cloud. This use case does not take advantage of any of the additional network configuration or optimization that AWS has in order to simulate a user who may be unfamiliar with these advanced cloud constructs. In this use case all of the instances are provisioned in the same AZ and within the same subnet without any additional parameters specified.

The second use case is multi-AZ. This use case was chosen as executing in multiple AZs is useful for large scale HTC workloads that require a lot of computational resources which may not all be available in one AZ. This is also a typical use case when executing HTC workloads utilizing the AWS Spot Market as the price for each instance type varies per AZ. Due to Spot Pricing being based upon supply and demand, if a user were to request all of their instances within the same AZ, they would drive the price up and therefore be unable to obtain all the resources they require for their specified price. However, if the user were to spread the instances out over multiple AZs, the user would diffuse the demand and therefore not increase the price on themselves. While this strategy does not directly apply to preemtible instances on GCP, it is possible that there are resources in one AZ that do not exist in another AZ and that a user would have to utilize multiple AZs in order to obtain all the resources they needed for their workload.

The third use case is a single AZ with a clustered placement group. As previously mentioned, a clustered placement group within AWS is supposed to allow instances to communicate with other instances inside the placement group at peak network performance. By controlling the placement of the instances so that they are located in the same physical proximity, a clustered placement group

limits the distance that the network traffic between instances has to travel. This indicates that this configuration should out perform the single AZ configuration for applications that are network bound. This is another use case for large scale and tightly coupled HPC applications that utilize MPI as the additional throughput and lower latency offered by clustered placement groups should help increase performance. This use case simulates a user who is more cloud aware and wants to take advantage of some of the additional capabilities offered by the cloud in order to potentially increase the performance of their application.

The fourth use case is the single AZ with a spread placement group. As previously discussed, a spread placement group ensures that the instances within the group are placed on distinct underlying hardware. AWS specifies high availability and fault tolerance as the use case for spread placement groups, we utilize this use case as a worst case scenario where none of the instances provisioned are in close proximity to each other. Although we have no guarantees on where the hardware is physically located, since the stated goal of AWS is high availability we can safely assume that the instances in this group are on different failure planes which allow us to get a measurement of what one of the possible worst case scenarios is if a user happens to launch instances within a single AZ and gets instances that are not on the same hardware.

3.4.3 Latency Characterization

In this section, we describe our exploration of the network latency characteristics of both AWS and GCP cloud infrastructures. Our primary measurement of interest is the latency of a single message between two nodes at the MPI layer. We also include results from the related measurement of *throughput* between nodes, as latency can be a contributing factor in throughput degradation in guaranteed delivery protocols such as Transmission Control Protocol (TCP).

3.4.3.1 Methodology

We use two measurement tools to quantify internode latency and throughput. First, we wrote an MPI application to repeatedly measure the time taken to perform individual `MPI_Send` calls. This was chosen over other readily available tools because we required individual measurements to find their distribution, while other tools use a more standard process reporting an average latency measurement over many iterations. Second, we used the industry standard tool `iperf` [10] to measure the TCP throughput between nodes.

To help ensure a comprehensive study of the AWS and GCP cloud infrastructure, we made efforts to draw measurements from as many unique nodes as was feasible given our resources. In provisioning nodes for measurement, we were constrained by two factors. First, the `cluster` and `spread` placement groups were limited to 7 instance members within an availability zone. Second, the `us-east1` region has 5 availability zones with the `i3.metal` and `i3.16xlarge` instance types that we used. Therefore, our methodology for internode measurements was to provision 7 instances at a time when using placement groups, and 5 instances at a time when measuring between availability zones. We then conducted all internode latency and throughput samples between each unique pair, giving us 21 and 10 measurements per provisioning, respectively. Each provisioning was repeated 10 times over the course of 24 hours, resulting in measurements between 210 unique node pairs for each placement group, and 100 unique node pairs between availability zones.

While individual messages sizes in an MPI application can vary widely, we determined that it would be prohibitively time consuming to expand our independent variable matrix to include a variety of message sizes. Furthermore, it seemed unlikely that such an effort would yield interesting differences in our results. To test this assumption, we measured the latency between pairs of nodes in AWS using a spectrum of message sizes, and plotted the mean and range of the measurements in Figure 3.29. We observe that the mean latency is similar for message sizes up to 1024 bytes, which is expected as the transmission time with a 25 Gb/s interface is only 0.33 nanoseconds per byte. For message sizes larger than 1024 bytes, the Maximum Transmission Unit (MTU) size of 1460 for GCP and 1500 for AWS causes packets to be fragmented into multiple Ethernet frames, which is reflected by the increasing mean latency. This, coupled with the similar measurement range between each message size, gives us reason to believe that restricting our further characterization to a single message size will yield results that are broadly applicable. We chose a message size of 1 byte for further latency measurements in this section.

Within the AWS cloud, the option to provision non-virtualized instance types offers an opportunity to quantify the effects of the AWS hypervisor on internode latency. These instances, labelled by AWS as "`i3.metal`" and referred to as *metal* in the following text and figures, use [to the best of our knowledge, reword] hardware identical to the "`i3.16xlarge`" instances to which we compare them.

Choosing a single availability zone that contained both virtualized and bare metal instance types, we studied the internode latency of both types with a sequence of individual measurements

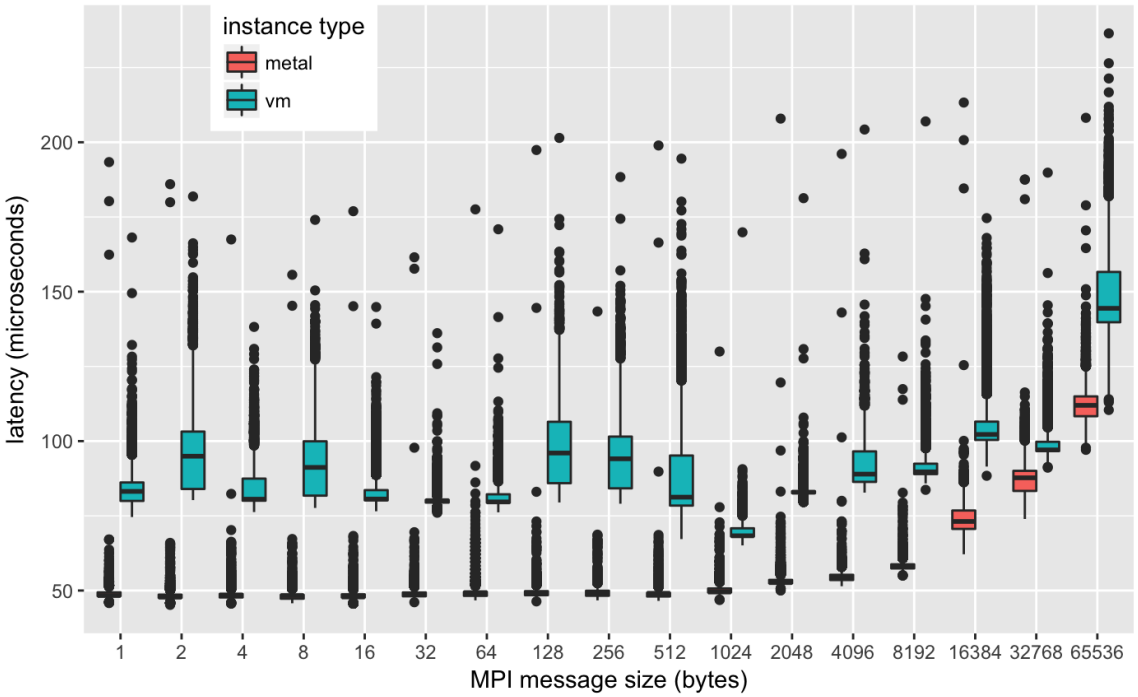


Figure 3.29: Mean and interquartile ranges of internode latency on AWS bare metal instances, AWS VMs, and GCP VMs.

and plotted the cumulative distributions, shown in Figure 3.30. We observe that the distribution of measurements between the virtualized instances exhibits higher median value and variation than measurements between bare metal instances. Furthermore, the vast majority of the 230 samples drawn from virtualized instance pairs feature a clear bimodal distribution similar to the given example. We believe that these characteristics are due to an extra level of queueing added by the AWS hypervisor. Other samples using different node pairs and availability zones exhibit very similar characteristics, and have been omitted for brevity.

In order to perform an accurate assessment of the AWS cloud infrastructure, we have chosen to treat virtualization as a control variable, and evaluate its effects independently. For the remainder of this section, we will use the AWS bare metal instances to measure characteristics of the cloud infrastructure, and then conduct a more thorough investigation of the effects of virtualization on network latency at the end.

One of the challenges we encountered in our exploration of cloud network characteristics was that, given identical provisioning parameters, the latency measurements between one pair of nodes could be significantly dissimilar from internode measurements of another pair. As an example,

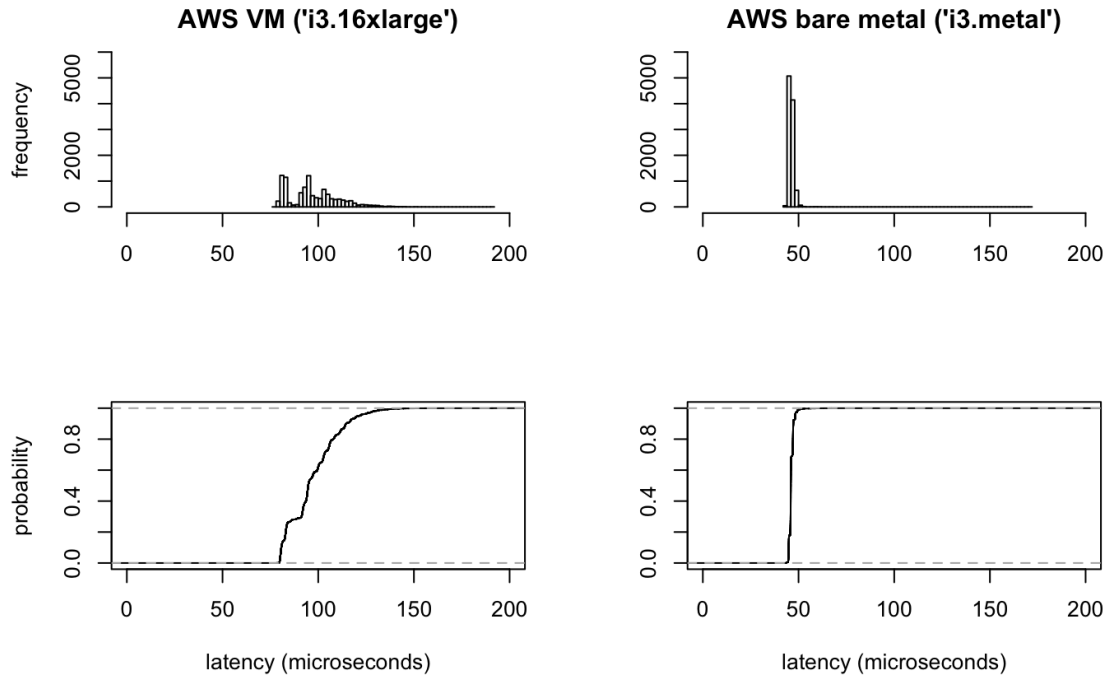


Figure 3.30: Histograms and cumulative probability distributions of internode latency measurements between virtualized and bare metal instance types in AWS.

Figure 3.31 shows measurement studies taken from four different pairs of nodes, while within the same AZ and using the `cluster` placement group. The sets of measurements all exhibit long-tailed characteristics, but have a range of different median values, variances, and modality.

Exploring this phenomenon further, we observed that the distributions could be classified visually, which supported our intuition that the dissimilarity in latency distributions could be explained by differences in network cabling and packet forwarding devices between nodes in a pair. We classified the distributions by their similarity by first building a matrix of 2-sample Komogorov-Smirnov (KS) test statistics [117] comparing each distribution to all other distributions, and then used K-medoids clustering to group the most similar distributions. Cluster sizes were chosen using the `pamk` implementation of the partitioning about medoids (PAM) method [103]. Sizes were verified using the "elbow" method of visualizing the inflection point of information gain using the total within-cluster sum of squares metric.

A visualization of the results are shown in Figure 3.32. We observe that there are reasonably clear distinctions between groups, and upon further visual inspection, the histograms of samples belonging to the same cluster are clearly more alike than those between different clusters. While we

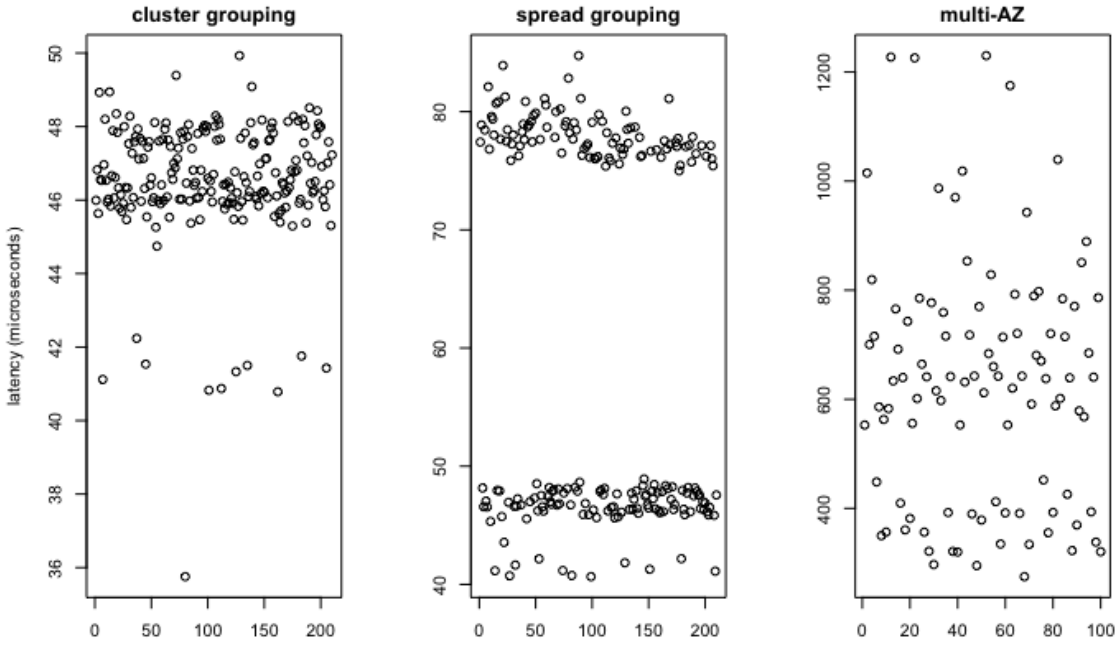


Figure 3.31: Median latency values between node pairs in AWS using `cluster` grouping, `spread` grouping, and between AZs.

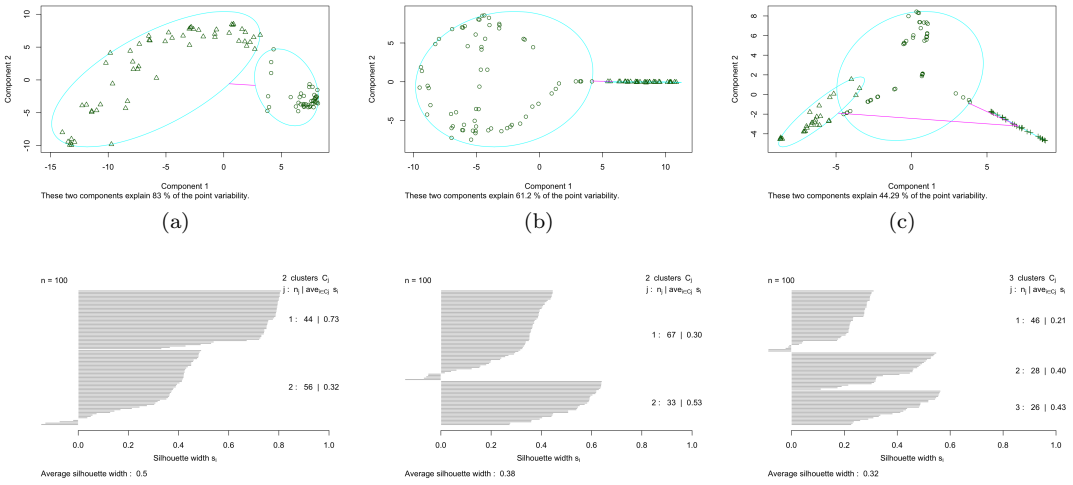
cannot know the infrastructure differences that cause these clusters to arise, we surmise that our clustering may be a reasonable approximation of similar network paths between nodes.

For further measurements in this section, we have specifically chosen node pairs that belong to the same similarity cluster as a control variable. This is done to ensure that our results are repeatable and focused on the independent variable in question. In other sections of this paper, where we will be interested in studying the cloud environment as a whole, we will clearly state whether or not experiments use a subset of nodes that share similar network characteristics.

3.4.3.2 Node Grouping Parameters

We explored the effect of the available node grouping options in AWS on network characteristics by conducting sequences of internode latency and throughput measurements between pairs of instances assigned via the grouping parameter.

With the `cluster` placement group option, referred to as *cluster*, we expect instances to have a short network distance, possibly within the same datacenter rack. The alternative placement group option `spread` is described as maximizing availability within an AZ, so we hypothesize that internode



(a) (b) (c)

(d) `cluster` group, $n = 100$, $k = 3$ (e) `spread` group, $n = 100$, $k = 5$ (f) Separate AZ, $n = 100$, $k = 10$

Figure 3.32: K-medoids clustering of internode latency distribution similarity measured by 2-sample KS test statistic.

network distance could be greater, possibly in different facilities. Finally, with no placement group option selected and nodes selected from different AZs, referred to as *multi-AZ*, the network distance could be even greater.

Table 3.7: AWS Internode Latency by Grouping Option

grouping	min	Q1	med.	mean	Q3	max	var.
cluster	65.9	78.8	90.9	93.3	102.4	178.4	243.9
spread	187.8	203.8	211.6	215.2	223.5	418.8	341.9
multi-AZ	523.9	584.1	589.7	590.2	595.9	803.2	230.3

The differences in measurements taken from each grouping type are readily observable, as shown in the sequence plot in Figure 3.33 and described in Table 3.7. As expected, the *cluster* grouping measurements exhibited a lower mean value, followed by the *spread* grouping measurements. However, perhaps counterintuitively, the measurements between nodes in separate AZs exhibited lower variance than those that were supposedly connected by a shorter network path.

3.4.3.3 Virtualization Effects

Using the same node pairs, we also conducted a throughput study using `iperf` over a 60 second sustained transfer with measurements taken at 1 second intervals. We saw no significant

Figure 3.33: Internode latency measurements from AWS with different node grouping options.

difference in median throughput between virtualized and bare metal instances, but we did observe a number of low outlier measurements in the virtualized instance pairs, as depicted in Figure 3.34. These outliers accounted for 1.1% of the total measurements. Of further note is that the observed maximum throughput is well below the 25Gb/s theoretical maximum of the instance's network interface. This is explained by the AWS documentation of a cap on throughput within an availability zone at 10Gb/s as of the time of this writing. We believe that further effect of virtualization on the throughput may be masked by the AWS cap.

3.4.4 Implications to Synthetic Benchmarks

We first conducted a series of trial measurements to ensure that our problem sizes in both NPB and LAMMPS were large enough to capture normal network performance variance and operating system interference. Since we sought to compare the performance of dispatching on a cloud infrastructure to that of a typical HPC cluster, we used the Palmetto Cluster [cite] as a baseline. Problem sizes were adjusted until the 99% confidence interval width over 30 runs was within 5% of the mean.

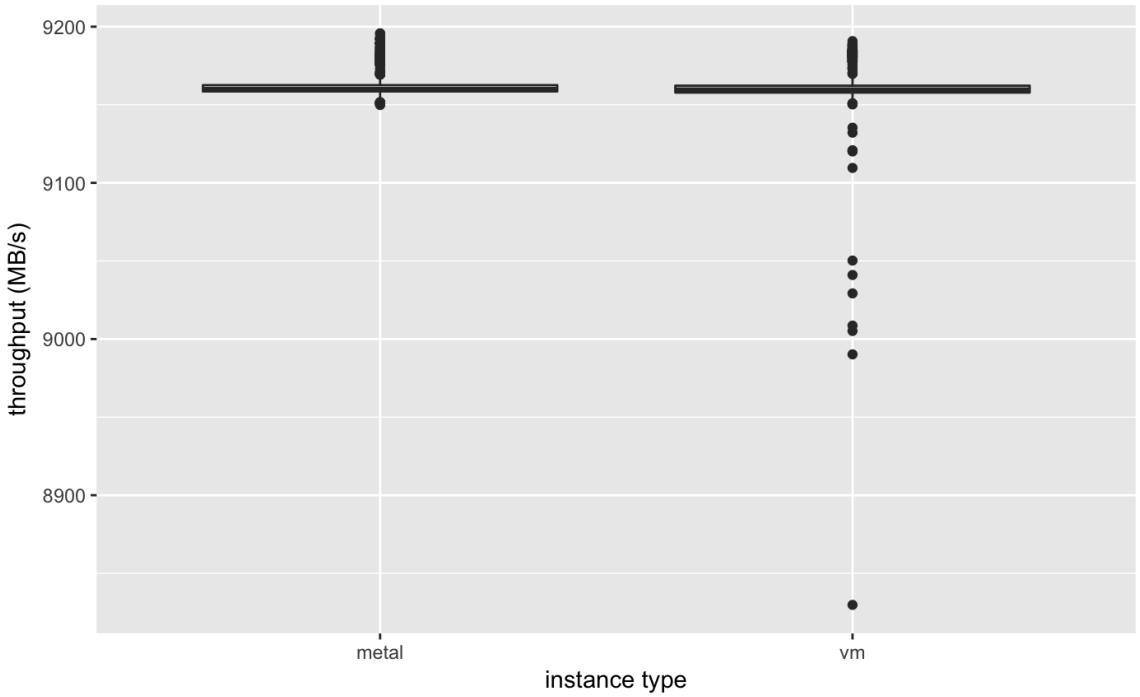


Figure 3.34: Internode throughput measurements between virtualized and bare metal instance types in AWS.

We then conducted the benchmarks with the same parameters on AWS and GCP cloud instances provisioned according to the placement configurations, repeating each trial until achieving a confidence interval as described above. We visualize the runtimes of NPB subbenchmarks in Figure 3.35. We can see that both the mean and variance of samples varies widely depending on the particular benchmark. As expected, the performance of the EP benchmark is similar between each group of nodes, and is appears to be relatively unaffected by network characteristics. We note that the mean performance correlates with the CPU frequency of the nodes, and that the range of measurements is not readily classified by virtualized vs bare metal instance types.

In other benchmarks, we observed that most benchmarks exhibited performance degradation when using multiple availability zones. The notable exceptions include EP, MG, and SP. We also note that on GCP, the BT performance seems to drop when using multiple AZs.

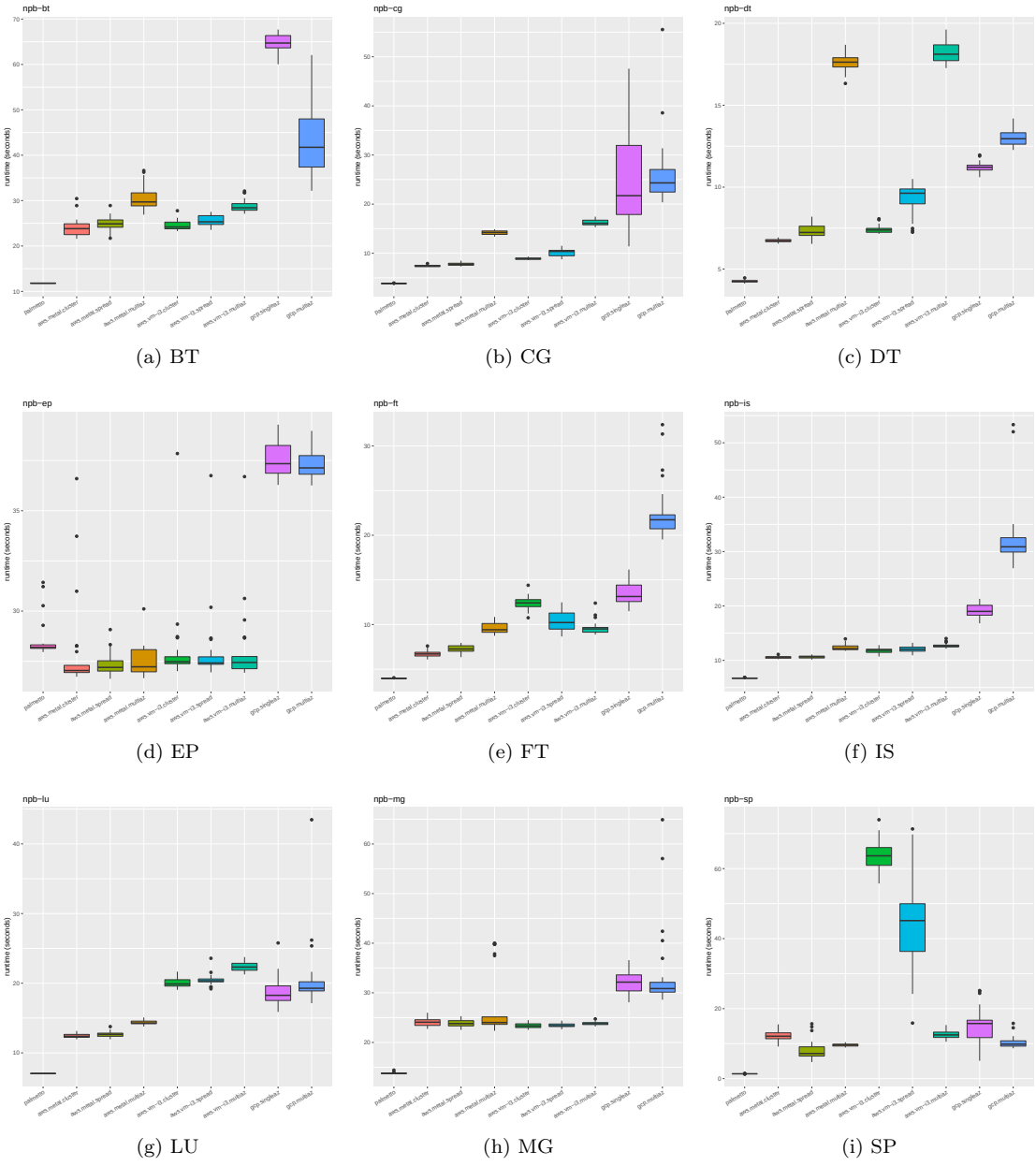
Again using node allocation as provided by the service provider, the LAMMPS benchmark exhibits runtime dispersions that are as expected, shown in Figure 3.36. In AWS with all node types, we observe that the largest measurements using `spread` grouping are up to 73% higher in bare metal instances, and up to 92% higher in virtualized instances with mean runtime increases of

38% and 71%, respectively. Benchmarks run using multiple availability zones exhibit much greater degradation, with extreme runtime increases up to 312%, 178%, and 301% compared to the single availability zone extremes in AWS bare metal, AWS virtualized, and GCP instances respectively. Mean runtime increases for those node classes were 313%, 181%, and 252% respectively compared to the means of single AZ measurements without group requirements.

3.4.5 Summary

In this work we used a custom MPI benchmark tool and the industry-standard benchmark `iperf` to generate network traffic, measuring the latency characteristics and throughput of network paths between cloud computing instances provisioned with different placement strategies. Informed by our results, we observed the relationship between network performance and distributed applications using the NAS Parallel Benchmarks and LAMMPS to generate synthetic network patterns.

We observed that the parameters to provisioning cloud resources can impact the network performance characteristics of internode communication. Furthermore, we saw that for some applications, suboptimal performance in network latency may be reasonably correlated with a degradation in application performance. Our results can be summarized as confirming that latency variation does indeed have a significant impact on the performance of synthetic benchmarks with a high degree of synchronous communication, and is strikingly apparent in computing environments with long-tailed internode latency distributions.



(a) BT (b) CG (c) DT

(d) EP (e) FT (f) IS

(g) LU (h) MG (i) SP

Figure 3.35: Dispersion of NAS Parallel Benchmark runtimes on AWS, GCP, and the Palmetto Cluster using available network proximity options.

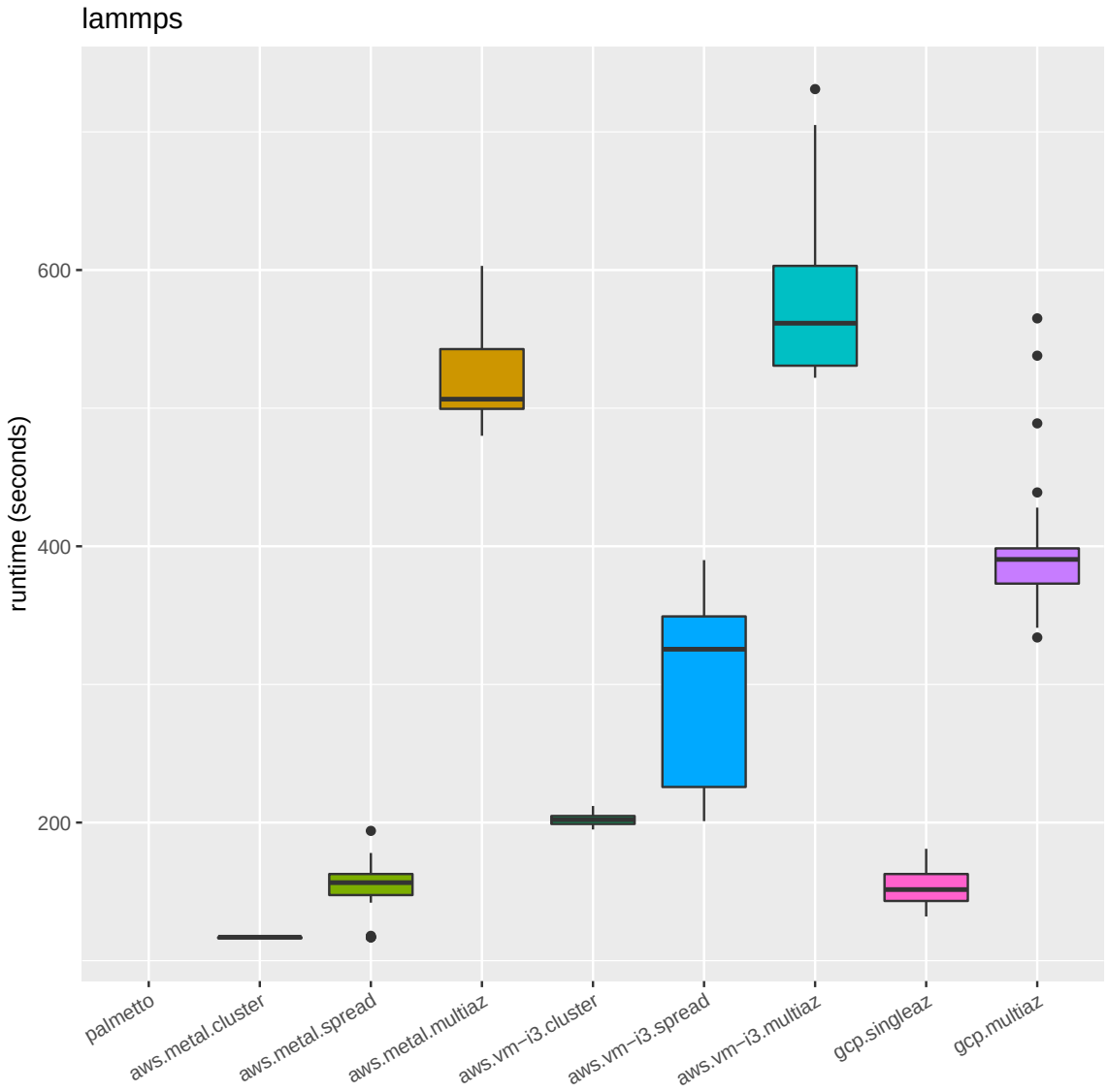


Figure 3.36: Dispersion of LAMMPS LJ benchmark runtimes on AWS, GCP, and the Palmetto Cluster using available network proximity options.

3.5 Conclusions

In this progression of research studies, we used a variety of custom and industry-standard benchmarking tools to generate synthetic network traffic patterns. These benchmarks allowed us to observe relationships between network characteristics and application performance at all levels of the network stack.

In Section 3.1, we used the `netperf` tool to generate traffic and measure the effects of software switches and virtualization on aspects of network performance. We found that lightweight OS-level virtualization tools such as Docker can offer network performance benefits compared to hardware virtualization in Xen.

Section 3.2 describes our work in using custom microbenchmarks using MPI primitives to generate low-level network traffic patterns and observe the effects of container virtualization and software switches on measures of latency variation, distributed synchronization, and internode throughput. We then employed the NAS Parallel Benchmarks to generate distributed traffic patterns common to HPC applications and characterize the higher-level performance impact, finding that high latency variation correlates with performance degradation in a number of benchmarks that depend on tight node synchronization.

In Section 3.3 we applied our findings to a more traditional HPC environment with low-latency InfiniBand networking. We used low-level synthetic benchmarks from the OFED `perftest` to gauge network performance at the hardware level and validate our tool to introduce artificial latency. We then measured the impact of artificial latency with custom MPI benchmarks at the library level and NPB at the application level, and concluded that latency variation has a significant role in performance degradation in tightly synchronized applications.

Finally, in Section 3.4 we extended our work to an emerging non-traditional HPC environment by measuring the effects of network performance on NPB and LAMMPS in AWS and Google Cloud. Using custom MPI benchmark tools and the industry-standard synthetic traffic generator `iperf`, we found large differences in network performance dependent on instance placement strategies and observed correlated degradation in application performance on synchronization-heavy benchmarks using deployments with high latency variation.

In the larger context of this dissertation, our research and results demonstrate that algorithmic synthetic traffic generation tools play an important role in the research and validation of

complex networking systems.

Chapter 4

Synthetic Data in Infrastructure Validation

The concept of Internet of Things (IoT) [30] is rapidly moving from a vision to being pervasive in our everyday lives, and is creating unprecedented opportunities for industry. Raw data is being collected at an increasing rate from a multitude of devices and sensors – sources that include connected homes, smart meters, manufacturing, healthcare, fitness trackers, mobile devices, vehicles, and more. The ingestion and integration of raw data, the extraction of valuable business information from the raw data, and planning for infrastructure capacity and analytic capability for analyzing this data are new challenges in the era of big data [119]. This results in a need for the development of infrastructure support and analytical tools to handle IoT data, which is naturally big and complex. In addition to developing new infrastructure models, there exists a need to use appropriate data to benchmark new and existing infrastructure to determine its performance and capacity.

Industries in the position to collect such data are not typically in a position to conduct this form of research and development, and commonly look to outsource these problems to third parties. But, research on IoT data can be constrained by concerns about the release of privately owned data, which may stem from the inclusion of personal records, proprietary information, or both.

Synthetic data offers a potential solution to these problems, as anonymized data modeled after real IoT data can be used for the design and validation of infrastructure as well as fundamental research and the development of improved systems. However, the complexity of IoT data can pose

a significant challenge. Unlike rigidly structured transactional data that follows simple schemas and can be stored in two-dimensional tables, IoT data is dynamic and self-describing [60] and is often represented in a hierarchical structure using XML or JSON [78, 14] with complex and potentially recursive nested schemas. Systems designed generate a synthetic analogue must capture the complex aggregated structural and statistical characteristics of this original data.

In this chapter, we present original work on two related aspects of synthetic data for infrastructure research and validation. First, in Section 4.1 we describe our research on the development and implementation of a synthetic IoT data generation framework. that is capable of generating terabytes of structurally similar synthetic data from a highly complex and nested original data source. Second, in Section 4.2 we detail methods and results of benchmarking a scalable cloud-based message passing system using generated data.

4.1 Generation of Complex Hierarchical Data for IoT

In this original research, we undertook the development and implementation of a synthetic IoT data generation framework capable of generating terabytes of structurally similar synthetic data from a highly complex and nested original data source. Our primary motivation for developing the framework is to enable the design, development, and testing of large scale data analytic tools and data infrastructures to support IoT research. We evaluate our approach on a real world data source that includes data that have been collected over a period of several months from hundreds of sensors on millions of electronic objects located at geographically different locations. We report on the results of our efforts in two ways. We compare the structural characteristics of the generated data to the original data, and we evaluate the performance of a data access framework on both the original and synthetic data.

4.1.1 Background

As a rapidly developing paradigm, the Internet of Things (IoT) presents the concept that all of the things surrounding us can communicate and exchange information via the Internet, and by doing so, they enable "autonomic responses to challenging scenarios without human intervention" [14]. However, as IoT data are extremely heterogeneous, noisy, and large-scale, this presents a major challenge to the process of cleaning, integrating, and processing the data. The complex,

nested, and potentially recursive schemas of hierarchical data formats used in many IoT sources are distinct from the more predictable schemas used in tabular or relational databases. For the purposes of this chapter, we characterize the simpler case as *structured* data, and refer to the more complex schemas of IoT as *unstructured* data.

Synthetic data generation in more rigidly structured formats, such as tabular data and relational databases, is typically accomplished by developing statistical distributions for a set of samples from data that were directly measured, and then creating new values in the same format as the real data from these distributions. This problem area is relatively simple compared to document collections with more complex schemas.

Unstructured data require a more complex approach for the generation process. The work by Aboulnaga et al. [12] utilizes the Markovian structures of all paths from the root to every possible leaf of the XML tree to generate the synthetic data. This work does not evaluate how structurally similar the synthetic data are compared to the original data. The work by Cohen [58] enables users to generate synthetic XML documents, but requires users to provide a target DTD (document type definition) document and detailed global and local constraints on the output synthetic XML structures. XTaGE [134] uses GUI support to let users define a base XML structure from which synthetic XML documents can be generated. Todic and Uzelac use pre-defined benchmark queries to derive the XML structures that can support the relevant benchmarks [171].

These approaches require *a priori* knowledge of the detailed foundational XML template for the original data. Today's IoT data sources exhibit a very large base XML template with millions of possible XML paths, and prior knowledge of the XML template is often not available. The goal of this research is to design and implement a scalable system for creating synthetic XML that captures the complexity and variety of presented IoT sources. Our approach is not constrained to XML data and can also be applied to other hierarchical data formats.

4.1.2 Characterization of IoT Data

The IoT synthetic data generation proceeds in two major phases. The first phase is to perform structure and value extraction from the original XML, resulting in a data characterization called the *synthesis set*. The second phase uses the synthesis set as input to generate and output synthetic XML data. In this section we present the steps of phase one, structure and value extraction.

To begin, we provide a formal definition of attributes (as in columns of a data table and

not the XML attribute construct) of a data object stored as an XML document. The text contents of an XML document are the collected measurements of the sensors on an object. Each of them represents a unique attribute of that object. The term *value* is used instead of *attribute* in order to distinguish an object's data attributes from its XML attribute construct and *path* to refer to the list of opening tags and attributes that categorize the value. The name of an object attribute is determined by the path from the root tag to the first opening tag before the text content of that attribute. The path includes all opening tags and their accompanying XML attribute constructs but excludes the closing tags. For example, there are four attributes as shown in the first column of Table 4.2 for the object *device 0001* stored in the following XML:

```
<?xml version="1.0"?>
<device id="0001">
  <sensor name="a">
    <type>module 01</type>
    <weight>4.0</weight>
  </sensor>
  <sensor name="b">
    <type>module 02</type>
    <temperature>60</temperature>
</device>
```

Data characterization is based on a few assumptions. First, due to the absence of strict schemas, we assume that XML elements at the same nesting level are unordered since order is not a requirement for well-formed XML [113]. Secondly, we assume that tag attributes are an identifying characteristic of a path in the XML tree and consider paths with variations in attribute values to be distinct. We also assume that values in the XML tree can be classified as either numeric or categorical, as described below.

Data characterization includes: 1) extracting the structure of the data so that similar documents can be generated, and 2) characterizing the values that exist within the dataset. Figure 4.1 outlines the process. First, the values are removed and the tag/attribute trees are stored along with an MD5 hash and their frequency of occurrence into the *structure_data* file. The original XML is examined again, extracting each value along with an MD5 hash of the identifying path and the number of occurrences of that path/value combination. Non-numeric and infrequently occurring

Table 4.1: Examples of the *structure_data* table.

Structure	Hash	Frequency
<device id="0001"> <sensor name="a"> <type></type> <weight></weight> </sensor> </device>	9641ABEF...	1
<device id="0001"> <sensor name="b"> <type></type> <temperature></temperature> </sensor> </device>	C5D7CA98...	3

Table 4.2: Values and the identifying MD5 hash of their path, along with the frequency of which that pair was encountered.

Tag	Path Hash	Freq.	Value
<device id="0001"><sensor name="a"><type>	B3EE890F...	1	mod. 01
<device id="0001"><sensor name="a"><weight>	0E742374...	1	4.0
<device id="0001"><sensor name="b"><type>	57518DAC...	3	mod. 02, mod. 03, mod. 04
<device id="0001"><sensor name="b"><temperature>	15FDF201...	3	60, 65, 69

values are classified into the *categorical_values* file. The remaining values are classified as numeric for distribution fitting. The results are stored in *numeric_distributions*. The three resulting files make up the synthesis set. The condensed nature of this set also facilitates obscuring details of the data through string substitution, as will be discussed in Section 4.1.3. Next, we describe the details of these steps.

4.1.2.1 Structure Extraction

The first step of data characterization is to extract the structure of the data. We view the XML structure as a tree of tags along with their associated attributes but which is separate from the document's values. We use Hadoop MapReduce [184] to ingest the XML-based IoT data. The MapReduce framework processes each XML document individually. The initial map phase removes all of the node values of an XML document while leaving the tags and attributes. Also in the map phase, the MD5 hash of the remaining XML string is calculated. This hash and the XML structure become the key/value pair output of the map phase. The reduce phase aggregates these pairs and emits a list of unique triples whose first value is the MD5 hash, the second value is the frequency of this hash in the entire document set, and the third value is the stripped XML structure. Table 4.1 is an example of the output from this process, which we refer to as *structure_data* within the synthesis set. The entire XML structure is stored, retaining the hierarchical characteristics of

the XML documents.

4.1.2.2 Distribution Modeling

The next step in data characterization is the calculation of the statistical distributions of the different values of the data. For a tabular data set, the set of values to be considered would be the data columns, but there is no predetermined set of values for hierarchical and schema-less IoT data. The distributions of the data values must be computed across tens of millions of IoT data entries, each of which has different combinations of sensor measurements.

The values of an XML object are encoded within the path of the object's XML structure. Hadoop MapReduce is used to extract the paths into an intermediate form for classification. A recursive algorithm is used to descend the XML tree, appending each node's name and attribute list to a string. If a text or numeric value is encountered, an MD5 hash of the path string is generated and emitted with the value. In the reduce phase of the job the hash-value combinations are counted and output as the intermediate *value_data* file. For example, the XML paths and their respective hashes, reduced counts, and values of the example XML structures in Table 4.1 are shown in Table 4.2.

The *value_data* is the input into the next phase, where the contents of the values are classified as numeric or categorical. Any set of contents for which there are 30 or fewer unique values is classified as categorical. In this initial design and implementation we make the assumption that the values are independent of each other for the purpose of calculating the statistical distributions.

For a categorical path A_i, we are interested in the possible values that the path might have on different objects. Let the domain of A_i be $1, 2, ..., d_k$. Then the categorical path domain is $D = \Pi_{i=1}^{k}$, which forms a contingency table. The probability of a particular categorical path can be approximated as $\widehat{\pi}_d = \frac{x_d}{n}$ where x_d is the frequency from the contingency table and $n = \sum_{d=1}^{D} x_d$. The path hash, frequency, and value of categorical data remain in the same format as the *value_data* file, and are output as the *categorical_values* component of the synthesis set.

Given a numeric path Z_j, we use a fitting method to determine the best distribution to represent the data, and then calculate n, max, and min. For each distribution, the relevant parameters of the data are also written out. For example, the lambda distribution would include λ - the sample mean, while the normal distribution would have $\overline{x}_n, s_n$ where $\overline{x}_n$ is the mean and s_n is the standard deviation. We consider a range of distributions that includes beta, Cauchy, chi-squared, exponential, F, gamma, geometric, log-normal, negative binomial, normal, Poisson, t, and Weibull distributions.

Table 4.3: Example table entries of distribution information for numeric paths

Distribution	n	Parameter 1	Parameter 2	IsInteger	Maximum	Minimum
Poisson	234	728.1 (lambda)	-	True	35827	1
Poisson	7752	1489.6 (lambda)	-	True	6948	0
normal	437	9.9 (mean)	14.9 (stddev)	False	77.3	0.0
geometric	92	0.0004 (probability)	-	True	2506	2358
Cauchy	629	80.5 (location)	20.4 (scale)	True	122	-48

These distributions were chosen as they are both supported in the R MASS package [179] and the Java Statistical Distribution Library, JDistlib . The R MASS package is used to determine the best fit and parameters of the data. R's MASS package includes a function `fitdistr` that performs a maximum-likelihood fitting for univariate distributions. JDistlib [1] is used to randomly generate numbers for the synthetic data from the maximum-likelihood fittings of the distributions.

The results of the `fitdistr` function are compared and the distribution with the best fit is selected. The `fitdistr` function can also estimate the parameters of a given dataset and distribution. These values, along with their associated path hashes, are recorded in the *numeric_values* component of the synthesis set to be used by the synthetic data generator. An example of data in this file is shown in Table 4.3.

4.1.3 Synthesis

The second major phase of the IoT synthetic data generator framework is the use of the synthesis set to create synthetic XML data. The data generator has a few design requirements. First, it must scale to handle millions of structural representations, millions of unique paths, and many millions of possible categorical values. Secondly, it must support an effective method of anonymizing categorical values to provide an additional level of privacy for industrial and consumer information. In our approach, we use a simple string substitution mechanism to map every unique string in the *structure_data* and *categorical_values* components of the synthesis set to a randomly generated string of the same length as the original. A new obfuscated synthesis set is then generated using the string map which retains the qualities of correlation between tag names, attribute names, attribute values, and categorical values. The string map can be retained by the owner of the data for mapping synthetic results back to original strings. Note that the string substitution step is optional and could be replaced with a more sophisticated technique where one is needed. The synthetic generator performs identically with an original or anonymized synthesis set.

The workflow diagram of the synthetic IoT data generation process is illustrated in Figure 4.2. The synthesis set is sufficient to create synthetic documents with characteristics similar to the original data. In this diagram, we obfuscate the textual fields with random strings of the same length as the originals, while maintaining the relationships between those strings throughout all generated files. The obfuscated synthesis set is sufficient to export to researchers without exposing sensitive data; conclusions drawn from that set can only be mapped to the original data with the obfuscated map set.

One design challenge is to manage millions of possible categorical values associated with a single path. In our test data some paths have over 5 million possible text values, each with an associated frequency of occurrence. In order to randomly select from these sets, we insert the possible values into unbalanced Huffman trees, implementing the linear time Huffman construction algorithm described in [109]. This allows for fast random selection, but it requires that the tree be kept in memory to avoid being rebuilt for each matching path.

4.1.3.1 Single Machine Synthesizer

To prepare the descriptive data for generation of synthetic XML files, we first import it into a database for fast lookups. A synthesis set consisting of three tables is used to describe the original dataset: *structure_data*, *categorical_values*, and *numeric_distributions*. Each entry of the *structure_data* table contains a unique structure hash, frequency of occurrence, and the bare XML structure that generates the structure hash. Each entry in the *categorical_values* table contains a hash of a non-unique path, frequency of the hash, and the categorical data in the path. Each entry in the *numeric_distributions* table contains a hash of a unique path, frequency of the hash, and distribution parameters of the path's values.

Our Java-based reference synthesizer preloads the numeric distributions and categorical values along with their frequencies from their respective files and into Huffman trees. Structures are then read from the structure data file and the XML strings are converted into W3C Document Object Model (DOM) [127] trees for simple parsing. Algorithm 2 describes the recursive method used to construct the path string for each node as shown in Table 4.2. A hash of the path string is used to select the relevant value tree, and then a random value is selected from the tree to fill the document node. During the intermediate calculation it is possible for the path hash to appear in both the *numeric_distributions* and *categorical_values* datasets. The data type chosen is based on

Algorithm 2 Synthesize Document Values

```
1: procedure SYNTHDOCUMENT(doc)
2:     SYNTHNODE(doc.head, "<")
3: end procedure
4: procedure SYNTHNODE(node, path)
5:     path ← path + node.name
6:     for all attr in SORT(node.attributes) do
7:         path ← path + " " + attr.name + "=" + attr.value
8:     end for
9:     path ← path + ">"
10:    if LEN(node.children) == 0 then                          ▷ base case
11:        hash ← MD5HASH(path)
12:        tree ← GETVALUETREE(hash)
13:        node.value ← GETRANDOMVALUE(tree)
14:        return
15:    end if
16:    for all child in node.children do
17:        path_copy ← COPY(path)
18:        SYNTHNODE(child, path_copy)                          ▷ recurse
19:    end for
20: end procedure
```

the frequency of occurrence for those values. Finally, the synthesized document is output as a file for later analysis.

There are limits to the scalability of the Java-based approach. Numeric distributions require very little memory to cache, but the Huffman tree holding the possible values and frequencies for categorical fields can contain millions of nodes. Consequently, the multi-threaded generator is limited by the memory available to the process, and does not scale well to a very large dataset. In our initial tests, generating data using a 24 GB set of categorical values could not complete on a machine with less than 100 GB of memory. This is partly due to the overhead of Java objects, but it was evident that the approach did not scale for larger datasets. However, it was relatively fast, generating approximately 30,000 synthetic XML documents per minute using 12 cores and document sizes averaging 450 KB.

4.1.3.2 Distributed Synthesizer

Hadoop MapReduce was used to implement a distributed synthetic generator. One design goal in this case was to minimize the movement of the large amounts of categorical data across the network. Since any single structure could require loading many of the large value/frequency trees in the dataset, a design parameter of our approach was to group data that needed to be computed

together. This translated to a design that performs two joins: one to group the structure hashes with a list of related path-value combinations, and another to join the path-value combinations to the structure. This is accomplished with a technique known as a reduce-side join (or repartition join), which we implemented in a manner similar to the Improved Repartition Join described in [41].

In the first phase of the two-part process, shown in Figure 4.2, the *structure_data*, *numeric_distributions*, and *categorical_values* are ingested by instances of the same map class. For structures, hashes for each path are computed as shown in Table 4.2, and emitted with the frequency of occurrence and the structure hash in which they were discovered. Numeric and categorical values are passed through to the reduce phase. The interphase sort ensures that map outputs are grouped by their path hashes. Each reduce instance can then construct a single Huffman tree for the categorical values related to each of its assigned path hashes, and randomly select enough values to satisfy every structure. The generated values are keyed by the requesting structure hash, and emitted with the related path hash.

The second phase of the process begins with an identity mapper that ingests and emits the results of the first phase, keyed by structure hash, along with the structure data. The interphase sort groups all values by the structure hash. Finally, each reduce instance has a set of structures paired with enough values to fill them, accomplished by recursively generating path hashes to match to the input values as described above. As security is a primary concern, we developed a method for obfuscating the textual fields of the data while preserving the relationships between the text values. A series of map-reduce jobs were used to derive a set of all strings in the structure data and categorical attributes, as shown in Figure 4.2. These strings were then mapped to randomly generated strings of the same length, giving us a dictionary of obfuscated string pairs. The encoding map was then used to generate obfuscated structure and categorical data, as well as a mapping between the MD5 hashes of the original and obfuscated paths for each value in the data sets. This obfuscated set can be used in the same manner as the original synthesis set to generate XML documents. The resulting obfuscated path and string maps can be retained by the owners of the original data, so that conclusions about the synthetic documents can be mapped to the real data.

4.1.4 Evaluation

Our primary goal in generating synthetic data is to create an experimental framework that can be used to evaluate the performance tradeoffs of various data infrastructure tools for very

Table 4.4: Descriptive statistics of the XML tags, attributes, and paths of the test data

	Individual XML Tags	Individual XML Attributes	Individual XML Paths	Individual Objects
Unique Count	110	29	8,716,624	3,193,783
Max Frequency	143,935,646	150,997,555	115,328,502	2,014
Min Frequency	11	1,383	1	1
Avg Frequency	1,966,444	75,34,407	2,582.4	1.99
Median	60,640	353,358	7.0	1.0

complex data. Our framework creates synthetic data that have values and a structure that match the statistical characteristics of the original data. To evaluate our framework we start with an original data set that was collected from various sensors on 3,193,783 electronic objects over a period of several months. The total size of the data set is more than 3 terabytes. The data entries arrive to the data warehouse in XML format containing 6,347,462 individual XML documents. The dataset is pre-processed using a simple string substitution for anonymization of identifiers.

The framework is used successfully to extract descriptive statistics of the original data. Table 4.4 shows the descriptive statistics of the unique XML tags, attributes, and paths of these structures. While there are a fixed number of XML tags and XML attributes, there is no pre-determined XML schema or DTD document. Without a fixed schema, out of 6,347,462 XML structures, 5,758,590 are unique. The combination of different XML tags and attributes generate a total of 8,716,624 unique XML paths within the original data. Because XML tags, attributes, and paths can appear multiple times in a single XML structure, the tag seen most often appears more than 140 million times across the dataset and the most frequently seen attribute occurs more than 150 million times.

After extracting the unique XML structures and paths and identifying categorical and numeric values, the statistical distribution of each unique numeric value is also calculated. Just three distributions provide the best fit for 93% of the numeric paths: Poisson (58.1%), normal (23.5%), and geometric (11.7%). The geometric distribution used is $Pr(Y = k) = (1 - p)^k p$, which models the number of failures before the first success. This level of statistical accuracy of the values meets our primary goals of synthetic data generation. The parameters of these distributions are recorded for use in the data generation phase of the framework. Development of techniques that provide high statistical accuracy of synthetic data values to original data is possible within the framework.

The synthetic data generation system needed for our problem comprises three features: the sensor log data consists of approximately 20 terabytes, the generated data needs to be of any specified size, and the data, stored in XML files, does not have a fixed schema.

Table 4.5: Shuffle and output characteristics of the 2-phase MapReduce synthetic generator.

Input		Phase 1		Phase 2	
Months	Documents	Shuffle (gzip)	Output	Shuffle (gzip)	Output
1	1.61 million	371 GB (73 GB)	198 GB	487 GB (69 GB)	365 GB
2	3.09 million	715 GB (141 GB)	394 GB	975 GB (143 GB)	733 GB
4	6.35 million	1,470 GB (290 GB)	810 GB	2,004 GB (294 GB)	1,506 GB
8	13.72 million	3,177 GB (627 GB)	1,750 GB	4,331 GB (635 GB)	3,255 GB

We further evaluate the framework by comparing the performance of applications that use the IoT data when presented with the original and generated synthetic data of the same size. We use this as an additional measure of the structural similarity between the datasets. We generate data of equal size to the original IoT data collected in 1, 2, and 4 months, with sizes of 365, 733, and 1,506 GB, respectively. With each of these datasets we run a program to descend into each XML document and record the instances of certain XML tags with a specific attribute. We verify that the frequency of occurrence is as expected in the synthetic data, and use the runtime as a measure of the document complexity. Each test was run to a 95% confidence interval, as shown in Figure 4.3. In each of the test cases, the performance using synthetic data is similar to using the original data, supporting our assertion that the synthetic data is structurally similar to the original.

The final aspect of the evaluation of the framework is measurement of the time to generate synthetic data of various sizes from the descriptive statistics. In the multi-threaded single CPU implementation, we were able to generate synthetic IoT data in XML format at a rate of 230 MB/second on a machine with 12 cores and 128 GB of RAM. However, the speed of this approach is offset by its lack of scalability, since the memory requirement is approximately four times that of the size of *numeric_data* and *categorical_values* files. To evaluate the distributed framework we create isolated testing environments for the generation process using the Clemson high performance computing cluster. In this environment we dynamically allocate four different Hadoop clusters with 10, 20, 30, and 40 compute nodes. Each node is configured with two 8-core Intel Xeon E5-2655 CPUs, 64 GB RAM, 1 Gb/s network link, and a single 900 GB HDD. The memory requirements of the reduce phase make it necessary to allocate 4 GB of RAM to each reducer JVM. To fit within memory requirements, we configure Hadoop to spawn 32 map tasks and 4 reduce tasks on each node. Synthetic data is generated in 1, 2, 4, and 8 month chunks, with sizes as described in Table 4.5.

In both phases of the data generation, we observe roughly linear scaling of the completion time as the number of compute nodes and size of the data vary, as shown in Figures 4.4 and 4.5. We note that the speedup from doubling the number of nodes is approximately 1.5x in Phase 1 and

1.3x in Phase 2. We hypothesize that this is due to the 1Gb/s network link between the nodes, which throttles the shuffle phase of map-reduce. To support this hypothesis, we conduct limited experiments with compressed vs. uncompressed shuffle traffic, and found the speedup to be worse with uncompressed data.

We report the results of the data generation process using a truncated set of 278,793 structures (47 GB), along with the full 200,000 numeric distributions (15 MB) and 473 million categorical values (24 GB). In the first stage of the synthetic data generation process, distributions and values are passed through to the reducer, but the structures are parsed for hashes of paths to be filled, resulting in 800 million path hashes. The reducer matches the structure path hashes to generated random values, writing 643 million hash/value combinations (48 GB) to disk in preparation for stage two. These values, along with the original 278,793 structures, are read into the stage two identity mapper, which partitions them by structure hash and passes them straight to the reducer. The reducer inserts the values into their matching structures and writes the filled XML documents to disk, resulting in 66 GB of records. Table 4.6 shows the run time measurement of the two stages of the generation process on different Hadoop cluster sizes.

Table 4.6: Runtime measurement (in seconds) of the generation process

Stage	10 nodes (40M/20R)	20 nodes (80M/40R)	30 nodes (120M/60R)
1	1045	457	497
2	576	305	374

Table 4.7: Runtime measurement in seconds of data generation at different sizes for a 20-node cluster

Size (GB)	Stage 1 (s)	Stage 2 (s)	Total (s)
66	457	305	762
133	594	571	1166
199	832	455	1287

We observed over two times speedup as the number of nodes increased from 10 to 20. Adding additional nodes to the cluster did not scale the performance further. For Stage 2, the performance with 30 nodes was lower than the performance with 20 nodes. Further examination of the output logs indicates that the performance bottleneck lies with the garbage collection processes of the map and reduce tasks. With the optimal 20-node configuration, we also varied the sizes of the synthetic data to 66GB, 133GB, and 199GB. As shown in Table 4.7, the rate of run time increase was less than the rate of data size increases. This was due to the initial constant load time of the synthesis

set which was amortized as the sizes of the synthetic data to be generated increased.

4.1.5 Summary

In this work, we have described methods of synthesizing large scale IoT data with noisy and complex structural characteristics in a scalable extraction and synthesis framework. The framework enables access by researchers to IoT data for the development and testing of tools and algorithms, and enables research by organizations that need to ensure the privacy of their sensitive data.

The modular design of the framework allows for streamlined extensions of the generator to support inclusion of different statistical distributions as necessary. In addition, the condensed form of structural and categorical data offers an option for ensuring privacy of confidential data within the synthetic data set. The preliminary efforts with string substitution have been successful. The use of Apache Hadoop and MapReduce for parallel processing makes the framework highly scalable, and enables a faster turnaround time in research and development.

Future work on the framework is possible. While the majority of the XML structures are unique in our test case, we need to support the case where there exists a common subtree in all of the XML structures. In this situation, pruning of the common subtree will help to reduce the size of the structural table and improve performance. The framework will be further modularized to allow users to specify their own set of distribution functions, including complex functions such as inter-dependent and correlated marginal distributions.

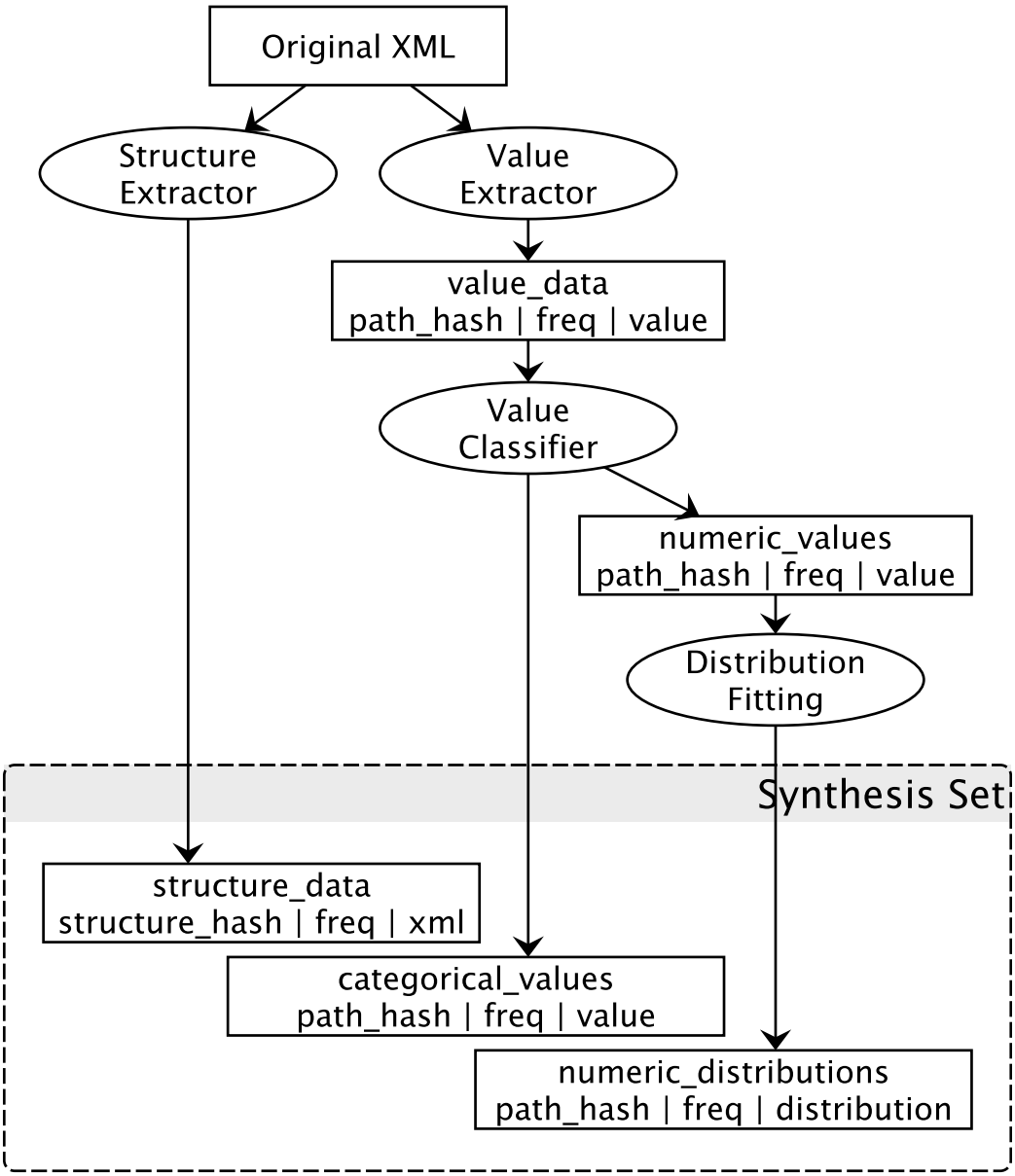


Figure 4.1: Structure/Value Extraction – Cascading series of steps to extract the data patterns from complex XML documents.

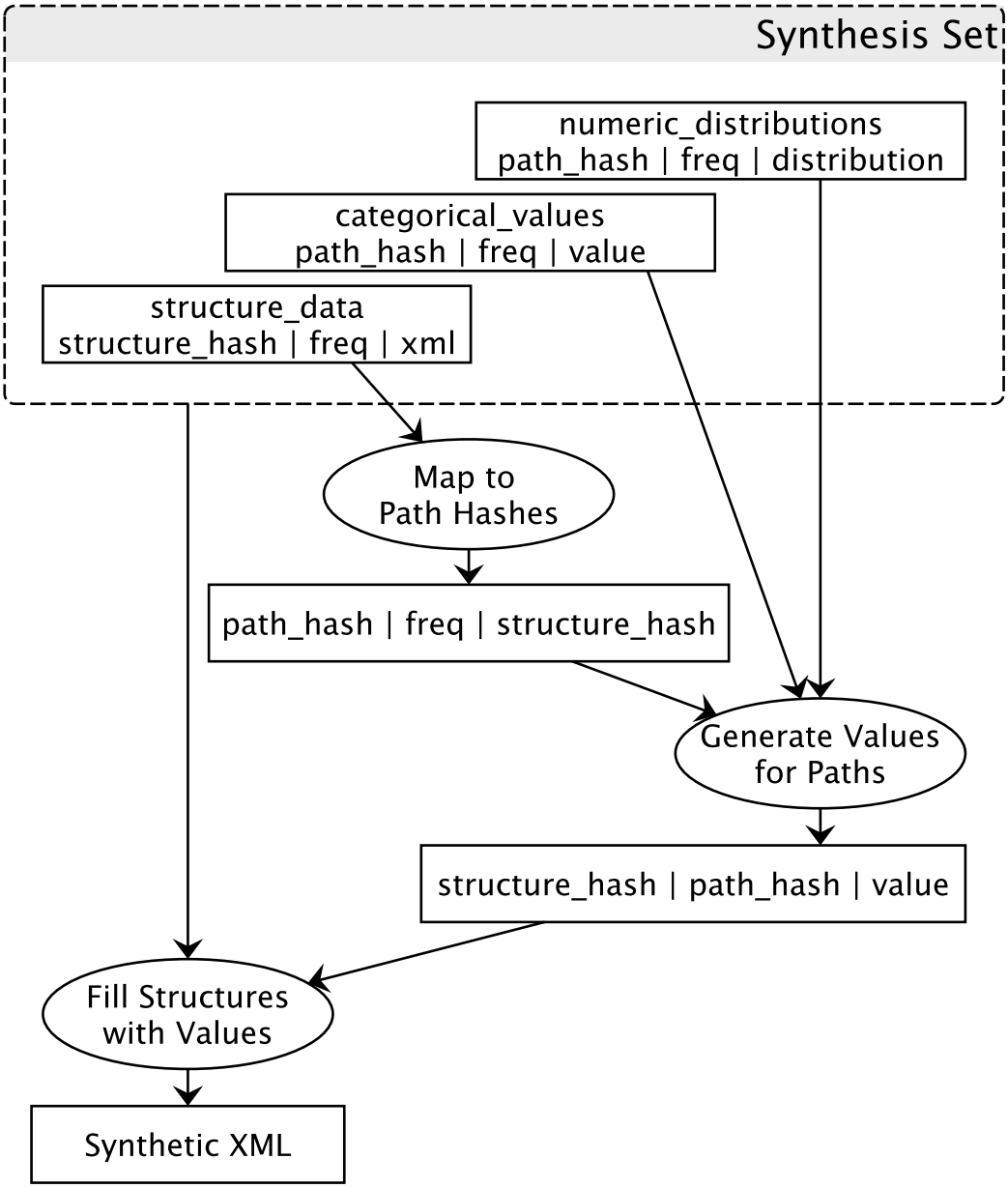


Figure 4.2: XML Synthesis – The workflow of generating synthetic IoT data. Reduce-side joins are used to combine structure path hashes with possible values, the results of which are recombined with the structures to fill empty tags.

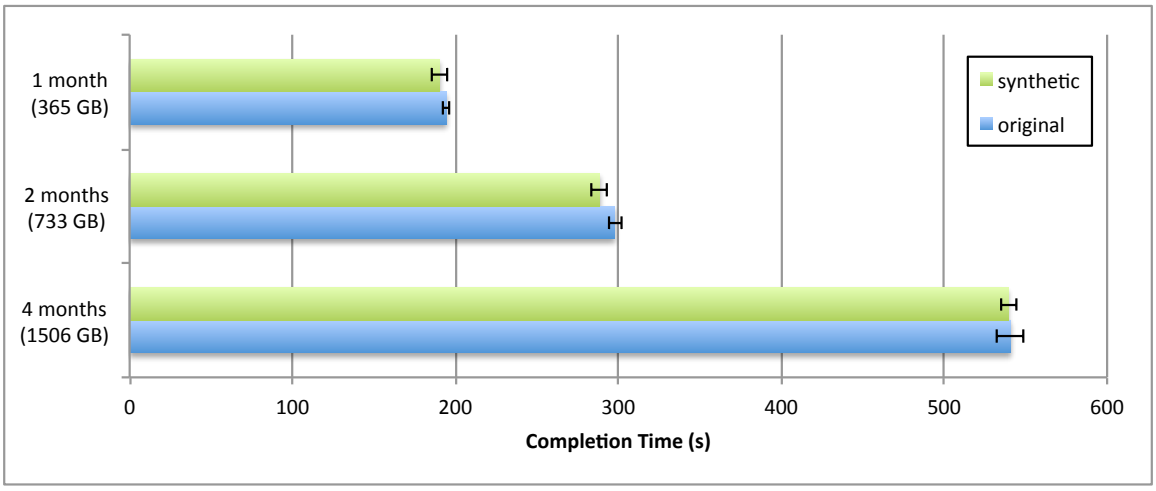


Figure 4.3: Structural validation by performance comparison of tag-attribute searching, with 95% confidence interval.

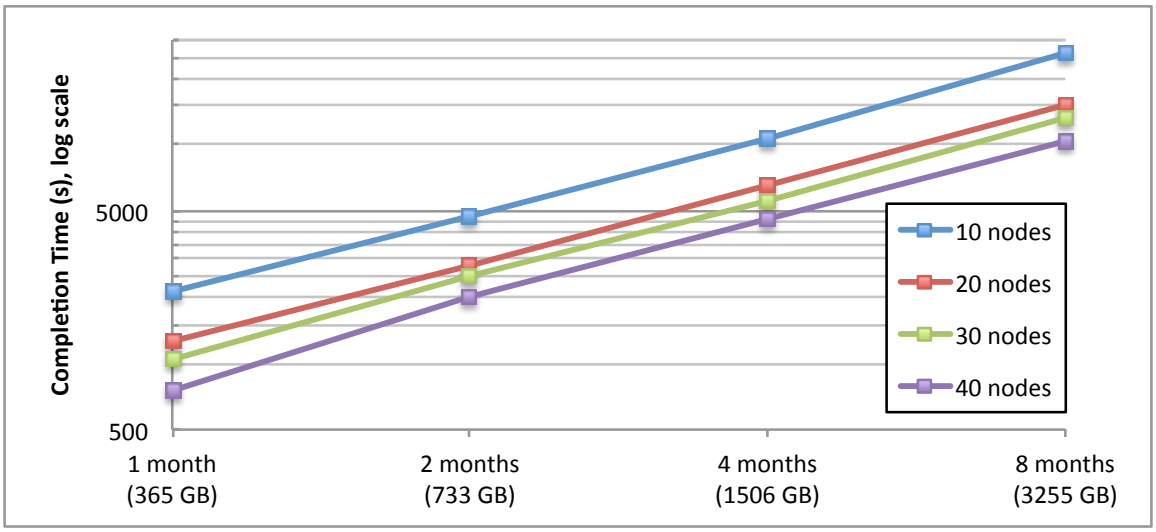


Figure 4.4: Synthetic generation Phase 1.

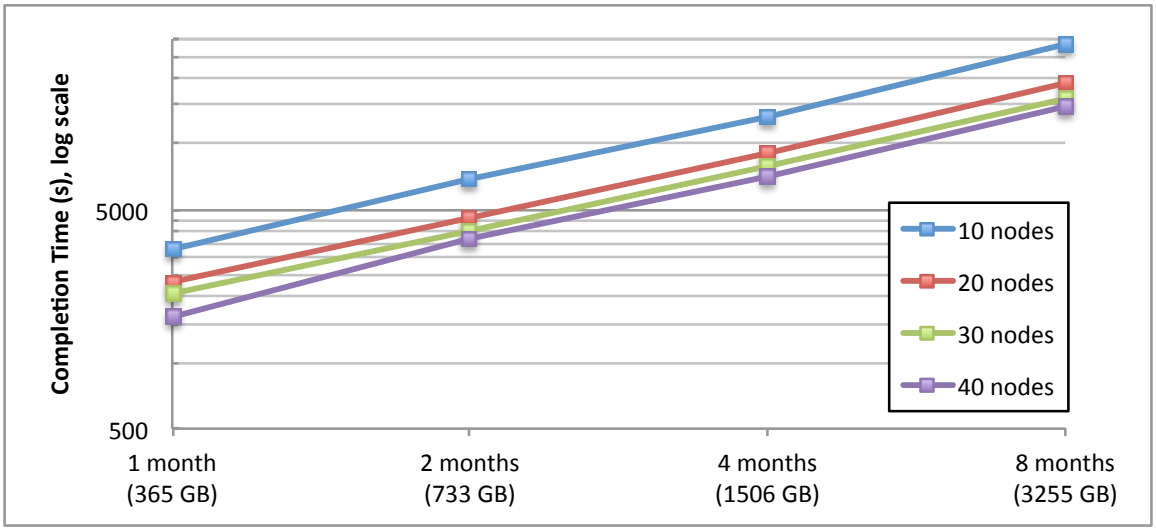


Figure 4.5: Synthetic generation Phase 2.

4.2 Benchmarking of Cloud-based Messaging Systems

The use of sensors within manufacturing has rapidly increased in the last ten years due to initiatives such as Industry 4.0 and IIoT (Industrial Internet of Things). These initiatives involve the collection and use of data throughout all aspects of the manufacturing process. Measurements from these sources are used in many areas, such as predictive maintenance, optimized logistics, and flexible production. The growing amount of data presents challenging processing questions: how much data can be moved to and processed by cloud-based systems under a soft real-time deadline, where edge versus cloud processing [163], and what costs are involved [126].

In this section we present research on using synthetically generated communication patterns modeled on real-world IoT devices to benchmark the capacity of Azure IoT Hub [7], a scalable message passing infrastructure. The generated workloads emulate conditions observed in a large automotive manufacturing plant, and are comprised of synthetic sensor data modeling data sources such as temperature and humidity, equipment vibration, acceleration, and power consumption. The synthetic data captures patterns similar to proprietary data of the plant and permits large-scale experiments. Our experiments measure latency and how it is affected across a range of parameters. We simulate scaling up to thousands of devices in a manufacturing plant by leveraging the Clemson supercomputer to generate the messages sent to IoT Hub. The goal of this research is to provide advice regarding the number of cloud-connected sensors that can be installed within a manufacturing plant.

4.2.1 Background

Azure IoT Hub uses HTTPS, AMPQ, or Message Queuing Telemetry Transport (MQTT) to facilitate communication between a device and the cloud. One common message protocol is Message Queueing Telemetry Transport (MQTT) [33] and the extended MQTT-SN for use with low power and low bandwidth devices such as wireless sensors [90]. MQTT is an extremely lightweight publish-subscribe protocol designed for Internet of Things connectivity. Azure IoT Hub implements MQTT v3.1.1.

Each IoT device, emulated or physical, has a unique connection string between it and IoT Hub. The connection string contains the hostname for the IoT Hub, the device ID, and the shared access key. These connection strings map the cloud-side systems processing the messages to one and

only one actively connected device, enabling securely sending messages.

4.2.1.1 IoT Hub Service Levels

Iot Hub is split into three *editions*, which are further separated into Basic and Standard *tiers*, with extra features such as Device Twins, Cloud to Device messages, and Azure IoT Edge. Operation throttling limits are identical for the two tiers, so we used the Basic tier as the Standard tier features are beyond the scope of this project.

Table 4.8: Iot Hub Editions and Throttling Limits

Edition	Max. D2C[a] Send Operations	Max. Messages per Day
B1/S1	100 per sec[b]	400,000 per unit
B2/S2	120 per sec per unit	6,000,000 per unit
B3/S3	6000 per sec per unit	300,000,000 per unit

[a] Device-to-Cloud

[b] This is 12 per sec per unit if it results in a higher value.

Since IoT Hub is a shared resource, the service runs on the same hardware as other IoT Hubs [35]. Different editions of IoT Hub come with different quotas, which are throttling limits for daily message allowance, maximum aggregate messages per second, and maximum new connections per second. IoT Hub lists quotas in terms of per IoT Hub unit. These limits are summarized in Table 4.8 [5] [3]. We chose the B3 edition as we needed to send thousands of messages every second. Quota limits scale linearly with number of units.

For example, a single unit of Edition 3 IoT Hub allows for 300M messages a day and up to 6000 messages per second. Adding another unit will double the number of messages per day to 600M and the number of messages per second to 12000. Because of this pricing and quota model, we choose to evaluate the Azure IoT Hub in this project.

4.2.1.2 Event Hub and Partitions

Azure IoT Hub is built on Event Hub, an event processing service designed for horizontal scalability. It employs a partitioned consumer model to ingest data [8]. IoT Hub has additional features that are specifically designed with IoT in mind, such as providing unique identities to each device, servicing millions of simultaneous connections, and sending cloud-to-device messages [4] [106].

When creating a new IoT Hub using the online Azure portal, one can request between 4 and 32 partitions, with a default of 4. However, it is possible to request up to 128 partitions if one creates their IoT Hub using the Azure Command Line Interface. The number of partitions is fixed upon creation, so it is recommended to plan for the long term when provisioning IoT Hub.

Customers can determine how many partitions they need by deciding how many concurrent readers of the event stream their downstream applications require. It is recommended to have one reader per partition, though a maximum of five per partition is allowed. New messages are tagged to the end of a partition and stored for a specific retention time. Events cannot be explicitly deleted but must expire. Data is added to partitions in the underlying Event Hub in 32 MB segments. A segment is not deleted until it is full and the retention period for the newest message stored in that segment has expired. This means that messages can be read after their retention period. Old messages are retained until their segment is full and all messages in their segment have exceeded their retention period. As such, partitions grow at their own rates [170].

Each unique device has a unique device ID and is deterministically hashed and assigned to a partition. Messages sent from a single device ID go to the same assigned partition and retain their order. Best service is provided by having approximately 100 times as many devices as partitions [170].

4.2.1.3 Edge processing

Manufacturing environments can have millions of sensors and devices transmitting data. Azure provides a service, IoT Edge, that can control edge devices for processing data. It can act as either a transparent or opaque gateway. In the case of a transparent gateway, devices retain their identity through the pipeline and can therefore be routed to different partitions. However, in the case of an opaque gateway, all data coming through IoT Edge will act as though it is coming from a single device. This means all data passing through an opaque gateway will be routed to the same partition within IoT Hub [170]. Therefore, the results of our study can be applied to environments with comparable numbers of sensors as well as environments with a much large number of devices that route their messages through a comparable number of opaque gateways.

4.2.2 Related Work

Subramanian et al. [165] studied generating synthetic seismic monitoring station readings using task parallelization in Azure. They conclude that cloud computing is an "ideal platform for

the rapid generation delivery for synthetic seismograms." However, their pipeline focuses on data creation in the cloud, not on the transportation of data to the cloud, as is the case with IoT data.

Wireless sensor network simulators such as OMNeT++ [178], J-Sim [160], and ns-3 [149] offer the means to simulate content generation and data sinks to study sensor traffic interactions at scale. However, simulators are not well suited for studying the effects on third party cloud-based sinks due to layers of unknown network complexity.

Various performance studies have found significant differences in the application-layer protocols commonly used to move data from wireless sensors to message broker services. Thangavel et al. [168] found that Constrained Application Protocol (CoAP) had lower bandwidth overhead than MQTT in scenarios with low packet loss, at a cost of higher latency. Luzuriaga et al. [112] compared MQTT and AMQP in the context of an unstable network environment, finding the protocols to exhibit similar recovery time and message arrival time jitter with intermittent drops in wireless connectivity. However, this study only used a uniform distribution of message dispatch times, which we expand upon in this paper.

Other studies such as Nguyen et al. [126] have found tradeoffs between cloud-based message broker services that make choosing a provider highly use case dependent.

4.2.3 Software and Connectivity Architecture

In our study we emulate data produced by a manufacturing environment. Here, we explain 1) Palmetto, 2) software architecture, 3) the synthetic data generator, 4) the experimental loop, and the 5) connection process.

4.2.3.1 Palmetto Cluster

We utilized ten nodes of Palmetto, the Clemson supercomputer, to execute the clients in our experiments [9]. Thousands of sensors can be emulated by executing our clients across multiple cores and nodes. We utilize the Clemson network to emulate the network of a real manufacturing plant. Clemson maintains a complex enterprise computing network to support a wide variety of applications. Complex multi-level network policies and protocols are implemented to satisfy different SLAs [53].

Palmetto has direct access to the Internet, and being a high performance computing cluster with low latency interconnects to local storage, there is typically very little network contention for outbound Internet traffic. To verify a sufficiently high bandwidth network path for our experiments,

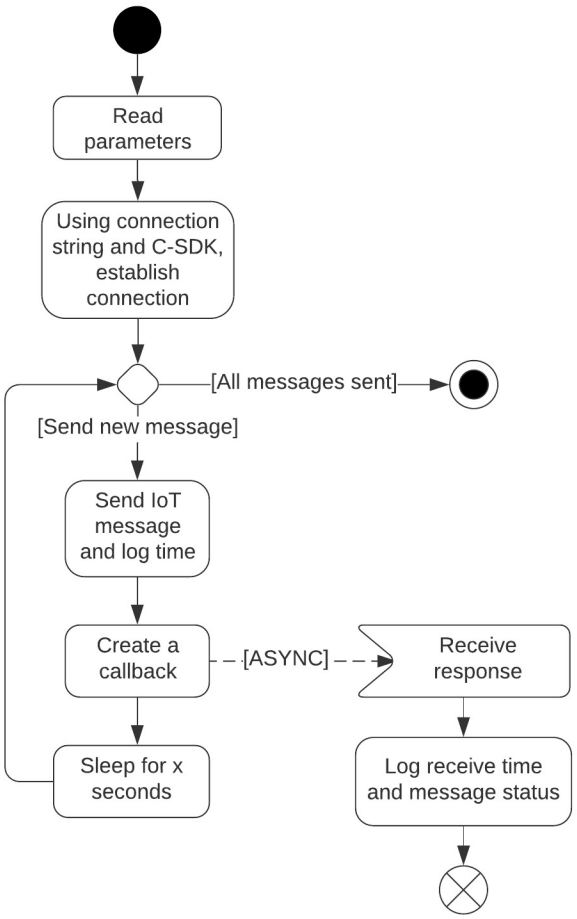


Figure 4.6: Synthetic device flow diagram.

we traced packets sent from Palmetto to an IoT Hub provisioned in the Azure East US 2 region, which is geographically located about 500km away in Virginia. Individual compute nodes are connected by 10Gb links and share a 100Gb path to the Internet gateway, which is narrowed to 10Gb for commodity traffic before entering Microsoft's internal network infrastructure.

4.2.3.2 Software Architecture

The flow of the data is shown in Figure 4.6. We built a synthetic data generator in C++ with a small memory footprint. The generator represents a single physical sensor in a manufacturing plant that reads data periodically and sends it to IoT Hub. Simulating 10 sensors requires 10 instances of the generator running simultaneously. The data generator utilizes many parameters to specify the behavior of the sensor so as to cover a wide scope of sensors in a real manufacturing environment.

Table 4.9: Synthetic Data Generator Parameters

Parameter	Range
Statistical Distribution	Constant OR Pareto
Inter-message Gap Time	10 ms - 1000 ms
Message Size	512 B - 32,768 B
Experimental Run Time	5 min - 90 min
Network Protocol	MQTT

Each instance of the data generator accepts a time parameter to specify how long to send messages. Each instance writes its own individual log file, logging for each message an ID, send time, callback receive time, and callback status. The user can choose any of the protocols available from the SDK. For this work we use the default MQTT, as seen in Figure 4.7.

4.2.3.3 Data Generator Configuration

Parameters that differentiate data generator behavior are shown in Table 4.9. They include the message generation frequency, distribution and parameters (constant or Pareto), the message protocol, payload size, and approximate experiment duration.

4.2.3.4 Experiment Loop

With given parameters, our client generates data modeled after the specified template and sends it immediately to IoT Hub. The send time and message ID are stored. The SDK automatically creates threads for every message callback, which asynchronously listens for a response. The amount of individual data points to generate is based on the the time specified. Between each data point the main thread sleeps. Responses from Azure are received by the async thread at any time. The response from Azure reports how IoT Hub processed that message. That status and the time our client receives the response are both logged.

With these logged values, we calculate the round trip latency with sub-millisecond precision, and whether the message was successfully processed. The status of a message can be OK (Azure successfully processed the message), destroyed (Azure could not or would not process the message and rejected it), or other. We use Python's data libraries to calculate and measure latencies in aggregate. We measure round trip latency as we do not include the time costs of any other Azure

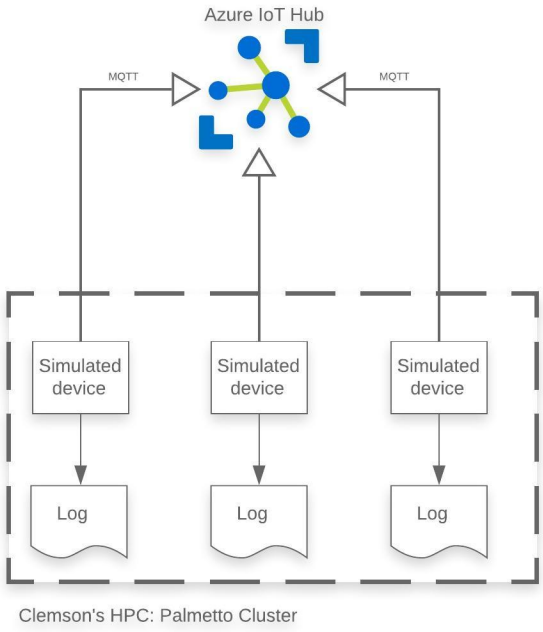


Figure 4.7: System architecture diagram.

services that could be used to process the data from IoT Hub. Some of the latency we observe is due to the C-SDK creating a separate thread to asynchronously receive the confirmation message. This is reflective of the actual end-to-end latency in a production application.

4.2.3.5 Connection Process

As noted earlier, everything that connects to IoT Hub is represented as a device. Each instance of the data generator has a unique connection string so that it acts as an individual device. Thus every instance of the data generator is a device from Azure's perspective ans as such we refer to our data generator as a device for the remainder of this paper.

The initial connection process between device and Azure requires some setup and has higher than normal latency, so we drop these initial messages (approximately the first 5%). After that point the individual device has reached steady state and latencies are indicative of a production environment.

Experiments use the default of four partitions except when comparing the latency perfor-

mance of different numbers of partitions. We choose the shortest retention time available for events, which is one day.

4.2.4 Experimental Study

We designed experiments to isolate and measure the effects on IoT Hub processing performance when varying the count of client devices, different message sizes, intermessage gap times (IMT), and IoT Hub partition count. We used representative synthetic data and performed a validation of the characteristics of the synthetic data generator prior to the experiments. In all statistical calculations we disregard approximately the first 5% of messages from each device, which are sent prior to the device reaching a steady state.

4.2.4.1 Validating Synthetic Data Characteristics

We evaluated the output of the generator to ensure it matched expected characteristics. Inputs into the data generator were the sending frequency, distribution parameters, and the type of sensor to emulate, which dictated the payload of each message. In our experiments we used either a constant distribution with a fixed IMT or a Pareto distribution with shape and scale parameters. Message payloads were JSON strings similar to those sent by real sensors, but for the purposes of this paper, we are concerned with the total message size in bytes.

To perform validation, the generator was configured to emit a series of 1000 messages with target median IMTs of 50, 100, 500, and 1000ms. We used the network packet sniffing tool `tcpdump` [6] to accurately record the host timestamp of packets sent by our data generator. We executed this part of the validation on an AWS EC2 instance since we lacked sufficient root privileges on Palmetto. For constant IMT, we calculated the median and standard deviation of the resulting message distribution. Table 4.10 compares the actual IMT to the target. We observed that standard deviation was less 0.5 ms in all but the 50 ms class, which had a standard deviation of 1.8 ms. The median for all cases stayed within 0.5 ms of the target.

For the long-tailed Pareto distribution we utilized three different shape parameters and computed scale parameters to produce median IMT of 50, 100, 500, and 1000 ms. We applied the Kolmogorov-Smirnov (KS) test [180] to evaluate the goodness of fit of the resulting distributions. The KS test results showed that in all cases the generated IMTs were a good fit for Pareto distributions with the input distribution parameters. The test statistics were all $<= 0.3$ and the p-values all fell

Table 4.10: Constant Inter-message Gap Statistics

Target Gap	Median	Std. Dev.	Mean	Max.	Min.
1000 ms	1000	0.45	1000	1005	998
500 ms	500	0.41	500	502	499
100 ms	100	0.48	100	102	99
50 ms	50	1.80	50	70	19

between 0.43 and 0.78.

4.2.4.2 Effects of Different Message Sizes

We examined whether increasing the message size correlates with increased IoT Hub round-trip latency. To observe the effect of message size on latency, we conducted experiments using 512B, 2048B, 8,192B, and 32,768B messages sizes across the three editions of IoT Hub Basic tier. Each edition uses the default 4 partitions. We repeated 30 trials for each of the 12 permutations of message size and edition, with each trial being run sequentially to avoid interference. All other data generation parameters were held constant: 10 sensors, constant IMT of 200ms, and running time of 120 seconds.

During an initial run, we exceeded the message throttling limit for the Edition 1 IoT Hub. This caused the subsequent trials of larger message size for Edition 1 to yield latencies with very large outliers. We re-ran this experiment at a later date to confirm that Edition 1 followed the same latency pattern for large message sizes as Editions 2 and 3, which was the case. We only include the results from our first set of experiments and focus on Edition 3, as we are primarily interested in best case performance.

Figure 4.8 illustrates the very high latencies of the outliers when the message throttling limit is exceeded. The box and whisker plot is shown for each permutation of message size and Edition compared to latency. Figure 4.9 shows the same data but focused on the median. Both the outliers and the median do not differ much within each edition as a result of size, nor do they differ much from edition to edition. The outliers are all between a latency of 42 ms and 500 ms, while the median hovers around 35 ms across permutations.

Runs using 512B message sizes have a higher interquartile range. We suspected that the Azure IoT SDK optimizes the sending of 512B messages, perhaps batching messages together, which

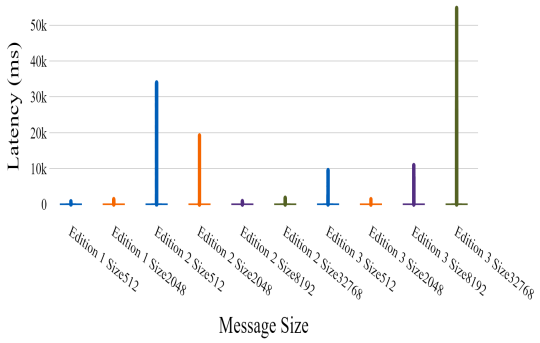


Figure 4.8: Latency outliers of 10 devices, constant IMT of 200 ms.

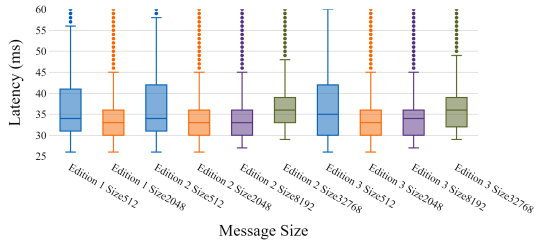


Figure 4.9: Latency IQR of 10 devices, constant IMT of 200 ms, enlarged.

would explain the relatively higher standard deviation. However, using `tcpdump` on AWS as in subsection 1, we verified that messages were being sent in individual packets so that hypothesis is not supported.

In Figure 4.10, we plot the Cumulative Distribution Function (CDF) of Edition 3 messages. 2048 B and 8192B messages follow nearly the same trace, crossing the 50% probability mark at 32ms. 32,768B messages show a higher latency until the curve plateaus and joins the other messages.

The 512B messages follow the other message sizes at first when P(L) <50%. However, between 38 ms to 41 ms latency, the distribution dips as a result of the high standard deviation. As latency increases, though, the 512B curve joins the message sizes in plateauing so that all message size cross the 95% probability mark at virtually the same spot.

4.2.4.3 Effects of Varying the Intermessage Gap Time

In a manufacturing environment, devices may send data at various frequencies, ranging from thousands of times a second to one time a minute or less. Sending data to the cloud at a high frequency can quickly exceed throttling limits and can overwhelm the message consumers. We

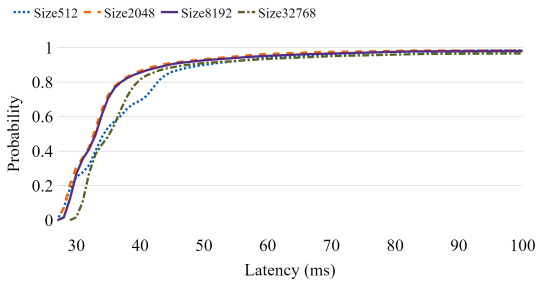


Figure 4.10: Latency CDF of 10 devices, Edition 3, constant IMT of 200 ms.

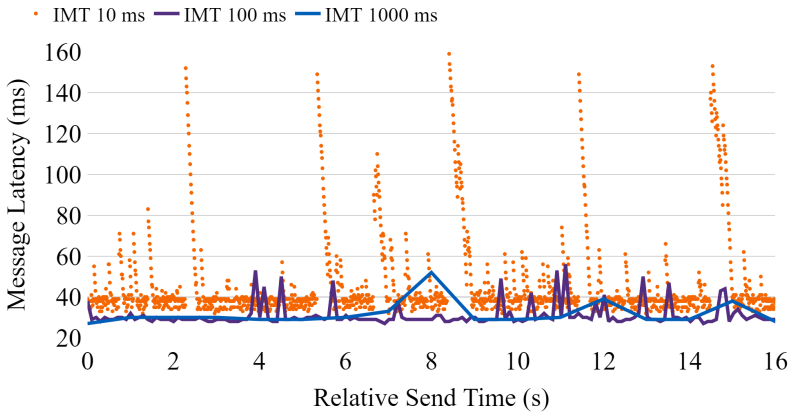


Figure 4.11: Latency of 1 device, 4 partitions, constant IMT distributions.

designed an experiment in which data was sent to the cloud at different frequencies while keeping the generation time constant.

We provisioned a B3 Edition IoT Hub with 4 partitions, and generated messages for approximately 300 seconds in each trial, sending data at one of three constant IMTs: 1000, 100, or 10 ms. We ran six sequential trials for each IMT, holding the message size constant at 2048B.

We repeated this experiment using the Pareto distribution for IMT instead of the constant distribution. We chose shape and corresponding scale parameters that would produce median IMTs of 1000, 100, and 10 ms. Table 4.11 displays the values used to reach the desired median IMTs.

Finally, we split the experiment into two groups. Group 1 had one sensor sending messages at the specified IMT, while Group 2 had 10 sensors. We performed the Pareto variation of the experiment with one sensor. All trials stayed within the throttling limits of 1 unit of B3 edition. In total, we ran 54 trials, the first group having 36 trials including the Pareto trials and the second group having 18 trials.

Table 4.11: IMT Latency Experiment: Pareto Distribution

Shape	Scale	Target Median
1	500	1000
1	50	100
1	5	10
2	707	1000
2	71	100
2	7	10
3	794	1000
3	79	100
3	8	10

For both constant and Pareto trials, the mean latency was between 31.1 ms and 53.9 ms. When high frequency runs (those with a median IMT of 10 ms) are removed the mean latency falls to range of 31.1 to 33.1 ms. The standard deviations of Pareto runs are higher than their constant counterparts.

Table 4.12: Comparing Constant and Pareto Distributions

Distribution	Median IMT	Mean Latency	Latency Std. Dev.
Constant	1000	32.7	8.7
Constant	100	31.3	7.5
Constant	10	48	26.6
Pareto α=1	1000	32.3	15.9
Pareto α=1	100	33.1	25.2
Pareto α=1	10	44.4	28
Pareto α=2	1000	31.1	11
Pareto α=2	100	33.3	15.5
Pareto α=2	10	53.9	69.2
Pareto α=3	1000	31.9	11
Pareto α=3	100	31.5	9.3
Pareto α=3	10	52.5	35.3

The Pareto trials tested a variety of shape and scale parameters as shown in Table 4.12, but for space reasons we have only included a graph showing the results from the Pareto distributions with a shape of 3.

For both constant and Pareto distributions the target median IMT of 1000 ms had very few

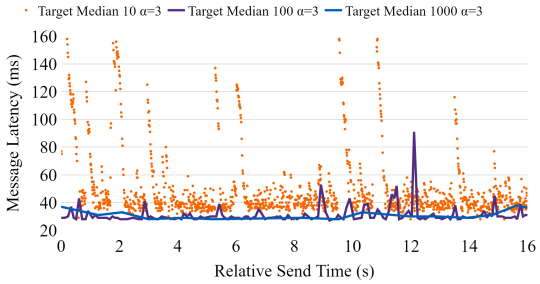


Figure 4.12: Latency of 1 device, 4 partitions, Pareto distributions with α=3.

spikes in latency, while the target median IMT of 100 ms saw slightly more spikes. However, with a target median IMT of 10 ms, spikes were very frequent and considerably worse in latency as seen in Figure 4.11 for constant and Figure 4.12 for Pareto. In all figures relative send time refers to the amount of seconds elapsed since the first kept message (after the 5% drop). For the constant distribution these spikes have a regular pattern but this was not the case for the Pareto distribution.

Due to the way partitions work, all of the messages from a single device go to a single partition in the underlying Event Hub. Subsequently, IoT Hub ensures that all messages from a single device are persisted in the order they were received. As part of this ensuring of delivery order, there are limits imposed that are being hit at the very fastest send rate (IMT 10ms). The processing pipeline will temporarily pause reading of new messages until it has flushed some of the existing messages to the underlying Event Hub. Thus the short latency spike.

Per Microsoft, the IoT Hub team already has an update planned in the message processing pipeline for an upcoming release of IoT Hub that they believe will mitigate these small spikes.

4.2.4.4 Effects of Varying the Partition Count

IoT Hub customers specify between 4 and 32 partitions when provisioning an IoT Hub, where the default value is 4. Once a partition number is set, it cannot be changed. However, the partition count does not affect the price. Increasing the number of partitions increases the number of concurrent readers for incoming messages. Thus, we tested whether latency decreased as the number of partitions increased.

We created four B3 edition IoT Hubs with partition counts of 4, 8, 16, and 32. We sent messages at a constant IMT of 100 ms and 10 ms for each partition. Each IMT was repeated in five non-overlapping trials, each using ten client devices, 2048B messages, and a 300s duration. This

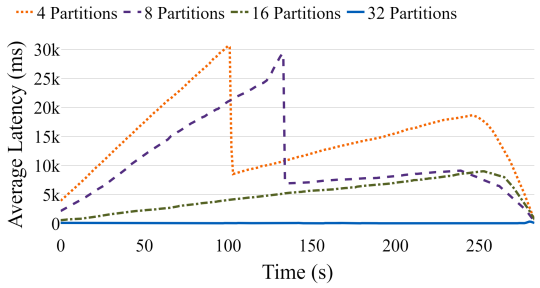


Figure 4.13: Latency mean of 10 devices, constant IMT of 10 ms.

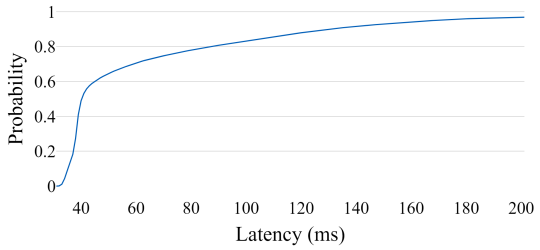


Figure 4.14: Latency CDF of 10 devices, 32 partitions, constant IMT of 10 ms.

resulted in running 40 trials total.

Figure 4.13 illustrates the best case trial based on the lowest latencies, showing the latency over time for 10 devices sending 100 messages a second, for a total of 1000 messages per second. This is well below the 6000 messages per second throttling limit for Edition 3 of IoT Hub. However, the latency was extremely high for 4, 8 and 16 partitions. The reason for this result is that IoT Hub device IDs are deterministically hashed, with each device assigned to a partition. There is no load balancing between partitions within IoT Hub, and a device always sends to the same partition. This means that for a small number of devices sending to small number of partitions, there is a high likelihood that partitions will experience an unbalanced load. For instance, the worst case would be that all 10 of our devices would end up sending messages to a single partition. The IoT Hubs with fewer partitions have a higher probability of higher average latencies and that rise drastically over time.

Sixteen partitions had a more gradual climb in latency than either 4 or 8 partitions for the trial displayed in Figure 4.13, but this was not the case for every trial. In certain trials that used different devices with unique device IDs, the 16 partition IoT Hub performed similarly or even worse than the 4 and 8 partition IoT Hubs. This is consistent with the unbalanced loads possible due to the deterministic device ID hashing.

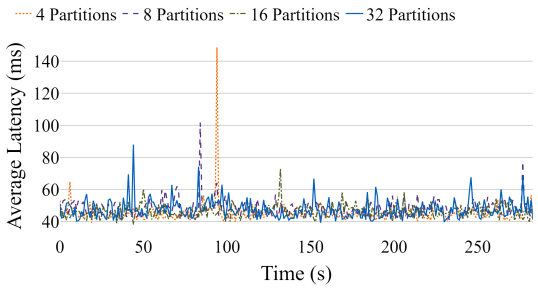


Figure 4.15: Latency IQR of 1000 devices, constant IMT of 333 ms, varying partitions.

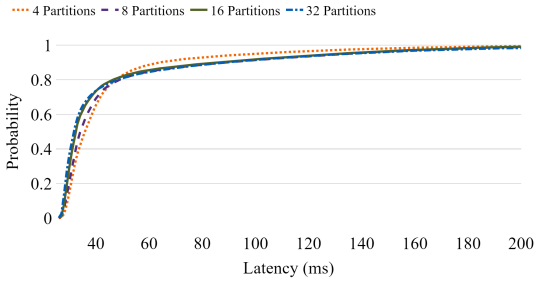


Figure 4.16: Latency CDF of 1000 Devices, constant IMT of 333ms, varying partitions.

A count of 32 partitions handled this load well. The CDF of the 32 partition latency is shown in Figure 4.14. 65% of the messages have a latency under 50 ms, while 83% have a latency under 100 ms. Finally, 97% of the messages have a latency under 200 ms. We note that there still exists the possibility of unbalanced device assignments between partitions, even with 32 partitions. However, the best performing workload for Azure IoT hub, and the one for which the design is optimized, is one with a large numbers of partitions and with messages that come from a large number of devices with a modest message frequency.

We ran another set of partition experiments to test the latency characteristics across 4, 8, 16, and 32 partitions with 1000 devices. Each device sent messages with IMT of 333 ms, or about three messages per second. Figure 4.15 shows that the performance of 4 partitions is poorer than higher partition counts, but that this workload is manageable for IoT Hubs across all numbers of partitions. While latency spikes are still present they are far less extreme and there is no evidence of climbing latency over time.

Figure 4.16 shows the CDFs of our partition experiments with 1000 devices across all trials. The CDFs for 8 partitions, 16 partitions, and 32 partitions all follow similar curves. These IoT Hubs had an 80% probability of latency under 50 ms, a 91% probability of latency under 100 ms,

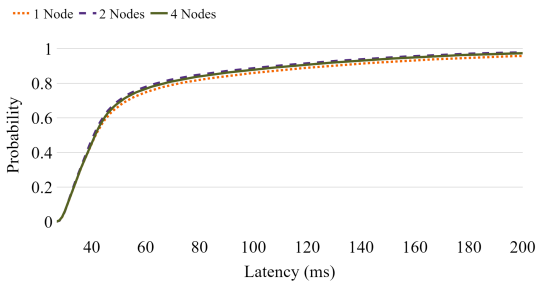


Figure 4.17: Latency CDF of 2000 Devices, constant IMT of 200 ms, sending to two Edition 3 IoT Hubs with 4 partitions each.

and a 99% probability of latency under 200 ms. The CDF for 4 partitions follows a slightly different curve with a 74% probability of latency under 50 ms, a 94% probability of latency under 100 ms, and a 99% probability of latency under 200 ms. These results are consistent with IoT Hub being best equipped to handle a large number of devices sending at a modest rate.

4.2.4.5 Scaling Experiments

In this section we describe scale-up experiments with a large number of client devices sending to Azure IoT Hub. Palmetto is comprised of multiple computing nodes. To simulate large-scale manufacturing workloads we tested a large number of client devices executing on Palmetto nodes and sending to Azure IoT Hub within throttling limits. Each computing node used for this experiment has 40 CPU cores.

Our first use case trial consisted of 1000 devices placed on 1, 2, or 4 computing nodes, each sending to a single unit of edition B3 IoT Hub. The messages all had constant IMT of 200 ms, meaning we sent 5000 messages per second. This is close to the the throttling limit (6000 msg/sec), however, we never exceeded it. All of our trial runs lasted for 90 minutes and sent a total of 27,000,000 messages.

Figure 4.17 displays the CDF of Latency across all experimental trials with devices being simulated on different numbers of nodes. Each CDF follows the same curve with a 66% probability of latency under 50 ms, a 85% probability of latency under 100 ms, a 95% probability of latency under 200 ms, and a 99% probability of latency under 300 ms.

We scaled up this experiment by increasing the number of receiving IoT Hub units. We ran 2000 devices sending to 2 units of IoT Hub and 4000 devices sending to 4 units of Iot Hub. We simulated these devices on 1, 2, or 4 Palmetto computing nodes. During these larger trials we

sent 10,000 messages per second and 20,000 messages per second respectively. We stayed below the throttling limits given for the multiple IoT Hub units. Trials continued to be run for 90 minutes. Message latency did not show any substantive change as the count of devices was scaled up. The CDFs from each experiment all follow the same curve. Azure IoT Hub behaved well under these large, stable workloads, even with message send rates above 80% of its throttling limit.

4.2.5 Summary

Using generated messages that synthesize environmental measurements and emulate patterns similar to real-world industrial IoT devices, we measured the latency and capacity of Azure IoT Hub. We simulate scaling up to thousands of devices in a manufacturing plant by leveraging the Clemson supercomputer to generate the synthetic messages.

Our work shows that when the target system does not match the workload requirements the performance is poor. For a well engineered system that fits within the specifications of Azure IoT Hub the results are predictable and well behaved. A system that stresses the specifications of Azure IoT Hub will have worse results as demonstrated by our Partition Experiment. Azure IoT Hub has been designed to scale horizontally and achieves the best results when there is a large number of devices sending data at a rate within the stated throttling limit for the specific IoT Hub Edition being used.

4.3 Conclusions

In this chapter, we have presented research on both the generation and application of synthetic semi-structured data to facilitate the validation of IoT infrastructure.

In Section 4.1 we have described methods of synthesizing large scale IoT data with noisy and complex structural characteristics in a scalable extraction and synthesis framework. The framework enables access by researchers to IoT data for the development and testing of tools and algorithms, and enables research by organizations that need to ensure the privacy of their sensitive data.

Section 4.2 motivates one use case for synthetic IoT data, where used synthetically generated communication patterns modeled on real-world IoT devices to benchmark the capacity of Azure IoT Hub. Our approach demonstrates one aspect of how scalable synthetic data generation can be a useful tool in the development of IoT pipelines.s

Chapter 5

Synthetic Data in Deep Learning

In the field of image classification and segmentation with deep learning systems, access to sets of labelled training images with sufficient quantity and quality can be a formidable barrier to training an accurate model. Collecting, segmenting, and labelling high quality images can be prohibitively expensive both in time and monetary cost. In some cases, the barrier can be lowered by pretraining a model with a generic dataset such as ImageNet [63] and then fine-tuned on a smaller set of images more directly related to the project goals. However, depending on the specificity requirements for the final model, a generalized dataset may not be useful.

A common alternative to vast quantities of readily available general images and costly task-specific images is synthetic image generation, where a 3D computer model of a scene relevant to the deep learning model is rendered to an image, segmented and/or classified, and then used to augment the training data available to the model. Synthetic image data has been used successfully in a growing body of research, in many cases reducing the overall cost of training a model.

Advantages to using synthetic images are not limited to overcoming the time and safety constraints of capturing and annotating real images. 3D modeling systems are very flexible – scenes and assets can be changed and re-rendered with a cost likely far less than the real world equivalent. For example, in the use cases presented in this work, the cost of changing the vehicle CAD model to a brand new vehicle or a new model year and then generating a new training set is far less than that of acquiring new real world examples, especially when the goal is to have a working detection system before the model enters production. The costs of developing a synthetic image generation pipeline specific to a model's goals can be further recuperated in cases where similar images can be

used to train other models, potentially requiring only minor alterations to the generator.

While the body of work around using synthetic images in deep learning models has become broadened in recent years, we have found little exploration of using synthetic images to pretrain a multistage segmentation model such as the recently proposed DoubleU-net which has been shown to be highly accurate in some applications. Our motivations in this work are to explore the performance effects of training such a model in various combinations of synthetic and real images.

In this section, we present our research on synthetic image training in the context of a real world anomaly detection system, including the results of testing on a large set of annotated proprietary production images. We believe the methodology presented here can be readily applied to other systems, and make the case that synthetic images can replace the real images and still achieve a potentially useful level of performance.

5.1 Background

Synthetic data can be used to train deep learning models in a number of ways.

First, in one extreme the model may be trained with only synthetic images, which can be useful in models where acquiring examples of desired detection conditions can be time consuming or unsafe. For example, sufficient examples of rare flaws in products on an assembly line could be time consuming to capture for a quality control model, and examples of unsafe conditions may be challenging to acquire for a video surveillance system. There has been some success with using purely synthetic data to train models [148, 164, 85], and may be a good option depending on the use case. Real images, if they exist, can be used as all or part of the test set to prove the model's accuracy.

Next, synthetic images may be mixed with real images in some combination, augmenting the size and/or variation of the training set presented to the model. In published research, this method has been used to successfully decrease model training cost or improve model accuracy, and in some cases both [133, 66, 166].

Finally, synthetic images can also be used to pretrain a model in a two-stage process, either by fitting a model to the synthetic set and then increasing the model bias toward real world examples by iterating over the real image set, or in a multi-model system such as DoubleU-net [99], which is the primary focus of this paper. This method is similar to using generalized image sets to pretrain a

system (such as robotic vision) on patterns common to the real world, and then secondary training to adapt the model to a specific environment. [sort out citations for examples here]

5.2 Related Work

Jhang et al. [100] demonstrated training a Faster R-CNN [146] object detection model using synthetic images annotated with Unity Perception and generated at scale with Unity Simulation. They found that while a model trained purely on a large (400,000) set of synthetic images performed poorly at detecting objects in situations with occlusions and low lighting, augmenting the synthetic images with a small number of real images significantly improved the detection accuracy over a model trained purely on a small (760) set of real images. Their work was inspired by and complements findings from Hinterstoisser et al. [85], who described a method for domain randomization by composing a backdrop of random objects in front of which the objects of interest are rendered and labeled. Our process is distinguished by using a randomly oriented "skybox" surrounding the subject of interest, which achieves domain randomization with lowered scene complexity and randomized reflections.

Another method of domain adaptation to insert simulated objects of interest into real images, such as in [189].

Rendered images of 3D scenes have been used to train object detection models for a long time, as exemplified by [125] and [111]. More recently, advances in 3D rendering techniques have made photorealistic image generation practical. [86] [193] [110]

Other researchers have applied full domain randomization [169] to synthetic image generation with varying degrees of success. [85, 173, 45]. Our approach is a hybrid between full domain randomization and photorealistic rendering, varying the lighting and subject/background orientation and random sampling from a set of realistic textures. The approaches described in [121], [138] and [172] are most similar to our own in this regard.

Successful specialization of U-net models has been achieved [92, 71] using VGG encoders [157] pretrained on the ImageNet [63] dataset.

Recent research has shown that models developed using synthetic data can be used as a basis for more specific models. This *transfer learning* can be used in many tasks, such as enhancing detection of object position in [93, 189] and separating target objects from visual distractors in [191].

5.3 Synthetic Image Generation

In this section we describe the tools and workflow developed for creating synthetic images, followed by our experiment designs for validating the output images and using the generated data to train a deep learning model. Software used includes Unity 2020.1, Unity High Definition Rendering Pipeline (HDRP) 7.4.1, and PiXYZ Plugin 2019.2.1.14.

5.3.0.1 3D Modeling

The image generator was built as a set of scene descriptions, models, and scripts in the Unity 3D game development platform.

For our use case, a vehicle model was translated from its native CATIAv5 CAD format [check name] into a Unity asset with the PiXYZ plugin. Importing the CAD object was relatively labor intensive due to a technical difficulty in mapping part materials to Unity textures, which is an area of current work. The work-around for our purposes was to manually assign textures to the approximately 10,000 visible surfaces in the imported Unity asset.

5.3.0.2 Realistic Rendering

In general, synthetic images for model training need to exemplify the characteristics of real images that the model relies on for accurate classification. While these qualities could be vastly different depending on the model, for our use case we needed images that embody the broad range of shadows and reflections seen in the production environment. Rather than attempting to identify and optimize for the most important image features, our approach was to create images as accurately as possible with a goal of being indistinguishable from real images by a human observer.

Images were rendered using the Unity High Definition Rendering Pipeline. We relied on a number of Unity features designed for high rendering accuracy, and avoided many approximation features designed to improve rendering performance in a game setting requiring high framerate with limited hardware resources. The Unity "camera" object was configured to mimic the properties of the physical camera used to capture real images. A full disclosure and justification of the rendering settings we used would be lengthy and beyond the scope of this paper, and will be made available on publication.

We found that several external resources were very helpful in creating realistic image render-

ing, especially in our use case with automotive models. In particular, Unity's automotive industry-focused Measured Materials library [114] helped us simulate the paint, glass, rubber, and plastic textures of a real vehicle. Skyboxes were sampled from the Unity HDRI pack, captured using techniques described by Lagarde et al. [108].

5.3.0.3 Domain Randomization

We chose a hybrid approach to domain randomization, rendering the image subject as accurately as possible with ambient lighting similar to the production environment. Randomized attributes included subject position relative to the camera within plausible constraints, vehicle exterior paint colors from a set of possible values, and a single light source (the sun) with varying position.

To separate the subject from the background, we used a background skybox with a very "busy" texture, and then randomized its orientation on all 3 axis for every scene. This served a secondary purpose in creating randomized reflection patterns on all surfaces of the vehicle.

Randomization of objects in the scene was accomplished with a set of scripts written in C-sharp, used natively in Unity for game logic.

5.3.0.4 Segment Labeling

We labeled image segments by capturing multiple images from each randomized scene – one fully rendered image, and then one false color image for each segment. This could have been achieved in many ways, but the approach we found to be most performant in Unity was to maintain a second "mask" copy of the subject model completely colored with an "unlit" black texture, locked to the same position as the color model. Two identical cameras in the same position were used, one able to see the color model, background, and lighting and the other camera only able to see the mask model.

After the normal image was captured with the color camera, the segment capture phase would iterate through groups of components comprising each segment, recolor the group with an unlit white texture, capture an image with the mask camera, and then recolor the group to the unlit black texture. Figure 5.1 shows the resulting image segments. This approach had the performance advantage of minimizing the retexturing of materials on the model. This also allowed us to capture occlusions by components not part of the segment of interest, such as the door handles in the example

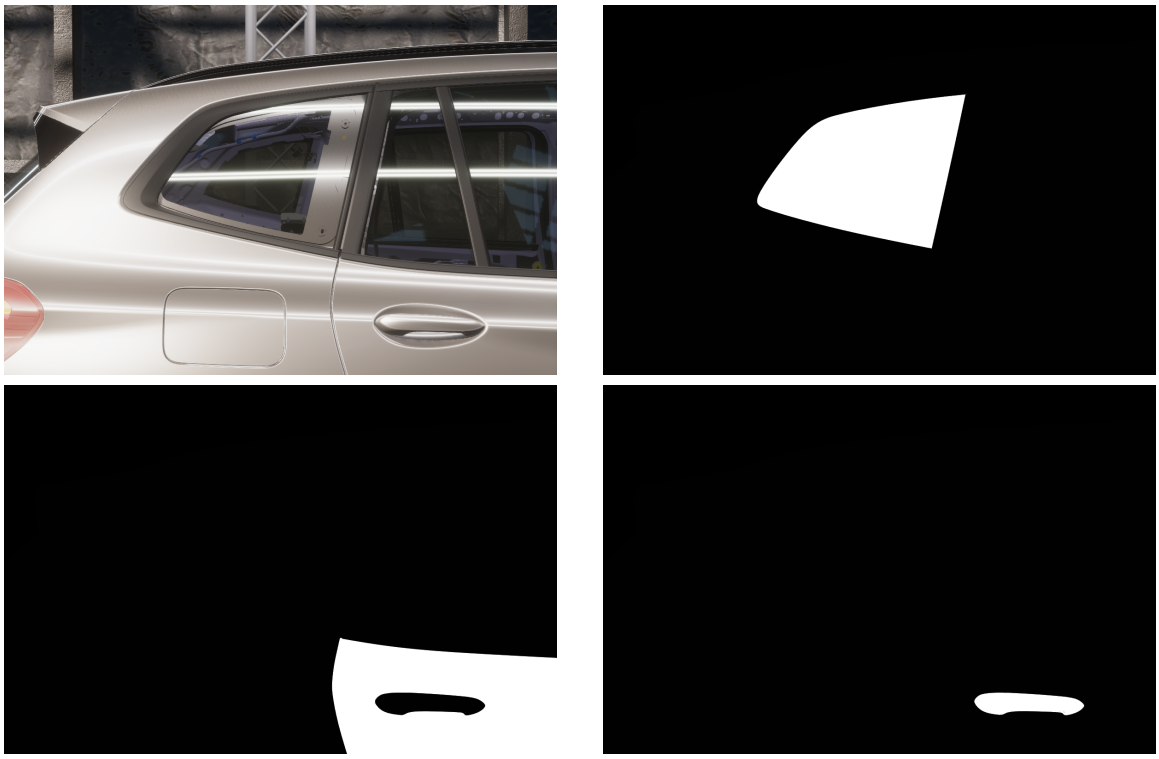


Figure 5.1: A 3D generated image (top left) in addition to a series of one-hot encoded masks segmenting each object class.

images.

5.4 Model Training

To validate the effectiveness of the synthetic image generator, we conducted experiments comparing models trained with varying amounts of real labelled images augmented with synthetic data. Our available data consisted of 14,125 labelled images of real vehicles in a production line, each of which contained one or more examples of eight distinct feature classes. From this dataset, a 10% holdout set was randomly selected for validating models, leaving 12,712 images in the real dataset R for training. The frequency of each feature's appearance is described in Table 5.1, where the subset of the real image set R with one or more pixels belonging to a feature class f is given as $R_f = \{e|e \in R \text{ and } f \in e\}$, and an example frequency of $|R_f|/|R|$.

Using the synthetic image generator described in Section 5.3, we rendered a set of 40,406 synthetic images and labels S with the same feature classes as R. Due to a slightly smaller horizontal

	real images R		synthetic images S	
feature	examples	frequency	examples	frequency
back door	5,994	47.09%	40,231	99.57%
back window	5,854	45.99%	40,263	99.65%
rear window	4,844	38.05%	24,080	59.60%
front door	6,599	51.84%	22,308	55.21%
front window	5,985	47.02%	26,171	64.77%
door handle	4,670	36.69%	40,084	99.20%
mirror	3,897	30.62%	6,932	17.16%
tail light	4,511	35.44%	8,501	21.04%

Table 5.1: Feature Example Frequency in Image Sets

range of camera freedom, some classes were represented more or less heavily in the synthetic set, as detailed in Table 5.1. However, as we weight each class equally in our metrics and present aggregate statistics over the entire dataset, we deemed that the example frequency weights would not affect the conclusions.

5.4.1 Training Methodology

Images and labels were used to train U-net [150] convolutional neural network models implemented in TensorFlow [11] 2.0.0 and Keras [57] 2.2.4-tf. Models were trained using an NVIDIA DGX-2 with Tesla V100 GPUs running Ubuntu 18.04.4 LTS.

In this section, all U-net model structure and parameters are identical with the exception of input datasets. The U-net implementation was derived from code provided by Debesh et al. in their Double-U-net supplement, to be consistent with the further work in Section 5.5. From the original U-net description, the only significant difference is the use of batch normalization [94] after the convolutional layers along the contracting path, which resulted in more consistent training and better generalization in our use case.

A hyperparameter search using real and synthetic datasets revealed optimal parameters that were similar enough to avoid differentiation between the domains. As the purpose of this work is to explore the tradeoffs of synthetic vs real data, we chose parameters that resulted in consistent and stable training sessions rather than strictly optimizing for the highest possible accuracy. For our datasets, a dropout probability of 0.30, a batch size of 64, and a learning rate of 0.0020 resulted in models that converged quickly and consistently within a reasonable limit on training time and generalized well to the validation data.

As synthetic data can be seen as a form of data augmentation, we chose to forego any traditional augmentation techniques (randomized cropping, gamma shifts, etc.) to present clear results, with the single exception of randomly flipping all training images horizontally to match the real dataset's imaging of both sides of the vehicle. During training, models were evaluated each epoch against the disjoint validation set. To prevent overfitting, we used an early stopping mechanism to halt training and revert to the best weights if no improvement in validation set prediction loss was made over 30 epochs.

5.4.1.1 Metrics

While the image generation and training techniques share applicability with object detection and instance segmentation models with more actionable metrics, we quantify the performance of a standard multiclass U-net segmentation model simply with per-pixel mean intersection-over-union (mean IoU) with uniform class weighting and a prediction threshold of 50%.

5.4.2 Real Dataset Supplementation

To determine how supplementing a dataset of real images with synthetic images would affect model training and accuracy, we trained instances of multiple model classes with different mixtures of images from both sets. Subsets of the real image set R of sizes $N = \{0, 16, 32, ..., 8192\}$ were paired with subsets of the synthetic image set S from the same size range, forming the axes of the 11x11 matrices shown in Figure 5.2 with model classes at each intersection. For each model class, random samples from R and S were used to train individual U-net segmentation models with parameters reported above. The number of models trained in each class was sufficient that the confidence interval ($\alpha = 0.95$) width of the mean truth/prediction IoU measurements on the real image validation set was less than 5% of the mean value, requiring between 7 and 30 model instances for each image set size pair. We refer to the resulting set of segmentation models as M, where $m_{r,s,i} \in M : r \in N, s \in N, i \in [0..|M_{r,s}|)$ is one instance of a class of U-net models trained on (r, s) random images from datasets R and S, and we report aggregate statistics over the model class $M_{r,s}$ at each cell in the matrices of Figure 5.2.

Figure 5.2a aggregates the mean IoU predictions of each trained model class on the unseen validation set from the real image domain. We observe a general trend of increasing accuracy with larger samples of real images, with diminishing returns as the training images grow to sufficiently

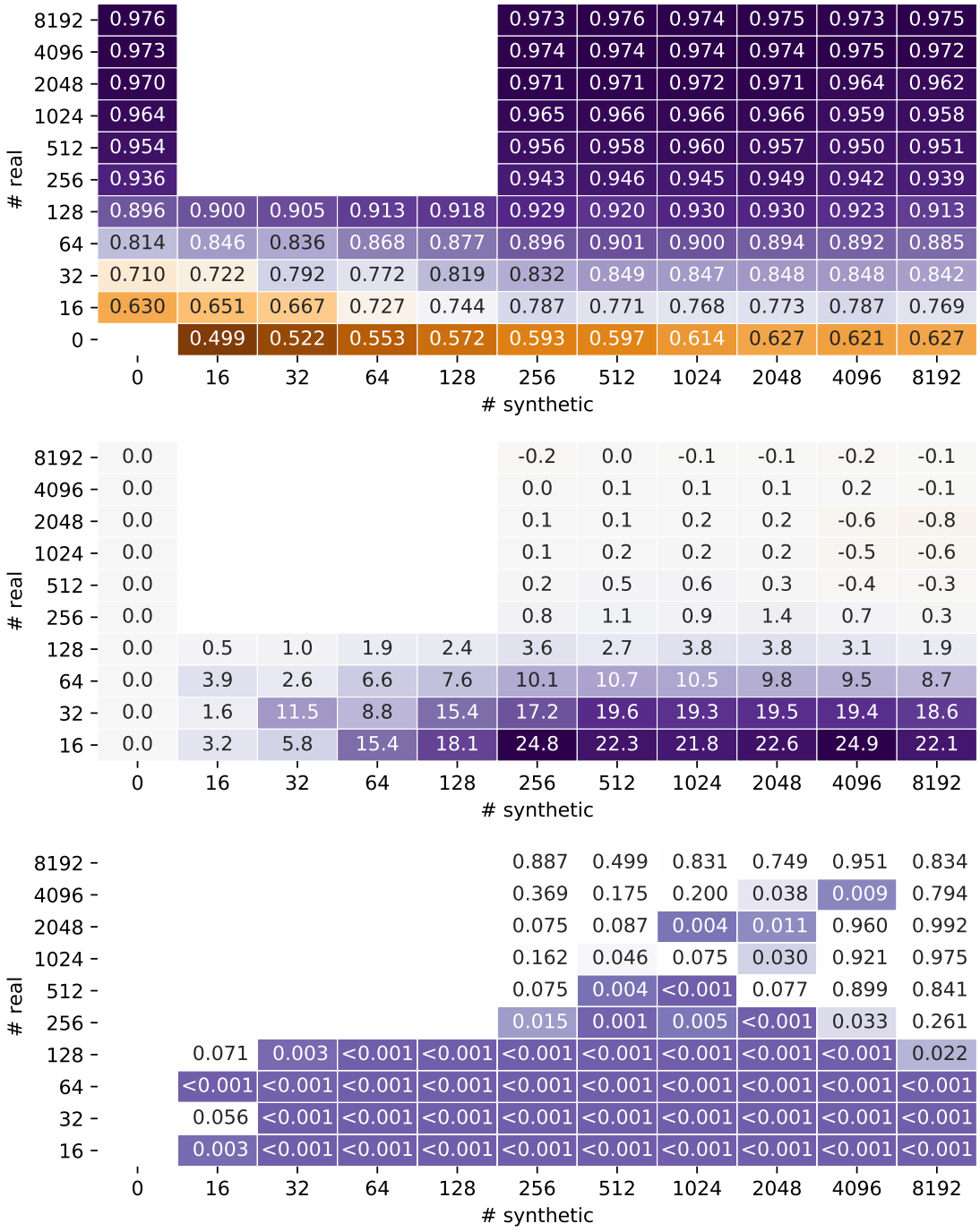


Figure 5.2: Aggregated mean prediction IoU (a) of U-net models trained on random samples from real and synthetic datasets. Models augmented with synthetic data showed up to 24.9% higher prediction accuracy (b) than the baseline, particularly with limited amounts of real training images. The p-values (c) of one-sided T-tests, $H_a : \overline{IoU}(M_{r,s}) > \overline{IoU}(M_{r,0})$, show significant accuracy increases ($p \leq 0.05$, highlighted) in most models trained with 256 or fewer real images.

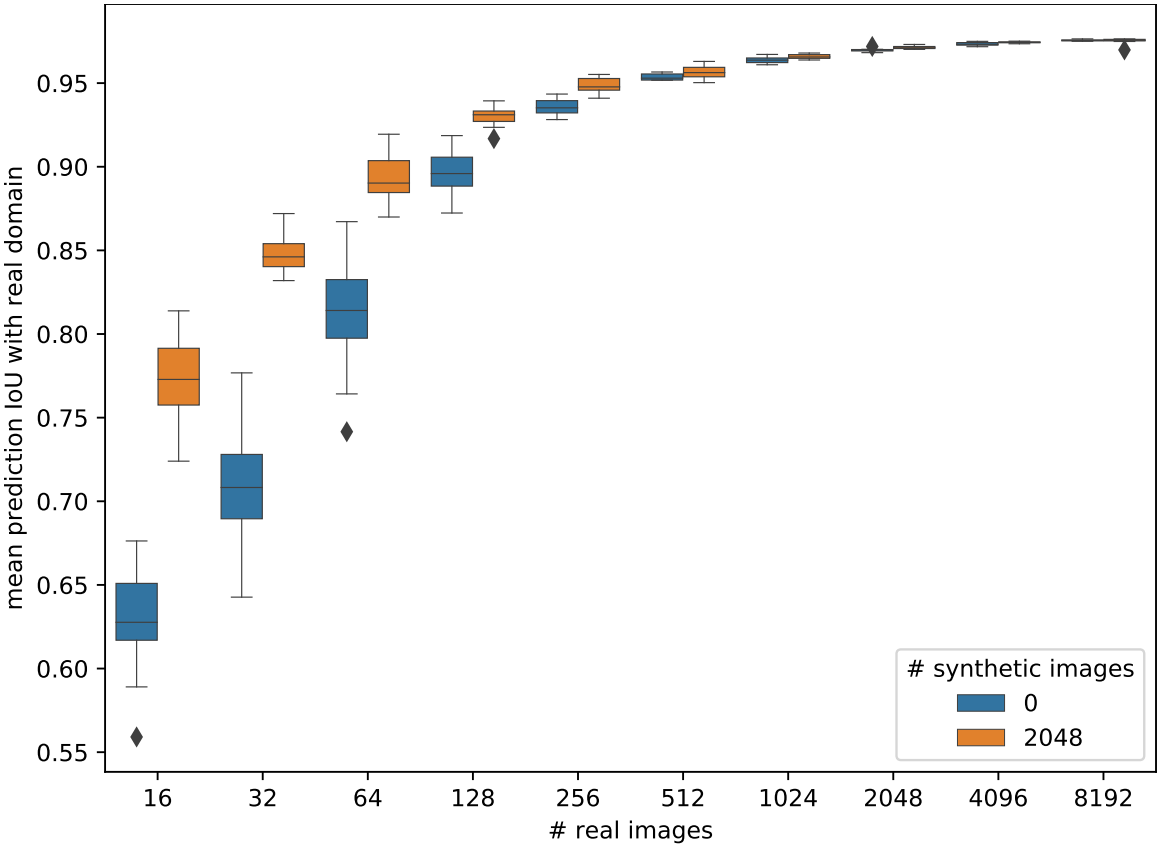


Figure 5.3: Mean IoU of model predictions on a validation set of 1176 real images. Each IQR plot describes between 7 and 30 individual U-net models trained on random subsets of real and synthetic images. In general, augmenting smaller (≤ 256) sets of real images resulted in higher accuracy and less variation in the trained models, with diminishing returns as the real data became sufficiently representative of the domain.

represent the domain features. Along the horizontal axis, we see that augmentation with synthetic data tended to increase accuracy, with greater yields in models trained on smaller real datasets. We also observe that models trained on purely synthetic data tend to poorly predict the real domain, even with thousands of examples.

To discuss the results of synthetic data augmentation, we first look at the effects of augmentation on model reliability. Figure 5.3 shows the summary statistics of mean validation set predictions for model classes trained on purely real images and those augmented with 2048 synthetic images, which details columns 0 and 2048 from Figure 5.2a. Models trained with smaller random samples of real images tended to show more variation in their resulting prediction accuracy. We observe that augmentation tended to increase mean accuracy and decrease variance in models

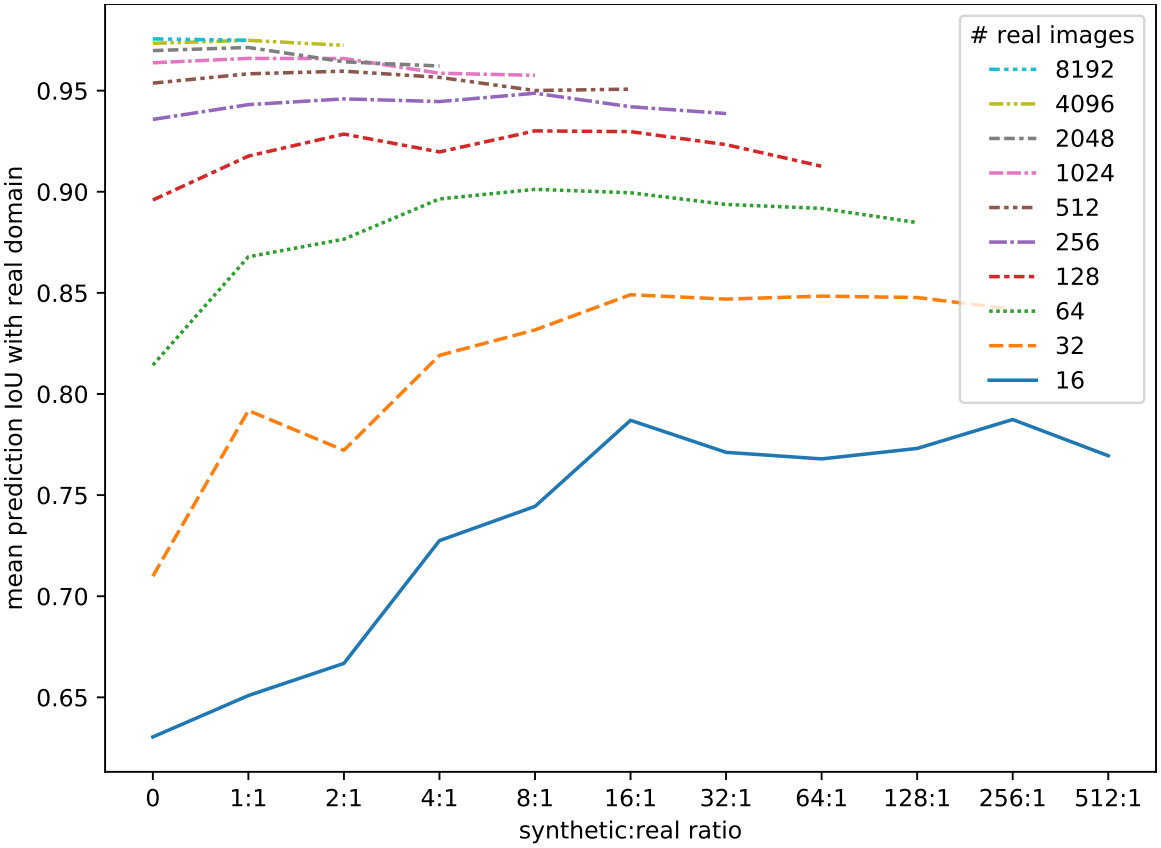


Figure 5.4: Mean prediction IoU of U-net models on real images, viewed by the ratio of real to synthetic data in the training datasets. Each trend exhibits an inflection point where accuracy decreased, presumably due to limited capacity of the model to encompass both the real and synthetic domain.

trained with less than 256-512 real images.

Augmenting the real training sample with varying amounts of synthetic data yields better results, depending on how accurate the model is to begin with. Figure 5.2b reshapes the data in Figure 5.2a as a percentage increase in mean prediction IoU relative to that of the pure real set (column 0). We can see that augmenting models trained with 512 or more real images only results in a marginal increase, at best 0.6%. However, in models trained with 256 or fewer real images, the accuracy increase is substantial, up to 25.0% when only 16 real images are available. We can also see that the addition of any amount of real images results in models that are more accurate than those trained on synthetic data alone. This is supported by the p-values of one-sided T-tests, $H_a : \overline{IoU}(M_{r,s}) > \overline{IoU}(M_{r,0}) \; \forall \; r, s \in N$, shown in Figure 5.2c with $p < 0.05$ highlighted.

Figure 5.2b also shows that in some cases, particularly in those with 512 or more real

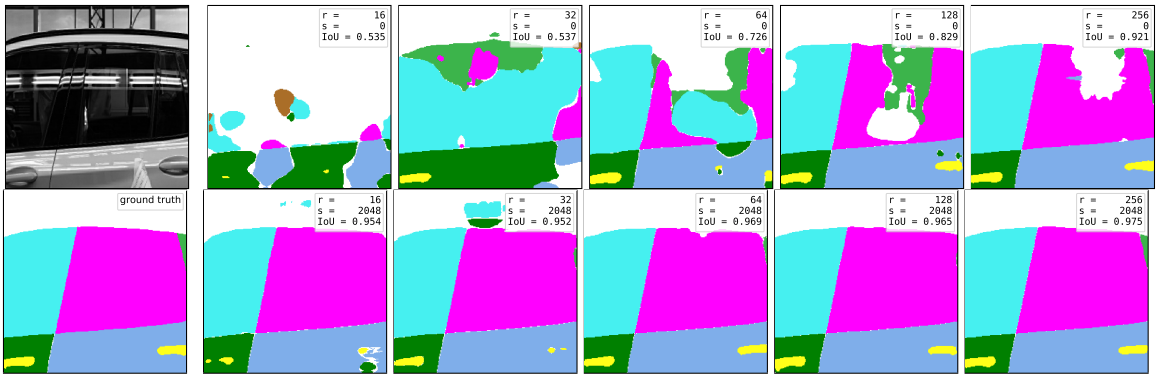


Figure 5.5: Segmentation map predictions of U-net models trained with pure real images (top) vs. the same training sets augmented with 2048 synthetic images (bottom). The input image and ground truth are shown on the left for reference.

images, the addition of large amounts of synthetic data correlate with a slight decrease in prediction accuracy, presumably due to dilution of the samples from the real domain and a limited capacity of the model to encompass both the real and synthetic domains. We can observe this trend more clearly when viewing the relationship between real and synthetic image set sizes as a ratio, shown in Figure 5.4. Each real image set size exhibits an inflection point where accuracy declines, which we suspect is dependent on the capacity of the model and similarity between real and synthetic data in a particular use case.

To visualize the differences in prediction accuracy, Figure 5.5 presents the segmentation maps predicted by 10 different models, trained on 16-256 real images and augmented with either 0 or 2048 synthetic images. In contrast to the randomly selected images used to train the models Figure 5.2, each real dataset larger than 16 images is a superset of the smaller datasets, and the same real datasets and 2048-image synthetic dataset are reused in each of the augmented models. For this example image, the quality of the predictions are fairly low in the pure real models, limiting usefulness depending on the use case. The addition of synthetic images results in clearly defined door/window boundaries with even the smallest real training set, and better identification of smaller features such as the door handles at 64 real images compared to requiring 128 without augmentation.

5.5 Transfer Learning

Another potential use case for synthetic data is in pretraining models for later improvement with real data, either as a base for multiple specialized models or as a starting point for incremental

training as real data becomes available. Our results from the previous section indicate that U-net models trained with 256 or fewer images from our real image dataset suffer from low applicability to new images, so in this section we will focus on pretrained model refinement with small numbers of real images.

The goals and requirements for transfer learning can vary widely, but in our exploration we will focus on use cases stemming from unavailability of real labelled training images and from the need to specialize a general model for a particular task. As such, we will quantify results in terms of accuracy (in this case, mean prediction IoU on real data) and training time of the model specialization training.

5.5.1 U-Net

There are many strategies for transfer learning using the U-Net model [cite], most involving freezing, reinitializing, adding, or removing layers. It is beyond the scope of this work to explore the many factors involved in choosing the optimal strategy for a particular use case. We will instead focus on a relatively simple technique that compares well to our work with a more advanced model in the next subsection, which to train a U-Net with purely synthetic data, and then continuing training with real images while optionally freezing or replacing part of the model. Our base synthetic-trained U-Net model uses parameters as described in the previous section, trained with a larger dataset of 36,480 synthetic images, which achieved 0.954 mean prediction IoU on the holdout set from the same synthetic domain. Accuracy on segmentation of real images was similar to the experiments with large pure synthetic datasets in the previous section, only achieving a mean prediction IoU of 0.618 on that domain.

Starting with an identical U-Net base model initialized with random weights, experiments were configured as follows:

- *synth-random* - only the contracting path (*encoder*) was initialized with weights from the pretrained base, allowing the untrained expanding path (*decoder*) to train completely on real data;
- *synth-synth* - both the encoder and decoder were initialized with pretrained base weights;
- *VGG19-random* - the encoder part of the model was replaced with VGG19, detailed below, and the decoder left with random weights;

- *VGG19-synth* - the encoder was replaced with VGG19, and the decoder initialized with pre-trained base weights;
- *control* - the base model was used without freezing or replacing layers, and the initial random weights were unchanged. Note that this is the same configuration as models in the previous section, and the resulting model is trained on purely real data.

Finally, we doubled the above configurations with another parameter, choosing to either freeze the layers of the encoder portion of the model or allow the secondary training with real data to propagate and update the encoder weights. Our expectations were that freezing the encoder section of the model would reduce training time as there were less parameters to update with each back-propagation, but could reduce the model's ability to adapt to the new data. Table 5.2 details the number of trainable parameters and mean training time per image-epoch for the four resulting model architectures, which indeed shows decreased time per image with less parameters to update.

For some experiments, the encoder layers of the model were replaced with a VGG19 [157] model pretrained with weights from ImageNet [63], following the same procedure as the work done in [99] for comparability. With the models initialized with pretrained weights, we continued training using randomly selected subsets of real images until convergence, using the stopping criteria described in the previous section. All model variant and real image sample size permutations were repeated 30 times.

Table 5.2: Model Size and Training Time

		parameters (millions)		training time per
model	variant	total	trainable	epoch-image (s)
U-net		7.77	7.77	0.0130
U-net	frozen encoder	7.77	3.05	0.0120
U-net	VGG19 encoder	23.86	23.86	0.0176
U-net	frozen VGG19 encoder	23.86	3.83	0.0153
W-net	frozen 1st U-net	10.11	2.34	0.0146
W-net	frozen VGG19 encoder	26.59	6.56	0.0195

We first compare on the frozen/trainable encoder variable, visualized in Figure 5.6. In models using VGG19 as the encoder, we observed greater prediction accuracy and lower training time, while models using our encoder pretrained on synthetic data tended to perform better when the encoder was not frozen during secondary training. This is perhaps due to the large difference in the number of encoder neurons, as propagating the training feedback from each example through

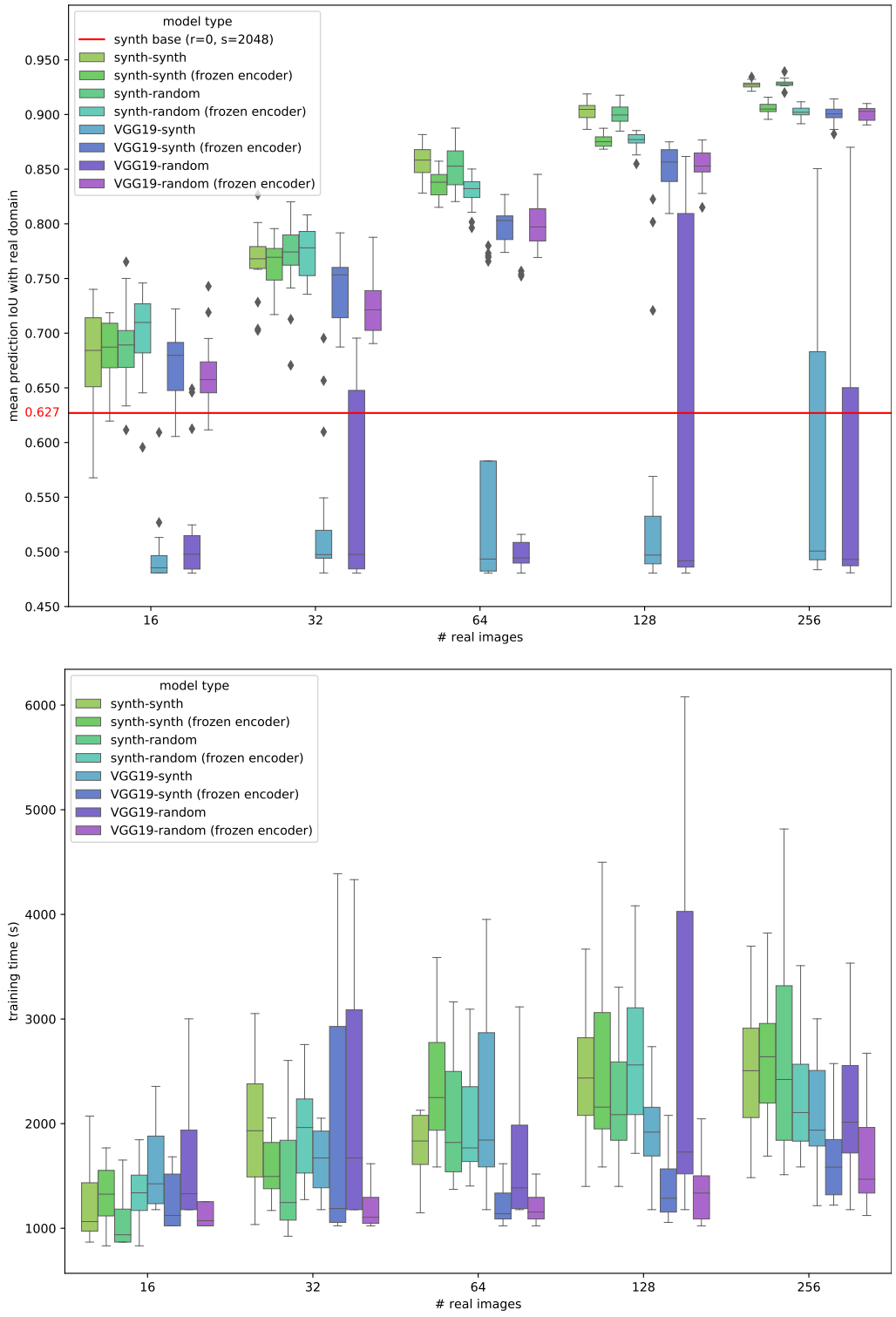


Figure 5.6: Comparisons of mean prediction IoU (a) and training time (b) of secondary training of pretrained U-Net models, with the weights of the contracting path (*encoder*) either trainable or frozen.

the larger VGG19 encoder is more costly and less impactful. We speculate that limiting the neurons being updated each epoch lead to faster model convergence while the models with more trainable weights slowed in training progress enough to trigger early stopping. The training logs support this conjecture, showing extremely slow improvement before training was terminated. It is possible that, given enough time, the accuracy differences between trainable and frozen versions of the same model would minimize. However, since all models use the same early stopping criteria, we present the results as comparable in a practical sense. In the remainder of this work, comparisons with these models will use the better-performing frozen encoders in the case of VGG19, and trainable encoders for the synthetic data-trained models.

Next, we compare the mean prediction accuracy of the retrained models with frozen encoders to the control models trained from randomly initialized weights. We observed that in cases with 64 or fewer real images, we saw an increase in accuracy over a control model trained on purely real data. However, in larger real image classes and with all control models trained on a mix of real and synthetic data, we saw significantly lower accuracy in the specialized models. We again speculate that the model training may have slowed enough to trigger our early termination criteria, and that a combination of refined learning rate, early termination parameters, and lengthened training time may result in improved accuracy. Our goals in this work are in comparability between experiments, though, so we present these results as a baseline to be improved upon.

In comparing the prediction accuracy of U-net models with different decoder weights, we saw mixed results; the pretrained synthetic data weights appeared to result in lower performance in models with synthetic weighted encoders trained on 16 or 32 real images, while having the opposite effect in models with VGG19 encoders. In models trained on 64 or more real images, the results were less clear; and a two-sided T-test showed insufficient difference to conclude that the results are drawn from different distributions at $p = 0.05$.

Comparing encoder paths of the different model classes was more consistent, in that the U-net default layers trained with synthetic data resulted in higher mean prediction accuracy than models using the VGG19 encoder trained on ImageNet, across all real data sample sizes. We conclude from these findings that a relatively small encoder (4.72m parameters) trained on a few thousand images drawn from a *similar* synthetic domain to the target can outperform the already impressive feature extraction of a large (23.03m parameters) encoder trained on over a million generic real images.

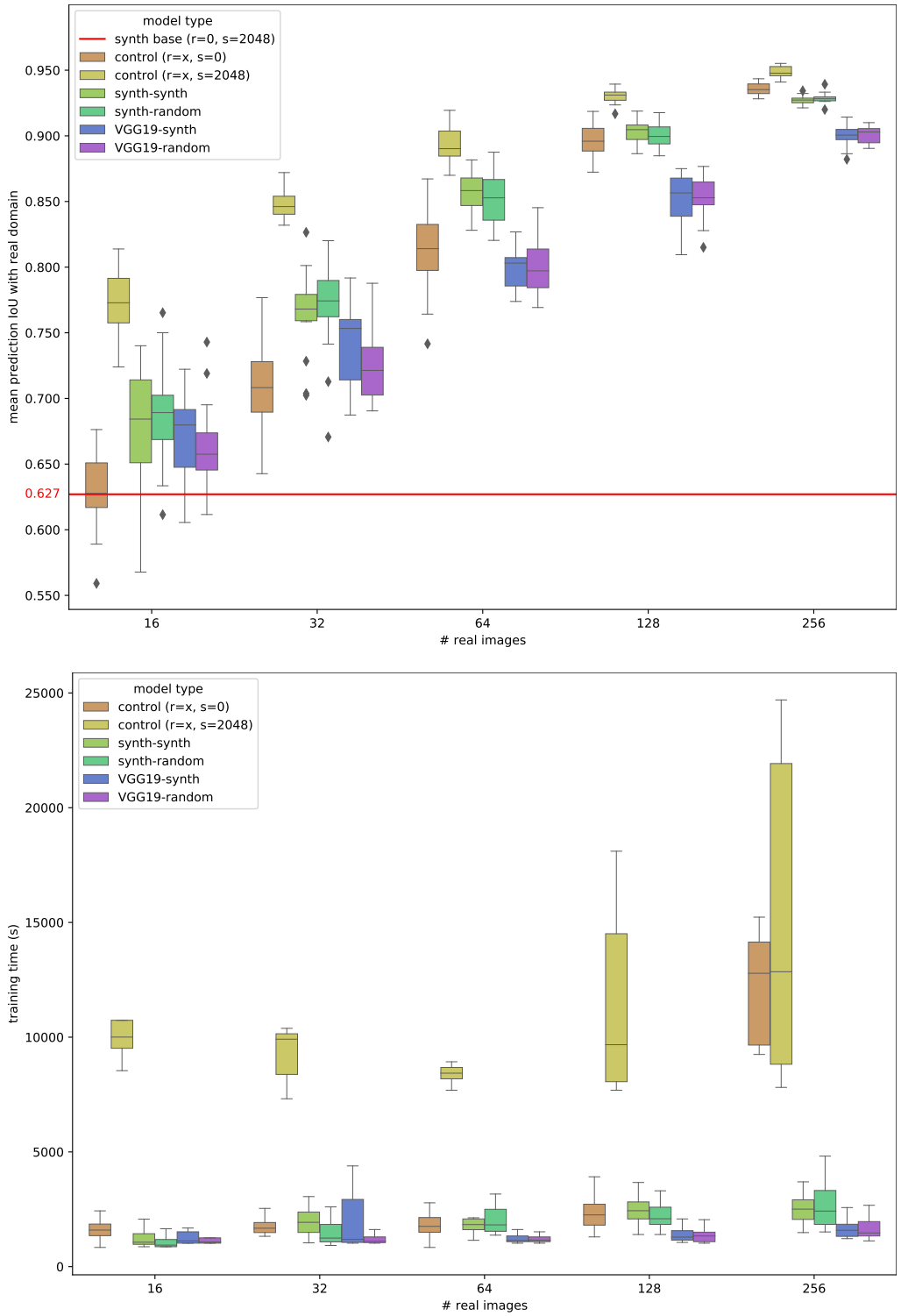


Figure 5.7: Limiting to encoder type and decoder initial weights (synthetic pretrained vs. random) model permutations, we observed a sizeable tradeoff between mean prediction IoU (a) and training time (b) when compared to models initialized from randomness.

Figure 5.7b compares the training times of retrained models to those of the control model for each real sample size class, with results between 10.0% and 20.8% of the time required for the control. The time can be accounted for in both the number of trainable parameters in the retrained models with frozen encoders, and the number of epochs required to converge. As the mean training time for a purely synthetic U-net (r=0, s=2048) is 11,648 seconds, the training time for a retrained U-net is comparable to that of the control.

5.5.2 Double-U-Net

Since the introduction of U-net in 2015, a number of derivative models have been proposed that improve its applicability to certain use cases. One of these, the Double-U-net [99], improves upon the localization of segment instances by dividing the task between, as the name suggests, two U-net models linked together. The first U-net, using a VGG19 encoder trained on ImageNet, outputs feature maps from each level of the encoding process as well as an intermediate segmentation map from the decoder. The segmentation map is paired with the original image as input to the second U-net, while feature map outputs of the first U-net are linked to corresponding layers of the second U-net decoder. The authors' results showed impressive accuracy gains over a standard U-net on a variety of medical segmentation datasets.

As an exercise in applying transfer learning to a more complex model, we chose the Double-U-net (abbreviated *W-net* for the remainder of this work) because of its intuitive design as a logical extension to the standard U-net, as well as having experience and success using the model in some production use cases. Our experiments in this section will expand on the previous section for ease of comparison, with the caveat that we made some implementation choices toward this goal while potentially sacrificing some peak performance. For example, the authors of W-net used *squeeze-excite blocks* [88] at the end of each convolutional block, which is not part of the original U-net specification. Additionally, in our image set, vehicle features were largely scale-invariant, as the images were captured from a fixed viewpoint with a low variation in the vehicle's distance from the camera. This warranted omission of the Atrous Spatial Pyramid Pooling (ASPP) block between the encoder and decoder in each U-net, which was used in [99] to handle feature scaling. We conducted a limited exploration and found these features to contribute little to no performance gains on our particular use case, so we believe that the simplified model is a better comparison to transfer learning results on a simple U-net in the previous section.

Our W-net implementation is simply two U-net models, identical to the implementation described in the previous section, with the following two additions. First, as in the [99], the U-nets are connected with a pixel-wise multiplication layer, such that the second U-net receives the original image augmented with the segmentation map output of the first U-net. Second, the encoder layer-wise feature maps from the first U-net are concatenated to the inputs of the second U-net decoder, in the same manner as the feature maps from the second U-net encoder.

Following the work in the previous section and as an analog to [99], we chose to construct Double-U-nets with two model variations. In the first model, we use a U-net trained on synthetic data as described above, with the entire first U-net frozen. The second model, analogous to [99], uses a frozen VGG19 encoder and a trainable uninitialized decoder. In both models, the second U-net is initialized with random weights and is fully trainable. Our hyperparameter search revealed optimal parameters very close to those used to train the individual U-nets, so we opted to keep the original parameters for comparability.

Our results, shown in Figure 5.8, show accuracy improvements using the W-net model with the VGG19 encoder over all training image size classes, and similar or better results with the synthetic-trained first U-net. The accuracy improvements correlate with a training cost increase, however, especially with the VGG19-based models with more layers to train. The conclusion we draw from these results is that secondary training with a multipart model like W-net can be a viable accuracy enhancement if the time cost can be justified.

5.6 Conclusions

We found that, for this image segmentation problem, synthetic images were an effective technique for augmenting limited sets of real training data. We observed that models trained on purely synthetic images had a very low mean prediction IoU on real validation images. We also observed that adding even very small amounts of real images to a synthetic dataset greatly improved accuracy, and that models trained on datasets augmented with synthetic images were more accurate than those trained on real images alone. We noted that for this domain, 256 to 512 images seemed to be enough to train a reasonably accurate model, with rapidly diminishing returns on adding synthetic images to the mix, eventually resulting in lower accuracy as the real:synthetic ratio dropped.

In use cases that benefit from incremental training or model specialization, we found that

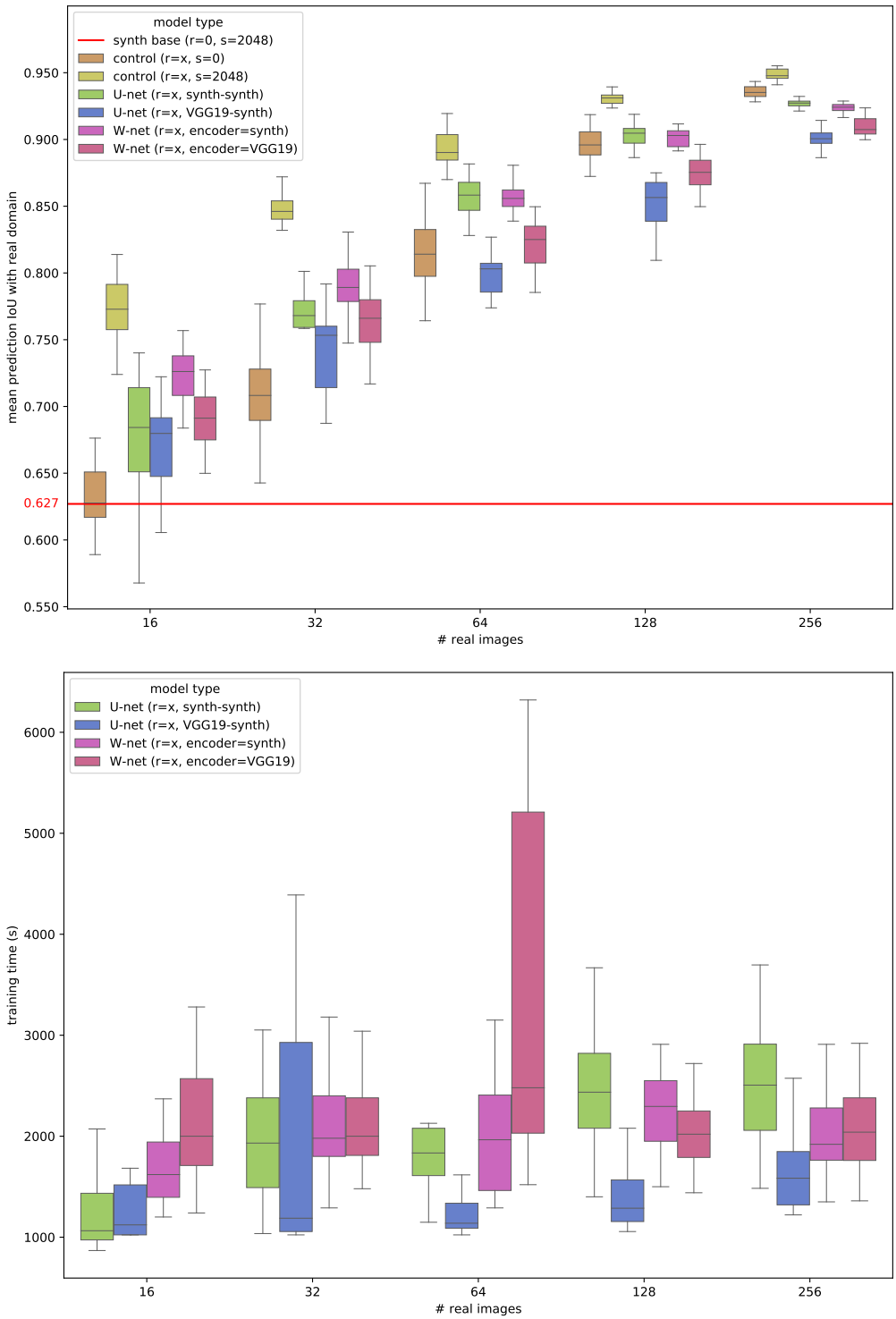


Figure 5.8: Results of transfer learning on U-net and W-net models with (first) encoders trained on synthetic data or VGG19/ImageNet, compared to training of *control* models initialized with random weights. The mean prediction IoU (a) and training time (b) suggest improved accuracy of synthetic-trained encoders, but in some cases with a time cost.

pretraining on synthetic images provided a usable base model for transfer learning. While we observed that models trained in a single session outperformed those pretrained on synthetic images and retrained on real data, we also saw that up to 90% of the total training time could be completed in the pretraining phase.

We conclude that synthetic image generation can be beneficial to segmentation model training when insufficient images are available to train a satisfactory model. However, testing must be done to find the break point where adding more synthetic images does not result in higher mean accuracy.

Chapter 6

Conclusion

In this dissertation, we have used examples from original research to show that, using appropriate models and input parameters, synthetic data that mimics the characteristics of real data can be generated with sufficient rate and quality to address the volume, structural complexity, and statistical variation requirements of research and development of digital information processing systems.

In Chapter 3 we presented a progression of research studies using a variety of custom and industry-standard benchmarking tools to generate synthetic network traffic patterns. These benchmarks allowed us to observe relationships between network characteristics and the performance of HPC applications at all levels of the network stack. In Section 3.1, we used the `netperf` tool to generate traffic and measure the effects of software switches and virtualization on aspects of network performance. Section 3.2 described our work in using custom microbenchmarks using MPI primitives to generate low-level network traffic patterns and observe the effects of container virtualization and software switches various network performance measures. We then employed the NAS Parallel Benchmarks to generate distributed traffic patterns common to HPC applications and characterize the higher-level performance impact. In Section 3.3 we continued our work in a traditional HPC environment with low-latency InfiniBand networking, using low-level synthetic benchmarks from the OFED `perftest` to gauge network performance at the hardware level.We then measured and characterized the impact of artificial latency with custom MPI benchmarks at the library level and NPB at the application level.The project concluded in Section 3.4, where we applied the previous work to an emerging non-traditional HPC environment by measuring the effects of network performance

on NPB and LAMMPS in AWS and Google Cloud. These results demonstrate that algorithmic synthetic traffic generation tools play an important role in the research and validation of complex networking systems.

Chapter 4 presented research on both the generation and application of synthetic semi-structured data to facilitate the validation of IoT infrastructure. In Section 4.1 we described methods of synthesizing large scale IoT data with noisy and complex structural characteristics in a scalable extraction and synthesis framework. The framework enables access by researchers to IoT data for the development and testing of tools and algorithms, and enables research by organizations that need to ensure the privacy of their sensitive data. Section 4.2 motivates one use case for synthetic IoT data, where used synthetically generated communication patterns modeled on real-world IoT devices to benchmark the capacity of Azure IoT Hub. Our approach demonstrates one aspect of how scalable synthetic data generation can be a useful tool in the development of IoT pipelines.

Finally, Chapter 5 details the contributions made toward synthetic image generation for deep learning models. We found that, for this image segmentation problem, synthetic images were an effective technique for augmenting limited sets of real training data. We observed that models trained on purely synthetic images had a very low mean prediction IoU on real validation images, that adding even very small amounts of real images to a synthetic dataset greatly improved accuracy, and that models trained on datasets augmented with synthetic images were more accurate than those trained on real images alone. In use cases that benefit from incremental training or model specialization, we found that pretraining on synthetic images provided a usable base model for transfer learning. While we observed that models trained in a single session outperformed those pretrained on synthetic images and retrained on real data, we also saw that up to 90% of the total training time could be completed in the pretraining phase.

Publications of Jason Anderson

First author

- NetGames 2013 – Short Paper – *Towards a System for Controlling Client-Server Traffic in Virtual Worlds using SDN* [26]
- IEEE BigData 2014 – Short Paper – *Synthetic Data Generation for the Internet of Things* [27]
- ICNC 2016 – Full Paper – *Performance Considerations of Network Functions Virtualization using Containers* [25]
- AnNet 2017 – Workshop Paper – *Random Access in Nondelimited Variable-length Record Collections for Parallel Reading with Hadoop* [24]
- (in submission to AIJ) - Journal Article - *Synthetic Image Data for Deep Learning*

Co-author

- DIDC 2014 – Workshop Paper – *A Reconfigurable Network Testbed for Evaluation of Datacenter Topologies* [122]
- SC '16 – Poster – *Narrowing the Gap: Effects of Latency with Docker in IP Networks* [84]
- ICPE 2018 – Full Paper – *Measuring Network Latency Variation Impacts to High Performance Computing Application* [176]
- ICPE 2021 - Short Paper - *Viability of Azure IoT Hub for Processing High Velocity Large Scale IoT Data* [81]

Bibliography

[1] JDistLib, 2014. `http://jdistlib.sourceforge.net`.

[2] TPC Benchmarks, 2014. Available at `http://www.tpc.org/information/benchmarks.asp`.

[3] Azure IoT Hub pricing, 2019.

[4] Connecting IoT devices to Azure: IoT Hub and event hubs, 2019.

[5] Reference - IoT Hub quotas and throttling, 2019.

[6] TCPdump & libPCAP, 2019.

[7] Azure IoT Hub, 2020.

[8] Features and terminology in Azure event hubs, 2020.

[9] Palmetto Cluster documentation, 2020.

[10] NLANR/DAST : Iperf - the TCP/UDP bandwidth measurement tool. http://dast.nlanr.net/Projects/Iperf/, Accessed 2007.

[11] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[12] Ashraf Aboulnaga, Jeffrey F Naughton, and Chun Zhang. Generating synthetic complex-structured XML data. In *Procs. of WebDB*, 2001.

[13] John M. Abowd and Lars Vilhuber. How protective are synthetic data? In *International Conference on Privacy in Statistical Databases*, pages 239–246. Springer, 2008. ZSCC: 0000095.

[14] Charu Aggarwal, Naveen Ashish, and Amit Sheth. The Internet of Things: A survey from the data-centric perspective. In *Managing and mining sensor data*, pages 383–428. Springer, 2013.

[15] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 63–74. ACM, 2010.

[16] OpenFabrics Alliance. Openfabrics enterprise distribution, 2012.

[17] Moustafa Alzantot, Supriyo Chakraborty, and Mani Srivastava. Sensegen: A deep learning architecture for synthetic sensor data generation. In *International Conference on Pervasive Computing and Communications (PerCom) Workshops*, pages 188–193. IEEE, 2017.

[18] Amazon Web Services. Amazon EC2 FAQs, 2018.

[19] Amazon Web Services. Amazon EC2 instance types, 2018.

[20] Amazon Web Services. Amazon placement groups, 2018.

[21] Amazon Web Services. Amazon regions and availability zones, 2018.

[22] Amazon Web Services. Optimizing cpu options, 2018.

[23] Amazon Web Services. What is cloud computing? - amazon web services, 2018.

[24] Jason Anderson, Christopher Gropp, Linh Ngo, and Amy Apon. Random access in nondelimited variable-length record collections for parallel reading with hadoop. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 965–970. IEEE, 2017.

[25] Jason Anderson, Hongxin Hu, Udit Agarwal, Craig Lowery, Hongda Li, and Amy Apon. Performance considerations of network functions virtualization using containers. In *Computing, Networking and Communications (ICNC), 2016 International Conference on*, pages 1–7. IEEE, 2016.

[26] Jason Anderson and Jim Martin. Towards a system for controlling client-server traffic in virtual worlds using sdn. In *2013 12th Annual Workshop on Network and Systems Support for Games (NetGames)*, pages 1–2. IEEE, 2013.

[27] Jason W Anderson, Ken E Kennedy, Linh B Ngo, Andre Luckow, and Amy W Apon. Synthetic data generation for the internet of things. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 171–176. IEEE, 2014.

[28] Christian Arnold and Marcel Neunhoeffer. Really Useful Synthetic Data–A Framework to Evaluate the Quality of Differentially Private Synthetic Data. *arXiv preprint arXiv:2004.07740*, 2020. ZSCC: 0000004.

[29] Christian Arnold, Marcel Neunhoeffer, and Sebastian Sternberg. RELEASING DIFFERENTIALLY PRIVATE SYNTHETIC MICRO-DATA WITH BAYESIAN GANS. 2018. ZSCC: 0000000.

[30] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer networks*, 54, 2010.

[31] Laura Aviñó, Matteo Ruffini, and Ricard Gavaldà. Generating synthetic but plausible healthcare record datasets. *arXiv preprint arXiv:1807.01514*, 2018. ZSCC: 0000018.

[32] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The nas parallel benchmarks. *Int. J. High Perform. Comput. Appl.*, 5(3):63–73, September 1991.

[33] Andrew Banks and Rahul Gupta. MQTT version 3.1.1. 29, 2014.

[34] Anat Reiner Benaim, Ronit Almog, Yuri Gorelik, Irit Hochberg, Laila Nassar, Tanya Mashiach, Mogher Khamaisi, Yael Lurie, Zaher S. Azzam, and Johad Khoury. Analyzing medical research results based on synthetic data and their relation to real data results: systematic comparison from five observational studies. *JMIR medical informatics*, 8(2):e16492, 2020. ZSCC: NoCitationData[s0] Publisher: JMIR Publications Inc., Toronto, Canada.

[35] Nicole Berdy. IoT Hub throttling and you, 2016.

[36] Alan Mark Berg, Stefan T Mol, Gábor Kismihók, and Niall Sclater. The role of a reference synthetic data generator within the field of learning analytics. *Journal of Learning Analytics*, 3(1):107–128, 2016.

[37] Abhinav Bhatelé and Laxmikant V Kalé. Quantifying network contention on large parallel machines. *Parallel Processing Letters*, 19(04):553–572, 2009.

[38] Abhinav Bhatelé, Kathryn Mohror, Steven H Langer, and Katherine E Isaacs. There goes the neighborhood: performance degradation due to nearby jobs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 41. ACM, 2013.

[39] Vincent Bindschaedler, Reza Shokri, and Carl A. Gunter. Plausible deniability for privacy-preserving data synthesis. *arXiv preprint arXiv:1708.07975*, 2017. ZSCC: 0000113.

[40] Mark S Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D Underwood, and Robert C Zak. Intel® omni-path architecture: Enabling scalable, high performance fabrics. In *High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on*, pages 1–9. IEEE, 2015.

[41] Spyros Blanas, Jignesh Patel, Vuk Ercegovac, Jun Rao, Eugene Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in MapReduce. In *Procs. of ACM SIGMOD*, 2010.

[42] Nanette J Boden, Danny Cohen, Robert E Felderman, Alan E. Kulawik, Charles L Seitz, Jakov N Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[43] Verónica Bolón-Canedo, Noelia Sánchez-Maroño, and Amparo Alonso-Betanzos. A review of feature selection methods on synthetic data. *Knowledge and information systems*, 34(3):483–519, 2013. ZSCC: 0000621 Publisher: Springer.

[44] Verónica Bolón-Canedo, Noelia Sánchez-Maroño, and Amparo Alonso-Betanzos. Recent advances and emerging challenges of feature selection in the context of big data. *Knowledge-based systems*, 86:33–45, 2015. ZSCC: 0000240 Publisher: Elsevier.

[45] Joao Borrego, Atabak Dehban, Rui Figueiredo, Plinio Moreno, Alexandre Bernardino, and José Santos-Victor. Applying domain randomization to synthetic data for object category detection. *arXiv preprint arXiv:1807.09834*, 2018. 00013.

[46] Claire McKay Bowen and Joshua Snoke. Comparative study of differentially private synthetic data algorithms from the NIST PSCR differential privacy synthetic data challenge. *arXiv preprint arXiv:1911.12704*, 2019. ZSCC: 0000006.

[47] Scott Bradner and Jim McQuaid. Benchmarking methodology for network interconnect devices. Technical report, 1999.

[48] Anna L. Buczak, Steven Babin, and Linda Moniz. Data-driven approach for creating synthetic electronic medical records. *BMC medical informatics and decision making*, 10(1):1–28, 2010. Publisher: BioMed Central.

[49] Elvijs Buls, Roberts Kadikis, Ričards Cacurs, and Jānis Ārents. Generation of synthetic training data for object detection in piles. In *Eleventh International Conference on Machine Vision (ICMV 2018)*, volume 11041, page 110411Z. International Society for Optics and Photonics, 2019.

[50] Jan Pablo Burgard, Jan-Philipp Kolb, Hariolf Merkle, and Ralf Münnich. Synthetic data for open and reproducible methodological research in social sciences and official statistics. *AStA Wirtschafts-und Sozialstatistisches Archiv*, 11(3):233–244, 2017. Publisher: Springer.

[51] Rajkumar Buyya, Saurabh Kumar Garg, and Rodrigo N Calheiros. SLA-oriented resource provisioning for cloud computing: Challenges, architecture, and solutions. In *Procs. of the Int. Conf. on Cloud and Service Computing*, 2011.

[52] Gregory Caiola and Jerome P Reiter. Random forests for generating partially synthetic, categorical data. *Trans. Data Priv.*, 3(1):27–42, 2010.

[53] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. *SIGCOMM Computer Communication Review*, 37(4):1–12, 2007.

[54] Anne-Sophie Charest. How can we analyze differentially-private synthetic datasets? *Journal of Privacy and Confidentiality*, 2(2), 2011. ZSCC: 0000061.

[55] Kristina Chodorow. *MongoDB: the definitive guide*. O'Reilly Media, Inc., 2013.

[56] Edward Choi, Siddharth Biswal, Bradley Malin, Jon Duke, Walter F Stewart, and Jimeng Sun. Generating multi-label discrete patient records using generative adversarial networks. In *Machine learning for healthcare conference*, pages 286–305. PMLR, 2017.

[57] François Chollet et al. Keras. `https://keras.io`, 2015.

[58] Sara Cohen. Generating XML structure using examples and constraints. In *Procs. of the VLDB Endowment*, 2008.

[59] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Procs. of the ACM SoCC*, 2010.

[60] Joshua Cooper, Anne James, et al. Challenges for database management in the Internet of Things. *IETE Technical Review*, 26(5), 2009.

[61] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. Logp: Towards a realistic model of parallel computation. In *ACM Sigplan Notices*, volume 28, pages 1–12. ACM, 1993.

[62] Daniel Bristot de Oliveira. [rfc] workqueue: avoiding unbounded wq on isolated cpus by default, 2015.

[63] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. 00000.

[64] Jörg Drechsler. Using support vector machines for generating synthetic datasets. In *International Conference on Privacy in Statistical Databases*, pages 148–161. Springer, 2010.

[65] Jörg Drechsler. *Synthetic datasets for statistical disclosure control: theory and implementation*, volume 201. Springer Science & Business Media, 2011.

[66] Debidatta Dwibedi, Ishan Misra, and Martial Hebert. Cut, paste and learn: Surprisingly easy synthesis for instance detection. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1301–1310, 2017. 00164.

[67] Hadi Keivan Ekbatani, Oriol Pujol, and Santi Segui. Synthetic Data Generation for Deep Learning in Counting Pedestrians. In *ICPRAM*, pages 318–323, 2017.

[68] ETSI. Network Functions Virtualisation (NFV): Service Quality Metrics. *ETSI GS NFV-INF 010*, 2014. Available at http://www.etsi.org/deliver/etsi_gs/NFV-INF/001_099/010/01.01.01_60/gs_NFV-INF010v010101p.pdf.

[69] Roberto R Expósito, Guillermo L Taboada, Sabela Ramos, Juan Touriño, and Ramón Doallo. Performance analysis of hpc applications in the cloud. *Future Generation Computer Systems*, 29(1):218–229, 2013.

[70] John Fink. Docker: a software as a service, operating system-level virtualization framework. *Code4Lib Journal*, 25, 2014.

[71] Maayan Frid-Adar, Avi Ben-Cohen, Rula Amer, and Hayit Greenspan. Improving the segmentation of anatomical structures in chest radiographs using u-net with an imagenet pre-trained encoder. In *Image Analysis for Moving Organ, Breast, and Thoracic Images*, pages 159–168. Springer, 2018.

[72] Chrystel Gaber, Baptiste Hemery, Mohammed Achemlal, Marc Pasquet, and Pascal Urien. Synthetic logs generator for fraud detection in mobile transfer services. In *2013 International Conference on Collaboration Technologies and Systems (CTS)*, pages 174–179. IEEE, 2013.

[73] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. Bigbench: Towards an industry standard benchmark for big data analytics. In *Procs. of the ACM SIGMOD*, 2013.

[74] Google. Google compute engine faq, 2018.

[75] Google. Regions and zones, 2018.

[76] Google. Vpc resource quotas, 2018.

[77] Google. What is cloud computing? — google cloud, 2018.

[78] Dominique Guinard, Vlad Trifa, Friedemann Mattern, and Erik Wilde. From the Internet of Things to the Web of Things: Resource-oriented architecture and best practices. In *Architecting the Internet of Things*, pages 97–129. Springer, 2011.

[79] Ankush Gupta, Andrea Vedaldi, and Andrew Zisserman. Synthetic data for text localisation in natural images. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2315–2324, 2016. ZSCC: 0000933.

[80] Isabelle Guyon, Steve Gunn, Masoud Nikravesh, and Lofti A Zadeh. *Feature extraction: foundations and applications*, volume 207. Springer, 2008.

[81] Wajdi H. Halabi, Daniel N. Smith, John C. Hill, Jason W. Anderson, Ken E. Kennedy, Brandon M. Posey, Linh B. Ngo, and Amy W. Apon. Viability of azure iot hub for processing high velocity large scale iot data. In *Companion of the ACM/SPEC International Conference on Performance Engineering*, ICPE '21, page 73–76, New York, NY, USA, 2021. Association for Computing Machinery.

[82] Qiming He, Shujia Zhou, Ben Kobler, Dan Duffy, and Tom McGlynn. Case study for running hpc applications in public clouds. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 395–401. ACM, 2010.

[83] Joshua Higgins, Violeta Holmes, and Colin Venters. Orchestrating docker containers in the hpc environment. In *International Conference on High Performance Computing*, pages 506–513. Springer, 2015.

[84] Corbin Higgs and Jason Anderson. Narrowing the gap: Effects of latency with docker in ip networks. In *The International Conference for High Performance Computing, Networking, Storage and Analysis, Student Poster*, number 2016, 2016.

[85] Stefan Hinterstoisser, Olivier Pauly, Hauke Heibel, Marek Martina, and Martin Bokeloh. An annotation saved is an annotation earned: using fully synthetic training for object detection. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*, October 2019. 00004.

[86] Tomáš Hodaň, Vibhav Vineet, Ran Gal, Emanuel Shalev, Jon Hanzelka, Treb Connell, Pedro Urbina, Sudipta N. Sinha, and Brian Guenter. Photorealistic image synthesis for object instance detection. In *2019 IEEE International Conference on Image Processing (ICIP)*, pages 66–70. IEEE, 2019. 00000.

[87] Torsten Hoefler, Lavinio Cerquetti, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. A practical approach to the rating of barrier algorithms using the logp model and open mpi. In *Parallel Processing, 2005. ICPP 2005 Workshops. International Conference Workshops on*, pages 562–569. IEEE, 2005.

[88] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018.

[89] Zengyi Huang and Paul Williamson. A comparison of synthetic reconstruction and combinatorial optimisation approaches to the creation of small-area microdata. *Department of Geography, University of Liverpool*, 2001.

[90] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. MQTT-S—A publish/subscribe protocol for wireless sensor networks. In *3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE)*, pages 791–798. IEEE, 2008.

[91] IBM. What is cloud computing?, 2018.

[92] Vladimir Iglovikov and Alexey Shvets. Ternausnet: U-net with vgg11 encoder pre-trained on imagenet for image segmentation. *arXiv preprint arXiv:1801.05746*, 2018.

[93] Tadanobu Inoue, Subhajit Choudhury, Giovanni De Magistris, and Sakyasingha Dasgupta. Transfer learning from synthetic to real images using variational autoencoders for precise position detection. In *2018 25th IEEE International Conference on Image Processing (ICIP)*, pages 2725–2729. IEEE, 2018. 00018.

[94] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[95] Keith R Jackson, Lavanya Ramakrishnan, Krishna Muriki, Shane Canon, Shreyas Cholia, John Shalf, Harvey J Wasserman, and Nicholas J Wright. Performance analysis of high performance computing applications on the amazon web services cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 159–168. IEEE, 2010.

[96] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM Computer Communication Review*, volume 18, pages 314–329. ACM, 1988.

[97] Max Jaderberg, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Synthetic data and artificial neural networks for natural scene text recognition. *arXiv preprint arXiv:1406.2227*, 2014. ZSCC: 0000712.

[98] Raj Jain and Shawn Routhier. Packet trains–measurements and a new model for computer network traffic. *IEEE journal on selected areas in Communications*, 4(6):986–995, 1986.

[99] Debesh Jha, Michael A. Riegler, Dag Johansen, P\aal Halvorsen, and H\aavard D. Johansen. DoubleU-Net: A Deep Convolutional Neural Network for Medical Image Segmentation. *arXiv preprint arXiv:2006.04868*, 2020. ZSCC: 0000003.

[100] You-Cyuan Jhang, Adam Palmar, Bowen Li, Saurav Dhakad, Sanjay Kumar Vishwakarma, Jonathan Hogins, Adam Crespi, Chris Kerr, Sharmila Chockalingam, Cesar Romero, Alex Thaman, and Sujoy Ganguly. Training a performant object detection ML model on synthetic data using Unity Perception tools, September 2020. 00000.

[101] Ana Jokanovic, Jose Carlos Sancho, German Rodriguez, Alejandro Lucero, Cyriel Minkenberg, and Jesus Labarta. Quiet neighborhoods: Key to protect job performance predictability. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 449–459. IEEE, 2015.

[102] Alexandros Kalousis, Julien Prados, and Melanie Hilario. Stability of feature selection algorithms: a study on high-dimensional spaces. *Knowledge and information systems*, 12(1):95–116, 2007.

[103] Leonard Kaufman and Peter J Rousseeuw. Partitioning around medoids (program pam). *Finding groups in data: an introduction to cluster analysis*, pages 68–125, 1990.

[104] Xiaohong Jiang Kejiang Ye, Dawei Huang Siding Chen, and Bei Wang. Analyzing & Modeling the Performance in Xen-based Virtual Cluster Environment. 2010.

[105] G. Maurice Kendall. *The Advanced Theory Of Statistics*, volume 1. Charles Griffin and Company Limited, 42 Drury Lane, London, 1948.

[106] Scott Klein. *IoT Solutions in Microsoft's Azure IoT Suite Data Acquisition and Analysis in the Real World*. Apress, 2017.

[107] Vipin Kumar and Sonajharia Minz. Feature selection: a literature review. *SmartCR*, 4(3):211–229, 2014. ZSCC: 0000395.

[108] Sébastien Lagarde, Sébastien Lachambre, and Cyril Jover. An artist-friendly workflow for panoramic HDRI. In *ACM SIGGRAPH 2016 Courses*, SIGGRAPH '16, New York, NY, USA, 2016. Association for Computing Machinery. 00000 event-place: Anaheim, California.

[109] J. Van Leeuwen. On the construction of Huffman trees. In *Proc. 3rd International Conference on Automata, Languages, and Programming*, Edinburgh University, 1976.

[110] Zhengqi Li and Noah Snavely. Cgintrinsics: Better intrinsic image decomposition through physically-based rendering. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 371–387, 2018. 00000.

[111] David G. Lowe. Three-dimensional object recognition from single two-dimensional images. *Artificial intelligence*, 31(3):355–395, 1987. 01904.

[112] Jorge E Luzuriaga, Miguel Perez, Pablo Boronat, Juan Carlos Cano, Carlos Calafate, and Pietro Manzoni. A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks. In *12th Annual Consumer Communications and Networking Conference (CCNC)*, pages 931–936. IEEE, 2015.

[113] Eve Maler, Jean Paoli, C. M. Sperberg-McQueen, François Yergeau, and Tim Bray. Extensible markup language (XML) 1.0 (third edition). first edition of a recommendation, W3C, 2004. http://www.w3.org/TR/2004/REC-xml-20040204.

[114] Edward Martin and Luc Vo Van. We have you covered with the Measured Materials library, February 2019. 00000.

[115] Richard P Martin, Amin M Vahdat, David E Culler, and Thomas E Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 85–97. ACM, 1997.

[116] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 459–473. USENIX Association, 2014.

[117] Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951.

[118] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[119] McKinsey Global Institute. Big data: The next frontier for innovation, competition, and productivity, 2011.

[120] Microsoft Azure. What is cloud computing? a beginner's guide — microsoft azure, 2018.

[121] Chaitanya Mitash, Kostas E. Bekris, and Abdeslam Boularias. A self-supervised learning system for object detection using physics simulation and multi-view pose estimation. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 545–551. IEEE, 2017. 00063.

[122] William Clay Moody, Jason Anderson, Kuang-Ching Wange, and Amy Apon. Reconfigurable network testbed for evaluation of datacenter topologies. In *Proceedings of the sixth international workshop on Data intensive distributed computing*, pages 11–20, 2014.

[123] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. *IEEE Symposium on Security and Privacy*, pages 111–125, 2008.

[124] A. Narayanan and V. Shmatikov. De-anonymizing social networks. *IEEE Symposium on Security and Privacy*, pages 173–187, 2009.

[125] Ramakant Nevatia and Thomas O. Binford. Description and recognition of curved objects. *Artificial intelligence*, 8(1):77–98, 1977. 00606 Publisher: Elsevier.

[126] D. Nguyen, A. Luckow, E. Duffy, K. Kennedy, and A. Apon. Evaluation of highly available cloud streaming systems for performance and price. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 360–363, 2018.

[127] Gavin Nicol, Lauren Wood, Mike Champion, and Steve Byrne. Document object model (dom) level 3 core specification, 2001.

[128] Gottfried E. Noether. *Elements of Nonparametric Statistics*. The SIAM Series in Applied Mathematics. John Wiley and Sons, Inc., New York, 1967.

[129] Beata Nowok, Gillian M. Raab, and Chris Dibben. synthpop: Bespoke creation of synthetic data in R. *Journal of statistical software*, 74(1):1–26, 2016. ZSCC: 0000153.

[130] Noseong Park, Mahmoud Mohammadi, Kshitij Gorde, Sushil Jajodia, Hongkyu Park, and Youngmin Kim. Data synthesis based on generative adversarial networks. *arXiv preprint arXiv:1806.03384*, 2018.

[131] Synthetic data. In Sybil P. Parker, editor, *McGraw-Hill Dictionary of Scientific and Technical Terms*. McGraw-Hill Education, 6th edition, 2003.

[132] Neha Patki, Roy Wedge, and Kalyan Veeramachaneni. The synthetic data vault. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 399–410. IEEE, 2016. ZSCC: 0000145.

[133] Xingchao Peng, Baochen Sun, Karim Ali, and Kate Saenko. Learning deep object detectors from 3d models. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1278–1286, 2015. 00274.

[134] María Pérez, Ismael Sanz, and Rafael Berlanga. Xtage: a flexible XML collection generator. In *Procs. of the ACM SIGMOD*, 2010.

[135] Fabrizio Petrini, Darren J Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: achieving optimal performance on the 8,192 processors of asci q. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 55–55. IEEE, 2003.

[136] Gregory F Pfister. An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001.

[137] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of computational physics*, 117(1):1–19, 1995.

[138] Aayush Prakash, Shaad Boochoon, Mark Brophy, David Acuna, Eric Cameracci, Gavriel State, Omer Shapira, and Stan Birchfield. Structured domain randomization: Bridging the reality gap by context-aware synthetic data. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 7249–7255. IEEE, 2019. 00036.

[139] Gillian M. Raab, Beata Nowok, and Chris Dibben. Practical data synthesis for large samples. *Journal of Privacy and Confidentiality*, 7(3):67–97, 2016. ZSCC: 0000040.

[140] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[141] Trivellore E Raghunathan, Jerome P Reiter, and Donald B Rubin. Multiple imputation for statistical disclosure limitation. *Journal of official statistics*, 19(1):1, 2003.

[142] Jerome P. Reiter. Satisfying disclosure restrictions with synthetic data sets. *Journal of Official Statistics*, 18(4):531, 2002. ZSCC: 0000210 Publisher: Statistics Sweden (SCB).

[143] Jerome P Reiter. Inference for partially synthetic, public use microdata sets. *Survey Methodology*, 29(2):181–188, 2003.

[144] Jerome P Reiter and Satkartar K Kinney. Inferentially valid, partially synthetic data: Generating from posterior predictive distributions not necessary. *Journal of Official Statistics*, 28(4):583, 2012.

[145] Jerome P. Reiter, Quanli Wang, and Biyuan Zhang. Bayesian estimation of disclosure risks for multiply imputed, synthetic data. *Journal of Privacy and Confidentiality*, 6(1), 2014. ZSCC: 0000055.

[146] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *IEEE transactions on pattern analysis and machine intelligence*, 39(6):1137–1149, 2016. 23133 Publisher: IEEE.

[147] Robert Ricci and Eric Eide. Introducing cloudlab: Scientific infrastructure for advancing cloud architecturesand applications. *; login:*, 39(6):36–38, 2014.

[148] Stephan R. Richter, Vibhav Vineet, Stefan Roth, and Vladlen Koltun. Playing for data: Ground truth from computer games. In *European conference on computer vision*, pages 102–118. Springer, 2016. 00781.

[149] George F Riley and Thomas R Henderson. The ns-3 network simulator. In *Modeling and Tools for Network Simulation*, pages 15–34. Springer, 2010.

[150] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015. 00000.

[151] Donald B Rubin. Statistical disclosure limitation. *Journal of official Statistics*, 9(2):461–468, 1993.

[152] Cristian Ruiz, Emmanuel Jeanvoine, and Lucas Nussbaum. Performance evaluation of containers for hpc. In *European Conference on Parallel Processing*, pages 813–824. Springer, 2015.

[153] Stephen M Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K Ousterhout. It's time for low latency. In *HotOS*, volume 13, pages 11–11, 2011.

[154] Yunus Saatci and Andrew Wilson. Bayesian gans. In *Advances in neural information processing systems*, pages 3624–3633, 2017.

[155] Connor Shorten and Taghi M. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):1–48, 2019. ZSCC: 0002070 Publisher: Springer.

[156] Ashish Shrivastava, Tomas Pfister, Oncel Tuzel, Joshua Susskind, Wenda Wang, and Russell Webb. Learning from simulated and unsupervised images through adversarial training. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2107–2116, 2017. ZSCC: 0001519.

[157] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 00000.

[158] Dennis Smith, Lydia Leong, and Raj Bala. Magic quadrant for cloud infrastructure as a service, worldwide, May 2018.

[159] Joshua Snoke, Gillian M. Raab, Beata Nowok, Chris Dibben, and Aleksandra Slavkovic. General and specific utility measures for synthetic data. *Journal of the Royal Statistical Society: Series A (Statistics in Society)*, 181(3):663–688, 2018. ZSCC: 0000070 Publisher: Wiley Online Library.

[160] Ahmed Sobeih, Jennifer C Hou, Lu-Chuan Kung, Ning Li, Honghai Zhang, Wei-Peng Chen, Hung-Ying Tyan, and Hyuk Lim. J-Sim: a simulation and emulation environment for wireless sensor networks. *Wireless Communications*, 13(4):104–119, 2006.

[161] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.

[162] Neil Spring, Larry Peterson, Andy Bavier, and Vivek Pai. Using planetlab for network research: myths, realities, and best practices. *ACM SIGOPS Operating Systems Review*, 40(1):17–24, 2006.

[163] Aishwarya Srivastava, Siddhant Aggarwal, Amy Apon, Edward Duffy, Ken Kennedy, Andre Luckow, Brandon Posey, and Marcin Ziolkowski. Deployment of a cloud pipeline for real-time visual inspection using fast streaming high-definition images. *Software: Practice and Experience*, 50(6):868–898, 2020.

[164] Hao Su, Charles R. Qi, Yangyan Li, and Leonidas J. Guibas. Render for cnn: Viewpoint estimation in images using cnns trained with rendered 3d model views. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2686–2694, 2015. 00567.

[165] Vedaprakash Subramanian, Liqiang Wang, En-Jui Lee, and Po Chen. Rapid processing of synthetic seismograms using Windows Azure cloud. In *2nd International Conference on Cloud Computing Technology and Science*, pages 193–200. IEEE, 2010.

[166] Baochen Sun and Kate Saenko. From Virtual to Reality: Fast Adaptation of Virtual Object Detectors to Real Domains. In *BMVC*, volume 1, page 3, 2014. 00150 Issue: 2.

[167] Latanya Sweeney. K-anonymity: A model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5), 2002.

[168] Dinesh Thangavel, Xiaoping Ma, Alvin Valera, Hwee-Xian Tan, and Colin Keng-Yan Tan. Performance evaluation of MQTT and CoAP via a common middleware. In *9th International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, pages 1–6. IEEE, 2014.

[169] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 23–30. IEEE, 2017. 00851.

[170] Joey Brakefield Todd Whitehurst, Steve Busby. Private Communication, July 2020.

[171] Milos Todic and Branislav Uzelac. Combined XML/XQuery generator. In *Procs. of the International Workshop on Testing Database Systems*, 2012.

[172] Jonathan Tremblay, Aayush Prakash, David Acuna, Mark Brophy, Varun Jampani, Cem Anil, Thang To, Eric Cameracci, Shaad Boochoon, and Stan Birchfield. Training deep networks with synthetic data: Bridging the reality gap by domain randomization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 969–977, 2018. 00256.

[173] Jonathan Tremblay, Thang To, Balakumar Sundaralingam, Yu Xiang, Dieter Fox, and Stan Birchfield. Deep object pose estimation for semantic robotic grasping of household objects. *arXiv preprint arXiv:1809.10790*, 2018. 00168.

[174] Jonathan Turner. New directions in communications (or which way to the information age?). *IEEE communications Magazine*, 24(10):8–15, 1986.

[175] Peter Turney. Bias and the quantification of stability. *Machine Learning*, 20(1):23–33, 1995.

[176] Robert Underwood, Jason Anderson, and Amy Apon. Measuring network latency variation impacts to high performance computing application performance. In *Performance Engineering (ICPE), 2018 International Conference on*, pages 1–12. ACM/SPEC, 2018.

[177] Tim Van den Bulcke, Koenraad Van Leemput, Bart Naudts, Piet van Remortel, Hongwu Ma, Alain Verschoren, Bart De Moor, and Kathleen Marchal. SynTReN: a generator of synthetic gene expression data for design and analysis of structure learning algorithms. *BMC bioinformatics*, 7(1):1–12, 2006. Publisher: Springer.

[178] András Varga and Rudolf Hornig. An overview of the OMNeT++ simulation environment. In *1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems*, page 60. ICST, 2008.

[179] W. Venables and B. Ripley. *Modern Applied Statistics with S*. Springer, 4 edition, 2003.

[180] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[181] Guohui Wang and TS Eugene Ng. The impact of virtualization on network performance of amazon ec2 data center. In *Infocom, 2010 proceedings ieee*, pages 1–9. IEEE, 2010.

[182] Qi Wang, Junyu Gao, Wei Lin, and Yuan Yuan. Learning from synthetic data for crowd counting in the wild. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8198–8207, 2019. ZSCC: 0000236.

[183] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. High-resolution image synthesis and semantic manipulation with conditional gans. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8798–8807, 2018. ZSCC: 0002070.

[184] Tom White. *Hadoop: The Definitive Guide*. O'Reilly, 2012.

[185] Jon Whiteaker, Fabian Schneider, and Renata Teixeira. Explaining packet delays under virtualization. *ACM SIGCOMM Computer Communication Review*, 41(1):38–44, 2011.

[186] Miguel G Xavier, Marcelo V Neves, Fabio D Rossi, Tiago C Ferreto, Timoteo Lange, and Cesar AF De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 233–240. IEEE, 2013.

[187] Lei Xu and Kalyan Veeramachaneni. Synthesizing tabular data using generative adversarial networks. *arXiv preprint arXiv:1811.11264*, 2018. ZSCC: 0000052.

[188] Andrew Yale, Saloni Dash, Ritik Dutta, Isabelle Guyon, Adrien Pavao, and Kristin Bennett. Privacy preserving synthetic health data. In *ESANN 2019-European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, 2019. ZSCC: 0000016.

[189] Mengyuan Yan, Iuri Frosio, Stephen Tyree, and Jan Kautz. Sim-to-real transfer of accurate grasping with eye-in-hand observations and continuous control. *arXiv preprint arXiv:1712.03303*, 2017.

[190] Joris Toonders Yonego. Data is the new oil of the digital economy, Jul 2014.

[191] Fangyi Zhang, Jürgen Leitner, Michael Milford, and Peter Corke. Sim-to-real transfer of visuomotor policies for reaching in clutter: Domain randomization and adaptation with modular networks. *world*, 7(8), 2017.

[192] Lixia Zhang, Scott Shenker, and Daivd D Clark. Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic. *ACM SIGCOMM Computer Communication Review*, 21(4):133–147, 1991.

[193] Yinda Zhang, Shuran Song, Ersin Yumer, Manolis Savva, Joon-Young Lee, Hailin Jin, and Thomas Funkhouser. Physically-based rendering for indoor scene understanding using convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5287–5295, 2017. 00000.

[194] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2223–2232, 2017. ZSCC: 0010213.