Workload Characterization for Microservices

Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara
IBM Research - Tokyo
IBM Japan, Ltd.
Tokyo, Japan
{tkueda, nakaike, ohara}@jp.ibm.com

Abstract— **The microservice architecture is a new framework to construct a Web service as a collection of small services that communicate with each other. It is becoming increasingly popular because it can accelerate agile software development, deployment, and operation practices. As a result, cloud service providers are expected to host an increasing number of microservices that can generate significant resource pressure on the cloud infrastructure. We want to understand the characteristics of microservice workloads to design an infrastructure optimized for microservices. In this paper, we used Acme Air, an open-source benchmark for Web services, and analyzed the behavior of two versions of the benchmark, microservice and monolithic, for two widely used language runtimes, Node.js and Java. We observed a significant overhead due to the microservice architecture; the performance of the microservice version can be 79.2% lower than the monolithic version on the same hardware configuration. On Node.js, the microservice version consumed 4.22 times more time in the libraries of Node.js than the monolithic version to process one user request. On Java, the microservice version also consumed more time in the application server than the monolithic version. We explain these performance differences from both hardware and software perspectives. We discuss the network virtualization in Docker, an infrastructure for microservices that has non-negligible impact on performance. These findings give clues to develop optimization techniques in a language runtime and hardware for microservice workloads.**

Keywords— microservices; microservice architecture; Node.js; WebSphere Liberty; Java; Docker; container

I. INTRODUCTION

As an increasing number of companies provide their services through the cloud, the development speed of Web services has a direct impact on business outcomes. Developers use agile development [16] with DevOps to accelerate Web-service deliverables. With the widespread use of agile development, the microservice architecture proposed by James Lewis and Martin Fowler [17] is attracting much attention. The microservice architecture is a design methodology for building a Web service with *"a suite of small services"*. Each service communicates with other services *"with lightweight mechanisms, often an HTTP resource API"*. A recent common style in microservice development involves a developer building a Web service as a Docker container image on his/her laptop then transfers the image to a production cloud. A Docker container image includes all the libraries that the service needs to run. With the isolation mechanism of the container technology, developers can avoid conflicts among the runtime environments, even when they deploy multiple microservices on the same machine. Furthermore, the microservice architecture enables developers to continuously update an Web service without shutting down the entire service. Web-service developers are eager to adopt the microservice architecture to accelerate agile software development and DevOps.

Microservices running in containers are expected to generate more pressure on computer systems than traditional monolithic service running as a native process. It is easy to imagine that microservices consume CPU cycles to communicate with each other through API calls. The Docker container technology relies on the functions of operating systems, such as cgroups and iptables, to isolate each container from other containers. Thus, microservices generate more pressure on the operating-system layer. To make matters worse, it is becoming a common practice to control scale-up or scale-out performance by changing the number of containers with orchestration tools such as Swarm and Kubernetes. For fine grained control, a container should include only one process, even though a container can maintain multiple processes, which results in increasing the number of containers. If the language runtime, such as Node.js, adopts a single-thread processing model, we need multiple containers to use multiple CPU cores for processing incoming requests. The increased number of containers running on an operating system is expected to increase the impact on performance.

Efficiently executing microservices is important to both developers and cloud vendors. Developers need to carefully consider the impact on performance before transforming a monolithic service into microservices. With the changing trend in Web-service development, cloud vendors need to host a large number of containers to run microservices. Thus, cloud venders need to investigate the optimization opportunities in microservices to build efficient software and hardware platforms for cloud services. It is also important for both cloud vendors and developers to understand microservice behaviors when we collocate containers into one physical machine because container collocation is effective in reducing cloud-operation cost. Villamizar et al. evaluated microservices on a cloud [3][4]. They reported the performance differences between monolithic services and microservices. However, they did not provide detailed analysis of runtime and hardware layers that resulted in these differences. There has been no empirical study on the performance analysis of microservices.

978-1-5090-3896-1/16/$31.00 ©2016 IEEE

In this paper, we used a public benchmark, Acme Air [9]-[12], that is implemented in two different service architectures, monolithic service and microservice, and in two different languages, Node.js and Java. The implementations in the two different languages help us cover both single-thread and multi-thread processing models. We analyzed the performance differences from the aspects of hardware and language runtime levels. We believe this is the first study that analyzes microservices. Even though we only focused on Acme Air, the results offer generally applicable insights for optimizing microservices. We explain the following interesting points from our measurement results:

- *Noticeable impact on performance of microservices* - We observed that developers needed to pay up to 79.2% performance penalty in Node.js and 70.2% in Java to transform the Acme Air monolithic version into the microservice version. This is a larger performance penalty than previously reported [3]. Developers should carefully consider this fact when transforming a monolithic service into microservices.
- *Frequent CPU data cache misses caused by large number of containers* - The number of containers increases when using the microservice architecture. The system in our experiments on microservices had 68 containers with Node.js on 16 CPU cores. The number of entries in iptables increased as the number of containers increased. Regarding communication, checking the access control lists that have a large number of entries is necessary. As a result, microservices caused a large number of cache misses, which degraded performance. This result explains an optimization opportunity of the operating-system layer.
- *Primary components of Node.js and Java that caused throughput differences between microservices and monolithic service* - We analyzed the profiles of both Node.js and Java to investigate which part of the runtimes had an impact on the throughput differences. We found the libraries for HTTP communication consumed much time. Network communication is the key source of performance degradation. This result explains the opportunities of communication optimization in the language runtimes and microservices.
- *Non-negligible impact on performance of network virtualization* - Docker supports multiple network virtualization configurations. The bridge network can provide full virtualization. However, it degraded the throughput up to 33.8%. Developers should carefully choose the network virtualization policy by considering application requirements.

The rest of the paper is structured as follows. Section II explains the background and reason we conducted this research. Section III describes the experimental environment for this research. Section IV summarizes the measurement results. Section V analyzes the measurement results based on code path lengths and CPI numbers. Section VI explains the performance analysis from the aspects of language runtime levels. Section VII discusses previous work and our contributions. Section VIII concludes this paper.

II. BACKGROUND

The microservice architecture is a new design style in Web service development. This section explains the background of microservices with container technology and language runtimes.

A. Microservices

James Lewis and Martin Fowler proposed microservices in their blog [17]. They define that a microservice is *"a particular way of designing software applications as suites of independently deployable services"*. Each microservice runs in its own process and communicates with others with lightweight mechanisms, often HTTP resource APIs. Developers can update a part of an Web service without shutting down the entire service. This property is the key requirement for continuous integration in agile development. Since the microservice architecture is a worldwide development trend, cloud vendors cannot escape from hosting a large number of microservices.

Note that Lewis and Fowler found a similarity between service oriented architecture (SOA) and the microservice architecture [17]. It is our understanding that microservices do not use a common communication bus such as the enterprise service bus (ESB), which is the key mechanism of SOA. In the microservice architecture, each service communicates with other services directly by using HTTP or any other standard Web communication protocol. In addition, developers tend to use Docker containers to build a Web service in the microservice architecture. Thus, we believe the microservice architecture creates new performance behaviors that we have not seen in SOA.

B. Container Technology for Microservices

Docker [15] provides operating-system-level virtualization. Each container is isolated from other containers by operating-system mechanisms such as cgroups and iptables. A process in a container runs as a native process of the host operating system. The performance of the process running in a container should be almost identical with that of the native process running without Docker. Docker provides two types of network configurations, host network and bridge network. The host network configuration exports the network interfaces of the host operating system to the process in a Docker container. The Docker bridge configuration exports a virtualized network interface to a container that is connected to a private network segment. Docker relies on iptables to transfer packets among the virtualized interfaces of containers and other physical networks.

With the isolation mechanism, Docker plays an important role in Web-service development. One common example of agile development is prototyping an Web service with a preferable environment, typically a laptop, then transferring the image of the development to a production cloud. Developers can consider a Docker image as an executable file, and a container as a native process. Docker can guarantee the

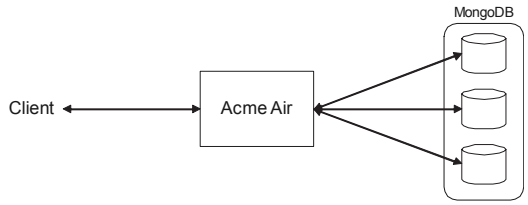

Fig. 1. Acme Air monolithic version. It uses single monolithic service to run the Web service. We used MongoDB as data store for benchmark.

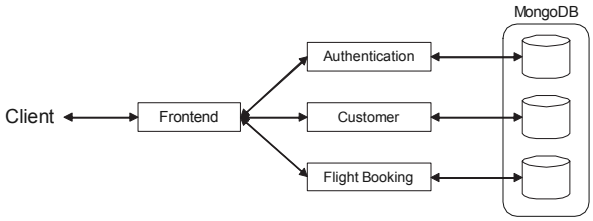

Fig. 2. Acme Air microservice version. It consists of four microservices. We also used MongoDB as data store for benchmark. Each microservice runs in its own process.

portability between a developer's laptop and production cloud with the container mechanism that duplicates all the system configurations including standard libraries. If a developer uses the bridge network, he/she can use any port numbers when implementing a service to run in a container because the virtualization mechanism avoids conflicts of port numbers among the services running in other containers. The container technology ensures that each process runs in an isolated software environment in which the service is developed.

Web service developers actively adopt microservices with container technology to accelerate agile development. The microservice architecture generates a larger number of containers to provide a Web service than that of the monolithic architecture. Thus, the network virtualization mechanisms generate pressure on software and hardware. For example, we can anticipate that the bridge network impacts performance. On the contrary, we can anticipate that network performance should not degrade if we use host configuration. Cloud vendors need to host an increasing number of microservices; therefore, we need to understand the overhead of the container technology.

C. Node.js and Java Application Server for Microservices

Node.js is quite popular in developing Web services. It adopts an event-based programming model. Developers can implement a Web service with third-party libraries by implementing callback functions. Node.js processes incoming requests asynchronously. In other words, it always has a runnable thread to process incoming requests. On the contrary, Java-based application servers, such as WebSphere, adopt a thread-based processing model. If all the threads in a thread pool wait for I/O completions, the application server cannot process new client requests. The single-thread processing model of Node.js is the key advantage over thread-based processing models such as Java. Paypal reported that Node.js can perform better than a Java-based application server [18].

However, Node.js requires multiple processes to achieve parallelized performance because a Node.js process can use only one CPU core due to the single-thread model processing. It is becoming common to control scale-up or scale-out performance by changing the number of containers. This means we need to use multiple Docker containers for providing scale-up or scale-out performance in microservices. As a result, microservices may generate more resource pressure on a machine. On the contrary, a Java-based application server can exploit multiple CPU cores with a single Docker container because of the multi-thread processing model.

III. Measurement Methodology

We used Acme Air [9]-[12], an open-source benchmark that emulates transactional workloads for a fictitious airline's Web site. The projects in GitHub [9][10] and forked projects [11][12] provide the implementations of the monolithic and microservice architectures and of the different languages, Node.js and Java. We collected the performance data by using Linux operating systems running on a single computer.

A. Acme Air Benchmark

Acme Air was originally developed as a monolithic service [9][10]. The projects in GitHub provide the Java [9] and Node.js [10] implementations. In the forked projects in GitHub [11][12], the developers transformed each monolithic implementation of the two different languages into a microservice implementation. Thus, by using the Acme Air benchmark, we can collect performance data of four types of implementations: Node.js monolithic service, Node.js microservice, Java monolithic service, and Java microservice. The Acme Air project provides an Apache JMeter [14] scenario as the default workload [13]. During the workload, each client logs in the Web site before making reservation requests. Then, each client searches an available flight between two cities on a preferable date and time. After that, each client books a flight.

The Acme Air monolithic version shown in Fig. 1 runs the Web service with a monolithic service. Every function of the Acme Air is implemented in the monolithic service. We used a MongoDB instance as a data store. As shown in Fig. 2, the microservice version divides the monolithic service into the following four microservices. The frontend service interacts with the clients. It renders the Web user interface and receives the requests from the clients. This service communicates with the other three microservices by using representational state transfer (REST) interfaces to perform authentications, flight searches, bookings, and cancelations. The authentication service performs user authentication and manages Web sessions. The flight-booking service handles the flight search requests from the clients and books the flights. The customer service manages information associated with each client including booked flights and user information.

B. Software and Hardware Environment

We used Node.js 4.2.2 and IBM Java 1.8 as the language runtimes. We configured each runtime to have a 1GB heap memory space. We used Docker 1.9.1 for the measurements, MongoDB 3.2.0 as the database, and Apache JMeter 2.13 [14] as a workload driver. We conducted Java experiments using

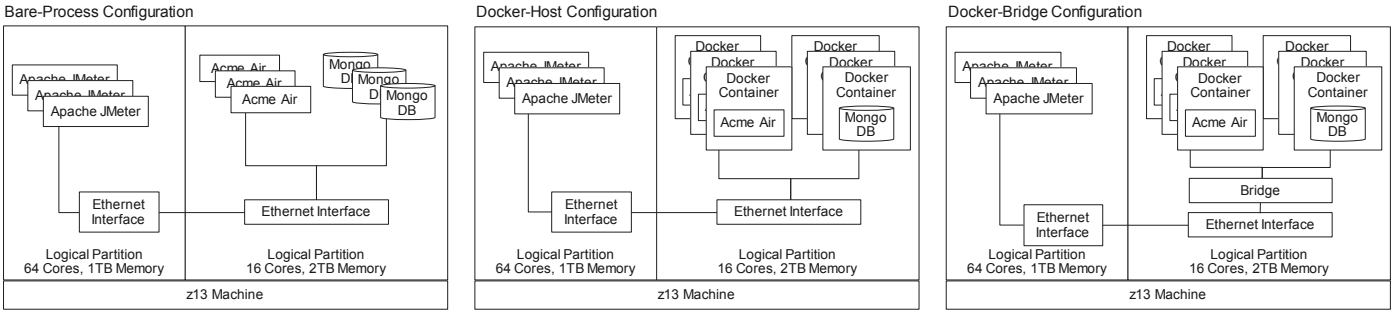

Fig. 3. Three experimental configurations. Bare process does not involve Docker containers. Docker host uses Docker containers connected to network interface of host operating system. Docker bridge uses Docker containers connected to virtualized network bridge.

IBM WebSphere Application Server Liberty Core 8.5.5.6 as an application server.

We used a single machine consisting of IBM z13 processors to evaluate the scale-up performance of the Acme Air implementations. The machine has a logical-partitioning mechanism (LPAR) to isolate multiple operating systems. As shown in Fig. 3, we used two operating systems (SUSE Linux Enterprise Server 12 SP1) running in two different logical partitions on the machine. We assigned enough memory for the two logical partitions. The Acme Air Web service and MongoDB run in a logical partition with 16 CPU cores and 2TB memory. Apache JMeter, the workload driver, runs on another logical partition with 64 CPU cores with 1TB memory. The impact on performance due to the collocation of the two partitions is minimal because of the highly optimized partitioning mechanism.

C. Experimental Configurations

We used the three experimental configurations shown in Fig. 3 for measuring the performance of each implementation of Acme Air. For the bare-process configuration, we measured performance when Acme Air ran as native processes outside the Docker containers. We also measured performance with the two different types of Docker network configurations, host and bridge. The host configuration allows the process running inside Docker containers to use the network interfaces of the host operating system. In the bridge configuration, the network interface in each container is connected to the virtual Ethernet bridge. The virtual Ethernet bridge is connected to the physical network of the host machine by using iptabes.

We collected the performance data on 1, 4, and 16 cores. For each number of CPU cores, we collected the performance data of the three configurations by using the monolithic and microservice architectures. We also measured performance on Node.js and Java. Thus, we had a total of 36 measurement configurations.

We made duplications of an Acme Air benchmark instance to emulate multiple tenants. In the configurations on the single and four cores, we used one Acme Air tenant. In the configurations on the 16 cores, we used 4 Acme Air tenants. We conducted Node.js experiments and assigned four Node.js processes for each Acme Air tenant because Node.js requires multiple processes to exploit multiple CPU cores. In this configuration, the system with 16 cores used 16 Node.js processes for the 4 Acme Air tenants in the benchmarks with the monolithic implementation. We chose this configuration so that the total number of Node.js processes in the monolithic configurations would be the same as the number of CPU cores. In the benchmarks with the Java monolithic implementation, we used the same number of Liberty processes as that of Acme Air tenants. We configured each Liberty process to have 100 threads in the thread pool for processing incoming requests.

IV. Performance Overview

Fig. 4 shows the throughput ratios based on the throughput of the Node.js monolithic implementation with the bare-process configuration running on the single CPU core. Fig. 5 shows the scalability ratios based on the single-core performance of each configuration. In comparison with the monolithic implementations, the Node.js microservice implementation degraded throughput up to 79.2% and the Java microservice implementation degraded throughput up to 70.2%. This is a noticeably worse performance penalty than previously reported [3]. From this fact, we argue that Web-service developers should carefully consider the impact on performance before transforming a monolithic implementation into a microservice implementation.

The Docker network configurations exhibited non-negligible impact on performance. The bridge network that Docker uses as default exhibited up to 33.8% performance degradation in the Node.js implementations compared to the bare-process configuration. As expected, the Docker-host configuration always exhibited better performance than the Docker-bridge configuration because Docker host uses the interface of the host machine without virtualization. However, when we use the host interface, applications may cause a conflict of network ports if they use the same port. From this performance trend, we argue that developers should select an appropriate Docker network configuration depending on whether the developers have to avoid port conflicts.

There are other interesting performance results. Java exhibited better performance than Node.js on the 4 and 16 cores, except the 16-core bare-process configuration. Java exhibited super-linear scalability on the four cores. On the contrary, Node.js outperformed Java on the single core.

In the following sections, we investigate why each of these performance gaps occurred from the hardware- and language-runtime-level perspectives.

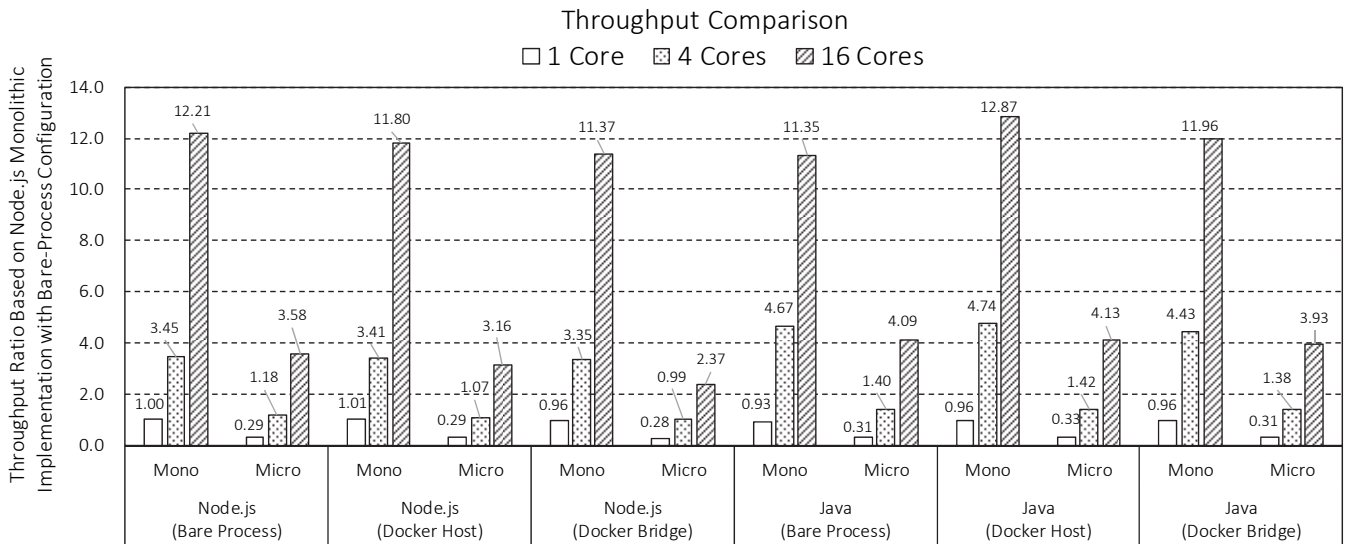

Fig. 4. Throughput comparison among Node.js and Java implementations with Bare-process, Docker-host, and Docker-bridge experimental configurations. Values are relative ratios of throughput based on throughput of Node.js monolithic implementation with bare-process configuration.

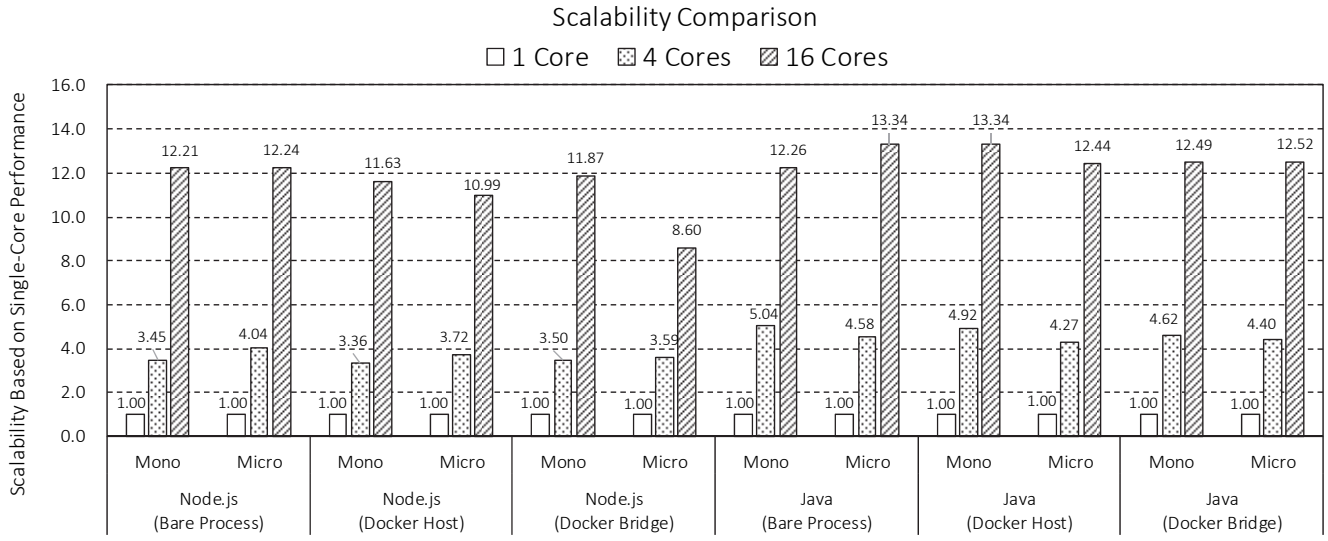

Fig. 5. Scalability comparison among Node.js and Java implementations with Bare-process, Docker-host, and Docker-bridge experimental configurations. Values are relative ratios of throughput based on 1-core throughput of each configuration.

V. Hardware-Level Analysis

We analyzed the code-path lengths and cycles per instruction (CPI) of each experimental configuration to investigate the trend in performance.

A. Path Length and CPI Trends

We used an in-house tool to calculate path lengths and CPI. In this paper, a path length means the number of CPU instructions to process one client request. Path lengths enable us to check if the code path changes in the different experimental configurations. We discuss the relative ratios of the path lengths in this paper instead of the actual number of instructions. The CPI represents the average number of clock cycles to complete one instruction. The CPI increases due to hardware-level bottlenecks such as cache miss and branch miss.

Fig. 6 shows the code-path lengths to process one client request in each experimental configuration. On average, the path lengths of the microservices were longer than the monolithic service by 3.05 times in Node.js and 3.00 times in Java. We need to refer to the outlier that occurred in the Java experiments on the single core. The path lengths on the single core were longer by 29.9% on average than those of the experiments on four or more cores. We explain the cause later in this section. There were two more trends outside of this anomaly. The first trend was that the path lengths in the Node.js experiments were longer than those in the Java

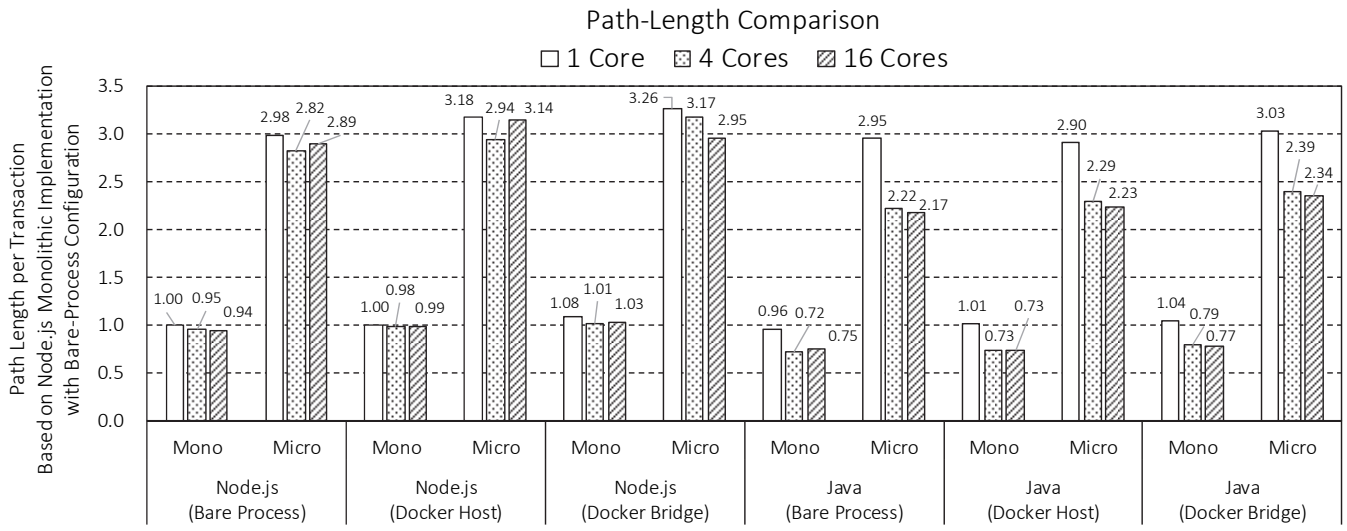

Fig. 6. Path-length comparison among Node.js and Java implementations with Bare-process, Docker-host, and Docker-bridge experimental configurations. Values are relative ratios of path lengths based on path length of Node.js monolithic implementation with bare process configuration.

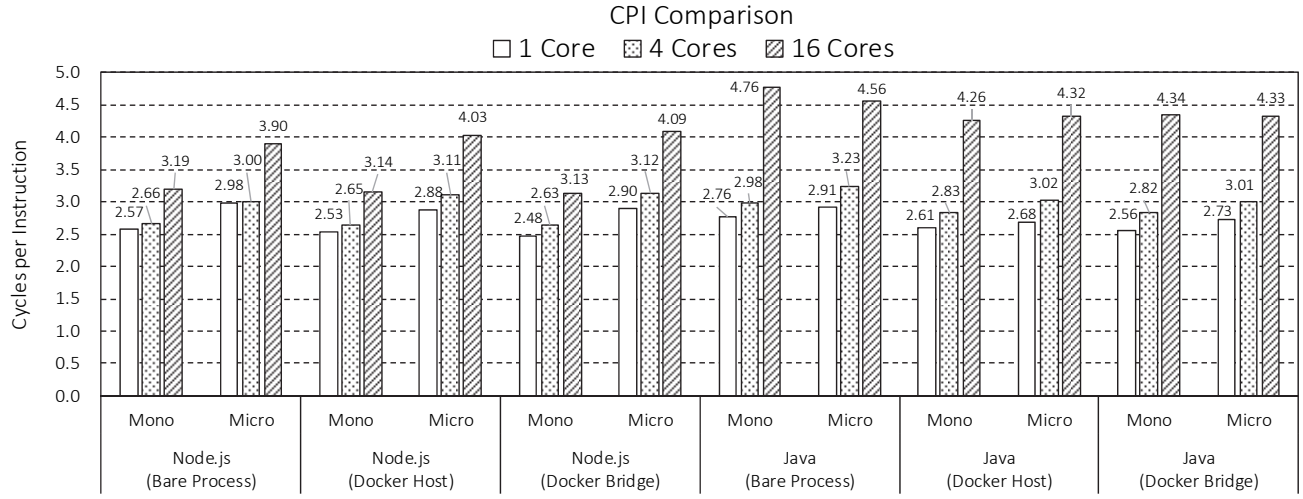

Fig. 7. CPI comparison among Node.js and Java implementations with Bare-process, Docker-host, and Docker-bridge experimental configurations

experiments. The second trend was that the path lengths were similar among the different numbers of CPU cores.

Fig. 7 shows the CPI of each experimental configuration. For the single and four cores, the CPI of Node.js and Java were similar. In the benchmarks on the 16 cores, Node.js always exhibited smaller CPI than Java. In the Node.js implementations, the microservice configurations exhibited 19.7% larger CPI than the monolithic configurations on average. In addition, the differences in the CPI between the Node.js monolithic and microservice implementations increased as the number of CPU cores increased. The Node.js microservice implementation with the Docker-bridge configuration on the 16 cores exhibited 30.8% longer CPI than that of the Node.js monolithic implementation. Surprisingly, the experiments with Docker on the 16 cores exhibited smaller CPI than those without Docker.

B. Breakdown Investigation and Discussion

Fig. 8 shows the breakdown of the path lengths of the Docker-bridge experimental configuration. The other experimental configurations had a similar path-length trend; thus, we omit the results. There is a different trend between Node.js and Java. In the Node.js experiments, the Node.js native layer implemented using C++ had major path lengths in the workloads. In the Java experiments, however, the major part was the just-in-time (JIT) compiled code, i.e. Java classes, not the Java runtime. We can explain this difference from the runtime architectures. Node.js handles the network operations including socket handlings in the C++ native layer. Java implements a large part of the network operations in Java libraries. Thus, the Java JIT code exhibited longer path lengths than Node.js JIT code.

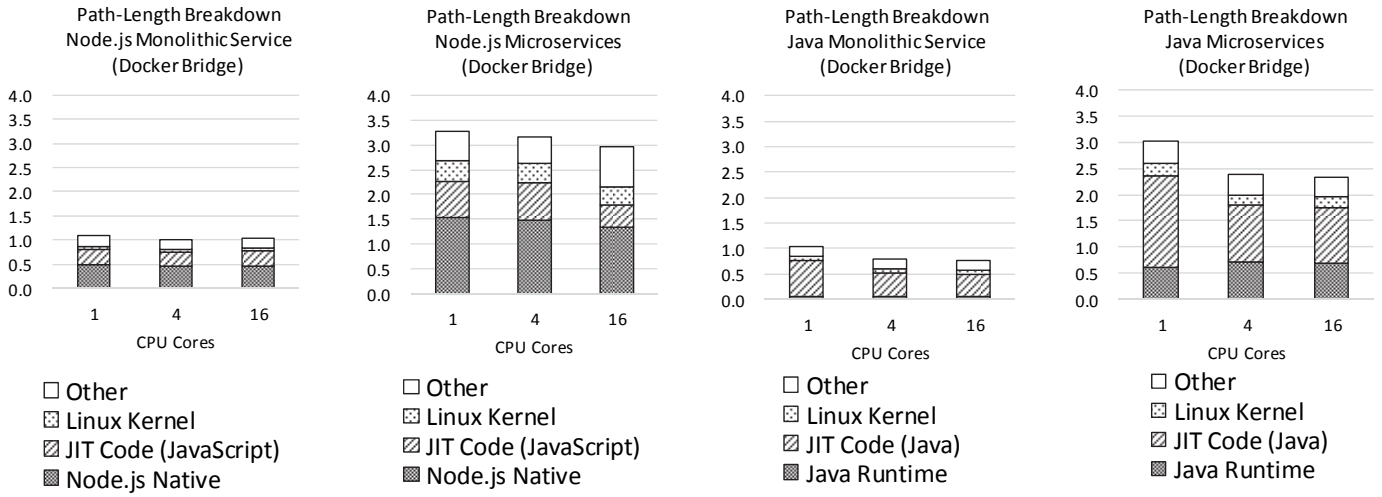

Fig. 8. Path-length breakdown of Node.js and Java implementations with Docker-bridge configuration. Values are relative ratios of path lengths based on path length of Node.js monolithic implementation with bare-process configuration, same as in Fig. 6.

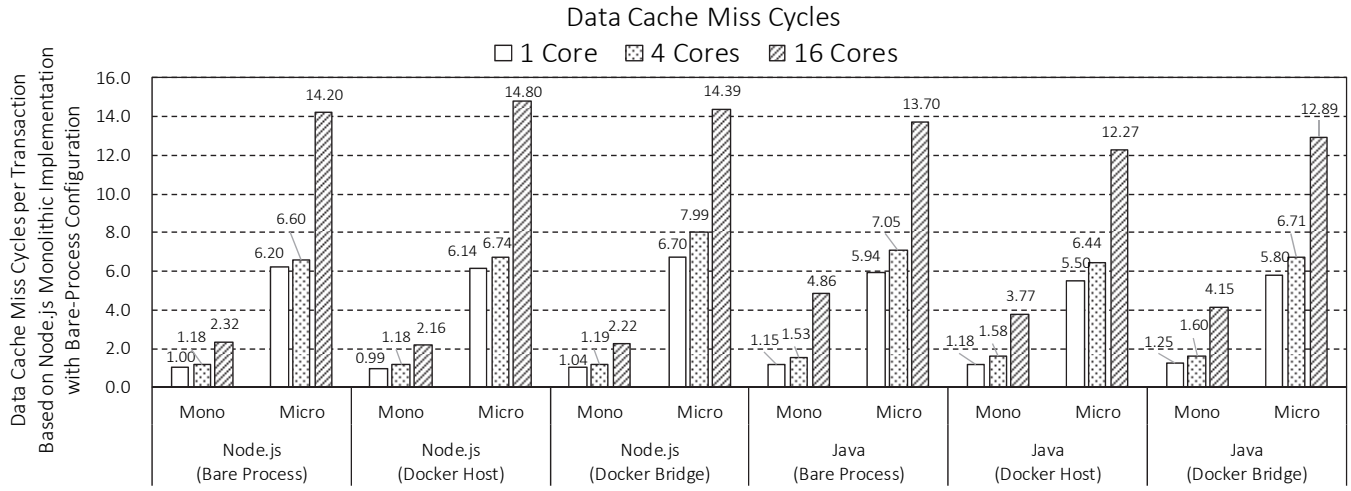

Fig. 9. Data cache miss cycle comparison among Node.js and Java implementations with Bare-process, Docker-host, and Docker-bridge experimental configurations. Values are relative ratios of cache miss cycles based on cache miss cycles of Node.js monolithic implantation with bare-process configuration.

The breakdown results in Fig. 8 show the cause of the anomaly in the Java experiments on the single core. We can see that the JIT code on the single core exhibited longer path lengths than on the 4 or 16 cores. We assume that the quality of the JIT code was poor, resulting in extended path lengths. To confirm this, we warmed up the benchmark on the four cores then took three cores offline and kept one online. This experiment forced the Java runtime to compile the source code on the four cores and use the code on the single core. We confirmed that performance on the single-core improved and became almost the same with that on the 4 and 16 cores. We argue that a single core is insufficient for collecting enough information for generating high-quality JIT code.

As described above, a Node.js process can use only a single core because it adopts a single-thread processing model. Thus, we tend to use multiple Docker containers for providing parallelized performance. When the system hosted microservices on 16 cores, it had 68 containers including 64 containers for Node.js and 4 containers for MongoDB instances. The number of entries in the iptables increased as the number of containers increased. As a result, the overhead of the Linux kernel and iptables included in "Other" in Fig. 8 increased. We also confirmed that acquiring spin locks in the Linux kernel contributed to the longer path lengths.

Fig. 9 shows the relative ratios of data cache miss cycles per transaction in each configuration. We can say that the trend in the total cache miss cycles is similar among the two language runtimes. When we changed the number of cores from 4 to 16, the cache misses rapidly increased compared to the change from the single core to four cores.

Fig. 10 shows the breakdown of the data cache miss cycles. We can see that the Node.js microservice implementation exhibited a larger number of cache miss cycles compared to the Java microservice implementation. The majority of the cache misses were nf_connt, which is a part of netfilter. Fig. 11

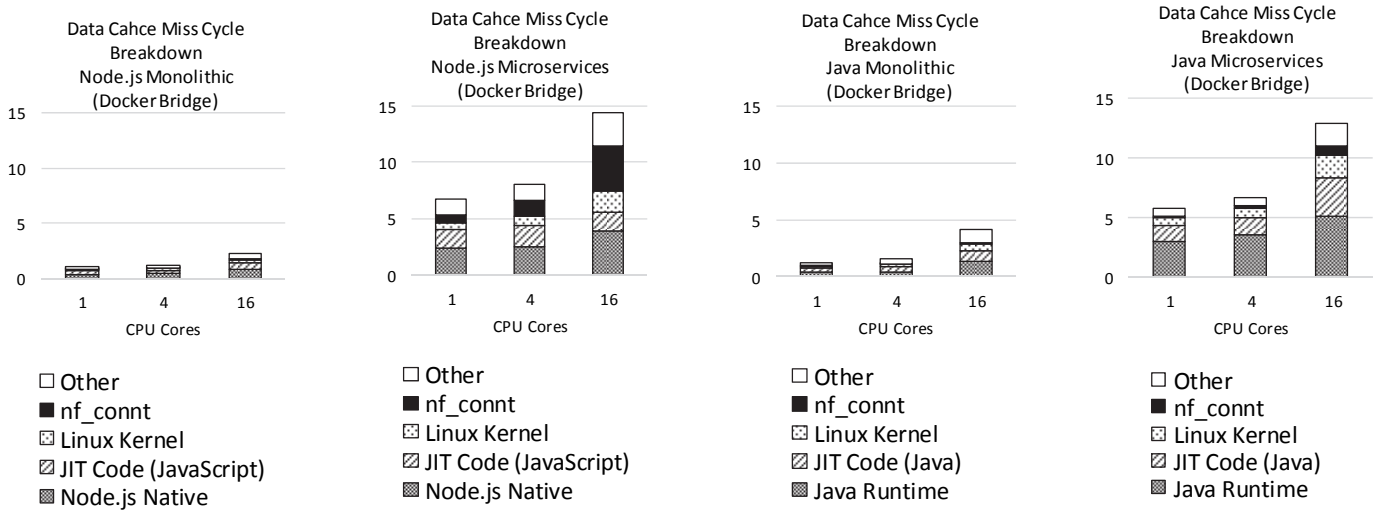

Fig. 10. Data cache miss cycle breakdown of Node.js and Java implementations with Docker-bridge configuration. Values are relative ratios of cache miss cycles based on cache miss cycle of Node.js monolithic implementation in bare-process configuration, same as in Fig. 9.

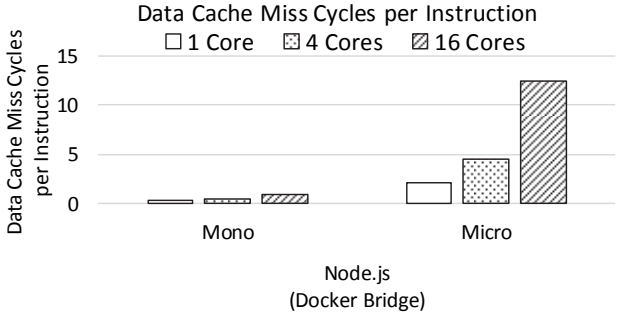

Fig. 11. Data cache miss cycles per instruction of nf_connt in Node.js implementations in Docker bridge configuration

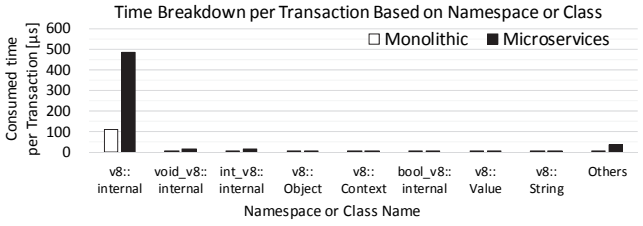

Fig. 12. Time Breakdown of native layer of Node.js

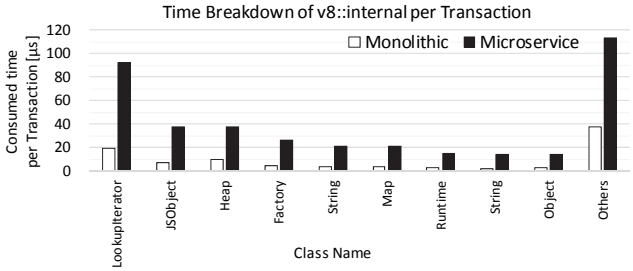

Fig. 13. Time Breakdown of v8::internal namespace of Node.js

shows the cache miss cycles per instruction of nf_connt. We confirmed that nf_connt increases the cache miss penalty when we have a large number of containers. We also found the function v8::internal::LookupIterator::Next() caused many cache misses. Node.js needs to support dynamic addition or deletion of properties to a class. It uses a hidden class to optimize accesses to such dynamically added properties. If Node.js fails in using the hidden class for optimization, it needs to use this function to look up the property. Java does not have such ability and the runtime did not consume much time compared to Node.js. This is the difference between the two language runtimes.

VI. Runtime Layer Analysis

In this section, we discuss the performance differences between microservices and monolithic services from the perspectives of Node.js and Java. As discussed in the previous section, the JIT code and language runtimes had longer path lengths in microservices than that in monolithic services. We explain which part of the language runtime generated the longer path lengths. We used a sample-based profiling tool to collect stack samples. We also collected symbol information of the Node.js JIT code by patching the Node.js source code to create call graphs. This enabled us to analyze the performance results by using the call graphs.

A. Node.js Breakdown Analysis

Fig. 12 shows the breakdown results of the Node.js native layer on the four cores. The functions in the v8::internal namespace consumed 84.0% of the running time of the Node.js native layer in the microservice architecture. Fig. 13 shows the further breakdown of the functions in the v8::internal namespace. The most time-consuming class was LookupIterator. As explained in the previous section, JavaScript supports dynamic property additions and deletions. LookupIterator is used to lookup properties in a class when the hidden class failed to optimize the lookup operations. The v8::internal namespace includes common functions; thus, the functions in the namespace were called in a variety of contexts including Acme Air application logics. To identify the bottleneck of the microservice architecture, we need to identify why these functions were called.

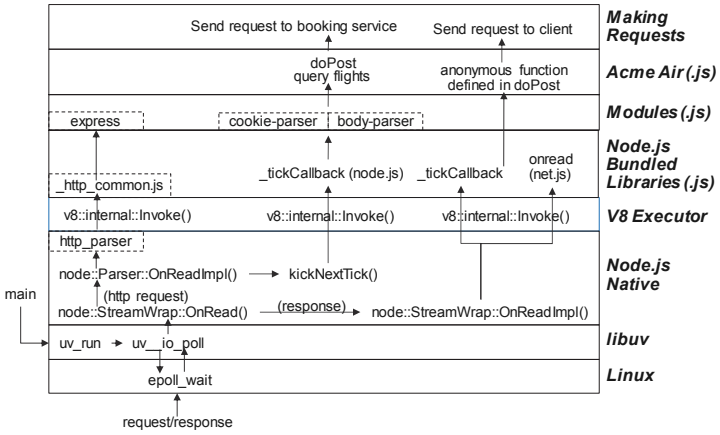

Fig. 14. Node.js software architecture including Acme Air stack

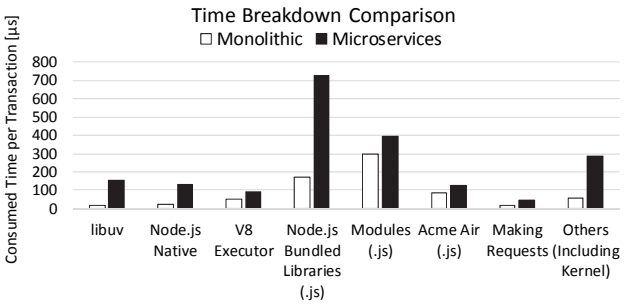

Fig. 15. Node.js breakdown classified by callers

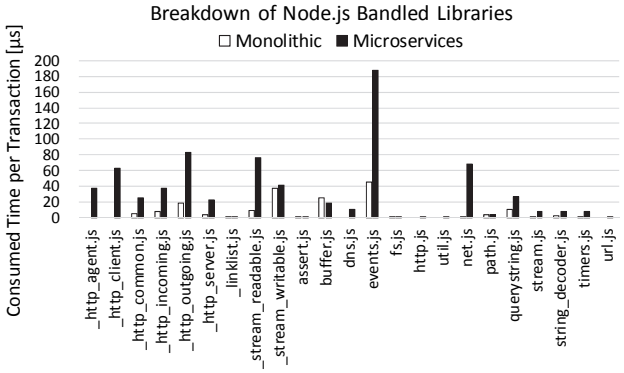

Fig. 16. Node.js breakdown in library level

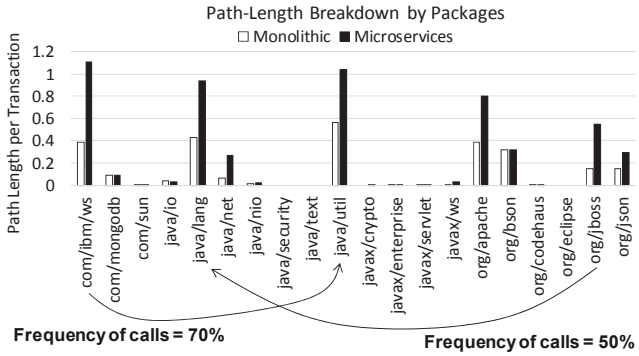

Fig. 17. Breakdown of Java implementations

Fig. 14 shows the Node.js software architecture including the Acme Air software stack. Node.js uses a single thread to process incoming requests besides helper threads for handling I/O blocking. Node.js relies on libuv to handle events that are notified from the operating system. The C++ layer of the Node.js runtime calls the APIs of V8 to invoke JavaScript code blocks. The core modules of Node.js are written in JavaScript. The libraries are stored in the /lib directory in the Node.js runtime directory. We call these libraries "Node.js bundled libraries". Events.js delivers events to the callback functions of applications. In normal cases, applications use Express, which is a library for building a lightweight Web server. The Express library actually invokes the callback functions of Acme Air.

We developed a tool to analyze the stack traces to calculate how long each layer in Fig. 14 consumes to process a transaction. The tool was used to analyze each stack sample and identify the callers. We collected the profiles on an x86 machine apart from the measurement machine because of the limitation of our in-house profiling tool. Fig. 15 shows the analysis results. The most time-consuming part was the Node.js bundled libraries. The microservice implementation consumed 4.22 times more time in the libraries than the monolithic implementation to process one user request. We broke down the profiles of the bundled libraries based on the file names. Fig. 16 shows the results. In the microservice implementation, the libraries for making HTTP requests consumed 37.1% of the time that the bundled libraries consumed. Another time-consuming part was events.js, which delivers events to callback functions. Events.js consumed 25.7% of the total time that the bundled libraries consumed in the microservice implementation. The Node.js event-handling mechanism depends on libuv, which consumed only 2.53% in the monolithic implementation and 7.98% in the microservice implementation. Since the microservices implementation of Acme Air generates more communication than the monolithic implementation, the libuv in the microservice implementation consumed more time than the monolithic implementation.

From these analyses, we found that the communication layer of Node.js, rather than the application logics of Acme Air, is the main source of microservices performance degradation. Optimizing the runtime layer, especially the communication stack, is effective in providing better performance in the microservice environment. In the Node.js case, we argue that the efficiency of delivering events to the callback functions is important for Node.js performance in the microservice architecture.

B. Java Breakdown Analysis

We also created a call graph of the Java implementations. Fig. 17 shows the breakdown of the Java experiments. The results show that the Liberty layer is the main time-consuming part of the implementations. The com.ibm.ws package includes the main implementation of Liberty Core. The java.util package also generated a long path because the application server uses java.util package intensively. The org.jboss package consumed much more time. This means the communication of the microservice architecture caused the major bottleneck. This result suggests that there are many

opportunities of performance optimization by optimizing the network communication layer.

VII. Discussion And Related Work

In this section, we review the history of microservices and related work to clarify our contributions. According to Martin Fowler's blog [17], they started a discussion about the idea of microservices in 2011 at a workshop. James Lewis, the co-author of the blog post, presented some of the ideas at a conference in 2012. Since they posted the article as a blog post [17], the microservice concept has been attracting many developers. They also discussed the difference between microservices and SOA. It is our understanding that microservices do not use a common communication bus such as ESB. Each microservice communicates with other services by using APIs that the developers of each service define.

There has been no empirical study on the performance analysis of microservices. Villamizar et al. compared a monolithic service and microservices on Amazon Web Service [3][4]. They reported the performance numbers and cloud operation costs of the monolithic service and microservices. However, they did not provide detailed analysis of runtime and hardware layers that resulted in these differences. Kang et al. used the microservice architecture to implement a Dockerized OpenStack instance [1]. Le et al. adopted the microservices architecture to construct the Nevada Research Data Center [6]. They adopted the microservice architecture to handle the demands of change requests for the system. Heorhiadi et al. presented a systematic resiliency testing framework for microservices [7].

There have been studies on the performance analysis of Node.js. Zhu et al. reported high instruction cache and TLB misses of Node.js workloads [8]. Ogasawara reported that Node.js spent much time in V8 libraries [5]. Lei et al. compared the performances of PHP, Python, and Node.js [2]. They reported that Node.js completely outperformed PHP and Python in high concurrency situations.

The term 'micro' has been used in different contexts. For example, we can see an example in the discussion of microkernel and monolithic kernel. Thus, we cannot say that the microservice architecture is a completely new concept that is not related to the traditional loosely coupled software architecture. However, the key point of the microservice architecture is that it is applied for Web-application development. The container technology additionally generates new resource pressure on computing systems.

VIII. Conclusion

The microservice architecture is becoming a popular design strategy in Web services. Developers build a Web service with small services. They publish their services as Docker images by transferring the images to a cloud. This is a worldwide development trend. Thus, cloud vendors cannot escape from hosting a large number of microservices. Understanding the microservice workload is important for optimizing systems for microservices, which will be beneficial in terms of cost for both cloud venders and developers.

We characterized the behaviors of the runtime layers and processor when hosting microservices. We confirmed that both runtimes consumed much time in the runtime functions rather than the application logics. We assume this will increase when the number of microservices increases. Even though microservices can accelerate agile developments, we must recognize the negative impact on performance. There has been no empirical research on the performance difference between the monolithic and microservice architectures. We expect this study to provide clues to optimization both for CPU architecture and open-source software.

Acknowledgments

The authors would like to thank our colleagues at IBM in Toronto, Böblingen, and Rochester. They gave us valuable advice to conduct this work.

References

[1] H. Kang, M. Le, and S. Tao, "Container and Microservice Driven Design for Cloud Infrastructure DevOps," In *Proc. of IC2E 2016*, pp. 202-211.

[2] K. Lei, Y. Ma, Z. Tan, "Performance comparison and evaluation of Web development technologies in PHP, Python and Node.js," In *Proc. of CSE 2014*, pp. 661-668.

[3] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy Web applications in the cloud," In *Proc. of CCC 2015*, pp. 583-590.

[4] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang, "Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures," In *Proc. of CCGrid 2016*, pp. 179-182.

[5] T. Ogasawara, "Workload characterization of server-side JavaScript," In *Proc. of IISWC 2014*, pp. 13-21.

[6] V. D. Le, M. M. Neff, R. V. Stewart, R. Kelley, E. Fritzinger, S. M. Dascalu, F. C. Harris, "Microservice-based architecture for the NRDC," In *Proc. of INDIN 2015*, pp. 1659-1664.

[7] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, "Gremlin: systematic resilience testing of microservices," In *Proc. of ICDCS 2016*, pp. 57-66.

[8] Y. Zhu, D. Richins, M. Halpern, V. J. Reddi, "Microarchitectural implications of event-driven server-side Web applications," In *Proc. of MICRO 2015*, pp. 762-774.

[9] https://github.com/acmeair/acmeair.

[10] https://github.com/acmeair/acmeair-nodejs.

[11] https://github.com/wasperf/acmeair.

[12] https://github.com/wasperf/acmeair-nodejs.

[13] Acme Air Workload driver, https://github.com/acmeair/acmeair-driver.

[14] Apache JMeter, http://jmeter.apache.org/.

[15] Docker - Build, Ship, and Run Any App, Anywhere, https://www.docker.com/.

[16] Manifesto for Agile Software Development, http://agilemanifesto.org/.

[17] Microservices a definition of this new architectural term, http://martinfowler.com/articles/microservices.html.

[18] Node.js at PayPal, https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/.