November 30, 2016

RT0973

Computer Science 11 pages

Research Report

Workload Characterization for Microservices

Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara

IBM Research - Tokyo
IBM Japan, Ltd.
19-21, Nihonbashi Hakozaki-cho
Chuo-ku, Tokyo 103-8510 Japan

Workload Characterization for Microservices

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

***Abstract*— The microservice architecture is a new service model to compose 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. We want to understand their characteristics to design an infrastructure optimized for microservices. In this study, we used Acme Air, an open-source benchmark application for Web services, and analyzed the behavior of the application in two models, microservice and monolithic, for two widely used language runtimes, Node.js and Java Enterprise Edition (EE). We observed a significant overhead due to the microservice architecture; the performance of the microservice model can be 79.1% lower than the monolithic model on the same hardware configuration. The microservice model spent much more time in runtime libraries to process one client request than the monolithic model by 4.22x on a Node.js application server and by 2.69x on a Java EE application server. We explain the performance impact due to the microservice architecture through detailed performance analysis. We also discuss the performance impact due to network virtualization associated with the Docker container framework, which is commonly used as an infrastructure for microservices. These findings give us clues to optimizing the language runtime and hardware architecture for microservices.**

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

I. INTRODUCTION

As an increasing number of enterprises are providing their Web services on a cloud, their development speed has a direct impact on business outcomes. Therefore, the microservice architecture proposed by James Lewis and Martin Fowler is attracting much attention [3][24]. It allows us to build a Web service with *"a suite of small services"*, where each service communicates with other services *"with lightweight mechanisms, often an HTTP resource API"*, and the small services *"can be maintained separately, scaled separately, or thrown away if needed"* [3]. These characteristics in the microservice architecture are expected to accelerate the development cycle of Web services [19][25].

The Docker container framework [21] plays a key role to simplify managing microservices [29]. Because each microservice can be developed by a different developer, the version of a runtime library may be inconsistent among microservices. Thus, it could be difficult to setup an operating system that provides all the versions of the runtime library to host such microservices. The Docker container framework mitigates this issue by separating the file system among containers and capturing all the runtime libraries that a service needs to run. This separation also makes it easy to deploy a service from a development system to a staging system for testing and finally to a production system because a container can run on a system even when the version of a runtime library differs between the container and the host system.

The microservice architecture is, however, expected to consume more computation resources than a traditional monolithic architecture to provide the same Web service. For example, it is easy to imagine microservices consuming extra CPU cycles to communicate with each other through API calls. The Docker container technology furthermore relies on operating system functions, such as iptables, to isolate each container from others. Since the microservice architecture typically uses more containers than a monolithic one, the former is expected to spend more CPU cycles in the operating system to process a client request than the latter.

Application servers, moreover, tend to rely on a large number of CPU cores on a symmetric multiprocessor (SMP) for high performance since the increase of CPU clock frequencies has slowed down. Because a Node.js application server adopts a single-thread event-driven processing model, it is a common practice to run multiple server instances to take advantage of multiple CPU cores. Even for a Java EE application server, which adopts a multi-thread processing model, it is also a common practice to run multiple server instances because the performance of an application server can typically scale only up to a certain number of CPU cores due to lock contentions in the application server and the Java virtual machine. The microservice architecture further increases the number of processes on a system because of the suite of small services, where each process is typically associated with a container because a container containing multiple processes may cause difficulties in managing, retrieving logs, and updating processes individually [8]. Thus, the number of containers tends to increase when microservices run on a large number of CPU cores. It is therefore important to understand the performance characteristics of a large number of containers running on a single system.

Villamizar et al. evaluated microservices on a cloud [6][7] and reported the performance difference between monolithic services and microservices. They however did not provide detailed performance analysis. There has been no empirical study on the performance analysis of microservices. In this

This technical report is an updated version of the conference paper written by the same authors which appeared in IISWC 2016 [10].

study, we used Acme Air [14]-[17], an open-source benchmark application that mimics transactional workloads for a fictitious airline's Web site, implemented in two different service models: monolithic and microservice, and on two different application server frameworks: Node.js and Java Enterprise Edition (EE). Since Node.js is based on a single-thread model while Java EE is based on a multi-thread model, studying the two server frameworks help us understand the effect of the thread model on Web service performance. We analyzed the performance difference from the aspects of hardware and language runtime levels. To the best of our knowledge, this is the first study that analyzes microservices with detailed performance profile data. Even though we focused on Acme Air, the results offer generally applicable insights into optimization opportunities in a computer system to host microservices. We discuss the following notable performance characteristics based on our measurement results.

- *Noticeable performance impact due to the microservice architecture* – we observed that the performance of Acme Air in a microservice model is much lower than that in a monolithic model by up to 79.1% on Node.js and by up to 70.2% on Java EE.
- *Frequent CPU data cache misses due to a large number of containers and their root cause* – the Docker container framework uses a network bridge to virtualize the network across containers in a default configuration, which uses iptables. As a result, as the number of containers increases, the number of iptables entries to be scanned for each communication operation increases; thus, the number of cache misses increases.
- *Primary software components in Node.js and Java EE application servers that contributed to the throughput difference between microservice and monolithic models* – we analyzed the performance profile of the two models to investigate which part of the runtime contributed to the throughput difference. We identified that Node.js and Java EE runtime libraries for handling HTTP communication consumed significantly more CPU cycles in the microservice model than in the monolithic one.
- *A non-negligible performance impact due to network virtualization associated with the Docker framework* – the Docker framework supports multiple network virtualization configurations. A default configuration with a network bridge degraded the throughput by up to 24.8% from another configuration that did not virtualize the network.

The rest of the paper is structured as follows. Section II explains the background of this research. Section III describes our experimental methodology. Section IV summarizes our experimental results, and Section V analyzes them from path length and cycles-per-instruction (CPI) perspectives. Section VI further analyzes them from language runtime perspectives. Section VII discusses previous work, and Section VIII concludes this paper.

II. Background

A. Microservices and Container Technology

James Lewis and Martin Fowler proposed microservices in their blog [24]. They define the microservice architecture as *"a particular way of designing software applications as suites of independently deployable services"*. Each microservice runs in its own process and communicates with others through a lightweight mechanism, typically an HTTP resource API. Thus, one can update a part of a Web service by testing and deploying a small service separately from other services [26]. Even if a newly deployed component causes a failure, the other parts of the system should continue running with graceful degradation [26]. The microservice architecture is expected to accelerate agile software development [23], deployment, and operation practices.

Docker [21] relies on a Linux container technology that isolates processes and their resources through operating-system mechanisms, such as cgroups and iptables. Since a process in a container runs as a native process of the host operating system, the container technology should not introduce extra overhead in terms of processor virtualization. In terms of network virtualization, on the other hand, a default configuration uses a network bridge to expose a virtualized network interface to each container that is connected to a private network segment. This virtualization relies on iptables to transfer packets among the virtualized network interfaces of containers and other physical networks. Thus, the network virtualization could introduce extra overhead due to the network bridge mechanism.

The Docker framework takes advantage of these isolation mechanisms for Linux containers to make DevOps (an integration of development and operation processes) easy. For example, one can develop a container image for a Web service on a preferable environment (e.g. a laptop computer), can transfer it to a staging system for system-level testing, and can transfer it to a production system. The file system isolation mechanism within the Docker framework allows each container to use its own runtime libraries, which can be different from those installed on the operating system of its host (i.e. development, staging, or production) system. A network isolation mechanism (e.g. a network bridge), if used, allows each container to use its own network port space avoiding a port conflict with other Web services or its host system.

The microservice architecture promotes a Web service that consists of multiple smaller services, where each service is typically developed and deployed as a container. Thus, for complex real-world applications, it can lead to a large number of containers running on a single system and can generate a significant amount of network traffic over a network bridge. Therefore, it is critical to understand the performance characteristics due to isolation mechanisms associated with the Linux container, as microservice-based Web services emerge.

B. Node.js and Java EE Application Server for Microservices

Node.js is one of the most popular server frameworks for Web services [27]. It adopts an event-driven processing model, where a Node.js server uses one server thread, which is always runnable, and processes incoming requests asynchronously.

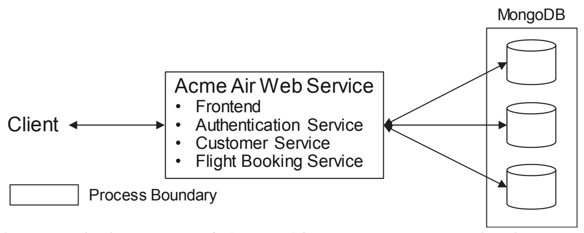

Fig. 1. Monolithic model of Acme Air benchmark application. It uses single process to run entire Acme Air Web service. We used MongoDB as data store.

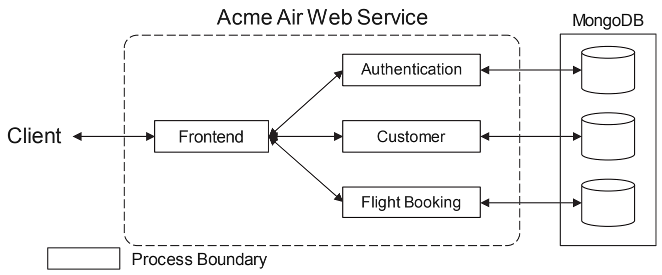

Fig. 2. Microservice model of Acme Air benchmark application. It uses four processes, one for each microservice. We also used MongoDB as data store.

The Java EE framework, in contrast, adopts a multi-thread processing model, where each server thread processes incoming requests synchronously, for example, by waiting for the completion of an I/O operation before accepting another incoming request.

A previous work by PayPal reported that the Node.js framework can perform better than the Java EE framework due to the difference in the processing model [28]. The Node.js framework, however, requires multiple server instances to take advantage of multiple CPU cores due to its processing model, while the Java EE framework requires only one server instance. This means that the Node.js framework needs to use more containers than the Java EE framework to use multiple CPU cores because, as described in the Introduction, each server typically runs as a container [8]. Thus, the performance of a Node.js application server may suffer more from the overhead due to isolation mechanisms associated with the Linux container than that of a Java EE application server.

III. Measurement Methodology

A. Acme Air Benchmark Application

We used Acme Air [14]-[17], an open-source benchmark application that mimics transactional workloads for a fictitious airline's Web site. Four projects on GitHub [14]-[17] correspond to four models of Acme Air, a combination of two service models (monolithic and microservices) and two server frameworks (Node.js and Java EE). We collected the performance data for these four models of Acme Air by running them on a Linux operating system of a single computer.

Another GitHub project provides a default workload scenario [18] to be driven by an Apache JMeter driver [20]. The scenario models typical user behavior at a flight booking site, where each client logs in the Web site, searches available flights between two cities for a preferable date and time, and books one of those flights.

The monolithic model of the Acme Air benchmark application shown in Fig. 1 runs the entire Acme Air Web service on a single application server. We used a MongoDB instance as a data store. The microservice model is, on the other hand, made of four microservices: frontend, authentication, flight booking, and customer services, as shown in Fig. 2. The frontend service interacts with clients as well as the other three microservices by using a representational state transfer (REST) API. The authentication service performs the client authentication and manages Web sessions. The flight booking service handles flight searching and booking requests from clients. The customer service manages information associated with each customer including booked flights and customer profile information.

B. Software and Hardware Environment

We used Node.js 4.2.2 and IBM WebSphere Application Server Liberty Core 8.5.5.6 with IBM Java Runtime Environment 1.8 as an application server runtime. We configured each runtime to have a 1GB heap memory space. We used Docker 1.9.1 as a container framework, MongoDB 3.2.0 as a database, and Apache JMeter 2.13 [20] as a workload driver.

We used a single host machine, IBM z13, to collect the performance profile data for the four models of the Acme Air benchmark application. As shown in Fig. 3, we used two logical partitions (LPARs) running on the host: one LPAR with 16 CPU cores and 2TB memory to host one or more application servers and MongoDB instances, and the other LPAR with 64 CPU cores and 1TB memory to host an Apache JMeter workload driver. We collected the performance profile data of the LPAR that hosts application servers and MongoDB instances. Both LPARs run a Linux operating system (SUSE Linux Enterprise Server 12 SP1). We allocated the two LPARs on the same computing system to avoid potential performance effects due to limitations in the bandwidth and latency of a network between the application servers and the workload driver. Because physical CPU cores and memory are dedicatedly assigned to each LPAR, the interference due to the fact that the two LPARs run on the same computing system is negligible.

C. Experimental Configurations

Fig. 3 also illustrates three container configurations we used to measure the performance of the Acme Air benchmark application for the four models. First, the bare-process configuration runs each service as a process without using a Docker container. This configuration allows us to exclude the performance effect due to containers. Second, the Docker-host configuration runs each service as a container without using a network bridge. This configuration allows us to exclude the performance effect due to network virtualization. Finally, the Docker-bridge configuration runs each service as a container using a network bridge. This configuration allows us to measure performance when the network is virtualized with a network bridge, and this is also a default configuration for Docker containers.

For each of these three container configurations, we ran an experiment for each of the four models of the Acme Air

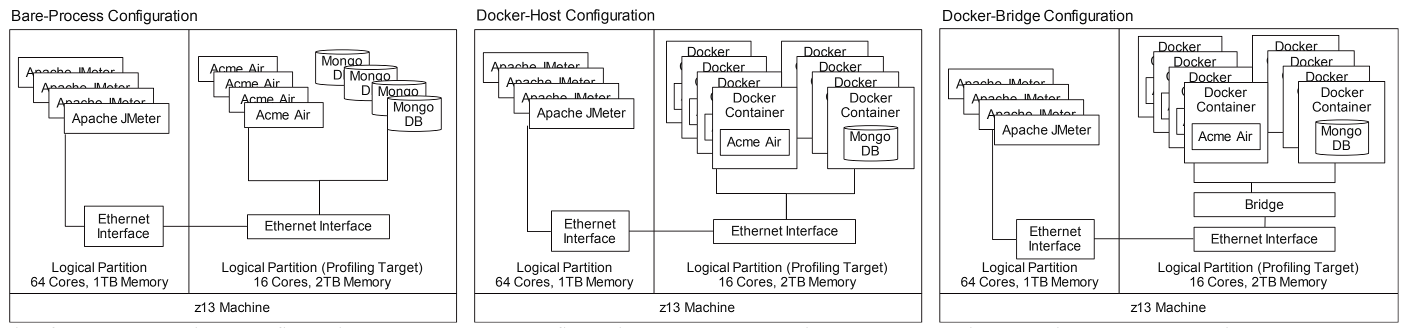

Fig. 3. Three container configurations. Bare-process configuration runs each service as process without using Docker containers. Docker-host configuration runs each service as Docker container without network virtualization. Docker-bridge configuration runs each service as Docker container with network virtualization (network bridge). We collected profile data of LPAR (Profiling Target) that hosts Acme Air Web service and MongoDB instances.

Table 1. Number of processes during measurements. Total number of containers can be calculated by summing one number from (A) and another number from (B) on same row.

# of CPUs	(A) Acme Air				(B) Data Store
	Monolithic		Microservices		MongoDB (# of Tenants)
	Java EE Server	Node.js Server	Java EE Server	Node.js Server	
1	1	1	4	4	1
4	1	4	4	16	1
16	4	16	16	64	4

benchmark application by enabling 1, 4, or 16 CPU cores for the application servers. Thus, we conducted an experiment for 36 configurations. Since a MongoDB instance may become a bottleneck if it accepts too many requests, we configured multiple Acme Air tenants for experiments on 16 cores to avoid such a bottleneck. One Acme Air tenant consists of one Acme Air Web service and one MongoDB instance. The three configurations shown in Fig. 3 are those with four Acme Air tenants.

A Java EE application server uses a thread pool to process incoming requests. To tune the number of threads in the thread pool, we collected the throughput of the monolithic model on 16 cores by changing the thread pool size. We found that the Java EE server showed peak throughput when the size is 100 threads. When the size was more than 100, the throughput degraded due to lock contentions in the server. Thus, we chose 100 threads as the thread pool size for all of the performance measurements.

We also collected the throughput of the monolithic model with a Java EE server on 1, 4, and 16 cores by changing the number of tenants. On 1 and 4 cores, a setup with a single tenant showed better throughput than those with 2, 3, or 4 tenants. On 16 cores, a setup with 4 tenants showed better performance than those with 1, 2, 3, 6, or 8 tenants. Thus, we chose to use a single tenant on 1 and 4 cores, and 4 tenants on 16 cores. We also applied the same number of tenants to Node.js experiments. We furthermore ran the same number of Acme Air services on Node.js as the number of CPU cores since Node.js runs in a single-thread processing model.

Table 1 shows the number of server processes used in our experiments. The number of processes in the microservice experiments were four times more than that of monolithic experiments because the microservice model uses four processes per tenant, as shown in Fig. 2. We used the same number of JMeter processes as the number of tenants. Each JMeter process sent client requests to a specific tenant. For Node.js experiments on 4 and 16 cores, each JMeter process uniformly sent client requests to multiple Node.js processes of one tenant.

IV. Performance Overview

Fig. 4 shows the throughput performance for the 36 experimental configurations, a combination of 12 software configurations and three different numbers of CPU cores (1, 4, or 16 cores). The 12 software configurations correspond to a combination of two application server frameworks (Node.js or Java EE), three container configurations (bare-process, Docker-host, or Docker-bridge), and two service models (monolithic or microservices). Each throughput is relative to that of a base configuration, which is a monolithic model of the Acme Air application running on a Node.js application server in the bare-process configuration with a single CPU core. Fig. 5 shows the CPU-core scalability for each of the 12 software configurations. Fig. 4 indicates that the throughput of a microservice model was always significantly lower than that of a monolithic model when the rest of the configuration options were the same. The throughput difference between the two models is up to 79.1% for a Node.js application server and up to 70.2% for a Java EE application server. These differences are larger than those previously reported [6].

The use of Docker containers exhibited a non-negligible impact on performance. The Docker-bridge configuration, a default in Docker, exhibited lower performance than the Docker-host configuration by up to 24.8% on a Node.js application server. The performance difference indicates that network virtualization alone has a non-negligible impact on performance. One can argue that network virtualization can be omitted if each application container can be configured to use an available network port on the host network when the container is deployed. Network virtualization, however, also plays a key role in isolating network traffic among applications for security purposes. Thus, the performance overhead may not justify omitting the network virtualization.

Figs 4 and 5 also show other interesting performance aspects. A Java EE application server outperformed a Node.js application server on the 4 and 16 cores, except the 16-core bare-process configuration, while a Node.js application server

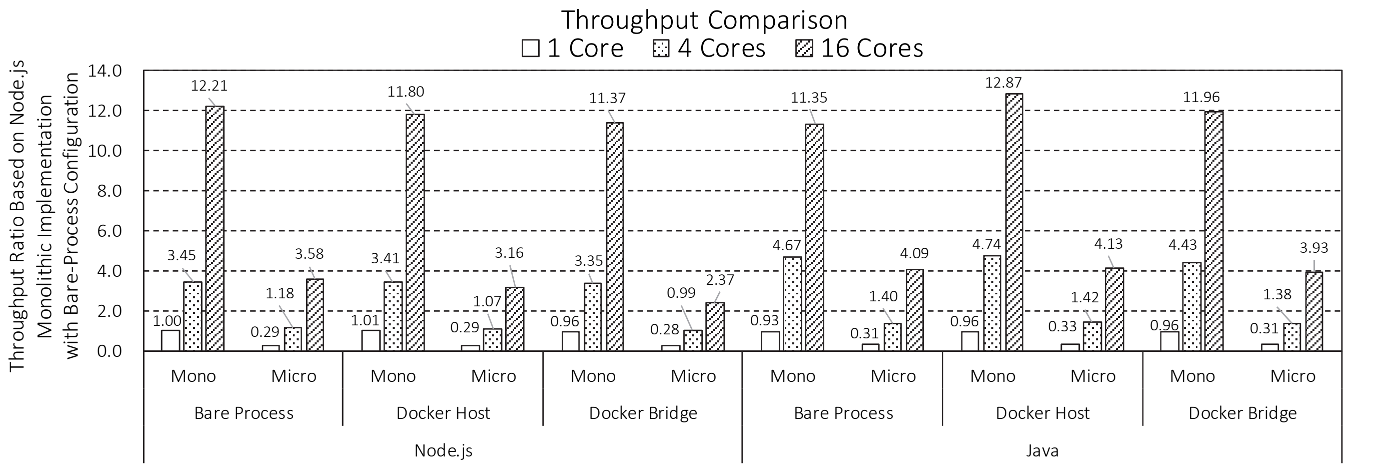

Fig. 4. Relative throughput performance of 36 configurations: combination of two application server frameworks (Node.js and Java EE), three container configurations (Bare-process, Docker-host, and Docker-bridge), two service models (monolithic and microservices), and number of CPU cores (1, 4, and 16 cores). Throughput numbers are relative to base configuration at left-most column (Node.js, Bare-process, monolithic, 1 core).

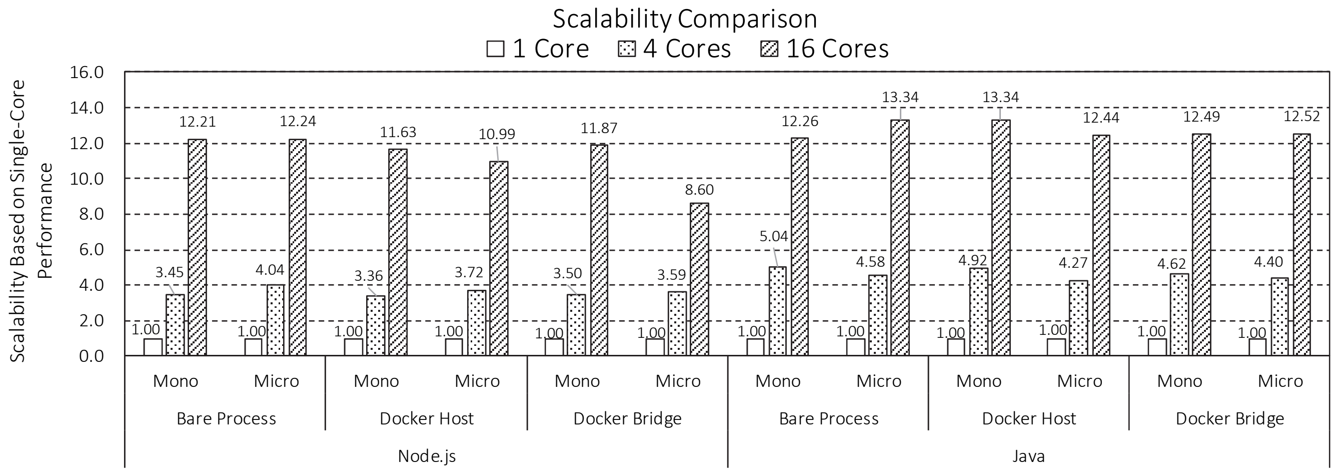

Fig. 5. Scalability comparison among Node.js and Java server frameworks with Bare-process, Docker-host, and Docker-bridge experimental configurations. Each value is throughput relative to 1-core throughput of each configuration.

outperformed a Java EE application server on a single core with the monolithic application. Furthermore, a Java EE application server exhibited super-linear scalability on four cores.

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

V. Hardware-Level Analysis

A. *Path Length and CPI Trends*

We used an in-house tool to calculate the path length and CPI for each experiment. In this paper, the path length means the number of CPU instructions to process one client request. We compared the path length among the 36 experimental configurations to identify how each configuration option affected the path length. To this end, we discuss a path length relative to a base configuration, rather than the actual number of instructions. The CPI represents the average number of clock cycles to complete one instruction. The CPI generally increases due to a hardware-level bottleneck such as cache and branch misses.

Fig. 6 shows the relative path length for each experimental configuration. Overall, the path length of a microservice model is longer than that of a monolithic model by 3.0 times on average on both Node.js and Java EE application servers. There is an anomaly on a Java EE application server with a single core. The path length on a single core is longer than that on 4 or 16 cores by 29.9% on average. We explain its cause later in this section. There were two more trends outside of this anomaly. First, the path length on a Node.js application server is generally longer than that on a Java EE application server. Second, the number of CPU cores does not significantly affect the path length (except for the anomaly).

Fig. 7 shows the CPI for each experimental configuration. The choice of the application server (Node.js or Java EE) does not significantly affect the CPI on 1 and 4 CPU cores. That is,

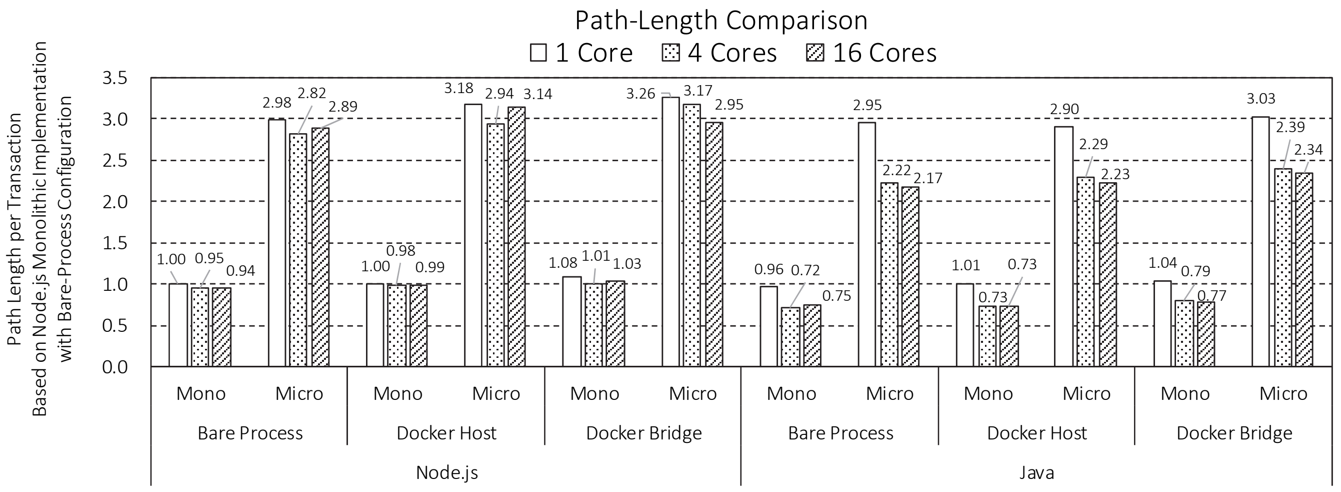

Fig. 6. Relative path length of 36 configurations: combination of two application server frameworks (Node.js and Java EE), three container configurations (Bare-process, Docker-host, and Docker-bridge), two service models (monolithic and microservices), and number of CPU cores (1, 4, and 16 cores). Path-length numbers are relative to base configuration at left-most column (Node.js, Bare-process, monolithic, 1 core).

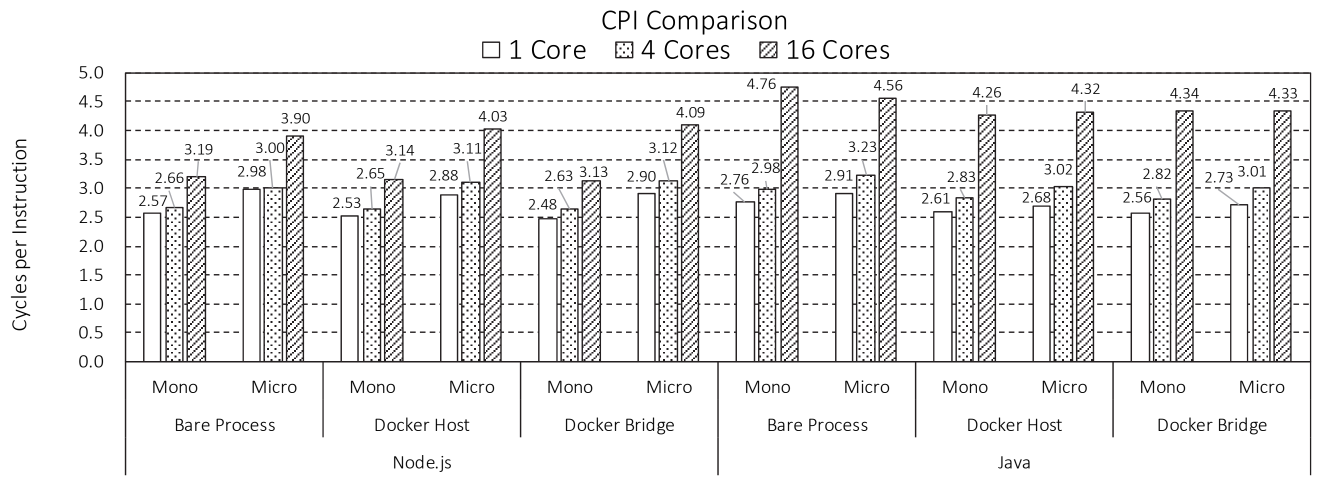

Fig. 7. CPI of 36 configurations: combination of two application server frameworks (Node.js and Java EE), three container configurations (Bare-process, Docker-host, and Docker-bridge), two service models (monolithic and microservices), and number of CPU cores (1, 4, and 16 cores).

Node.js and Java EE application servers exhibited a similar CPI when the number of cores is 1 or 4 and the rest of the configuration options are the same. When the number of cores is 16, however, a Node.js application server always exhibited a smaller CPI than a Java EE application server.

Let us examine the effect of the service model (monolithic or microservices) on the CPI. On a Node.js application server, the CPI with a microservice model is larger than that with a monolithic model by 19.7% on average when the rest of the configuration options are the same. The CPI increases as the number of CPU cores (i.e. the number of containers) increases. The difference between the two models is largest at 30.8% for a Node.js application server in a Docker-bridge configuration running on 16 cores.

B. Breakdown Investigation and Discussion

Fig. 8 shows a breakdown of the path length for the experiments in a Docker-bridge configuration. We omit the breakdown for the rest of the container configurations because they have a similar pattern. The components in the path length are quite different between the two application server frameworks. The largest component for Node.js application servers is Node.js native runtime written in C++, while that for Java EE application servers is JIT code (just-in-time compiled code) written in Java. We can explain this behavior from a difference in the two application server architectures. A Node.js application server handles runtime operations, such as socket API, mostly in a native runtime library written in C++, while a Java EE application server handles them mostly in a runtime library written in Java. Thus, the JIT code (Java) on a Java EE application server exhibited much a longer path length than the JIT code (JavaScript) on a Node.js application server.

The breakdown results in Fig. 8 can explain the cause of the anomaly we discussed earlier in which the path length on a Java EE application server with a single core is longer than that of the same software configuration with 4 or 16 cores, as shown in Fig. 6. Our hypothesis is that the quality of the JIT code is poor, resulting in an extended path length, when only

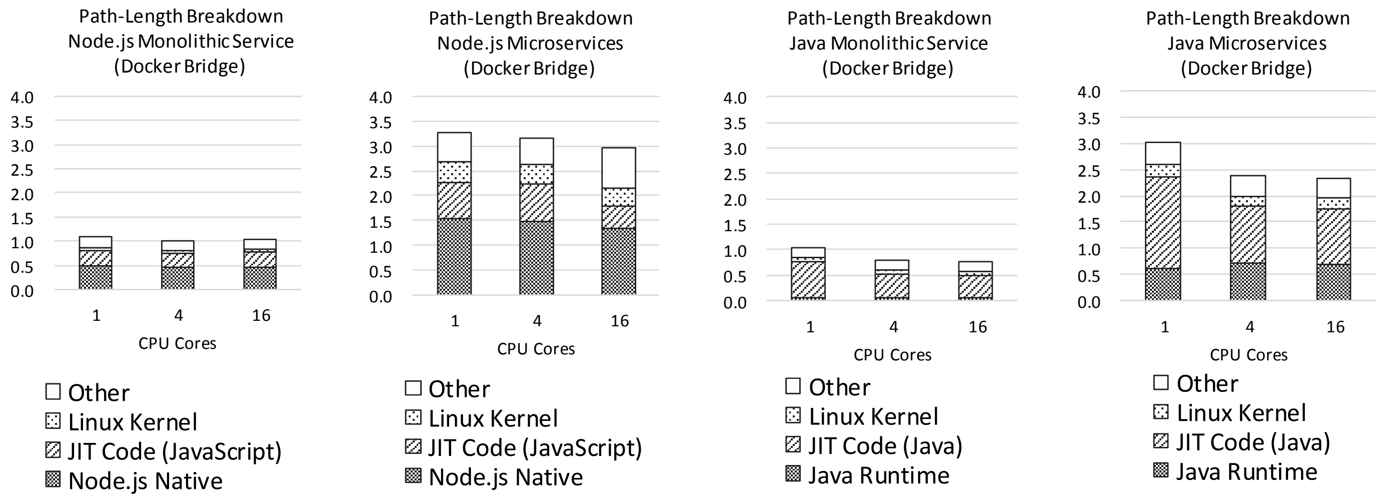

Fig. 8. Path-length breakdown of Node.js and Java server frameworks with Docker-bridge configuration. Each value is path length relative to that of Node.js monolithic model with Bare-process configuration, same as in Fig. 6.

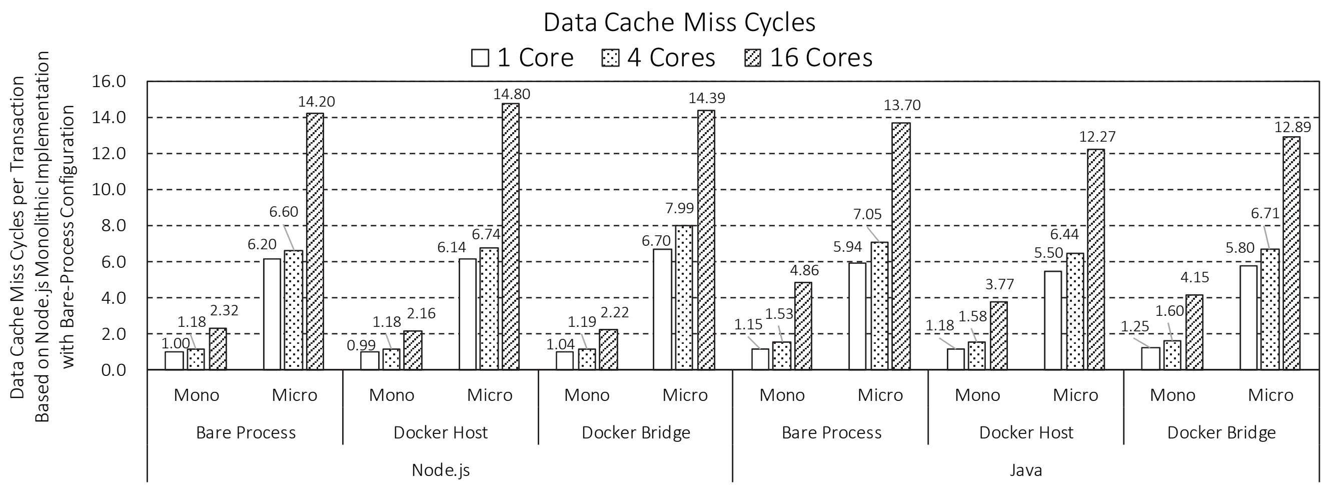

Fig. 9. Data cache miss cycle comparison among Node.js and Java server frameworks with Bare-process, Docker-host, and Docker-bridge experimental configurations. Each value is number of cache miss cycles per transaction, relative to that of Node.js monolithic model with Bare-process configuration.

one CPU core is available. To validate this, we started and warmed up an application server on four cores then took three of them offline at the Linux kernel level before measuring performance. This experiment allowed the Java runtime to compile the Java code on four cores but still allowed us to measure performance on a single core. The results of this experiment in fact validated our hypothesis; the path length is almost the same regardless of the number of cores when the Java code is compiled on four or more cores. We argue that a single core is insufficient for the JIT compiler to generate high-quality code when the core is also busy for serving client requests.

As described earlier, one Node.js application server can use only a single core because of its single-thread processing model. Thus, we need to use multiple application servers to take advantage of multiple cores. When the microservice model ran on 16 cores, the system hosted 68 containers: 64 for Node.js and 4 for MongoDB instances (See Table 1). 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.

Fig. 9 shows the number of data cache miss cycles per transaction, relative to the base configuration (a monolithic model running on a Node.js application server in a bare-process configuration with 1 core). This metric corresponds to the number of clock cycles to serve an L1 cache miss. We can say that the trend in the total number of cache miss cycles is similar among the two language runtimes when the number of CPU cores increases. That is, it increased slightly when the number of cores increased from 1 to 4, but it increased significantly when the number of cores increased from 4 to 16.

Fig. 10 shows a breakdown in the number of data cache miss cycles for a Docker-bridge configuration. We can see that a microservice model on a Node.js application server exhibited a slightly larger number of cache miss cycles than that on a Java EE application server, where the former caused a large

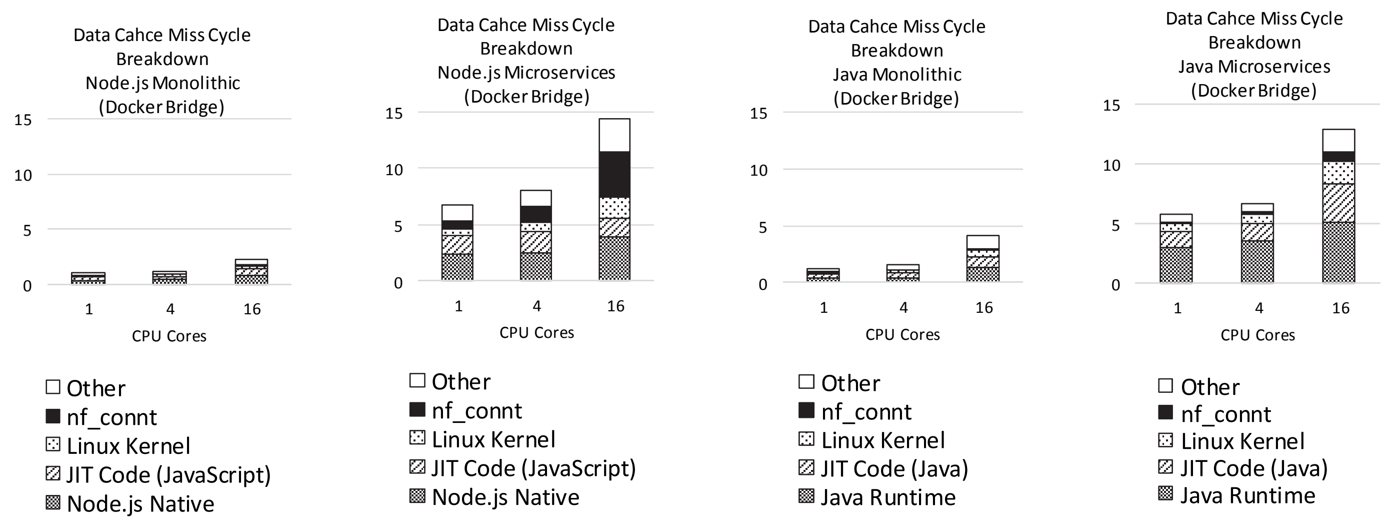

Fig. 10. Data cache miss cycle breakdown of Node.js and Java server frameworks with Docker-bridge configuration. Each value is number of cache miss cycles relative to that of Node.js monolithic model in Bare-process configuration, same as in Fig. 9.

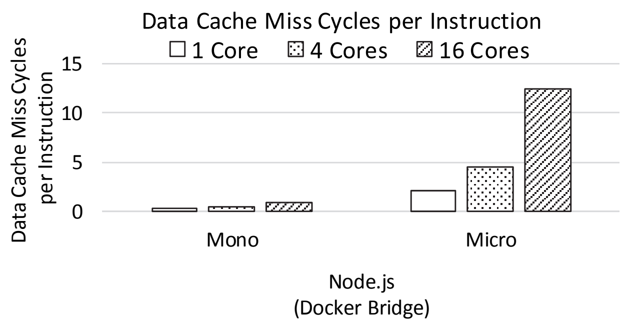

Fig. 11. Number of data cache miss cycles per instruction from nf_connt on Node.js application server in Docker-bridge configuration

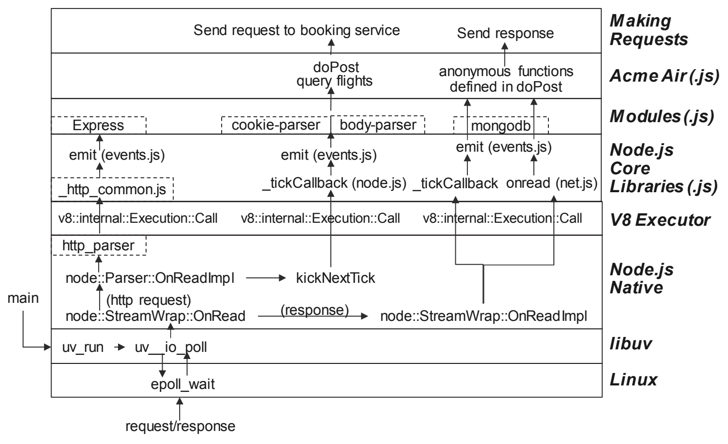

Fig. 12. Node.js software architecture including Acme Air application layer.

portion of the cache miss cycles in nf_connt, which is a part of netfilter. Fig. 11 furthermore highlights the behavior of nf_connt with the number of cache miss cycles per instruction from nf_connt, which increases as the number of containers increases.

VI. Runtime Layer Analysis

In this section, we discuss the performance difference between monolithic and microservice models on both Node.js and Java EE application servers. As discussed in a previous section, the path length for a microservice model was longer than that for a monolithic model on both Node.js and Java EE application servers. We explain which software functions contributed to the increase on both server frameworks by analyzing stack trace samples collected using a sample-based profiling tool. Note that typical method-level profilers do not necessarily give us much insight into a high-level software behavior because hot methods identified by profilers are typically used in many contexts. In this study we analyzed stack trace samples collected from both Node.js and Java EE servers to calculate the time spent for each high-level software function.

A. Node.js Breakdown Analysis

Fig. 12 shows the Node.js software architecture including the Acme Air application layer. A Node.js application server uses a single thread to process incoming requests besides helper threads for handling synchronous I/O operations. It relies on a library, libuv, to handle events notified from the operating system. The libuv library calls a callback function in the Node.js native runtime, which calls the V8 executor through a V8 API to invoke a JavaScript code block. In most cases, the V8 executor invokes a function in Node.js core libraries. The core libraries implement common utilities such as an HTTP communication handler (_http_common.js) and an event handler (events.js). The latter delivers an event to an appropriate JavaScript callback function, such as those in Express module, which provides a lightweight Web server. The Express module parses each HTTP request and typically generates another event, which eventually calls back a service handler in Acme Air, such as queryflights function.

We developed a tool to analyze stack trace samples to calculate how long each software layer shown in Fig. 12 takes

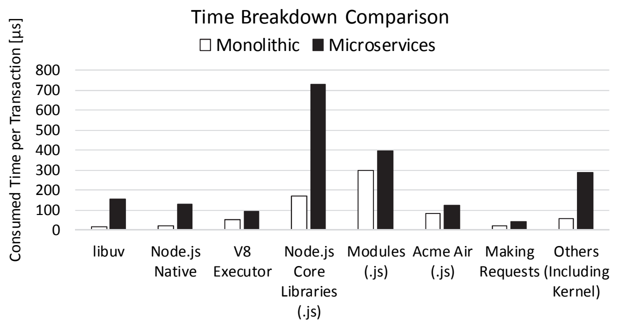

Fig. 13. Node.js execution time breakdown classified by software layer on single core with Bare-process configuration. We developed tool to analyze stack trace samples to calculate how long each software layer takes to process client request.

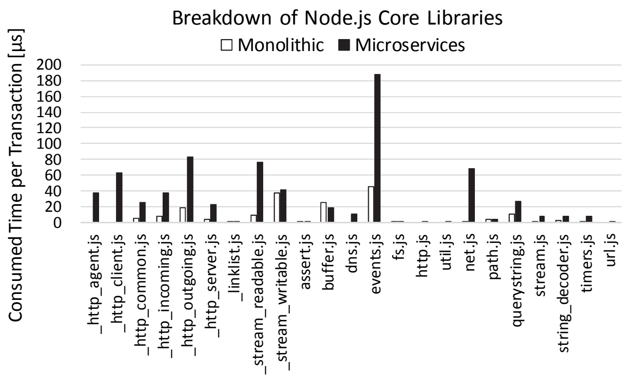

Fig. 14. Node.js execution time breakdown at Node.js core libraries layer, categorized by library file name. Libraries of which name begins with "_http_" and events.js consumed much longer time in microservice model than monolithic model. Those libraries represent different behavior of two models.

to process a client request. A stack trace sample includes an instruction pointer and a sequence of callers to reach the instruction when the sample was collected. The tool takes each stack trace sample to traverse the sequence of callers to identify one of the eight software layers shown in Fig. 12 that was being executed when the sample was collected. The tool furthermore accumulates the time spent for each layer by adding a sampling period to the identified layer for each sample and divides it by the number of transactions served to normalize it. We collected the profiles on an x86 machine apart from the measurement machine due to a limitation in our in-house profiling tool.

Fig. 13 shows the analysis results on a single core with Bare-process configuration. The most time-consuming layer for the microservice model was the Node.js core libraries layer. The microservice model spent 4.22 times more time in the layer than the monolithic model to process one client request. We further broke down the time spent in the layer based on the file name of libraries as shown in Fig.14. In the microservice model, the libraries of which name begins with _http_, which handle HTTP communication, consumed 37.1% of the time that the layer consumed. Another time-consuming library was events.js, which delivers an event to a callback function.

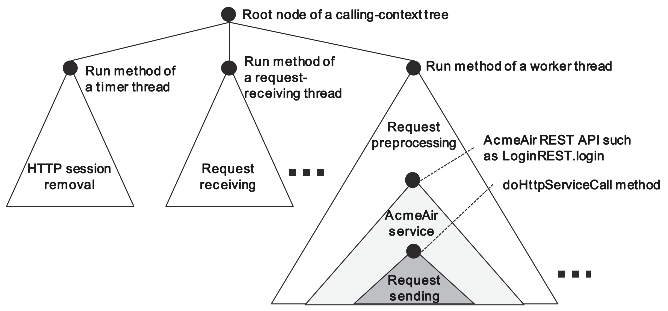
Root node of a calling-context tree
Run method of a timer thread
Run method of a request-receiving thread
Run method of a worker thread
HTTP session removal
Request receiving
Request preprocessing
AcmeAir REST API such as LoginREST.login
doHttpServiceCall method
AcmeAir service
Request sending

Fig. 15. High-level structure of Calling-context tree of Acme Air on Java EE server, common for both monolithic and microservice models

Events.js consumed 25.7% of the total time that the layer consumed for the microservice model. The Node.js event delivery mechanism depends on libuv, which consumed only 2.53% in the monolithic model and 7.98% in the microservice model, as shown in Fig. 13. Since the microservice model of Acme Air generates more HTTP communication than the monolithic model, the libuv in the microservice model consumed more time than the monolithic model to deliver more events caused by HTTP communication.

Our analysis has shown that the software layer which manages HTTP communication in the Node.js application server, rather than the Acme Air application layer, is the main cause of the performance gap between the two service models. Optimizing the software layer managing communication is a key to improve the performance for the microservice model. We furthermore argue that the performance of an event delivery to callback functions is also important for the Node.js application server because the number of events due to HTTP communication increases in the microservice model.

B. Java Breakdown Analysis

To analyze the performance difference between monolithic and microservice models on a Java EE framework, we collected a calling-context tree (CCT) profile [2][5] by using JPROF [22]. Fig. 15 illustrated a high-level structure of the CCT, which is common for both models. Each triangle corresponds to a subtree of the CCT. The three nodes next to the root correspond to three types of Java threads: timer, request-receiving, and worker. The timer thread removes HTTP session objects when they are not accessed for a certain period of time. The request-receiving thread receives a client request and stores it in a request buffer. The worker thread takes a client request from the request buffer and processes it. We divided the subtree for the worker thread further into three sections: request-sending, Acme Air service, and request-preprocessing. First, the request-sending section is one or more subtrees whose root node is doHttpServiceCall method. This section is responsible for sending a REST request to another microservice. Second, the Acme Air service section is one or more subtrees whose root node is a method to serve an Acme Air REST API (e.g. LoginREST.login()), excluding the request-sending section. This section is responsible for executing application-level services, such as login and flight booking by accessing a backend database. Finally, the request-preprocessing section is the rest of the subtree for the worker thread. This section is responsible for parsing an HTTP request, creating HTTP-related objects (e.g. session and cookie objects), and creating REST objects, which are used in a REST request handler in the Acme Air application. To understand the performance difference between monolithic and microservice models, we examine the time spent on a Java EE server for the following six sections: HTTP session removal (timer thread), request-receiving (request-receiving thread), request-preprocessing (worker thread), Acme Air service (worker thread), request-sending (worker thread), and others.

We collected the following two kinds of performance profiles to calculate the execution time of each section. One is an execution-time profile that provides the execution time of each Java method, collected by using an in-house tool on z Systems. The other profile is a calling-context tree (CCT) profile that provides the number of method calls at each node of the tree, collected by using JPROF. We collected two profiles from different runs of the Acme Air benchmark application because JPROF affects the execution time significantly due to its underlying instrumentation mechanism while it can produce an accurate CCT profile. We further collected the CCT profile for each service separately in the microservice model to reduce the profiling overhead.

We calculated the execution time of each section in the following three steps.

1. For each method m, we calculated the execution time per call $T_{\text{per-call}}(m)$ by using the following equation.

$$T_{\text{per-call}}(m) = \frac{T(m)}{N(m)}$$

where $T(m)$ is the execution time per transaction of a method m, which is obtained from the execution-time profile, and $N(m)$ is the number of calls per transaction for a method m, which is obtained from the CCT profile. Note that we approximate the execution time of each method call with the averaged execution time per call. The actual execution time of a method may be different per call.

2. We categorize all nodes (i.e. methods) in a CCT into six sections as discussed earlier.

3. For each section s, we calculated the execution time per transaction $T(s)$ by using the following equation.

$$T(s) = \sum_{m \in S} T(s, m)$$

where S is a set of methods appeared in section s and $T(s, m)$ is calculated by using the following equation.

$$T(s, m) = T_{\text{per-call}}(m) \times N(s, m)$$

where $N(s, m)$ is the number of calls per transaction for a method m appeared in section s.

Fig. 16 shows an execution-time breakdown of Acme Air on a Java EE server for monolithic and microservice models. Both models spent a longest time in the request-preprocessing section. Optimizing this section is a key to improve the performance on a Java EE server, especially in the microservice model since it consumes 78% of the total execution time. The time spent in the Acme Air service section is almost the same between the two models because it essentially conducts the same amount of work per transaction in both models. Somewhat surprisingly, the request-sending section was not a major cause of the performance degradation in a microservice model; it consumes only 0.1% of the execution time.

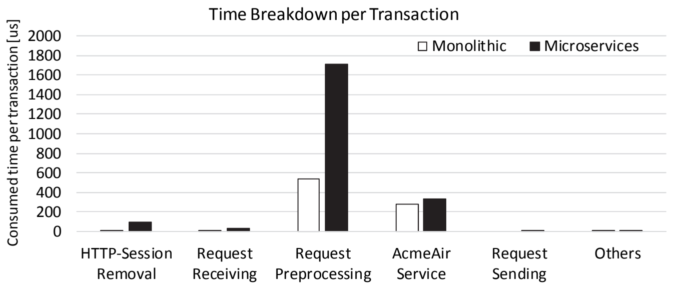

Fig. 16. Execution time per transaction of Acme Air on Java EE server for monolithic and microservice models categorized into 6 sections

Another cause of the performance degradation in the microservice model was an increasing execution time in the HTTP session removal section. The microservice model consumed 97 microseconds per transaction in the section while the monolithic model consumed only 0.01 microseconds per transaction. This is because the Acme Air benchmark application creates a new HTTP session per request for inter-microservice communications. Reusing HTTP sessions is expected to improve the performance of the microservice model.

VII. Discussion And Related Work

In this section, we review related work to clarify our contributions. According to Martin Fowler's blog [24], he and his co-author James Lewis started discussing an idea of microservices in 2011 at a workshop. Lewis, presented some of the ideas around microservices at a conference in 2012. After they posted an article as a blog [24], the microservice architecture has been attracting many Web service developers.

There has been no empirical study on the performance analysis of microservices. Villamizar et al. compared monolithic and microservice architectures on Amazon Web Service [6][7]. They compared the performance number and cloud operation cost between the two architectures. They, however, did not provide deep analysis to identify the root causes of the differences. Kang et al. used the microservice architecture to implement a Dockerized OpenStack instance [1]. Le et al. adopted the microservice architecture to construct Nevada Research Data Center [11]. They adopted the microservice architecture to respond to frequent system design changes. Heorhiadi et al. presented a systematic resiliency testing framework for applications in the microservice architecture [12].

There have been studies on the performance of Node.js application servers. Zhu et al. reported a high miss ratio for an instruction cache and a translation lookaside buffer (TLB) on a Node.js application server [13]. Ogasawara reported that a host running a Node.js application server spent a significant time in V8 libraries [9]. Lei et al. compared the performance of PHP, Python, and Node.js application servers [4] and reported that a Node.js application server completely outperformed the PHP and Python ones when many clients accessed the server concurrently.

VIII. CONCLUSION

The microservice architecture is gaining attention as it is considered to reduce the cycle time of Web service development, where a Web service is decomposed into a set of smaller "micro" services and each can be individually developed, tested, deployed, and updated as a container. Since each microservice runs in a container, a server system is expected to host an increasing number of containers communicating with each other, typically via a REST API on a virtualized network. It is, therefore, important to understand the workload characteristics of applications designed in such a microservice model.

To that end, we examined the workload characteristics of monolithic and microservice models of Acme Air, which is an open-source benchmark application that mimics an airline ticket reservation service. We also examined the interplay between the choice of two service models (microservice and monolithic) and the choice of application server frameworks (Node.js and Java EE) since the Acme Air application is available on both service models and both application server frameworks. Our experimental results have shown that the application in a microservice model spends a longer time in a server runtime, such as a communication library, than that in a monolithic model for both application server frameworks. The performance gap between the two service models is expected to increase as the granularity of services decreases. This suggests that we need to tradeoff between the benefit from agile development and the cost from performance overhead, both due to the microservice model. This study also suggests that optimization in the communication between services can potentially boost the performance of Web services in a microservice model.

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] J. M. Spivey, "Fast, accurate call graph profiling," Journal of Software: Practice and Experience, Vol. 34, Issue 3, pp. 249-264, Mar. 2004.

[3] J. Thönes, "Microservices," IEEE Software, Vol. 32, Issue. 1, pp. 113-116.

[4] 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.

[5] K. Vaswani, A.V. Nori, and T.M. Chilimbi, "Preferential path profiling: compactly numbering interesting paths," In *Proc. of POPL 2007*, pp. 351-362.

[6] 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.

[7] 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.

[8] R. Benevides, "10 things to avoid in docker containers," http://developers.redhat.com/blog/2016/02/24/10-things-to-avoid-in-docker-containers/.

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

[10] T. Ueda, T. Nakaike, and M. Ohara, "Workload Characterization for Microservices," In *Proc. of IISWC 2016*, pp. 85-94.

[11] 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.

[12] 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.

[13] 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.

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

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

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

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

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

[19] Adopting Microservices at Netflix: Lessons for Architectural Design, https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

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

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

[22] JPROF - Java Profiler, http://perfinsp.sourceforge.net/jprof.html.

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

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

[25] Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach, IBM Redbooks, Aug. 2015, http://www.redbooks.ibm.com/abstracts/sg248275.html?Open.

[26] Microservice Trade-Offs, http://martinfowler.com/articles/microservice-trade-offs.html.

[27] Node.js 2016 User Survey Report, https://nodejs.org/static/documents/2016-survey-report.pdf.

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

[29] The Definitive Guide To Docker Containers, https://www.docker.com/sites/default/files/WP-%20Definitive%20Guide%20To%20Containers.pdf.