Request Reconstruction in MirageOS Unikernels

Al Amjad Tawfiq Isstaif
Clare College

UNIVERSITY OF CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Master of Philosophy in Advanced Computer Science

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: aati2@cam.ac.uk

June 9, 2019

Declaration

I Al Amjad Tawfiq Isstaif of Clare College, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 14,849

Signed:

Date:

Request Reconstruction in MirageOS Unikernels

Abstract

Distributed tracing techniques have proven to provide an extremely useful and efficient approach to support a variety of performance management tasks for complex cloud distributed applications. Recently, unikernels have emerged as a promising cloud deployment model for network applications, as they provide a more secure, lightweight and elastic alternative to traditional virtualization options. A combined approach can provide a promising building block to address the autoscaling problem of microservices.

In this work, we present an aggregated distributed tracing model that can be used to identify the location of performance bottlenecks in microservice applications. The model enables each service to identify whether it is the root cause of a performance bottleneck or it is one of its descending dependent services. This is achieved based on aggregated measurements received by each service from its directly neighboring services. The aggregation feature allows a flexible tracing infrastructure which enables a local trace analysis at the level of each service as well as at the application level. This can be useful to support the development of both centralized and decentralized autoscaling policies.

We present protocol-agnostic generic execution wrappers that allow to implement the tracing model in Mirage unikernels. We leverage the existing tracing module in Mirage and focus on the two essential resources of CPU and network. These wrappers can be used to instrument protocol libraries which are based on synchronous inter-service communication and are written using Lwt lightweight threads. We use these wrappers to instrument the Cohttp library, which enables automatic instrumentation and trace generation for Mirage applications based on HTTP. We illustrate the service-centric analysis capability and show that such an approach can have a minimal overhead on throughput (2-5%) if appropriate sampling techniques are used.

Contents

1 Introduction **1**
- 1.1 Motivations and Aims . . . 1
- 1.2 Outcomes and Results . . . 2
- 1.3 Structure of the Thesis . . . 4

2 Background **5**
- 2.1 Distributed Tracing . . . 5
  - 2.1.1 Design of a Tracing Tool . . . 6
- 2.2 Unikernels . . . 7
  - 2.2.1 MirageOS Architecture . . . 7
  - 2.2.2 Lwt Lightweight Threads Library . . . 8

3 Related Work **11**
- 3.1 Autoscaling Microservices . . . 11
- 3.2 Distributed Tracing for Resource Management . . . 12
- 3.3 Relevant Tracing Frameworks . . . 13

4 Design and Implementation **15**
- 4.1 Tracing Model Design . . . 15
  - 4.1.1 Microservices Concepts and Design Requirements . . . 15
  - 4.1.2 Design Goals, Concepts and Assumptions . . . 17
  - 4.1.3 Example Usage of the Tracing Model . . . 21
- 4.2 Trace Points within Mirage . . . 23
  - 4.2.1 Overview of Cohttp Library Instrumentation . . . 23
  - 4.2.2 Propagational Trace Points . . . 25
  - 4.2.3 Trace Points within a Local Context wrapper . . . 25
  - 4.2.4 Trace Points for Network Waiting Time . . . 29
- 4.3 Implementation . . . 30
  - 4.3.1 Implementation Methodology . . . 30
  - 4.3.2 Implementation of Execution Wrappers . . . 32
  - 4.3.3 Accounting Context Switch Time . . . 34
  - 4.3.4 Trace Format and Serialization . . . 37
- 4.4 Requirement for Packet-Timestamping . . . 37
  - 4.4.1 Experimental Setup . . . 38
  - 4.4.2 Trace Analysis . . . 39

5 Evaluation **43**
- 5.1 Evaluation Goals . . . 43

5.2 Experimental Methodology 44
5.2.1 Benchmark Setup 44
5.2.2 Application Code 45
5.2.3 Trace Data Collection and Processing 46
5.3 Bottleneck Detection 47
5.3.1 Benchmark Performance 48
5.3.2 Waiting Times under a CPU Bottleneck 49
5.3.3 Waiting Times under a Network Bottleneck 49
5.4 Overhead Evaluation 53
5.5 Tuning the Number of Cooperative Context Switches 54

6 Summary and Conclusions **57**
6.1 Summary 57
6.2 Conclusions 58

List of Figures

4.1 A running example of a simple microservice application. 16
4.2 Illustration of waiting time aggregation in the tracing model . . . 18
4.3 Illustration of local and central trace collection. 19
4.4 Example autoscaling decision process based on the tracing model. 22
4.5 Overview of the usage of execution wrappers to instrument the Cohttp library 24
4.6 Trace points in a local context wrapper 26
4.7 Trace points to account for promise sleeping time. 27
4.8 Httperf benchmark latency under CPU and Network bottleneck in Mirage 40
4.9 Detection of an artificial network bottleneck based on packet timestamps 41
4.10 Mirage TCP stack trace analysis under a CPU bottleneck 42

5.1 Evaluation experimental setup 44
5.2 wrk Benchmark performance under a CPU and network bottleneck 48
5.3 Request processing times compared to benchmark latency 50
5.4 Waiting times under a CPU bottleneck 51
5.5 Waiting times under a network bottleneck 52
5.6 Throughput overhead with and without 1/1024 sampling 53
5.7 Local waiting time of backend service 55
5.8 Effect of Mirage connection establishment on backend service throughput 56

Chapter 1

Introduction

1.1 Motivations and Aims

MirageOS [1] is a library operating system that allows network applications to be written using high-level source code and compiled as single-purpose lightweight virtual machines that run on commodity hypervisors powering today's cloud platforms. With minimal memory footprints and fast startup times, unikernels effectively represent an efficient and secure alternative to release individual services compared to traditional virtual machines. However, end-to-end management of unikernel-based network applications composed of multiple services remains an independent problem.

In the past decade, distributed tracing has emerged as an extremely useful and efficient approach to perform a variety of performance management tasks for complex distributed systems in the cloud [2]. Distributed tracing provides the capability for an end-to-end reconstruction of a request path as it crosses the various boundaries within the distributed system. Traces extracted from these tools have been used to address various performance management tasks ranging from performance debugging and anomaly detection, to workload modelling and resource attribution. Furthermore, distributed tracing techniques have been used to develop effective dynamic resource management policies. Such policies allow to schedule system resources among requests belonging to multiple tenants in order to achieve fairness or satisfy tenant priorities [3, 4, 5].

Unikernels extended with appropriate distributed tracing capabilities can represent a promising building block to address the challenges of autoscaling microservice applications [6]. This is becoming an increasingly important problem with

the rise of software development practices that encourage larger-scale finer-grain service-based architectures as well as with increasing need for generic resource management solutions for such applications in emerging cloud and edge environments [7, 8]. On one side, distributed tracing can help tackle the problem of dynamic and real-time detection of resource bottlenecks across the distributed structure of microservice-based systems, which is essential to trigger effective scaling decisions that resolve such bottlenecks. On the other side, with the smaller granularity and the faster startup time of unikernels, resources can be provisioned reactively just-in-time [9, 10] and in finer grains.

While previous tracing models [3, 4, 5] supported the task of regulating existing system capacity, we build upon these works to propose a distributed tracing model that is appropriate to support the auto-scaling task, which is about adjusting system capacity in response to changing user demand. Another aim is to have a flexible tracing model that can support both centralized and decentralized autoscaling policies. This flexibility would allow to experiment with autoscaling policies that operate on the level of the entire request path as well as on a subset of this path, and down to the level of a single service. While centralized policies can ideally leverage a comprehensive view of the system state and request path, decentralized policies might be more desirable and effective for large scale applications or applications deployed across multiple data centers.

1.2 Outcomes and Results

In this work, we present an aggregated distributed tracing model which can support the development of autoscaling policies for microservice applications as well as a prototype implementation of this model for Mirage unikernels. The overhead is evaluated and the usage of this tracing model is demonstrated over a simplified microservice application structure.

A performance bottleneck can be detected by measuring the waiting time a request spends on a resource. We focus on waiting times during context switching as well as network transfer. These correspond to the CPU and network resources which are essential for any stateful or stateless service. Measuring waiting times allows autoscaling policies to identify the locations within the application structure where resource allocation can help reduce overall latency the most. This can be a horizontal scaling decision of adding a new service instance, a vertical scaling decision by increasing CPU share or network bandwidth, or a migration decision of the service to another host where more CPU time or network bandwidth are available.

The tracing model relies on the essential technique of metadata propagation to propagate request identifiers along with the network messages of the RPC call (e.g. HTTP request headers). These identifiers make it possible to correlate traces that belong to the same request across the distributed system. Furthermore, waiting time measurements are aggregated at each service and propagated in the opposite direction (e.g. HTTP response headers) to the directly neighboring *upstream* service initiating the request. Based on these aggregated measurements, each service can identify whether it is the root cause of a performance bottleneck or it is one of its dependent *downstream* services. This hierarchical aggregation allows a flexible tracing infrastructure that supports local resource allocation decisions based on traces collected from a subset of services on the request path, as well as global decisions based on end-to-end analysis of the request path. This flexible tracing infrastructure can then be used to develop both types of centralized and decentralized autoscaling policies as well as hybrid ones.

To implement the tracing model in Mirage unikernels, we introduce protocol-agnostic generic execution wrappers. These wrappers are integrated with the existing tracing module in Mirage and can be used to instrument Mirage protocol libraries used for inter-service communication. More specifically, the wrappers would be used to instrument the two sides of the service invocation if the protocol libraries are exposed as Lwt lightweight threads. On the server side, a *local context* wrapper is used to instrument the request handler and tracks its asynchronous execution to provide total execution time as well as waiting time during cooperative context switches for sequential execution paths. On the client side, a *remote context* wrapper is used to instrument remote calls to dependent services to account for network waiting time as well as to facilitate request information propagation between the upstream and downstream services.

We use these wrappers to instrument the client and server modules in the Mirage Cohttp library, which allows automatic instrumentation and trace generation for Mirage applications based on HTTP. We use a simple multi-tier application to illustrate the capability of useful service-centric trace analysis. We show how traces can be used to identify the service which is the root cause of a resource bottleneck and distinguish whether this is a CPU or network bottleneck. Our evaluation shows that the overhead of such an approach can depend on the amount of context switches, and can be reduced to have a minimal overhead on throughput (2-5%) if appropriate sampling techniques are used.

Finally, we identify the need to introduce packet-level timestamping to the network devices used by the unikernels in order to achieve reliable resource account-

ing. This is a necessary feature to distinguish genuine network bottlenecks from ones caused by CPU bottleneck as well as to accurately account for waiting time before request admission. This is due to the single-threaded architecture of Mirage unikernels, in which the TCP stack is embedded within the application code through non-preemptive lightweight threads or promises. In such an architecture, these lightweight threads are executed by an event loop, and any bottleneck in this loop would lead to a delay in network transfer, as this bottleneck prevents the TCP stack from processing packets in a timely manner.

1.3 Structure of the Thesis

In Chapter 2, we provide a background on distributed tracing techniques as well as the architecture of unikernels. In Chapter 3, we describe some of the related works and illustrate the contributions of this work. In Chapter 4, we present the design of tracing model and the prototype implementation in Mirage which is based on the general execution wrappers. In Chapter 5, we evaluate the accuracy and overhead of our prototype. In Chapter 6, we conclude.

Chapter 2

Background

2.1 Distributed Tracing

The work in [2] provides a systematic understanding of the area of workflow-centric distributed tracing. The authors use the term *workflow-centric* to refer to a wide spectrum of distributed tracing systems which rely on propagating request metadata information across the system boundaries in order to establish the connection between causally-related distributed events. Such an approach requires the altering of system components to perform what is referred to as *request metadata propagation* or *trace context propagation*, where traces are attached with request identifiers which allow to reliably reconstruct the execution path. This approach has become a widely-adopted approach in several production systems [11, 12] with several recent efforts to provide standardization to the semantics and APIs of metadata propagation [13, 14, 15].

Literature includes less-intrusive approaches to capture these causal relationships. Such approaches establish causality based on correlating shared variables in log messages [16] or timing-information of network messages [17]. These approaches are generally less accurate due to the lack of deterministic inference of causality. However, even when such a determinism is achieved, for example, through a developer-specified event schema as in Magpie [16], sampling techniques [18] which are essential to reduce the overhead of instrumentation cannot be implemented. This is due to the need for complete collection of log messages to establish causality [19]. Furthermore, metadata propagation is essential in order to execute performance management tasks in-band which is not possible through other approaches.

2.1.1 Design of a Tracing Tool

In this subsection, we aim to summarize some of the key design choices of a performance management tool based on distributed tracing as presented in [2]. This work is based on the previous experience of the authors in designing and implementing multiple popular tracing infrastructures as well as a systematic review of existing tools in the literature. The design choices presented are associated with tradeoffs that can be taken in light of the performance management task. Also, these tradeoffs are also influenced by whether the management task is executed in-bound (within the system itself) or out-of-bound (in a centralized system).

The authors identify two core choices that define the fundamental capabilities of the tracing infrastructure. The first is related to the scope of the causal relationships that need to be captured. A scope that includes all relationships is impractical to implement in production systems due to the high associated overhead. Therefore, this scope needs be limited to the relationships that support the performance task of the tracing tool. The second choice is related to the model used to express causal relationships. Specialized models such as paths and directed paths have the advantage of efficient storage and retrieval and can be used to express synchronous execution and simple concurrency. However, more expressive models such as directed acyclic graphs (DAGs) are required to express synchronization and dependencies within the request execution path.

The authors also identify two core software components that are required to implement a tracing tool. These are the metadata fields that need to be propagated along the request path, and the trace points that need to be added by the developer to the distributed system in order perform this propagation across boundaries of the system. These propagational trace points store records that indicate an event of serialization (e.g. network) or synchronization (e.g. fork-join). If the tracing tool executes out-of-bound management tasks, then only request identifiers and logical clock timestamps need to be propagated, which are stored and used later by the tracing tool to reconstruct the request execution path asynchronously. However, in a tracing tool which executes in-band management tasks, specific metadata fields are propagated from specific components and are then extracted by other components located on the request path. These components would use the value of these fields to support the execution of the performance management task and can thus reduce the need for trace persistence.

Alongside propagational trace points, value-added trace points can be introdced to the system to capture additional performance measurements, which can be

stored or propagated, depending on whether it is for an in-bound or out-of-bound management task. Overhead reduction mechanisms such as trace sampling and aggregation can also be implemented to allow the usage of the tracing tool in a production environment. Finally, for out-of-bound execution, a trace reconstruction component is necessary, which is executed over the traces stored asynchronously as described earlier.

2.2 Unikernels

The *unikernel* term was originally introduced through MirageOS [1] project which can be seen as a continuum to existing work in the library operating systems. The main insight behind the work was to overcome the hardware compatibility issue in library operating systems by writing drivers for prevalent virtual devices. The work introduced a clean-slate implementation of the entire system stack with a design approach of whole-system specialization and extreme minimization, as well as pervasive type-safety. By stripping unused kernel modules of a monolithic single-purpose virtual machine (VM), such an approach would provide the main benefit to have more secure, but also to be more efficient and easier to manage cloud applications. Furthermore, the work introduced an optimized language runtime for OCaml, a type-safe functional language, which allowed to have a robust implementation of the networking stacking, including low-level I/O, without severely compromising performance.

Other unikernel systems then emerged with different architectures specialized for different purposes. Examples include ClickOS, which is specialized for high-performance virtualized network middleboxes, and HermitCore which is specialized for HPC environments by supporting traditional parallel computing message passing libraries. Other unikernel systems were designed for compatibility with specific environments (such as IncludeOS, which is comptabile with C++) or more general source-level compatibility such as Rumprun (compatible with unmodified POSIX software) and OSv (which supports popular application stacks such as C, JVM, and Node.js). As the focus of this work is on Mirage unikernels, the rest of this section is devoted to its general architecture as well as Lwt, the lightweight threading library it uses.

2.2.1 MirageOS Architecture

Network applications deployed as Mirage unikernels are specialized at compile-time to single-address space VMs that include only necessary system libraries.

These unikernels can be compiled to run on multiple target hypervisors such as Xen and KVM, but also as POSIX binaries to run on Unix systems for debugging and prototyping purposes. Unikernels compiled as POSIX binaries can either include binaries to directly use the kernel socket subsystem or be compiled with Mirages OCaml TCP/IP stack with the required binaries to read Ethernet packets from a `tuntap` device. A recent proposal allows to run unikernels as confined processes [20], which can be desirable to leverage existing OS performance debugging and monitoring tools as well as to support nested virtualization use cases in today's cloud platforms.

The single-address space architecture allows to minimize the cost of data copying and user-kernel context switching. Furthermore, preemptive threading found in traditional operating systems is stripped away and replaced with an event-driven mechanism in which the single-threaded application will be notified asynchronously of external events such as I/O events or timeouts. The implementation of this mechanisms is dependant on the unikernel target and is extended with a lightweight threading library (Lwt) [21] written using OCaml. This library provides unikernel developers with the ability to write concurrent programs based on cooperative threads, also known as promises, which are run serially by an event loop engine. Therefore, the unikernel will be either continuously executing these lightweight threads or blocked waiting for the delivery of external events.

This single-threaded architecture is a direct consequence of the authors design decision to embrace a multi-kernel approach, where a unikernel is assigned a single vCPU, and scalability is achieved through instantiations of multiple VMs. The lightweight threads in Lwt can be assigned with priorities through thread-local variables, which allows custom scheduling to be implemented, for example, by rate limiting specific thread classes. In one sense and run as processes, the architecture of unikernels can be seen as single-threaded user-space networking applications with application-level thread management and scheduling which can be ported to multiple targets. This portability is achieved by interfacing the code of Mirage with generic modules such as `Netif` and `Blkif`, which can be used to implement target-specific networking and block device drivers.

2.2.2 Lwt Lightweight Threads Library

Lwt is a cooperative thread library written entirely using OCaml. In cooperative threading, threads are not preempted by the scheduler, but rather cooperate by yielding control explicitly through a call to `Lwt_main.yield` or implicitly when waiting for a blocking operation such as a blocking system call. Furthermore,

Lwt threads are much more lightweight compared to kernel threads which require a full stack. In contrast, Lwt threads are implemented as OCaml memory cells that hold the state representation of the threads and context switches are much cheaper as they involve no kernel intervention and require less function calls.

These threads are completely written using OCaml in a functional monadic style, but have semantics that resemble *futures* or *promises*. In the original paper [21] and early documentation, the *thread* term was used to convey a promise. However, the latest version of the documentation has adopted the promise term to avoid mistaking with the behaviour of system threads. When executing a promise, the simpler case is when it resolves immediately either terminating successfully with a returned value (*fulfilled promise*) or failing with an exception (*rejected promise*). However, if the promise represents an asynchronous operation it will return a *sleeping promise* which will act as a proxy for the original promise. Callbacks can be registered on this promise, which will be woken up when the result of the original promise is ready and resolved by Lwt.

Lwt threads can have three main internal states which correspond to the three types of promises we just mentioned: `Return v` (fulfilled promise with value `v`), `Fail e` (rejected promise with exception `e`) or `Sleep waiters` (sleeping promise with a set of callbacks `waiters`). Internally, a thread can be resolved only once by a call to `Lwt.wakeup_result t result`, which will terminate the thread into a fulfilled promise and the thread state will become immutable. This function will be either invoked directly by the main scheduler function `Lwt_main.run` or by the event loop engine which is also iteratively invoked by the scheduler through a call to `Lwt_engine.iter`. The scheduler will usually extract threads which have been explicitly yielded from designated queues, while the event loop engine will resolve any promises which were waiting for a timer or file descriptors to be ready.

The binding function `Lwt.bind` is the main compositional construct in Lwt which is also denoted as `>>=`. A call to `bind t f` will apply the second thread `f` on the output value `v` of thread `t` after it resolves successfully. Other sequential variants of bind exist such as `map` (binding a promise to a non-promise) and `catch` (binding to an error handler). Parallel composition functions such as `join l` allow the execution of a list of threads `l` concurrently and will resolve after all these threads have resolved. `pick l` will resolve with the first resolved thread and cancel others.

Functions such as `Lwt_main.yield` or `Lwt_main.pause` are usually called within a parent promise to give control to other threads at a specific point, to which

Listing 2.1: Recursive loop with yield points

```
let rec looper n () =
    match n with
    | 0 ->
        Lwt.return_unit
    | n ->
        Lwt_main.yield () >>=
        looper (n-1)
```

control will be returned in the next scheduler iteration. A call to such functions will result with adding this thread to a waiting queue and will return a sleeping thread, on which other threads can wait. In the next scheduler iteration, the sleeping thread will be resolved by the scheduler which will also wake up all promises waiting for this thread, allowing to execute the remainder of the original promise. Listing 2.1 illustrates a recursive loop which gives control to other threads at each iteration. The bind operation (`>>=`) is used to connect each iteration after the yielding point of the previous iteration.

Asynchronous I/O is implemented within the `Lwt_unix` module by wrapping blocking system calls with asynchronous promises. This is the only Lwt module where we have OCaml code which is linked to a C systems-level code. This is used in the Mirage Unix target to read and write from a file descriptor corresponding to the socket or network device. A call to `read fd buf` will immediately return a sleeping promise after creating a detached system thread performing the system call. The sleeping promise will be resolved when the read operation is ready, which will be tracked by the event loop engine `Lwt_engine`. The engine encapsulates an event loop library such as `libev` or a polling system call such as `select`. The engine also wakes up threads waiting for timers which can be used by calling `Lwt.unix_sleep interval`.

Chapter 3

Related Work

The contribution of this work can be split into two folds. The first is related to the tracing model based on hierarchically aggregated resource waiting times. More specifically, the contribution is the flexible tracing approach which is suitable to support centralized and decentralized policies simultaneously. The second is related to the implementation experience of the tracing model within Mirage. We believe this is the first attempt to describes an approach for automated resource accounting for application codes based on asynchronous promises or futures. This also includes the experience of instrumenting single-threaded user-level TCP stacks, and identifying the requirement for packet-timestamping.

In this chapter, a description of some of the previous works related to autoscaling microservice applications is provided, which were the inspiration behind our tracing model. We then describe how our work builds on previous works related to employing distributed tracing for dynamic resource management. Finally, we describe how our implementation relates to the implementation of relevant distributed tracing frameworks.

3.1 Autoscaling Microservices

The literature of auto-scaling of cloud applications has very little number of works that address the challenges of autoscaling service-oriented application architectures such as microservices. A recent survey [6] shows that literature is dominant by the suboptimal approach of considering the scaling requirements of a single service or application tier. Experimental approaches to configure autoscaling policies require extensive profiling of the application under a rep-

resentative workload, which can be a costly process given the large degree of freedom in workload and deployment options of microservices [22]. Such configurations are vulnerable to changes in microservice characteristics due to workload changes or new software releases [23, 24]. We believe our distributed tracing model, if deployed with lightweight virtualization options such as unikernels or containers [10], can be used to build generic autoscaling solutions that can work for arbitrary application structures and under dynamic workload changes, without the cost of highly-specialized predictive models [25].

The aggregated approach to address the auto scaling problem is directly inspired from the work in [26]. We adopt a similar approach which relies on hierarchically-aggregated performance values of an application structured as a directed acyclic graph (DAG) of services. However, the approach presented in this work is dependant on oversimplified service-level performance models, which are used to estimate performance gains or losses as a result of a potential resource allocation decision. Instead, our tracing model is based on measuring resource waiting times. We believe these provide a more generic and reliable alternative to performance models, as they provide ground truth measurements on which accurate and reliable *reactive* autoscaling policies can be developed. Furthermore, the facilities of distributed tracing allow a finer-grained approach for bottleneck detection, which takes into account the multiplicity of resource types and request classes.

3.2 Distributed Tracing for Resource Management

Recently, several works [3, 4, 5] have presented systems which employ distributed tracing techniques to perform dynamic resource management tasks for distributed systems. The focus of these works has been on enforcement mechanisms such as scheduling, rate limiting and admission control. These mechanisms are all related to regulating the usage of existing system resources. This problem is different from the autoscaling problem which is related to changing system capacity in response to changing user demand. To the best of our knowledge, this work is the first work to propose the usage of distributed tracing to address the autoscaling problem. In the context of microservice applications, the works in [4] and [5] show that the usage of distributed tracing mechanisms to regulate RPC queues is a more effective alternative to manual-tuned RPC libraries such as Twitter Finagle and Netflix Hystrix. As we outline in Subsection 4.1.2, our tracing model assumes that fairness and prioritization goals are

addressed with similar local resource management mechanisms.

Resource management mechanisms can be centralized [3] or completely decentralized [4, 5]. In a centralized strategy, traces are collected from individual system nodes and reconstructed in a central control node which enforces system-wide policies by remotely configuring local enforcement mechanisms within each node. In the decentralized counterpart, each node is responsible of configuring its own enforcement mechanisms based on local traces and align these configurations with those of its neighboring nodes. Our hierarchical tracing model can be seen as an enabler for both types of policies as well as for potentially hybrid policies.

3.3 Relevant Tracing Frameworks

Our instrumentation relates to implementations of some existing tracing frameworks, which are design for different purposes. Zipkin is a popular distributed tracing model based on Dapper [27] which employs metadata propagation. Zipkin trace model is based on a single span that represents both sides of an RPC call. The tracing model in this work is based on similar instrumentation points, which are taken on both sides of the call (more detail are provided in Subsection 4.2.4). In Zipkin, these trace points are collected in a centralized server. Differently, our trace model back-propagates measurements based on these points in the response of each RPC call.

Twitter Finagle [29] is an RPC library which can be configured based on request-related statistics propagated across the request path. Finagle is based on Scala Futures which can be considered very similar to Lwt lightweight threads used in Mirage. Similarly to Finagle, we use thread-local variables to introduce local contexts and propagate context data across RPC calls. However, our work involves additional instrumentation for the purpose of measuring waiting time during cooperative context switches and can be ported across various RPC-like protocols.

Finagle is based on JVM and is dependant on a an independent network stack, in contrast to Mirage which uses its own TCP stack. We show in Subsection 4.4 that this architectural difference introduces the requirement for packet-timestamping if reliable resource accounting is desired. We believe our experience in instrumenting Mirage is also relevant to network applications written as a composition of asynchronous functions [30] as well as applications structured using multiple isolated user-level single-threaded TCP stacks [31].

Chapter 4

Design and Implementation

In this chapter, an overall design and implementation of the tracing model is presented. Section 4.1 is devoted to the design goals, and how these goals are achieved through the two concepts of local and remote contexts. This includes a concrete example illustrating the usage of the model to support autoscaling policies. Section 4.2 introduces the generic execution wrappers which directly correspond to the two concepts of local and remote contexts. These wrappers encapsulate the trace points required to implement the tracing model within Mirage unikernels, and are used to instrument Cohttp, the HTTP library in Mirage. Section 4.3 goes into the details of how these execution wrappers are implemented within the Lwt lightweight threading library and how they integrate with the existing tracing module in Mirage. Finally, Section 4.4 highlights the requirement for packet-timestamping in the network devices used by Mirage unikernels in order to achieve reliable resource accounting.

4.1 Tracing Model Design

4.1.1 Microservices Concepts and Design Requirements

In this subsection we describe a running example representing a typical microservice application structure. We use this example to illustrate some of the concepts related to the microservices as well as the autoscaling problem. We also use this example in the following subsection to illustrate some of tracing model design concepts.

The running example is depicted in 4.1. This setup represents the simplest pos-

sible structure of a microservice application. This is composed of three classes of services: a client-facing service usually referred to as an *API gateway* which interfaces a *frontend service*. This service, in turn, is dependent on another *backend service*. A microservice application will have multiple frontend services each corresponding to a different subset of application functionality. Furthermore, any frontend service might be dependent on an arbitrary number of other services each of which may have further dependencies. As a result, the general structure of a microservice application will resemble a single-root directed acyclic graph (DAG).

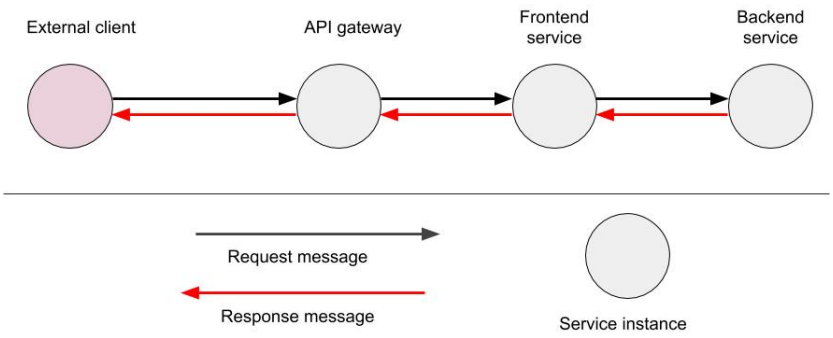


Figure 4.1: A running example of a simple microservice application.

We assume a synchronous inter-service communication style. The API gateway receives a request from an external client and initiates a single request to the frontend service. The API services relies on receiving a reply from the frontend service before it can respond to the client. Similarly, the frontend service depends on the backend service, which represents the final node in the request path. Given a request being processed by a service within the generic DAG structure, an *upstream service* is used to refer to one of the services from which the request had originated, while a *downstream service* is one of the dependent services by which the request has to be processed.

In a typical cloud environment, each service will be deployed within a virtual machine with its own virtual resources (e.g. virtual CPU and virtual network card). A service might be scaled *horizontally* by increasing the number of virtual machine instances of this service, or *vertically* by increasing the capacity of one of its virtual machines (e.g. increasing available network bandwidth). In the former case, a load balancer will distribute incoming requests among the available instances. An autoscaler or a resource manager is an external entity which can be configured in order to automatically take such resource allocation decisions. Such an autoscaler would continuously monitor the state of the application and would take decisions based on a user-specified autoscaling policy. Misconfigured

policies may lead to either under-utilized infrastructure due to over-provisioning of resources, or violating service-level-agreement (SLA) guarantee provided to the end user due to lack of sufficient resources.

Based on previous works related to configuring autoscaling policies of microservice applications (presented in Section 3.1), we identify that dynamic and real-time detection of resource bottlenecks within the application structure is a fundamental success factor for achieving effective auto-scaling policies. In the next section, we present a tracing model that aims to address this requirement in a generic and decentralized approach.

4.1.2 Design Goals, Concepts and Assumptions

The first design goal of our tracing model is to provide information related to performance bottlenecks in a microservice application based on synchronous RPC-like communication. The main purpose of this information would be to support resource allocation decisions made by an autoscaler. This autoscaler can be a single resource manager implementing a centralized autoscaling policy or can be composed of multiple resource managers implementing a distributed policy. Therefore, the second design goal is to make it possible for the generated traces to be collected and reconstructed in-band (i.e. locally) as well as out-of-band (i.e. centrally), so that it may be useful for both types of centralized and distributed autoscaling policies.

To achieve the first goal, two types of measurements are provided per request at the level of each service. The tracing model includes a measurement of the total request processing time as well as the waiting time for local resources. We focus on the the two resources of CPU and network as they are essential for any service. Processing time and waiting time measurements allow autoscaling policies to identify the opportunities of where time can be saved most to avoid a possible or actual latency-based SLA violation. The second goal is supported by aggregating resource waiting times at downstream services and propagating these values back to upstream services along with RPC responses. Given an application performance bottleneck, these values provide an upstream service with a local view of traces which is useful to distinguish whether it is the origin of the additional waiting time or it is one of the downstream services in the remaining request path.

We attach the aforementioned performance measurements as well as their aggregated values to the two concepts of a *local context* and a *remote context*. A Local context represents the request handler on a specific service while a remote

context is the client-side representation of the remote service invocation. A local context may have one or more child remote contexts which represent multiple RPC calls originating from a single request. Figure 4.2 illustrates an example of how these concepts can be projected on our running example. The backend service has a standalone local context as it has no further service dependencies. In contrast, each of the API and frontend services has a single local context with a single child remote context reflecting its single-service dependency.

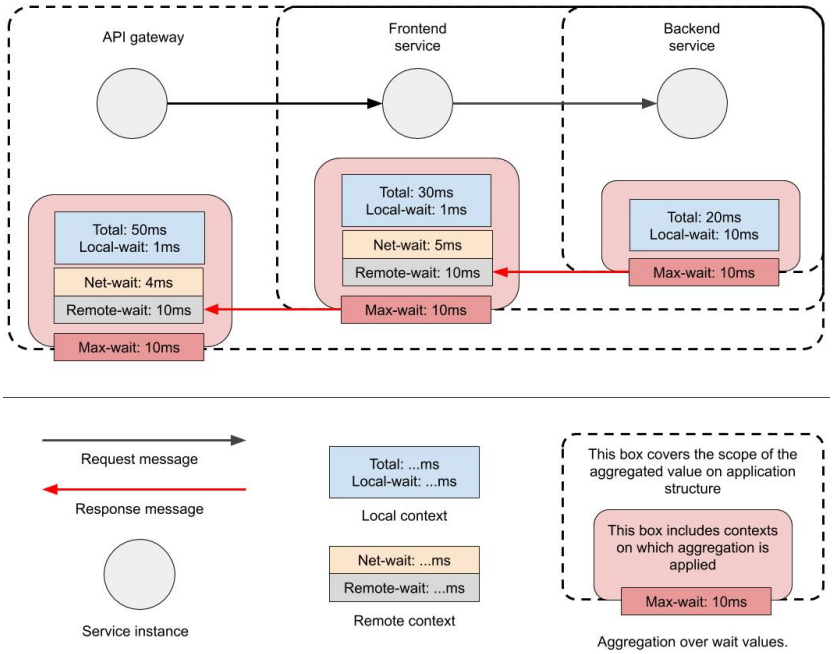


Figure 4.2: Illustration of waiting time aggregation in the tracing model

The illustration is applied on the running example in Figure 4.1. Each service is illustrated with its corresponding contexts beneath it. The red boxes and arrows illustrate the aggregation operations and propagation of the resulting aggregated values. The dashed boxes illustrate the scope of these aggregated values.

Local contexts hold the two measurements of `total` and `local_wait`. In the local context, `total` time represents the request processing time at this service while the `local_wait` represents waiting time during context switches. For example, the local context at the backend service in Figure 4.2, indicates that out of 20ms of processing time, 10ms are spent waiting to execute on the CPU. This time does not include network time for receiving the request and sending back the response, which is measured by `net-wait` on the corresponding remote context on the client side. In a remote context, `net-wait` represents time spent

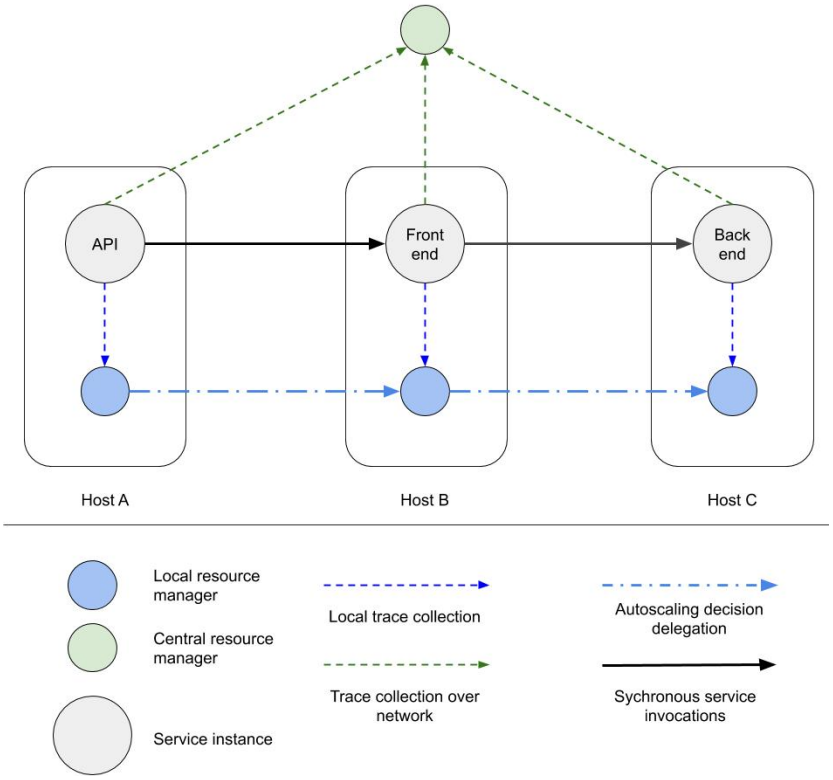


Figure 4.3: Illustration of local and central trace collection.

Local trace collection happens at the local host of each service, while centralized collection requires network transfer. Local resource managers can communicate as part of a decentralized autoscaling policy.

waiting time on the network link including connection establishment, sending the request and receiving back the response. The remote context at the frontend service in our example indicates 5ms of time spent waiting on the network link.

The other measurement held by a remote context is the `remote-wait` value, which represents the aggregated waiting time on resources belonging to all downstream services descending from the remote service including the resources of the remote service itself. Identifying the best approach to aggregate waiting time is beyond the scope of this work and is discussed in Subsection 4.1.3. For the sake of demonstration, we propose a simple aggregation policy based on maximum waiting time spent at any a single resource. This value is calculated as the the maximum of all local, network and remote waiting times at the remote end and propagated afterwards back to the client-side along with the network messages of the corresponding RPC response.

In an RPC call to a simple service with no further downstream services, the `remote-wait` will reflect a local bottleneck at the downstream service itself. In the running example, the 10ms of CPU waiting time in the backend service is propagated back and held in the corresponding remote context at the frontend service. However, if we have further service dependencies, as in the API call to the frontend service, `remote-wait` can be a result of waiting time at any resource on the path down to the backend service. In our running example, upon finishing request processing at the frontend service, the `remote-wait` received from the backend service (10ms) is larger than both network and local waiting times. As a result, we notice that the dominant CPU waiting time at the backend is propagated up to the API service.

The aggregated waiting times can allow a decentralized autoscaling policy to delegate the resource allocation decision to the next downstream service if the `remote-wait` is larger than waiting times on local resources (Figure 4.3). One approach to develop decentralized policies can be achieved by having the root service continuously measure request processing times, and trigger the process of resource re-provisioning based on specific thresholds (e.g. latency reaching 80% of SLA). The API service will either take a local resource allocation decision or delegate this process to one of the downstream services. In the example, the API gateway will identify that the remote waiting time is larger than both of `local-wait` and `net-wait` measurements, and will delegate the decision to the frontend service, which will also delegate again to the backend service.

The main purpose of the waiting times measured across the application structure is to guide the allocation or re-allocating of external resources in the most appropriate position within the application structure. We currently assume the resources of each service are completely isolated from one another. In other words, we currently assume no resource contention exists on the VM host and our focus is currently on detecting resource bottlenecks happening due to waiting times within the service boundaries.

Each service will generate a trace upon the execution of a local or remote context which will be available for collection by a local resource manager. Figure 4.3 illustrates how trace data can be collected by local resource managers as well as a centralized resource manager. We assume resource managers have additional metadata related to the application and network topology as they are responsible for allocating resources to their corresponding services. Therefore, it is possible to attach the amount of data transferred during a remote context, so that the resource manager can calculate the actual network bandwidth and compare it to the expected bandwidth of the link. This also requires internal mechanisms to ensure that no protocol-level bottlenecks are present (e.g.

underestimated TCP window size).

Classifying requests entering into the system is a common technique in distributed tracing and can be based on multiple customer classes or product functionalities. The introduction of multiple classes requires management policies and mechanisms that regulate local resources among these multiple classes. As this problem has already been addressed in previous works (Section 3.2), we choose to focus solely on the problem of allocating new resources and limit the current prototype to a single request class. In future work, the tracing model can be extended by additional measurements to support local resource regulation which are assumed to be analyzed on a smaller window scale (seconds compared to minutes) and from within the service, based on resource managers within the service [4].

4.1.3 Example Usage of the Tracing Model

In this subsection, we present an example usage of the tracing model based on a more general microservice structure. The example illustrates how the model can be used to explore interesting aspects of autoscaling policies of microservice applications. In the current prototype, the implicit interpretation of multiple remote contexts belonging to a single request, is that the corresponding service invocations are sequentially dependent on each other. This can be extended in future work by introducing wrapping contexts that indicate semantics of parallelism.

Figure 4.4 demonstrates how resource bottlenecks can be tracked in a DAG-structured microservice application. The application has a single-root service which sequentially issues two RPC calls to services B and C, both of which rely on backend services D and E respectively. The overall processing time, as seen by the root service, is 85ms. When this latency becomes closer a specific threshold, a resource manager of the root service can detect whether this is due to a local resource bottleneck or to a bottleneck in one of the downstream services. This resource manager would compare local waiting times to waiting times at the remote calls. In our example, local waiting time is less than waiting time spent at any of the two RPC calls and remote waiting times are larger than network waiting times for both RPC calls. The maximum waiting time in the RPC call to service C is 5ms and is due to an additional CPU waiting time on service E. However, we have a double of that time (10ms) on the RPC call to service B, which is due to a network bottleneck between services B and D.

Resource allocation decisions can be taken based on the different forms of ag-

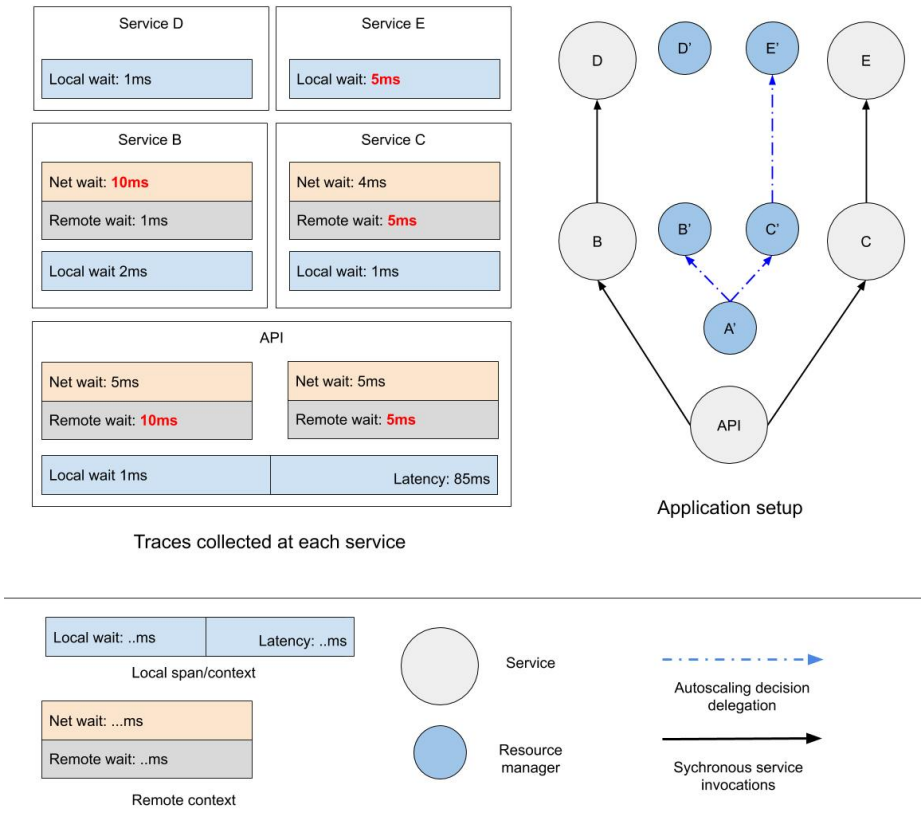


Figure 4.4: Example autoscaling decision process based on the tracing model.

The example service has a CPU bottleneck at service E and a network bottleneck at the link between services B and D. Traces (left) reflect these two bottlenecks by the additional waiting times (in bold red). Potential resource manager nodes participating in an autoscaling decision are illustrated on right.

gregated waiting times. The simple maximum aggregation suggested in the previous subsection might be desired if only one type of resource allocation steps is to be invoked. Another approach based on sum aggregation can be considered if multiple resource allocation actions can take place. In the latter case, the autoscaling decision will be delegated to resource managers of B and C simultaneously, and will be limited to B in the former case. An approach based on minimum aggregation can be also used to if a minimal resource allocation decision is required. Finally, more complex aggregations can be introduced to prioritize waiting times of specific resource types, for example, if these resources are associated with a smaller cost.

4.2 Trace Points within Mirage

In this section, we introduce generic execution wrappers corresponding to the concepts of local and remote contexts described in the previous section. These wrappers can be used to automatically introduce the trace points required to implement the tracing model within Mirage unikernels. We first describe how these wrappers are used to instrument the Cohttp library, as our current prototype implementation of synchronous service invocation is based on HTTP protocol. We then describe the trace points which are required to propagate request metadata and performance measurements between the two sides of the service invocation. We then move forward to describe trace points within the local context wrapper which can be used to measure the request processing time and waiting time during context switches. Finally, we illustrate how network waiting time is calculated based on measurements collected on both sides of the service invocation.

4.2.1 Overview of Cohttp Library Instrumentation

The local and remote context concepts introduced in the previous section can be used to represent any synchronous client-server communication mechanism. In other words, the execution wrappers are protocol-agnostic wrappers implemented within the Lwt library and can be used to instrument any appropriate libraries exposed through Lwt threads. In our prototype implementation, we instrument the the Cohttp library, the default HTTP library used in Mirage. More specifically, we instrument the HTTP server module as well as the HTTP client module which is used as a mechanism for RPC.

Figure 4.5 illustrates an overview of the usage of the execution wrappers. An

execution wrapper is essentially an Lwt thread that is used to wrap and instrument the execution of another Lwt thread. Local context wrappers are used to instrument request handlers in the HTTP server while remote context wrappers are used to instrument an HTTP client call. Furthermore, additional instrumentation points are introduced in order perform automatic propagation of request information. The HTTP client code is modified to propagate request identifiers into the headers of HTTP requests, and the HTTP server code is modified to propagate performance measurements in the HTTP response headers. Therefore, appropriate functions are introduced to facilitate data exchange between the execution wrappers and the instrumented library (these functions are described in Subsection 4.3.2).

No further instrumentation is required for any application code using the instrumented Cohttp library. Based on this instrumentation, all incoming requests as well as all outgoing HTTP calls will be automatically instrumented according to the tracing model and appropriate traces will be recorded using the existing tracing infrastructure of Mirage. Furthermore, request information will be automatically generated and propagated along the request path in any application structure where HTTP client calls are used for remote service invocations. Finally, we implement a simplified sampling mechanism in the HTTP server module of which its use is limited for the purpose of overhead evaluation.

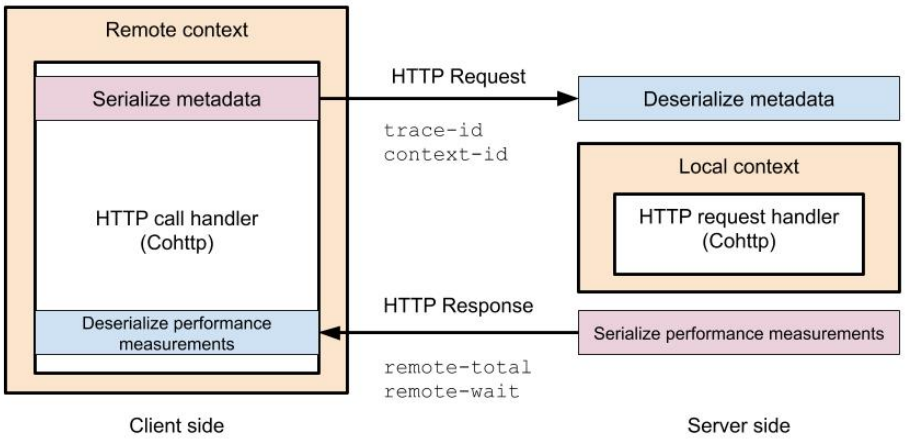


Figure 4.5: Overview of the usage of execution wrappers to instrument the Cohttp library

A local context wraps a request handler (server side, right) and a remote context wraps a client call (client side, left). Header fields used to propagate request metadata and performance measurements are illustrated below the HTTP messages.

4.2.2 Propagational Trace Points

A request spanning several services will be composed of several contexts. Requests and contexts are uniquely identified through randomly generated numbers. Each context will have its own unique identifier, and contexts belonging to the same request will have a shared identifier called trace ID. The first context at which the request is received is referred to as the root context. A unique trace ID will be generated for the first time when the incoming request enters into the system and will be stored in a local variable in the context wrapper. The root context will have its context ID the same as the trace ID. Any child context created from within this context will inherit the same trace ID, get a uniquely generated context ID and will have a parent context ID variable pointing back to the context ID of the parent context.

Figure 4.5 illustrates the request information propagated in both directions between the two sides of a service invocation. On the client side, serialization metadata data will be performed within a remote context, where this remote context will be wrapping the HTTP client call. The client side will extract the trace and context identifiers and propagate these two values as two HTTP request headers of `trace-id` and `context-id`. On the server side, these headers will be deserialized before the invocation of a request handler. This allows to instantiate the local context wrapping the request handler with the same trace ID of the remote context. The local context will have its own ID and have a parent context ID that points back to the ID of the remote context on the client side initiating the request.

In the opposite direction, the server will be propagating the two measurements of `remote-total` and `remote-wait` in the HTTP response headers. These values are extracted after the local context wrapping the request handler resolves. The `remote-total` value represents the total processing time of the request, while the `remote-wait` value represents the aggregated waiting time of this request up to this service (recall Figure 4.2). On the client side, the remote context will extract and store these two fields from the HTTP response headers. The processing time at the remote end (`remote-total`) is used by the remote context to measure network waiting time `net-wait` as we will explain in Subsection 4.2.4.

4.2.3 Trace Points within a Local Context wrapper

Local contexts are the default type of contexts and can be used to measure the total time a promise takes to resolve as well as the time it spends waiting

during context switches. In the current prototype, the context switch waiting time is limited to sequential composition of promises. However, we describe how local contexts can be used as building block to account for waiting time in asynchronous and parallel execution paths.

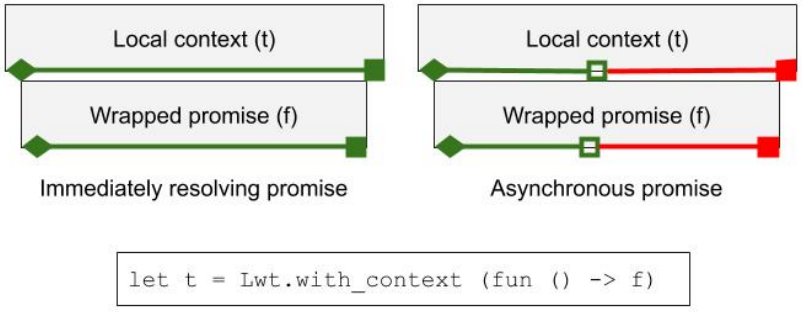


Figure 4.6: Trace points in a local context wrapper

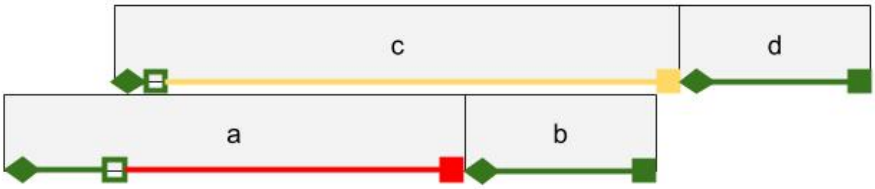
A full green diamond and a full green box are used to indicate the start and end of a promise. An empty box is used to indicate when a promise sleeps. The red line ending with a box indicates a sleep interval which ends when the the promise finally resolves.

A local context is essentially an Lwt promise that wraps the execution of another promise and tracks its execution until it is fully resolved. Figure 4.6 illustrates the two basic cases of a promise, which is either be resolving immediately or asynchronously. The promise should be passed to the `Lwt.with_context` wrapped within a function, which allows a timestamp to be taken before it is executed. If the target promise resolves immediately, the local context will end by resolving with the same result after taking another timestamp and measuring the total execution time of the promise. If the target promise sleeps, the local context promise will also sleep after registering a callback which will end the local context upon the wake up of the target promise. Having the local context resolve with the same state of the target promise allows inline instrumentation of existing libraries without altering the code which is dependant on the instrumented promises.

Accounting waiting time during context switches is currently limited to a promise that is a sequential composition of other child promises. We instrument Lwt main sequential composition functions (`bind`, `map`, `catch` and `try_bind`). Any of these functions is passed two promises and checks the internal state of the first promise. If it is already resolved, the second promise is applied over the result of the first promise immediately and no waiting time is measured in this case. If the first promise sleeps, the sequential operation will return a sleeping

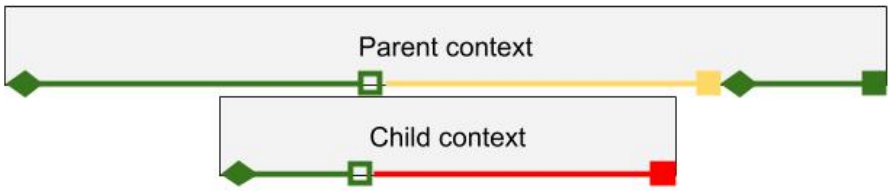

```
let t = Lwt.with_context (fun () -> a >>= b >>= c)
```

(a) Sequential promises




```
let t = Lwt.with_context (fun () ->
    let c = a >>= b in
    c >>= d)
```

(b) Nested sequential promises (only waiting time in red will be accounted)




```
let parent = Lwt.with_context (fun () ->
    ...
    Lwt.with_context (fun () -> f) >>=
    ...)
```

(c) Nested context waiting time (only waiting time in red will be accounted)

Figure 4.7: Trace points to account for promise sleeping time.

Illustration of the sequential composition scenarios addressed by the trace points. Refer to Figure 4.6 for diagram semantics.

promise as well, and a timestamp is taken before this point. The composition function, which is also a promise, will be woken up again after the first promise resolves. At this point another timestamp is taken and a thread-local variable within the local context is incremented with the corresponding time interval.

This can be considered a generic and automatic approach that allows to measure total waiting time for any sequentially-composed promise regardless of the cause of sleeping. However, the caveat is that the sleeping time within the last promise in the sequential composition is not tracked. Figure 4.7a illustrates a local context wrapping three promises connected sequentially through the bind operation (`>>=`). Both promises `a` and `b` sleep and this corresponds to two waiting intervals incremented to the a thread-local variable visible within the scope of the wrapping local context. As the third promise `c` resolves immediately, our accounting in this example will be accurate.

Our automatic instrumentation takes into account the compositional characteristics of Mirage libraries, where it is common for a promise which is part of a sequential composition to be also composed of a nested sequential composition. In such case, we will encounter a chain of sleeping promises where only the first waiting promise will account for the waiting time. An example of this case is illustrated in Figure 4.7b. The nested sequence (`a >>= b`) will be evaluated before the nesting sequence (`c >>= d`). Therefore, as the thread `a` sleeps first, this will also cause the promise `c` to sleep as well. To avoid accounting waiting time of promise `c`, the first sleeping promise will mark the context as sleeping, and only one timestamp will be taken regardless of the levels of nesting. When promise `a` is woken up, it will be the only promise to measure and increment a sleeping time interval.

Finally waiting times in child contexts are not accounted into the waiting time of a parent context. Our current usage of nested contexts is to wrap promises that handle and wait for external resources (RPC calls). We instantiate remote contexts to wrap HTTP calls within a local context representing the request handler. Therefore, this allows to avoid accounting waiting times for external resources as context switching time. Figure 4.7c illustrates an example where a child context is part of a bind in a parent context. While the parent will sleep waiting for the child context to resolve, this waiting interval will not be instrumented.

The local context with the trace points to measure waiting time can be used as a building block to account for waiting time in a non-sequential composition of threads, such as with parallel composition functions (`join` and `pick`) or asynchronous execution where we do not wait for a sleeping child thread (`async`).

Instead of directly measuring the total waiting time of the parent context, we would measure its actual execution time, which is the sum of the actual execution times of its asynchronous child promises. The actual time of each child be calculated by wrapping it with a local context and subtracting its waiting time from its total time. As we know all these child promises would need to be executed serially due to the single-threaded architecture of unikernels, the total waiting time of the parent context can be calculated by subtracting the measured actual execution time from its total time.

4.2.4 Trace Points for Network Waiting Time

To measure network waiting time in a service invocation, we use an approach similar to the popular distributed tracing framework Zipkin, which relies on four instrumentation points on both sides of the service invocation. These points are located as follows: a trace point (1) before initiating the call on the client side, two trace points (2) and (3) at the start and end of server processing the request, and finally a trace point (4) after fully receiving the response on the client side. The interval between trace points (1) and (4) can be captured by the total time of the remote context wrapping the HTTP call on the client side. The interval between trace points (2) and (3) can be captured by the total time of the local context wrapping the request handler on the server side. By propagating back the server processing time `remote-total` to the client side and subtracting this value from the total time of the remote context, we obtain the total time spent waiting on the network `net-wait`, including connection establishment as well as transfer of request and response messages.

Relying on recorded time intervals instead of timestamps allows to avoid the requirement for clock synchronization between the two sides of the service invocation. Using trace points on both of the client and server sides allows a more accurate estimation of network time compared to a measurement based on one side. A server-side measurement will exclude additional time spent on the client attempting to establish the connection in the case of dropped packets. Similarly, a client-side measurement will not be able to distinguish additional server processing time from a delayed network delivery of the first packets of a response message.

However, due to the single-threaded architecture of unikernels which embeds the TCP stack within the application code, a finer-grain instrumentation might be necessary to identify an artificial network bottleneck resulting from an overloaded CPU. Based on our experience we report in Subsection 4.4, we show that such instrumentation is possible based on packet timestamps collected from the

unikernel interface. However, our experience also shows that it is not possible to identify a CPU bottleneck affecting the TCP stack based on traces from within the unikernel. This is because a CPU bottleneck will cause packets to wait on the interface queue and it is not possible from within the unikernel to know whether a delayed packet is due to interface queuing or because of a genuine network bottleneck.

In order for an instrumentation of the TCP stack to be useful, it should be possible to access packet timestamps taken by network device from within the unikernel. This feature can be introduced to the network devices used by the unikernels in a similar way to the Unix sockets with the `SO_TIMESTAMP` flag. In such case, the current instrumentation can be extended by measuring the total time packets spend waiting on the interface for a given RPC call on both of the server and client sides. This allows to detect any CPU bottleneck affecting the TCP stack at both sides. This time can then be subtracted from `net-wait` to obtain true network waiting time. However, the packet waiting time on the client side cannot be propagated in the HTTP response headers as the network transfer will not have started yet. Instead, a custom HTTP implementation is required to propagate this time using HTTP trailer response headers.

Finally, our instrumentation scheme is most effective when both of the client and server sides are within the boundaries of the tracing system. However, when having an external client, the local context in the client-facing service will only measure server processing time and will exclude network transfer time to the client. Server-side measurement of network time allows the API gateway or root service in the microservice application, to perform a more accurate measurement of the actual QoS as experienced by the external client.

4.3 Implementation

In this section, the implementation of the local context and remote context execution is described. These wrappers are integrated into the existing Mirage tracing infrastructure in order to record traces that correspond to each local or remote context. We also describe the format of these traces, which hold request metadata information as well as performance measurements.

4.3.1 Implementation Methodology

The prototype implementation in this work is based on the Mirage `unix` target compiled with the `direct` network stack. Using this stack option is necessary

to make sure the current implementation is based on the same TCP stack used in other targets (such as the Xen `xen` and Solo5 `hvt` targets). We leverage the existing Mirage tracing module (provided by the `mirage-profile` package) to record traces at the end of each local or remote context. Execution wrappers are implemented within a forked version of the Lwt library which is integrated with the tracing module [1]. This fork is based on an older version of the Lwt library (v2.4.6 which is relatively old compared to the latest v4.2.1).

All introduced modifications are limited to the `core` and `tracing` modules within the forked Lwt library. The execution wrappers are protocol-agnostic and can be used to instrument libraries based on synchronous client-server communication which are exposed through Lwt threads. We use the local and remote context wrappers to instrument the a recent version of the Cohttp library (v2.1.1) and our instrumentation is limited to the `client.ml` and `server.ml` files within the `cohttp-lwt` module. This is the OS-independent package of the library which is used by Mirage. Therefore, the instrumentation of this library is portable across the various Mirage targets.

The `unix` target was essential to iterate over the initial instrumentation of the Lwt library and implementation of the local context wrapper. We first iterated our instrumentation over simplified OCaml programs before testing it on Mirage applications based on Cohttp. We debugged our traces by printing them out to the console, before integrating with the tracing module of Mirage (`mirage-profile`). Afterwards, an experimental setup described later in the evaluation (Chapter 5) was used to perform a more rigorous evaluation of the accuracy and overhead of the implementation. The initial results of these experiments helped us to identify several areas of improvements.

Due to the limited time of the project, the current implementation is still dependant on functions within the OCaml Unix module which is currently a build dependency for the Lwt `core` module. The OCaml Unix module is used for the two functionalities of random number generation and timestamping. Naturally, the next step would involve refactoring the implementation of the execution wrappers with generic interfaces where these target-dependent functionalities are provided by Mirage. As the overhead of the tracing infrastructure itself is not previously evaluated, a reproduction of the performance evaluation setup on the other targets is necessary to understand any overhead related to the tracing module itself. While is it possible to reuse most of the experimental setup of the Unix target for the Solo5 target, the tracing module is currently available only for the Unix and Xen targets.

[1] Available at `https://github.com/mirage/lwt` under the "tracing" branch

4.3.2 Implementation of Execution Wrappers

The `Lwt.with_context` is the main execution wrapper that corresponds to a local context, which wraps the execution of another promise until it is fully resolved. If the optional flag `~is_remote` is set, the wrapper will behave as a remote context, which is encapsulated through `Lwt.with_context_remote`. The main benefit of the execution wrapper is to hold request information such as request metadata and performance measurements. This is achieved by using the same mechanism used in the existing function `Lwt.with_value`, which provides a feature similar to thread-local variables in preemptive threading. Given a promise wrapped by a local or remote context, such a promise usually evaluates into a number of promises that are connected through sequential or parallel composition and all thread-local variables will be accessible to all child promises. When the execution wrapper is about to resolve, it will record a trace that includes the latest values stored in these variables.

An execution wrapper behaving as a remote context is mainly different from when behaving as a local context by the type of thread-local variables it uses. A local context wrapper uses the `total_sleep_time` variable to account for context switches time. This variable is used in our instrumentation of sequential composition functions (Subsection 4.3.3). The wrapper for a remote context will use the two variables of `remote-total` and `remote-wait` which can be set externally through the `Lwt.set_remote_measurements` function.

Multiple external setters and getters are used to expose these thread-local variables in our instrumentation of the Cohttp library. In the case of the HTTP client code, the `Lwt.set_remote_measurements` is used to save the remote performance values extracted from HTTP response headers. Furthermore, current trace and context IDs are extracted by the two `Lwt.get_trace_id` and `Lwt.get_context_id` functions in order to propagate these values in the HTTP request headers. In the HTTP server code, the `Lwt.extract_context_info` is used after a request handler resolves in order to extract the `remote-total` and `remote-wait` values, which are going to be propagated back through HTTP response headers. FInally, trace and context IDs extracted from HTTP request headers are passed as parameters to the local context wrapper, which are initialized before the execution of the promise.

Thread-local variables for an active context are stored in a hashmap that is held by a global mutable variable `current_data` accessible from within Lwt `core` module. When any child promise sleeps, it will save the current state of `current_data` and restore this state when it wakes up. A thread-local variable is defined through a key which is created by the `new_key` function. The `get`

Listing 4.1: Implementation of the local context wrapper

```
let with_context
    ?trace_id ?remote_span
    ?(is_remote=false) ?remote_host
    ?(is_parent=false) f =

    (* initialization of trace_id, span_id and parent_span_id ...
    *)

    let start = Unix.gettimeofday () in
    let result = f () in
    let t = repr result in
    match check_state t with
        | Return v ->
            end_of_context start ;
            current_data := parent_data;
            return v
        | Fail exn ->
            end_of_context start ;
            current_data := parent_data;
            fail exn
        | Sleep sleeper ->
            let res = temp t Bind in
            add_immutable_waiter sleeper (fun result ->
                end_of_context start ;
                (* restoring the previous waiting context *)
                current_data := parent_data;
                set_child_sleeping (Some false);
                match result with
                    | Return v -> connect res (return v)
                    | Fail _ as state -> fast_connect res state
                    | Sleep _ | Repr _ -> assert false;
            );
            (* going back to parent context *)
            current_data := parent_data;
            set_child_sleeping (Some true);
            res
        | Repr _ ->
            assert false
```

function takes a specific `key` in order to look up and return the value associated with that key. We introduce `set` function for internal use from within Lwt and only export the values of specific keys.

Listing 4.1 illustrates the `with_context` function which implements the context wrapper. A local context will be called from a parent context. If no parent context, `current_data` will represent data of the main event loop. In order to account for total time a promise requires to resolve, a timestamp should be taken at the beginning and end of the local context. The first timestamp is taken just before the execution of the promise, which might resolve immediately or go to sleep. When the promise resolves immediately. the `end_of_context` function is used to calculate the elapsed time and record appropriate traces. Furthermore, a snapshot of thread-local variables of the parent context is taken before entering into the new context (held by `parent_data`), which is used to restore previous state when returning to the parent context after the promise resolves or sleeps.

If the promise returns a sleeping thread, the `local_context` function will also return a sleeping thread (`res`), which will be resolved when the instrumented promise resolves. This is achieved by registering a callback that is added to the list of threads waiting for the instrumented promise to be resolved in the promise internal representation (`Sleep sleeper.waiters`). When this callback is invoked, `end_of_context` function is called and the `res` is resolved to the same state of the promise using the existing internal `connect` function.

At the end of each remote context, a single trace representing the RPC call is recorded after calculating `net-wait` and `remote-wait`. These measurements for all remote contexts executing within a local context will be saved within a hash map, so that they would be extracted again when the parent local context resolves. At that point, these measurements will be discarded after an aggregated waiting time will be calculated. The aggregated value as well as the total time of the local context will be stored in another hash map `context_store` (not visible in the listing). This will be be used by `extract_context_info` to extract request information based on a trace ID, and the corresponding hash entry will be removed afterwards.

4.3.3 Accounting Context Switch Time

We instrument Lwt main sequential composition functions (`bind`, `map`, `catch` and `try_bind`) to measure the waiting time during the sleep of any promise in a sequential chain of promises. This is achieved by instrumenting the `add_immutable_waiter_res`

Listing 4.2: Implementation of bind function (Lwt library)

```
let bind t f =
  let t = repr t in
  match check_state t with
    | Return v ->
        f v
    | Fail _ as state ->
        thread { tid = current_id (); state }
    | Sleep sleeper ->
        let res = temp t Bind in
        add_immutable_waiter_res res t.tid sleeper
          (function
             | Return v -> connect res (try f v with exn -> fail
    exn)
             | Fail _ as state -> fast_connect res state
             | Sleep _ | Repr _ -> assert false);
        res
    | Repr _ ->
        assert false
```

function which exists in the implementation of all these functions Listing 4.2 illustrates the usage of this function in the implementation of `bind`. This function is used if the first out of the two threads passed to the composition function returns a sleeping thread. In such case, the composition function returns a waiting thread (`res`), and registers a callback that is called back when the first thread resolves. The callback applies the second promise over the values returned by the first promise before it resolves the waiting thread `res` into the same thread representations using the internal `connect` function.

Listing 4.3 illustrates our instrumentation of the `add_immutable_waiter_res` function. We firstly check whether there is an existing trace ID, which will be absent when no instrumentation is active. This allows to reduce the overhead when sampling is active by calling the original code before instrumentation. If there is a trace ID, a timestamp is taken at the beginning, which corresponds to the start of the sleep interval. The `add_immutable_waiter` is called to do the actual registration of the callback on the wake up of the sleeping thread. In order to measure the time the promise has spent sleeping, we wrap the callback with a function that takes another timestamp before it passes the execution to the callback. Each sleep and wakeup will increment the thread-local variable `total_sleep_time` via the `inc_sleep_time`, which will be accessed at the end of the context to obtain the total waiting time.

Listing 4.3: Instrumentation of sequential composition functions

```
let add_immutable_waiter_res res tid sleeper cb =
    let vars, _tid = !current_data in
    let parent_trace = match get trace_key  with
        | None -> -1L
        | Some id -> id
    in
    if parent_trace == -1L then begin
        add_immutable_waiter sleeper (fun result ->
            current_data := (vars, (repr res).tid);
            cb result)
    end
    else begin
        let start = Unix.gettimeofday () in
        let is_first_sleeper =  match get_going_to_sleep () with
            | None -> set_going_to_sleep (Some true); true
            | Some t -> match t with
                | true -> false
                | false -> set_going_to_sleep (Some true); true
        in
        let is_child_sleeping = match get_child_sleeping () with
            | None -> false
            | Some t -> t
        in
        add_immutable_waiter sleeper (fun result ->
            if is_first_sleeper && (not is_child_sleeping) then
    begin
                current_data := (vars, (repr res).tid);
                let finish = Unix.gettimeofday () in
                let slept_for = finish -. start in
                inc_sleep_time slept_for;
                set_going_to_sleep (Some false); ()
            end else begin
                let sleep_time = get_sleep_time () in
                current_data := (vars, (repr res).tid);
                set_sleep_time sleep_time; ()
            end;
        cb result)
    end
```

4.3.4 Trace Format and Serialization

The tracing infrastructure provided by Mirage tracing module (`mirage-profile` package) is implemented in a similar way to the Linux Tracing Toolkit (LTT) approach. Traces are stored and grouped as a set of events in a small number of fixed-size packets. This allows to free up space efficiently by overwriting whole packets when storage is full. In the current prototype based on Mirage Unix target, traces are stored into a local file.

The forked Lwt library and the TCP stack were originally instrumented for the purpose of debugging performance problems in unikernels with a visual tool. Trace points within these libraries produce a large number of traces and were disabled as they introduce a significant overhead while serving a different purpose. These traces were disabled by modifying the tracing module to use a null tracer that replaces all previous types of trace points with an `ignore` function. The null tracer is then extended in the Mirage tracing module to include the two types of trace records, corresponding to local context and remote context wrappers.

The event header will include a 64-bit timestamp as well as a unique ID (8-bit) which enumerates the type of the trace (either correspond to a local context or to a remote context). Traces corresponding to a local context include two measurements of `local_wait` and `total`. A trace of a remote context will have the three values of `net_wait` and `remote_wait` as well as `total`. All measurements are serialized as standard 64-bit floating-point values (IEEE 754) with little endian byte order. The `ocplib-endian` is used to perform the serialization of trace fields. The format of the traces is serialized according to the Common Trace Format (CTF). Based on the trace file as well as a metadata file, traces can be analyzed by CTF-based tools such as Babeltrace.

Trace and context IDs are implemented as 64-bit integers. These are serialized and deserialized in the Cohttp library through OCaml `Int64` functions of `of_string` and `to_string`. Performance measurements are handled as `double` values are serialized using OCaml's `float_of_string` and `float_of_string`. More compact and efficient serialization can be considered when more amount of data is serialized over the network.

4.4 Requirement for Packet-Timestamping

In this section, an additional difficulty to fully implement the trace model in Mirage is described. This difficulty is imposed by the single-threaded architec-

ture of Mirage. A bottleneck in the event loop would lead to a delay in network transfer, as this bottleneck prevents the TCP stack from processing packets in a timely manner. We show that packet timestamps as collected from the network device attached to the unikernel can be used accurately address this problem. However, we also show that accurate timestamps cannot be obtained by instrumenting the Mirage TCP stack. We draw these conclusions from a set of exploratory experiments we conducted. We analyze packets captured from the network device attached to the unikernel as well as traces of a parallel instrumentation in the Mirage TCP stack. In future work, packet-timestamping feature in the network devices used by unikernels can be used to address this problem and integrated into the tracing model as described earlier (Subsection section:network-trace-points).

4.4.1 Experimental Setup

A performance evaluation of a simple Mirage TLS web server[2] is conducted using `httperf` workload generator. The generator and the unikernel are deployed over two machines connected through a 10 Gbps ethernet link. Full details of the machines are provided in Subsection 5.2. The Mirage unikernel is compiled to the Unix target using the full Mirage TCP/IP stack. The unikernel is attached to a Linux `tuntap` interface and host networking is activated and configured on the client machine so that the static unikernel IP address would be reachable.

`httperf` is a single-threaded event-based HTTP workload generator that establishes a new TCP connection for each request. Connections are generated using an internal timer according to a user-specified rate. We configure the generator to issue requests to a small file of 59 Bytes over SSL. All our experiments involve increasing the rate of generated connections (using steps of 5 or 10 req/sec) until we encounter a performance bottleneck, and further increasing this rate for a short range after this bottleneck. We run all experiments over 10 seconds after ensuring average latency does not change significantly after this period. We also run experiments for a sufficient number of runs (5 to 10) and report the median values of httperf measurements.

Httperf reports the average of reply time (interval between the first response received from server since the request has been sent) and as well transfer time of server response. We measure latency as the sum of both values which does not include connection establishment. Some experiments involve a CPU bottleneck, which is introduced through a loop over an arithmetic operation in the request

[2] Available at `https://github.com/mirage/mirage-skeleton/blob/master/applications/static_website_tls/`

handler. Some experiments also involve a network bottleneck, which is introduced by configuring one of the ethernet interfaces using Linux traffic shaper. We uncover the interleaving of packet traces belonging to different requests by using the port number of the client side.

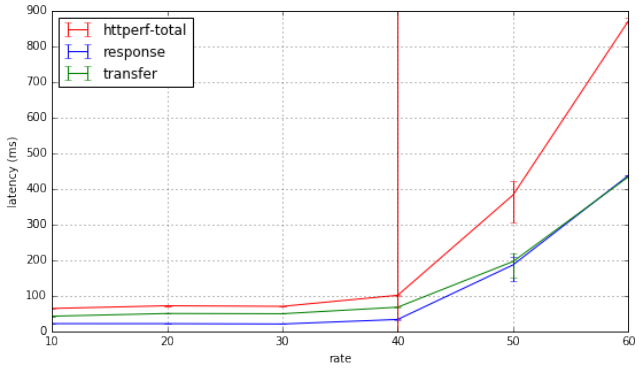
4.4.2 Trace Analysis

An artificial CPU bottleneck is introduced within the request handler. This leads to a bottleneck that is visible through latency measurements by httperf (figure 4.8a). Although this a CPU bottleneck, this also leads to a bottleneck in network transfer time measured on the side of the workload generator. This can be explained by the single-threaded architecture where a bottleneck in the event loop due to processing CPU-bound requests would lead to a delayed processing of acknowledgement packets and thus an artificial delayed transmission of data.

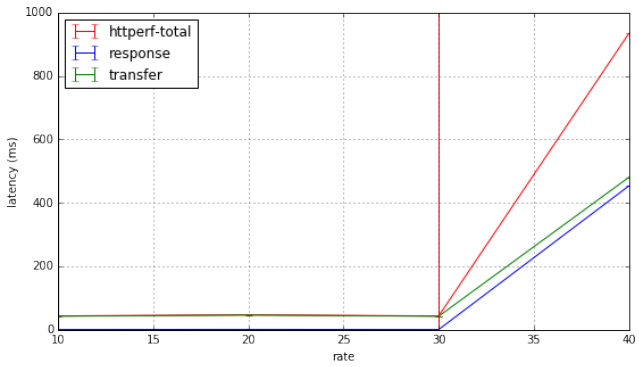
To address this problem, we propose to do a finer-grain measurement of network transfer based on packet arrival data on the network device. We calculate time difference of all ingoing and outgoing packets, which represent the time difference between packet arrival on the interface and the time of the previous packet (regardless of its type). We define server time as the total time difference for outgoing packets and client time as the total time difference for incoming packets. We use the server time to estimate the CPU-bound portion out of the timeline of a TCP connection, whereas client time to estimate the time spent waiting for the network.

Figure 4.9 illustrates how it is possible to use the two estimations of server and client time to distinguish an artificial network bottleneck from a genuine one. In figure 4.9b, a network bottleneck is introduced, and this bottleneck is detected using both client time based on packets as well as through httperf latency. The estimated server time correctly remains constant indicating a true network bottleneck. In contrast, when we introduce a CPU bottleneck as in figure 4.9a, the bottleneck is reflected in httperf measurements and server time. We observe a slight increase in network time which is insignificant compared to the increase in server time, which confirms this is largely a bottleneck in the event loop that is causing a delay for outgoing packets.

We instrument the Mirage TCP stack in a similar manner to calculate waiting time based on the timestamps of TCP segments packets, obtained as early as possible in the packet processing path. Figure 4.10 illustrates server time as measured from within Mirage with the same CPU bottleneck introduced in 4.9a. This time, traces from within Mirage indicate a constant TCP server time



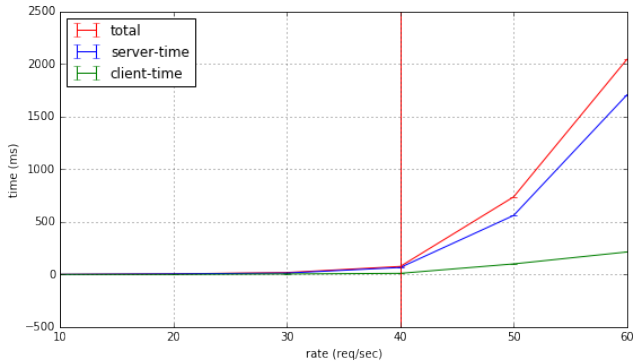
(a) CPU bottleneck



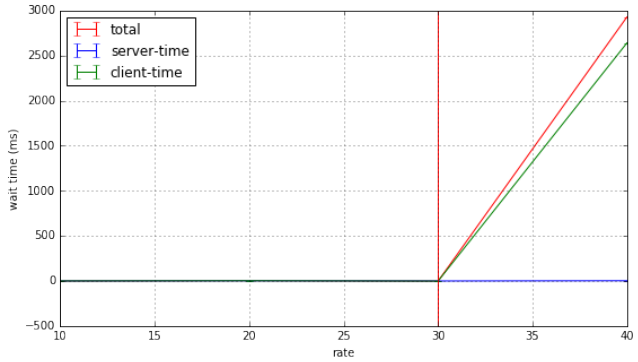
(b) Network bottleneck

Figure 4.8: Httperf benchmark latency under CPU and Network bottleneck in Mirage

The rate threshold at which the bottleneck occurs is illustrated by a vertical red line. Refer to Subsection 4.4.1 for response and transfer values.



(a) CPU bottleneck



(b) Network bottleneck

Figure 4.9: Detection of an artificial network bottleneck based on packet timestamps

Server time increases only on a CPU bottleneck. Client times increases only with a genuine network bottleneck.

despite the the CPU bottleneck introduced. A closer investigation of the TCP stack code and packet traces reveals that packets are transmitted immediately in response to an ACK. The actual delay happens outside the unikernel when reading the ACK packet itself from the network device. A bottleneck will cause the delay of reading and processing packets and the unikernel will have no information about the actual arrival time of the packet on the network device.

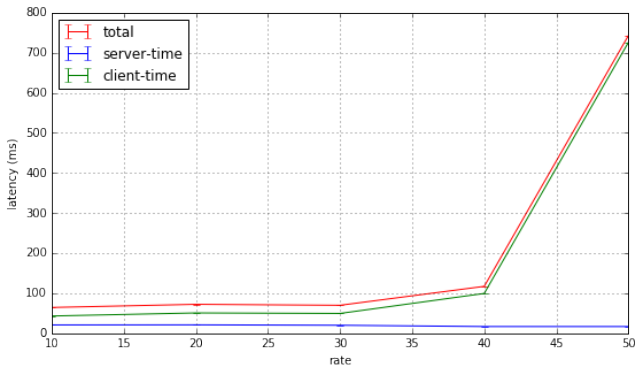


Figure 4.10: Mirage TCP stack trace analysis under a CPU bottleneck

Client time (measured from within the unikernel) increases despite a CPU bottleneck while the server time remains constant.

Chapter 5

Evaluation

This chapter presents the evaluation of our tracing model prototype implemented in Mirage. We first present the goals of the evaluation in Section 5.1, which is followed by the details of the experimental methodology and setup information in Section 5.2. In Section 5.3, we evaluate the accuracy of our instrumentation by illustrating the tracing model capability to detect bottlenecks. This is followed by Section 5.4, which presents our evaluation of the tracing model overhead. Finally, Section 5.5 illustrates the effect of the number of cooperative context switches on our experiments.

5.1 Evaluation Goals

The first goal of the evaluation is to illustrate the usage of the tracing model as well as the accuracy of the performance measurements which are based on the tracing points introduced by the execution wrappers. The second goal is to measure the overhead on throughput and the extent which sampling can reduce this overhead. We use the an application setup which resembles the running example presented in Section 4.1.1 for both goals.

For the first goal we illustrate how the tracing model can be used to distinguish a CPU bottleneck from a network bottleneck present at the backend service, and how this bottleneck is correctly mirrored at the traces collected at the frontend service and the API gateway.

For the second goal, we measure the overhead of the tracing model given our instrumentation of the Cohttp library is present at all of these services. When sampling is considered, we consider a sampling rate of 1 to 1024 requests (sim-

ilarly to [27]). Sampling is currently activated at each service independently for the purpose of overhead evaluation only. Performance is compared to baseline application performance configuration, where no instrumentation is present at the Cohttp library and the tracing model is not activated in the unikernel configuration file.

5.2 Experimental Methodology

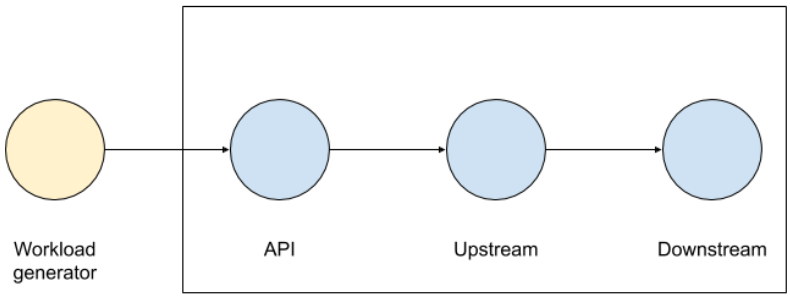


Figure 5.1: Evaluation experimental setup

5.2.1 Benchmark Setup

In all our experiments, we use a simple application structure composed of a chain of three services: an API service, a frontend service and a backend service. In order to fulfill a request, the API has to perform a single HTTP call to the frontend service, which in turn performs another single call to the backend service. We provide more details about the application code in the following subsection. Each service is connected to a `tuntap` network device and compiled with the Mirage TCP stack with the Unix target. All services are located on the same 6-core 2.40GHz machine with 64GB of RAM, which leaves a large room to eliminate CPU interference as each unikernel will not occupy more than a single core at any one time. The API service receives requests from the workload generator located on another machine over a 10 Gbps Ethernet network link. Static IPs are used in all network communications with no name resolution involved. Host networking is enabled to allow the hosts to deliver packets to the unikernels. Both hosts have Linux Ubuntu 18.04.1 LTS with kernel version 4.15.0-50.

We use the `wrk` HTTP benchmark which establishes a number of TCP connections at the beginning of the benchmark. These connections are continu-

Listing 5.1: Cooperative loop

```
let rec looper n () =
    match n with
    | 0 ->
        Lwt.return_unit
    | n ->
        Lwt_main.yield () >>=
        looper (n-1)
```

ously used to issue HTTP 1.1. keep-alive requests by a number of specified threads over a specified time interval. The usage of `wrk` allows to achieve higher throughput compared to `httperf` which is single-threaded and does not reuse any established connection. By avoiding connection establishment and using multiple threads, `wrk` is able to stress the API unikernel and therefore we consider it more suitable for our overhead evaluation. We assign a thread for each connection, and measure bandwidth and latency as we increase the number of open threads/connections.

We stress the application to its maximum throughput for both purposes of bottleneck detection and overhead evaluation. A CPU bottleneck is introduced by stressing the application beyond its maximum throughput. A network bottleneck is introduced by limiting the bandwidth of a single `tuntap` device through Linux traffic shaper `tc`. The benchmark runs for 15 seconds which we find sufficient to have well-converged measurements.

5.2.2 Application Code

We extend the Mirage TLS web server[1], without using TLS. Listing 5.2 illustrates the request handler code which is shared across the three services. The request handler performs a pattern matching on the request URI, where each URI corresponds to the service on which it is invoked. The backend service represents a rather simple service that performs a cooperative loop before returning a small HTTP response (several bytes). The loop is configured to be a sequential code block with a number of yield points (`Lwt_main.yeild`) (Listing 5.1). Each of the frontend and API services uses the HTTP client `get` function to invoke its corresponding dependency and responds with the same response received through the remote call.

This code is automatically instrumented based on our instrumentation of the client and server modules within the Cohttp library. The handler is automat-

[1] Available at `https://github.com/mirage/mirage-skeleton/blob/master/applications/static_website_tls/`

Listing 5.2: Request handler code shared across the services

```
1
2  let dispatcher get_ctx uri =
3      match Uri.path uri with
4      | "/downstream" ->
5          looper 1000 () >>= fun ()->
6          S.respond_string ~status:`OK ~body:"downstream-service" ()
7      | "/upstream" ->
8          get_ctx >>= fun raw_ctx ->
9          let resolv, cond = raw_ctx in
10         let ctx = Client.ctx resolv cond in
11         Client.get ~ctx (Uri.of_string "http://192.168.2.2:8080/
       downstream")
12         >>= fun (_resp,body)->
13         Cohttp_lwt.Body.to_string body >>= fun (body_str)->
14         Http_log.info (fun f -> f "body␣[%s]" body_str);
15         S.respond_string ~status:`OK ~body:body_str ()
16     | "/api" ->
17         get_ctx >>= fun raw_ctx ->
18         let resolv, cond = raw_ctx in
19         let ctx = Client.ctx resolv cond in
20         Client.get ~ctx (Uri.of_string "http://192.168.1.2:8080/
       upstream")
21         >>= fun (_resp,body)->
22         Cohttp_lwt.Body.to_string body >>= fun (body_str)->
23         Http_log.info (fun f -> f "body␣[%s]" body_str);
24         S.respond_string ~status:`OK ~body:body_str ()
```

ically wrapped by a local context and an HTTP client call is automatically wrapped by a remote context. The handler itself can be considered a sequential composition based on the `bind` function, which also has nested sequential compositions (concepts of which were illustrated earlier in Subsection 4.2.3).

In the experiments related to bottleneck detection, we tune the number of loops in the `looper` function to 1,000. This number directly corresponds to the number of yield points within a single request in the backend service, which represents the number of cooperative context switches. This high number creates a significant context switch cost which helps us to illustrate the capabilities of the tracing model under a CPU bottleneck without causing an artificial network bottleneck. Subsection 5.5, provides more details related to how performance differs in comparison to a 100 yield point configuration.

5.2.3 Trace Data Collection and Processing

Each unikernel stores its traces in a separate file, and these traces are also processed separately using the python API of `babeltrace`. For the API and frontend services, we have traces of local contexts representing the request as well as remote contexts representing the HTTP calls. In the backend service

we have traces of the former type only. We report the median values of the fields found in these traces: the values of `total` and `local-wait` in traces of local contexts, and the values of `remote-wait` and `net-wait` in traces of remote contexts.

For the overhead evaluation experiments, we compare the median of `wrk`throughput between the baseline configuration and when instrumentation is active with 5 benchmark runs. For the bottleneck detection experiments, we collect data using full sampling mode. We include the median based on a single benchmark run to allow a one-to-one comparison between the median obtained from the traces and the average reported by the benchmark for the same experiment. Also, to make sure sampling does not affect our ability to detect resource bottlenecks, we calculate median values based on a sample size of the dataset similar to the size used in overhead evaluation, including 1 request trace for every other 1024 trace.

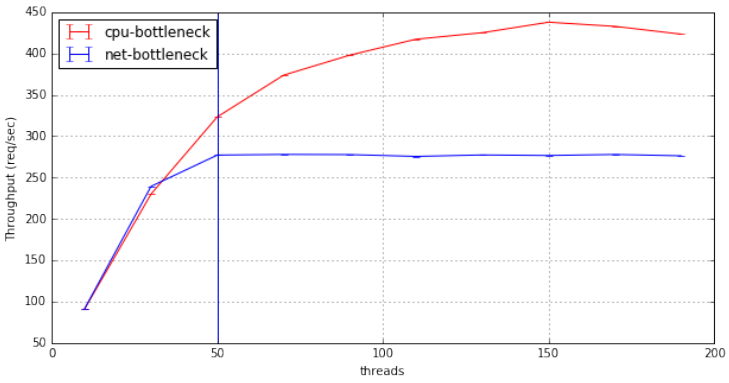
5.3 Bottleneck Detection

In this section, we illustrate the resource bottleneck detection feature of our tracing model. We analyze the traces collected from each service for two setups corresponding to a CPU bottleneck and a network bottleneck, both introduced at the backend service. A CPU bottleneck is introduced by stressing the application with more client threads until reaching the maximum throughput. We attribute this CPU bottleneck to the backend service based on our tuning of number of yield points in Subsection 5.5, where it has a larger amount of request processing compared to the other services. The network bottleneck is introduced by limiting the bandwidth of the network device used by the backend service. Traces from the backend service are used to identify the CPU bottleneck, whereas a network bottleneck will be always detected from the frontend service side according to our tracing model. Furthermore, we show that traces collected from the frontend and API services can be used to construct a consistent service-centric view of the traces.

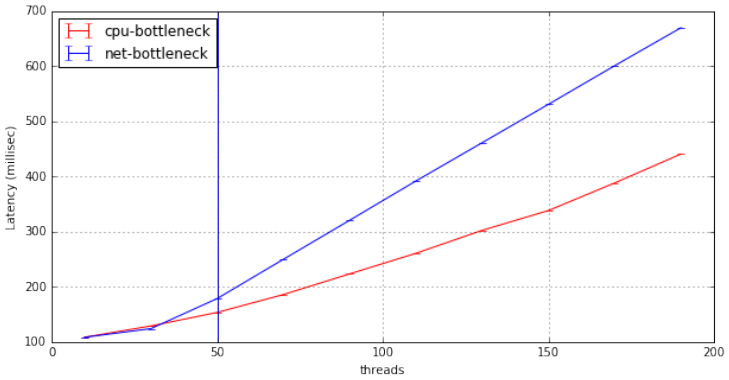
As we increase the number of threads and reach the maximum benchmark throughput, requests start to experience increased latency. We evaluate our measurements by comparing them to the benchmark latency and making sure they are consistent with our assumptions about the sources of the resource bottleneck. Given a CPU bottleneck, we show that `local-wait` increases in parallel to the request latency while `net-wait` remains constant. In reverse, and under a network bottleneck, request latency increases in parallel to `net-wait` while

`local-wait` remains constant. In both cases, we show that the bottleneck in latency is accurately propagated back to the frontend and API services through the `remote-wait` value. The total request processing time `total` at the API service should be always consistent with benchmark latency. However, the `total` value on the frontend and backend services will vary depending on the type of resource bottleneck.

5.3.1 Benchmark Performance



(a) Throughput



(b) Latency

Figure 5.2: wrk Benchmark performance under a CPU and network bottleneck

The blue vertical line indicates the threshold at which the effect of the network bottleneck becomes visible

Figure 5.2 illustrates the benchmark performance under the two configuration modes. Under the CPU bottleneck mode, the application approaches a maximum throughput slightly less than 450 req/sec at 150 concurrent clients. A network bottleneck causes the application throughput to level at 300 req/sec

starting from a much smaller number of 50 client threads. In both configurations, request latencies increase almost linearly as we increase the number of concurrent clients. This is due to the single-threaded architecture of unikernels. The introduction of a network bottleneck causes this latency to be higher after the bottleneck takes effect at the threshold of 50 concurrent clients.

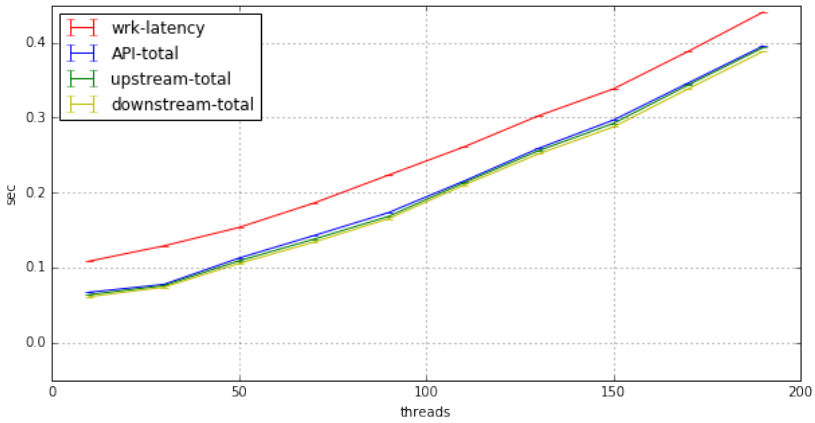
Figure 5.3 compares request processing times across all three services to the request latency as measured by the benchmark. Under a CPU bottleneck, all processing times increase in parallel with the request latency. This is consistent with our assumption of a CPU bottleneck causing an increasing processing time in the backend service. However, under a network bottleneck, the processing time at the backend service remains constant after the network bottleneck threshold at 50 concurrent. This is also consistent with our assumption that this is due a network bottleneck between the backend and frontend service, and therefore we have an increased processing time in the frontend and API services only.

5.3.2 Waiting Times under a CPU Bottleneck

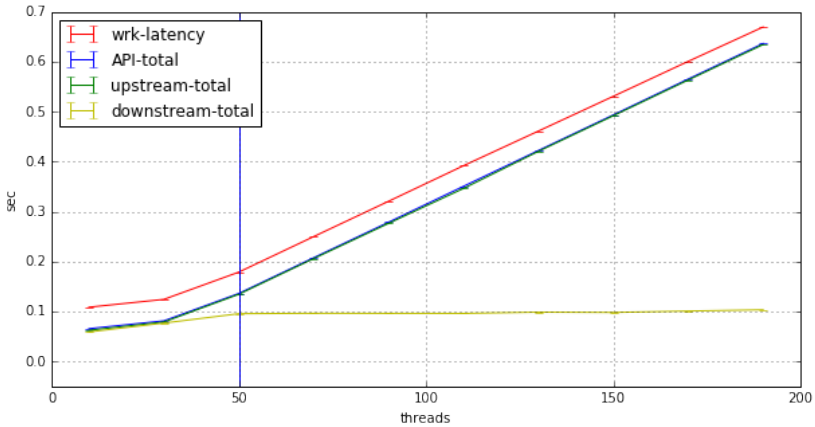
The `local-wait` measurement can indicate an additional waiting time at the event loop, which directly corresponds to a CPU bottleneck due to the single-threaded architecture of unikernels. Figure 5.4a shows how the increase in latency is parallel to the increased waiting time at the backend service. In the traces generated by the frontend and API services (Figures 5.4b and 5.4c), both of the local and network waiting times remain constant during the CPU bottleneck at the backend service. In contrast, the remote wait measurement correctly mirrors the backend bottleneck.

5.3.3 Waiting Times under a Network Bottleneck

When a network bottleneck is introduced, the local waiting time at the backend service remains constant after reaching the threshold corresponding to the the network bottleneck (vertical line in Figure 5.5a). Traces at the frontend service (Figure 5.5b) will detect an increasing network waiting time starting from 50 client threads (blue vertical line), which will remain smaller than the local waiting time at the downstream service until the threshold of 70 clients (green vertical line). Only after this threshold will the network waiting time be propagated back to the API service (Figure 5.5c). Remote waiting time as found in traces collected from the API, will reflect the maximum waiting time that can be saved in a autoscaling decision involving a single resource type.

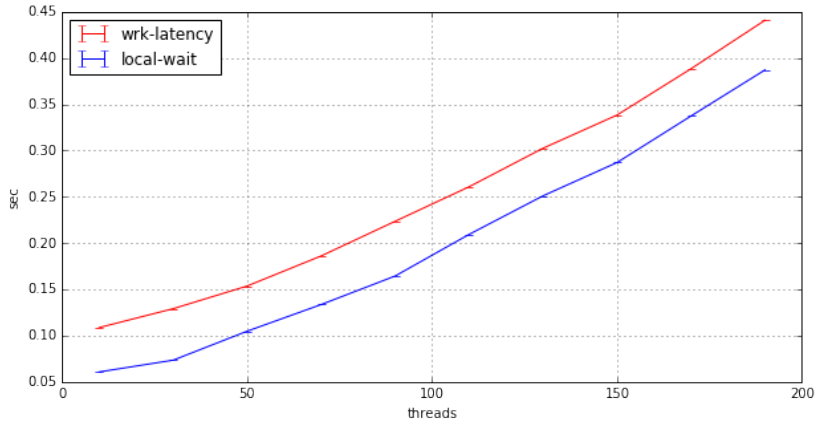


(a) CPU bottleneck

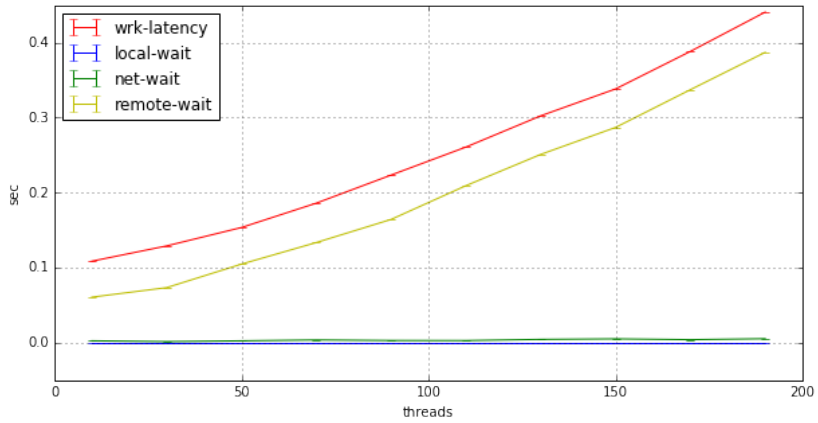


(b) Net bottleneck

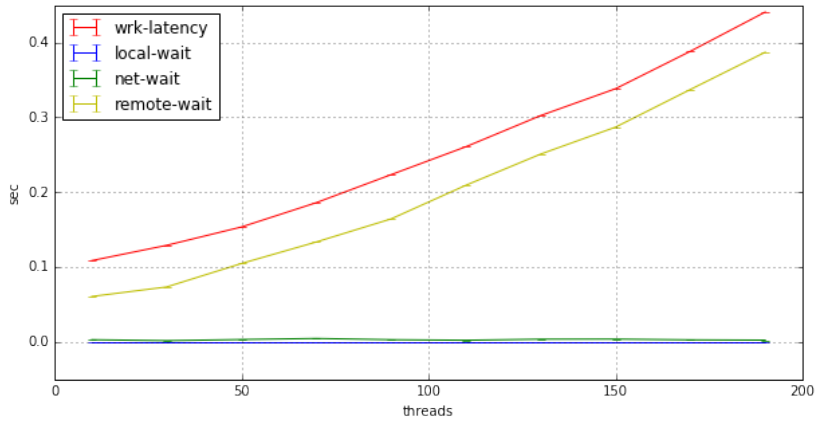
Figure 5.3: Request processing times compared to benchmark latency



(a) Backend traces



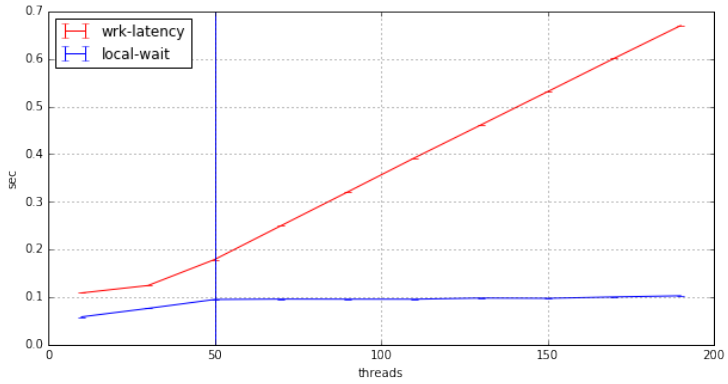
(b) Frontend traces



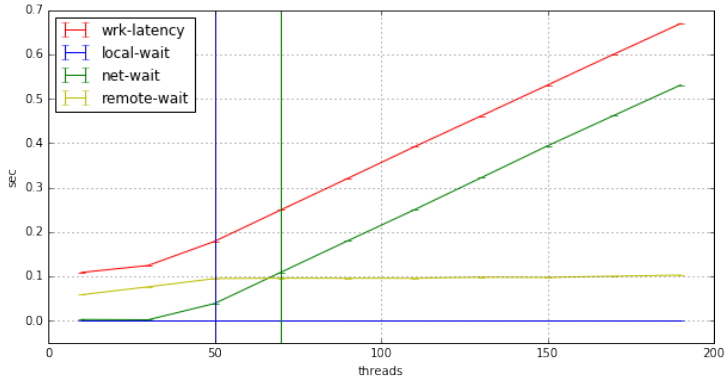
(c) API traces

Figure 5.4: Waiting times under a CPU bottleneck

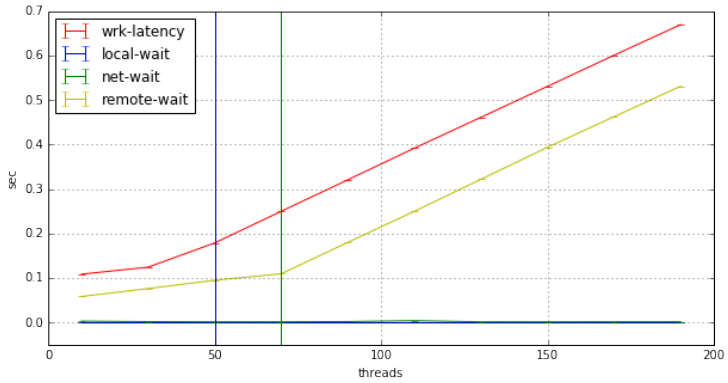
The increasing local wait at the backend indicates a CPU bottleneck, which is mirrored by the remote wait values at the frontend and API services



(a) Backend traces



(b) Frontend traces

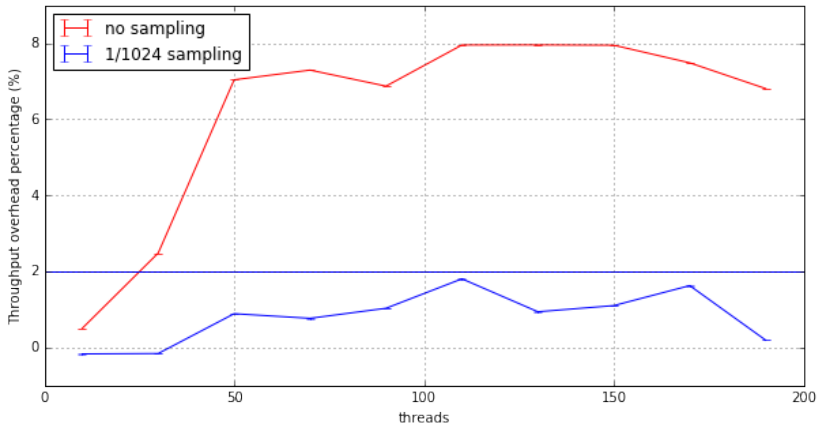


(c) API traces

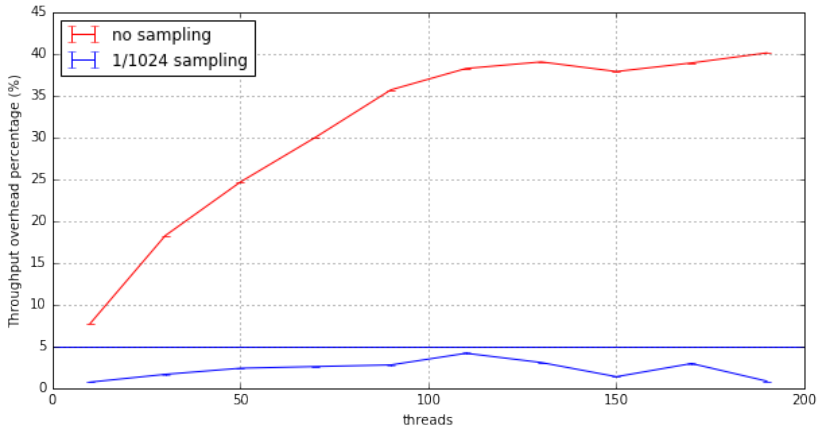
Figure 5.5: Waiting times under a network bottleneck

The network bottleneck at the backend service interface is reflected at the network wait time in the frontend service. The blue vertical line indicates the threshold corresponding to the network bottleneck. The green vertical line indicates when the network waiting time will be propagated to the API service

5.4 Overhead Evaluation



(a) 100 yield points



(b) 1000 yield points

Figure 5.6: Throughput overhead with and without 1/1024 sampling

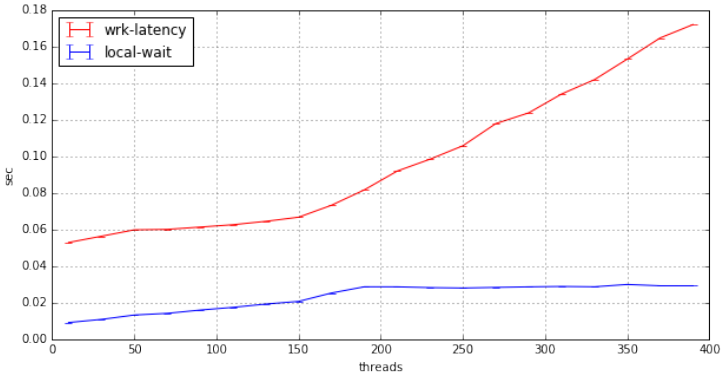
In this subsection, we present our measurement of the overhead of the tracing model on performance. We consider the overhead under 100 and 1,000 yield points. The latter configuration represents an unusual large number of explicit context switching points. This introduces a significant overhead regardless of our instrumentation (as we illustrate in Subsection 5.5). A large sequential code block composed of 1,000 or more threads will not necessarily have the same number of context switches, due to the implementation of sequential composition functions. These composition functions do not implicitly yield control between the execution of the threads they connect, unless the first thread returns a sleeping thread (Listing 4.2). In any case, a limit on the number of context switches should not affect code blocks which implement network protocols (e.g.

transfer of a large file over HTTP), as these blocks would be instrumented by a remote context wrapper where these trace points can be eliminated.

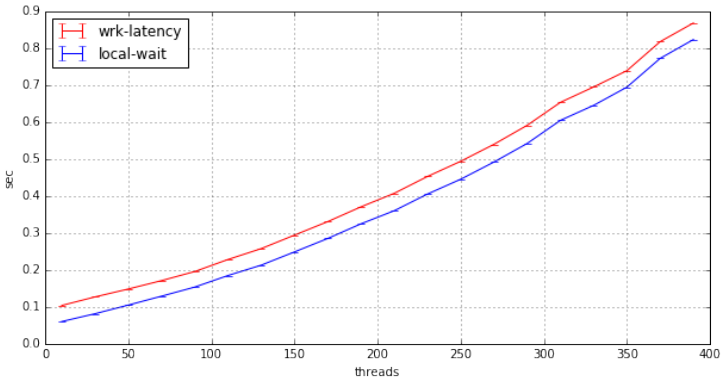
Our evaluation shows that our instrumentation can have a significant overhead if no sampling is introduced. Decreasing the number of context switches as well as using sampling techniques can significantly reduce this overhead. Figure 5.6 illustrates the overhead on throughput, given both a sampling rate of 1/1024 as well without sampling. If no sampling is used, the overhead of the tracing model increases as the load increases until the server reaches its maximum throughput. Under the 1000 yield point configuration, the maximum overhead of no sampling is significantly higher than under 100 yield points (40% compared to 8%). This is due to the additional instrumentation points required to account for waiting time during every context switch. Sampling can significantly reduce this overhead to less than 5% for 1000 yield points, and less than 2% for 100 yield points.

5.5 Tuning the Number of Cooperative Context Switches

The usage of additional yield points introduces a significant overhead but allows to introduce a CPU bottleneck in the backend without an artificial network bottleneck (an issue described earlier in Subsection 4.4). Under 100 yield points (Figure 5.7a), local wait time increases slightly until a CPU bottleneck in the network stack results with an artificial network bottleneck at a threshold between 150 and 200 client threads. This waiting time remains constant after the network bottleneck, as our instrumentation accounts for waiting time in the request handlers and not in the TCP stack. This is in contrast to the 1000 yield point configuration where the waiting time continues to increase in parallel with the request latency (Figure 5.7b). Furthermore, the high overhead of context switches allows to reduce the relative connection overhead that results from calling the backend service through an HTTP client call, where each request requires a new connection. This is illustrated in Figure 5.8 by comparing the performance of the backend being invoked directly from the benchmark to when invoked indirectly from either of the frontend or the API services.

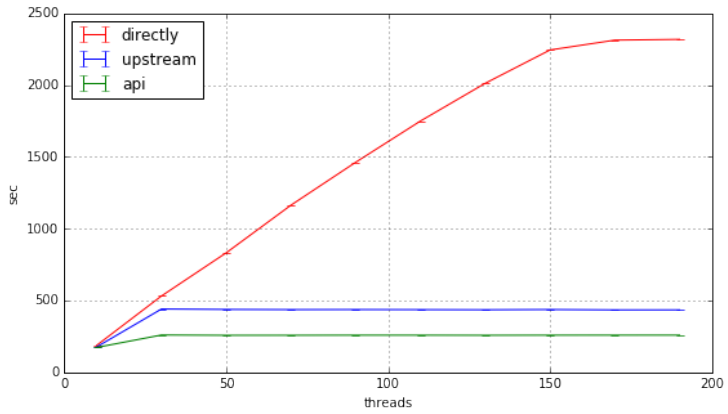


(a) 100 yield points

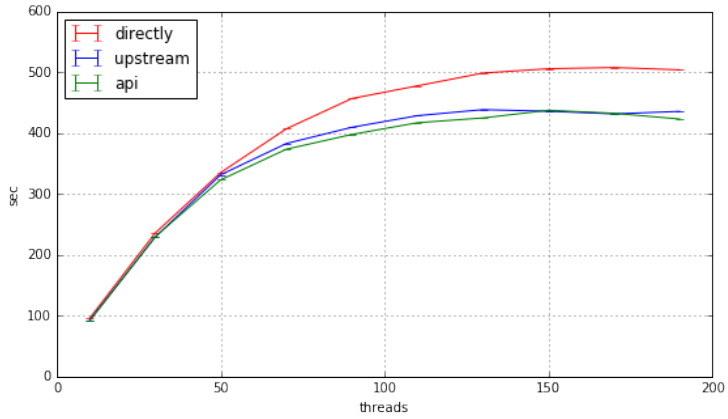


(b) 1000 yield points

Figure 5.7: Local waiting time of backend service



(a) 100 yield points



(b) 1000 yield points

Figure 5.8: Effect of Mirage connection establishment on backend service throughput

Calling the backend service from the benchmark directly reduces the relative overhead of connection establishment which occurs due to new connections created for every new request for Cohttp client call

Chapter 6

Summary and Conclusions

6.1 Summary

In this work, we have presented a hierarchically-aggregated tracing model which is appropriate to support both decentralized and centralized autoscaling policies. Given an increased latency, the essence of the model is to enable every service to identify whether the additional waiting time is due to a bottleneck in one of its local resources (CPU or network), or in some other resource in the downstream services. Having this model aggregated on each service based on performance measurements propagated from directly neighboring downstream services, the root client-facing service will be able to trigger an appropriate autoscaling decision, which is either taken locally or delegated appropriately down in the application structure.

We also described a generic and automatic approach to implement the tracing model in Mirage unikerenels. This approach is based on generic execution wrappers which encapsulate the necessary trace points and can be used to instrument any synchronous client-server protocol library used for remote service invocation. These wrappers integrate with the existing tracing infrastructure of Mirage and can instrument libraries exposed as Lwt threads. Based on our instrumentation of the default HTTP library of Mirage (Cohttp), we show how we can automatically generate traces for a simple HTTP-based microservice application. We show how these traces can be used to identify the service which is the root cause of a bottleneck, as well as to which resource it corresponds (CPU or network). Finally, the throughput overhead of the tracing model is dependant on the number of cooperative context switches as well as the degree of overload. However, we show that using standard sampling techniques can

reduce this overhead to a range between 2-5%.

6.2 Conclusions

The current prototype was useful to understand the overhead of the tracing model within Mirage unikernels and to demonstrate the usefulness of the service-centric trace analysis enabled by the hierarchical aggregation of resource waiting times. In order to evaluate the usefulness of the model to address the autoscaling problem, an evaluation based elasticity metrics and involving realistic microservice applications is necessary. However, the single-threaded architecture of Mirage which interleaves the TCP stack with application code, imposes two main difficulties that need to be addressed before. Firstly, a CPU bottleneck may lead to an artificial network bottleneck, and the unikernel will not be able to distinguish whether this is due to a genuine network bottleneck or a CPU bottleneck. This can be addressed by introducing packet-timestamping into the network devices used by Mirage, as this allows to measure packet waiting time before being processed by the TCP stack from within the unikernel. Secondly, TCP connection establishment can introduce a significant overhead which severely limits the overall throughput. This can be addressed by implementing connection pools or using advanced protocols such as HTTP2, which allows to multiplex the same TCP connection between concurrent requests.

The focus of this work has been on detecting resource bottlenecks assuming full resource isolation between services sharing the same host. However, operators are interested in consolidating their servers with more services sharing the same resources [6]. This introduces the requirement to extend the tracing model to detect waiting times due to host-level resource contention. In return, this would enable the tracing model to support service placement and orchestration decisions, such as migrating the service to a less consolidated host, or moving the service to an edge server with a less congested network link. This can be achieved through more advanced instrumentation such as using per-thread CPU clocks for detecting host-level CPU contention. Packet timestamping on the host level as well as in the network itself can be also used to detect contention within the application network infrastructure.

Overall, we believe this work identified the essential design dimensions and implementation challenges for building a tracing infrastructure that is suitable for further research towards realising effective autoscaling policies for self-scaling unikernels [32].

Bibliography

[1] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: library operating systems for the cloud. SIGPLAN Not. 48, 4 (March 2013), 461-472. DOI: https://doi.org/10.1145/2499368.2451167

[2] Raja R. Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H. Sigelman, Rodrigo Fonseca, and Gregory R. Ganger. 2016. Principled workflow-centric tracing of distributed systems. In Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16), Marcos K. Aguilera, Brian Cooper, and Yanlei Diao (Eds.). ACM, New York, NY, USA, 401-414. DOI: https://doi.org/10.1145/2987550.2987568

[3] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. 2015. Retro: Targeted resource management in multi-tenant distributed systems. In Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15). USENIX Association, Berkeley, CA, USA, 589-603.

[4] Lalith Suresh, Peter Bodik, Ishai Menache, Marco Canini, and Florin Ciucu. 2017. Distributed resource management across process boundaries. In Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17). ACM, New York, NY, USA, 611-623. DOI: https://doi.org/10.1145/3127479.3132020

[5] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. 2018. Overload Control for Scaling WeChat Microservices. In Proceedings of the ACM Symposium on Cloud Computing (SoCC '18). ACM, New York, NY, USA, 149-161. DOI: https://doi.org/10.1145/3267809.3267823

[6] Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya. 2018. Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey. ACM Comput. Surv. 51, 4, Article 73 (July 2018), 33 pages. DOI: https://doi.org/10.1145/3148149

[7] Vaquero, LM, Cuadrado, F, Elkhatib, Y, Bernal-Bernabe, J, Srirama, SN & Zhani, MF 2019, 'Research challenges in nextgen service orchestration' Future Generation Computer Systems, vol. 90, pp. 20-38. https://doi.org/10.1016/j.future.2018.07.039

[8] Kratzke, N. & Quint, P.-C., 2017. Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study. The Journal of systems and software, 126, pp.116.

[9] Madhavapeddy, A., Leonard, T., Skjegstad, M., Gazagnaire, T., Sheets, D., Scott, D.J., Mortier, R., Chaudhry, A., Singh, B., Ludlam, J., Crowcroft, J.A., & Leslie, I.M. (2015). Jitsu: Just-In-Time Summoning of Unikernels. NSDI.

[10] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17). ACM, New York, NY, USA, 218-233. DOI: https://doi.org/10.1145/3132747.3132763

[11] Sigelman, Benjamin H., Luiz Andr Barroso, Mike Burrows, P. C. Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. (2010).

[12] Apache Zipkin, https://zipkin.apache.org/

[13] Trace Context, W3C Candidate Recommendation 09 May 2019, https://www.w3.org/TR/trace-context/

[14] The OpenTracing project, https://opentracing.io/

[15] OpenCensus, https://opencensus.io/

[16] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using magpie for request extraction and workload modelling. In Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6 (OSDI'04), Vol. 6. USENIX Association, Berkeley, CA, USA, 18-18.

[17] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance debugging for distributed

systems of black boxes. In Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03). ACM, New York, NY, USA, 74-89. DOI: https://doi.org/10.1145/945445.945454

[18] Pedro Las-Casas, Jonathan Mace, Dorgival Guedes, and Rodrigo Fonseca. 2018. Weighted Sampling of Execution Traces: Capturing More Needles and Less Hay. In Proceedings of the ACM Symposium on Cloud Computing (SoCC '18). ACM, New York, NY, USA, 326-332. DOI: https://doi.org/10.1145/3267809.3267841

[19] Sambasivan, Raja R., Rodrigo Fonseca, Ilari Shafer and Gregory R. Ganger. So , youwant to trace your distributed system ? Key design insights from years of practical experience. (2014).

[20] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. 2018. Unikernels as Processes. In Proceedings of the ACM Symposium on Cloud Computing (SoCC '18). ACM, New York, NY, USA, 199-211. DOI: https://doi.org/10.1145/3267809.3267845

[21] Jrme Vouillon. 2008. Lwt: a cooperative thread library. In Proceedings of the 2008 ACM SIGPLAN workshop on ML (ML '08). ACM, New York, NY, USA, 3-12. DOI=http://dx.doi.org/10.1145/1411304.1411307

[22] Avritzer, Alberto, Vincenzo Ferme, Andrea Janes, Barbara Russo, Henning Schulz, and Andr van Hoorn. 2018. A Quantitative Approach for the Assessment of Microservice Architecture Deployment Alternatives by Automated Performance Testing. In Lecture Notes in Computer Science, 15974.

[23] Gotin, Manuel, Felix Lsch, Robert Heinrich, and Ralf Reussner. 2018. Investigating Performance Metrics for Scaling Microservices in CloudIoT-Environments. In Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering - ICPE 18. https://doi.org/10.1145/3184407.3184430.

[24] Gan, Yu, and Christina Delimitrou. 2018. The Architectural Implications of Cloud Microservices. IEEE Computer Architecture Letters 17 (2): 15558.

[25] Mendona, N.C. et al., 2018. Generality vs. reusability in architecture-based self-adaptation. In Proceedings of the 12th European Conference on Software Architecture Companion Proceedings - ECSA 18. Available at: http://dx.doi.org/10.1145/3241403.3241423.

[26] Jiang, Dejun, Guillaume Pierre and Chi-Hung Chi. Autonomous resource provisioning for multi-service web applications. WWW (2010).

[27] Sigelman, Benjamin H., Luiz Andr Barroso, Mike Burrows, Perry Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. (2010).

[28] Mace J, Fonseca R. Universal context propagation for distributed system instrumentation. Proceedings of the Thirteenth EuroSys Conference on - EuroSys 18. 2018. doi:10.1145/3190508.3190526

[29] Twitter Finagle, network stack for distributed systems, https://twitter.github.io/finagle/

[30] Marius Eriksen. 2014. Your server as a function. SIGOPS Oper. Syst. Rev. 48, 1 (May 2014), 51-57. DOI: https://doi.org/10.1145/2626401.2626413

[31] Michio Honda, Felipe Huici, Costin Raiciu, Joao Araujo, and Luigi Rizzo. 2014. Rekindling network protocol innovation with user-level stacks. SIGCOMM Comput. Commun. Rev. 44, 2 (April 2014), 52-58. DOI: http://dx.doi.org/10.1145/2602204.2602212

[32] Koleini, Masoud, Carlos Oviedo, Derek McAuley, Charalampos Rotsos, Anil Madhavapeddy, Thomas Gazagnaire, Magnus Skejgstad and Richard Mortier. Fractal: Automated Application Scaling. ArXiv abs/1902.09636 (2019): n. pag.