SERVICE BOOSTERS: LIBRARY OPERATING SYSTEMS FOR THE DATACENTER

Henri Maxime Demoulin

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2021

Supervisor of Dissertation

Boon Thau Loo and Linh Thi Xuan Phan
Professors of Computer and Information Science

Graduate Group Chairperson

Mayur Naik, Professor of Computer and Information Science

Dissertation Commitee

Vincent Liu, Professor of Computer and Information Science, Chair
Andreas Haeberlen, Professor of Computer and Information Science
Jonathan M. Smith, Professor of Computer and Information Science
Irene Zhang, Principal Researcher, Microsoft Research

SERVICE BOOSTERS: LIBRARY OPERATING SYSTEMS FOR THE DATACENTER

Acknowledgments

"Dissertations are not
finished; they are abandoned."
—Fred Brooks

None of this would have been possible without the unconditional love of she who was the first woman in my life. Thanks mom.

I cannot thank enough Linh Phan, Irene Zhang and Boon Loo for their support throughout my five years at Penn. I am a stubborn person who always wants to do things his way. My mentors had a hard time guiding me and without their patience I would not have graduated. I am forever grateful to Boon for accepting me in the program and giving me a chance to realize my dream. You have always had my back.

I have many people to thank at Penn and elsewhere. Vincent Liu is the most genuine, kind and gentle person I have ever met — in addition of being wicked smart. JMS has been a profound source of inspiration. When I think about Jonathan I think about the depth of humanity's quest for answers. We are very little things out there. Andreas Haeberlen helped me a lot finding my way during my first years. Marios Kogias taught me many things about systems research.

My friends at the Distributed Systems Laboratory have been very influential on my development as a researcher. I first wanted to write "X is very smart and helped me become smarter", but the fact is that all these individuals are insanely smart. So let's skip that part. Nikolaos Vasilakis, Joshua Fried and Isaac Pedisich have been profoundly inspiring. Nikos is a great example of how deep ideas combined to long lasting persistence can create breakthroughs. Our many conversations have changed me. To Josh, I am grateful I "cultivated" our relationship. You are a living example that one can be both a great computer scientist and a great human. Thanks for the time spent together, aloe. Without Isaac Pedisich, there would not be any Service Boosters. Out of everyone here, I*ss*ac is likely the one I harassed the most, night and day, with computer-related stuff. Likewise Tavish Vaidya was instrumental in my early research days.

Thanks to the great students I have been working with, Kelvin Ng, Jiali Xing and Pratyush Patel. Thanks to all the undergraduate and masters students at DSL, Chirag Shah, Robert DiMaiolo, Leon Wu, Ritika Gupta, Liana Patel, and Zachary Zhao.

I want to thank members of the NetDB group, Haoxian Chen, Nofel Yaseen and Nick Sultana. Qizhen Zhang has a special place in my heart. Thanks to the members of the Tundra lab, Saeed Abedi, Robert Gifford, Neeraj Gandhi and Edo Roth.

Interestingly Neeraj is one the most innocent and sweet person I met. The staff at Penn has always been helpful. Thanks to Cheryl, Liz and Britton.

This journey started way before Penn. Thanks to Benjamin Lee, Bruce Maggs, Jeffrey Chase, Theo Benson, Carlo Tomasi. Bruce trusted me and I will be forever grateful. You are a true friend. Thanks Edwin for introducing my to Philly. Thanks to my lifelong friends Aurelien, Benoit, Charles, Cyril, Cyprien, Fabienne, François (fat4), Jeanne, Karine, Lucas. Thanks to my little brothers Gabriel and Jeremy. Thanks Audrey. Thanks Julia and Iloa. Thanks Gilles and Marion for opening the curtains of possibilities.

Finally, to my second most fervent supporter, Christina, кристулик , I love you. You are bound to great things that, I guarantee you, are much more interesting than being a doctor for computers.

ABSTRACT

SERVICE BOOSTERS: LIBRARY OPERATING SYSTEMS FOR THE DATACENTER

Henri Maxime Demoulin

Boon Thau Loo and Linh Thi Xuan Phan

Cloud applications are taking an increasingly important place our technology and economic landscape. Consequently, they are subject to stringent performance requirements. High tail latency — percentiles at the tail of the response time distribution — is a threat to these requirements. As little as 0.01% slow requests in one microservice can significantly degrade performance for the entire application. The conventional wisdom is that application-awareness is crucial to design optimized performance management systems, but comes at the cost of maneuverability. Consequently, existing execution environments are often general-purpose and ignore important application features such as the architecture of request processing pipelines or the type of requests being served. These one-size-fits-all solutions are missing crucial information to identify and remove sources of high tail latency.

This thesis aims to develop a lightweight execution environment exploiting application semantics to optimize tail performance for cloud services. This system, dubbed Service Boosters, is a library operating system exposing application structure and semantics to the underlying resource management stack. Using Service Boosters, programmers use a generic programming model to build, declare and annotate their request processing pipeline, while performance engineers can program advanced management strategies.

Using Service Boosters, I present three systems, FineLame, Perséphone, and DeDoS, that exploit application awareness to provide real time anomaly detection; tail-tolerant RPC scheduling; and resource harvesting. FineLame leverages awareness of the request processing pipeline to deploy monitoring and anomaly detection probes. Using these, FineLame can detect abnormal requests in-flight whenever they depart from the expected behavior and alerts other resource management modules. Perséphone exploits an understanding of request types to dynamically allocate resources to each type and forbid pathological head-of-line blocking from heavy-tailed workloads, without the need for interrupts. Perséphone is a low overhead solution well suited for microsecond scale workloads. Finally, DeDoS can identify overloaded components and dynamically scale them, harvesting only the resources needed to quench the overload.

Service Boosters is a powerful framework to handle tail latency in the datacenter. Service Boosters clearly separates the roles of application development and performance engineering, proposing a general purpose application programming model while enabling the development of specialized resource management modules such as Perséphone and DeDoS.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

In the past decade, cloud applications have changed from — relatively — simple 3-tiers architectures to large graphs of microservices. Though the microservice design pattern enabled scaling engineering teams and building more complex applications, it also exacerbated the problem of *high tail latency* [**1**] for these systems. Tail latency arises when a small portion of the requests served by a machine are orders of magnitude higher than median response times. In the context of microservices, because end-users responses can depend on tens or hundreds of individual microservices, a single straggler can have disproportionate impact on end-to-end performance.

Meanwhile, there has not been significant changes to the abstraction between these cloud applications and the underlying hardware. Specifically, execution environments hosting microservices, such as Apache webserver [**2**] or Node.js [**3**], usually provide a coarse-grained thread pool design to handle incoming requests at a service, encapsulating the entire request processing pipeline in a single thread. This design essentially ignores application-level semantics and provide a one-size-fits all execution model to all requests. Unfortunately, taming high tail latency often relies on some form of application-awareness, for instance understanding the load at application queues [**4, 5**], observing requests' runtime [**6**], or understanding applications' state and data structures [**7**].

As we increasingly rely on cloud computing for high performance and predictable services, we need to rethink the design assumptions that make cloud execution environments oblivious of the application they host and instead transition toward well-defined exposition of application semantics to the resource management layer. This thesis argues that we should push the trend embraced by microservices a step further and allow users to declare and annotate each service's request processing pipeline. With such declaration of application-level semantics, the execution environment can implement advanced techniques for detecting and mitigating high tail latency. Building resource management techniques tailored to the application at hand typically raises concerns with respect to violating the end-to-end argument [**8**] and increasing costs of operation at scale. This thesis argues that finding novel solutions to harmoniously re-think the end-to-end argument and optimizing existing systems is much more viable economically and environmentally than pursuing with the existing abstractions. The traditional answer to high tail latency is to enlist more physical resources. Past a certain scale, more hardware means more data centers. As of 2020, there are an estimated 541 hyperscale data centers [**9**]. Building a new data center is very costly: Google can spend up to 600 million dollars per data center [**10**]. A price which pales in comparison to NSA's 1.5 billion dollars Utah data center [**11**]. In addition to being expensive to build, datacenters have currently a

significant environmental footprint: in 2016 they represented 1.8% of the total USA electric consumption [**12**] and in 2019 they were estimated to represent 1% of global energy usage [**13**]. Perhaps worse, cooling data centers often rely on potable water — even for leaders in data centers efficiency Microsoft and Google, more than 50% of the total water consumed is potable — which usage can have negative consequence on local habitats [**14**].

1.1. Contributions

This thesis introduces a library operating system (libOS) design for cloud services named Service Boosters. Service Boosters proposes a generic programming model to 1) expose application-semantics to the underlying execution environment and 2) develop custom management techniques for high tail latency. Using Service Boosters, users can declare their request processing pipeline as a graph of components and annotate this graph with information to be used by specialized resource management modules.

This thesis contributes three Service Boosters modules, FineLame, Perséphone, and DeDoS, that tackle high tail latency from three different angles. FineLame is a libOS module that exploits an understanding of the request processing pipeline to provide real time anomaly detection and predict when requests are likely to cause high tail latency. FineLame leverages recent advances in the Linux kernel and generalizes well to many cloud applications, beyond Service Boosters. Perséphone is a specialized resource management module leveraging the ability to classify incoming requests to implement in user-space a new request scheduling policy providing good tail latency for shorter requests in heavy-tailed workloads, without relying on interrupts. Perséphone is well suited to high performance services processing millions of RPCs per second. Finally, using knowledge of the request processing pipeline and the resource demand of each stage, DeDoS is a specialized resource management module responsible for dynamically scaling overloaded stages but harvesting only the necessary resources to mitigate tail latency.

1.2. Organization

The next chapter presents the design and components of Service Boosters. Chapters 3 and 4 will present FineLame and Perséphone. DeDoS is discussed in chapter 5. Finally chapter 6 proposes future work.

CHAPTER 2

Service Boosters

Service Boosters' first and foremost goal is to improve cloud services performance and predictability. It accomplishes these goal by supporting advanced techniques for detecting and mitigating high tail latency. Often, tail latency originates when the resource management layer antagonizes the characteristics of the workload or the request processing pipeline. Service Boosters provide a generic programming model to capture application-level semantic from cloud services and expose them to the resource management layer.

This chapter presents the overall design of Service Boosters (§2.1, 2.2), introduces the Boosters contributed in this thesis (§2.3) and the anomaly detection module (§2.4), the booster programming model (§2.5) and our prototype implementation (§2.6).

2.1. Overview

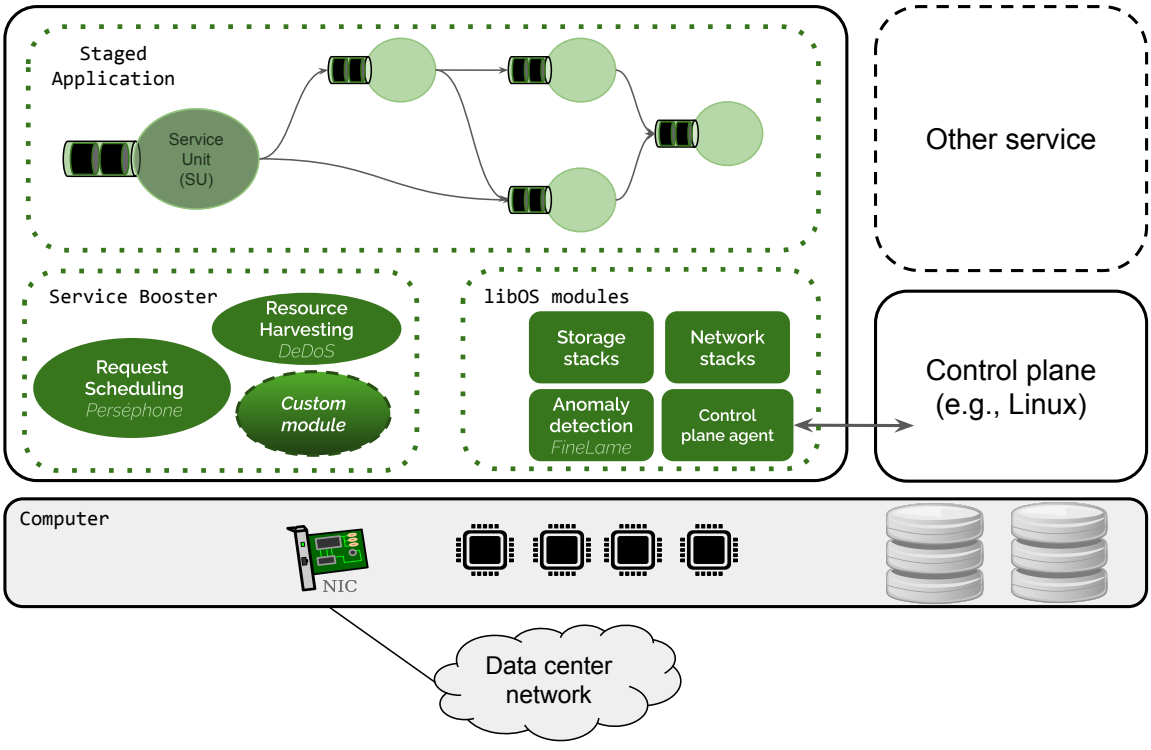


Fig. 2.1. Design of Service Boosters

Figure 2.1 presents the design of a Service Boosters runtime running alongside a control plane. This runtime is made of three domains: the application domain

Handler	Acceptable types	Description
UDP	Network	Process UDP payloads
TCP	Network	Process TCP payloads
NVMe	Storage	Process NVMe commands

Tab. 1. Native event handlers provided by Service Boosters

built using fine-grained building blocks named Service Units, specialized resource management modules named *Boosters*, and a set of libOS modules providing services for Service Units and Boosters. Service Units are responsible for processing part or entirety of incoming requests. Their goal is to be fine-grained and exhibit a well-defined resource consumption pattern. Each Service Unit in the application domain is a *typed unit of work* accepting a set of *input event types*. Programmers declare event types and fill a Service Unit event handling template, then compose Service Units together as a *dataflow graph* (DFG). Given a DFG declaration, the runtime will compile it and map each layer to a Service Unit. Boosters are specialized Service Units that take on resource management tasks. Boosters can operate inline with the application, such as the request scheduling Booster Perséphone using the DARC scheduling policy (chapter 4), or on the side of the pipeline, such as the resource harvesting Booster, DeDoS (chapter 5). The Service Boosters environment offers a library of modules, for example network and storage processing (Table 1), as well as a lower level API to program advanced Service Unit and Boosters. In addition, the runtime offers a monitoring subsystem and anomaly detection module (chapter 3) to profile Service Units at runtime and identify requests about to contribute to tail latency. These modules are services available to all Service Units and Boosters.

Finally, Service Boosters runtimes execute in a single process and runtimes can be distributed across a cluster. Typical Service Boosters applications are key-value stores, machine learning inference engines, search engines, statistic event processors, web front ends, file servers, *etc.*

2.2. Service Units

Service Units are essentially annotated event handlers. They are responsible for processing a set of event types from one or many input queue(s), which source can be network cards, storage devices, or other Service Units. By default, a Service Unit goes over input queues in a round robin fashion and dequeues events from a queue in a FIFO fashion. If the Service Unit is programmed to generate a response, it routes the message to the next stage in the DFG using a default least loaded queue dispatching policy across instances of the target stage. Messages flowing through the DFG are wrapped in a Service Boosters-specific protocol used for tracing and routing. Service Units localize and pass a pointer to application payloads to their programmed event handler. Figure 2.2 describes this sequence of actions.

The Service Unit programming interface is flexible, allowing programmers to break down RPC handlers into multiple Service Units that can be scaled independently.

```
1  /* Round robin through input queues */
2  for input_queue in input_queues:
3      if (!input_queue.empty()):
4          items = dequeue(input_queue, BATCH_SIZE);
5          for item in items:
6              /* Call associated event handler */
7              response = my_handler(item);
8              /* Dispatch response */
9              if (response):
10                 resp_type = classify(response);
11                 /* Configurable routing policy */
12                 out_queue = pick_dest(resp_type);
13                 push(out_queue, response);
```

Fig. 2.2. Service Unit event loop.

This is particularly useful to harvest resources from other machines in the data center, improving the service's resilience to sudden overloads in a stage of the request processing pipeline, and helping programmers identify bottlenecks. In addition, because Service Units embed application-level information, they enable the implementation of advanced load balancing techniques between stages of the pipeline. Finally, Service Units are independent of the underlying threading model (*e.g.*, N:M, 1:M). Section 2.6 describes a Linux implementation.

Event handlers: We provide a C++ template to program Service Units' event handlers. They accept a pointer to a payload and can either return nothing, a pointer to a response, or pass a response by copy. In addition, if a value is returned, it must include the type of destination (*i.e.*, which Service Unit should the runtime route the response to). Programmers have access to standard C/C++ libraries, in addition to a set of house-grown libraries (*e.g.*, buffer pool).

Event queues: Declared at configuration time, the set of event types an Service Unit accepts determines the set of queues it polls from. Network cards and storage devices expose descriptor queues to be polled for incoming packets or completed commands. Other Service Units expose memory queues in a one-to-one relationship with each other (so a Service Unit might poll over n input queues corresponding to the previous layer in the DFG, rather than a single one, to remove locking overheads from the communication pattern). The Service Boosters framework exposes a programming API to build custom event dequeuing policies.

Event routing: A Service Unit desiring to transmit messages to the next stage in the pipeline must choose an instance of the stage. Each stage in the DFG can be configured to use a specific dispatching policy. By default, Service Units uses a Decentralized, first come first serve (d-FCFS) policy which immediately dispatches messages to the least loaded instance of the target stage. The system also supports centralized dispatching modes where messages are first pushed to one or many local output buffer(s) at the Service Unit, then periodically dispatched. For example, we support a Centralized first come first serve (c-FCFS) policy with which messages are

dispatched to the next available target stage (in this mode, target stages have only input queues of depth 1 for the originating event type), and Dynamic Application-aware Reserved Cores (DARC, chapter 4), a policy classifying output event types through user-defined *request filters* and reserving Service Units to short requests in order to avoid head-of-line blocking. The Service Boosters framework exposes a programming API to build custom routing policies. Request filters are part of this API.

Booster's protocol: Requests entering the system are stored in memory buffers. A private zone — a buffer's headroom — is reserved for metadata: an internally generated request ID and the event type, plus potential sub-types.

2.3. Boosters

Boosters are special Service Units performing application-aware, customized resource management. This thesis contributes two boosters: a resource harvesting booster (DeDoS) and a request dispatching booster (Perséphone) .

Request Scheduling: A special Service Unit polling from NIC queues, classifying requests using request filters, and dispatching requests to the first stage in the DFG. This booster hosts a novel centralized scheduling policy, DARC, able to dedicate Service Units to specific event sub-types based on their profiled resource consumption. Fragmenting a DFG stage across event sub-types enables the separation of shorter requests from longer requests and eliminates head-of-line blocking otherwise resulting from mingling sub-types together. Chapter 4 describes this booster in details.

Resource harvesting: A special Service Unit responsible for scaling stages in the DFG if they appear overloaded. This booster uses the native Service Boosters monitoring subsystem to detect overloads at Service Units and periodically report load statistics and overload alerts to an external controller aware of the load across all Service Boosters nodes. When a Service Unit is overloaded the Booster attempts to locally scale the Service Unit and signals the overload to the controller. If a local solution was not found, the latter elects a machine where a new Service Unit can be created to host the overloaded handler. The decision is made based on the bottleneck resource exhibited by the overloaded handler. Chapter 5 describes this booster in details.

2.4. Detection

Service Boosters ships an internal monitoring and anomaly detection subsystem, FineLame [**15**], that keeps track of load and resource usage in two dimensions: Service Units and requests. These metrics can then be used by Boosters to perform specific tasks. For instance, the resource harvesting Booster uses Service Units input queuing delay to infer overloads. The subsystem also allows Booster to subscribe to alerts and be notified when a component or request goes rogue.

FineLame is made of three components: request mappers recording the request ID a given Service Unit is running, resource monitors maintaining a resource consumption profile for requests and Service Units, and an anomaly detection model to build a fingerprint of legitimate resource consumption behaviors. This fingerprint is used by resource monitors to detect in real time — in-flight — when a request or a Service Unit presents a suspicious behavior. Chapter 3 describes this subsystem in details.

2.5. API

In this section we discuss how to setup a Service Boosters application through concrete examples. The system offers a YAML configuration interface to compose DFGs and C++ templates for Service Units and boosters.

YAML:. The main section ("service_units") describes Service Units: which function template to instantiate, what initialization data to feed in for setup, which event types to accept, which nodes in the DFG responses should be sent to and which dispatch policy to use for this task. Service Unit can also be specified dependencies, that is other units that must be present on the same local runtime for correct execution. The YAML file comprises other sections specific to the implementation and cluster setup: for example, the monitoring subsystem can be wired to a database (MySQL or Postgres) to record statistics.

C++ template: Listing 2.1 presents the event processing function of a Service Unit used by programmers to write their service logic. Note that all messages flowing through the system use pre-allocated memory buffers by default, so that no dynamic memory allocation takes places for these messages, and the runtime can simply pass pointers around. Programmers can dynamically allocate memory within a Service Unit if they wish, but they are responsible for freeing it — potentially in another Service Unit.

Listing 2.1. C++ Service Unit template

```
#include <libos/ServiceUnit.hh>
int process_event(unsigned int event_in,
                  unsigned int event_out,
                  enum *event_out_target) {
    /* Cast event_in into an expected type */
    /* Process the request */
    /* Reuse event_in's buffer, or use a new one */
    /* Set up event_out_target */
    return 0;
}
```

Boosters use the same template than Service Unit. Some boosters take actual events (*e.g.*, network request dispatching) while others don't. The latter will be called with a dummy payload they can safely ignore.

Other facilities: The API offers a buffer pool library to acquire pre-allocated buffers, and other utilitarian libraries.

2.5.1. Example: REST service. The following describes a REST service made of the following components: network stack, TLS, HTTP, XML parsing, file storage. A typical requests enters the system through the network, gets deciphered and interpreted through the HTTP parser, then can either request a file or upload an XML file. Finally, a HTTP response is generated and sent to the user.

Each of these component might be vulnerable to performance outliers, that is a request could consume an unexpected amount of resources during the TLS, HTTP,

or XML stage. If our application was built from a single block, we would only be able to scale the entire monolith and maybe not be able to harvest sufficient resources to handle an entire new copy. With Service Boosters, we can instantiate a resource harvesting Booster which job is to detect overloads at Service Units and trigger a resource harvesting request. The Booster attempts to scale the overloaded unit locally but otherwise coordinates with an external resource controller to solve the bottleneck.

```
# REST service YAML configuration
    NETWORK: # This defines a "meta" input type
        handler: tcp_handler
        init_data: '1234' # Listening port
        destinations: [TLS] # Next layer in the DFG
    TLS:
        handler: tls_handler # e.g., implemented through OpenSSL
        sources: [NETWORK]
        destinations: [HTTP, NETWORK]
        dependencies: [network] # Needed for handshakes, etc.
    HTTP:
        handler: http_handler
        sources: [TLS]
        destinations: [XML, FILEIO, NETWORK]
        replicas: 3
    XML:
        handler: xml_handler
        sources: [HTTP]
        destinations: [FILEIO, HTTP]
        replicas: 2
    FILEIO:
        handler: fileio_handler
        sources: [HTTP, XML]
        destinations: [HTTP]
    HARVEST: # Resource harvesting Booster
        handler: resource_harvester
        init_data: '192.168.0.10 6789' # Controller's location
```

2.5.2. Example: KVS. The following describes a simple key-value store service. Requests enter the system though the network, and are then processed by a KV Service Unit. Responses are directly emitted by these units to the NIC. In this case, we rely on the network dispatching Booster to perform tail-tolerant scheduling through DARC (§4). This policy takes the name of a request filter, a user-defined function, to classify incoming requests before dispatching them to workers (here named "kvs_filter".

```yaml
# KVS service YAML configuration
service_units:
    NETWORK:
        handler: psp_booster
        init_data: '1234 DARC kvs_filter' # use DARC scheduling
        destinations: [KV]
    KV:
        handler: kv_handler
        sources: [NETWORK]
        replicas: 8
```

2.5.3. Example: Feature Generation service. This service is a typical map reduce example: a first layer of Service Unit extracts data from a file, performing an initial round of computation over the words, then a reducer computes a TF-IDF vector for each document in the dataset.

```yaml
# Feature Generation service YAML configuration
service_units:
    NETWORK:
        handler: udp_handler
        init_data: '1234'
        destinations: [KV]
    MAP:
        handler: feature_extractor
        sources: [NETWORK]
        destinations: [REDUCE]
        init_data: '/documents/storage/path'
    REDUCE:
        handler: feature_generator
        sources: [MAP]
        init_data: '/feature/storage/path'
```

2.6. Implementation

This section describes the implementation of Service Boosters over Linux. When an application is divided into a large number of fine-grained Service Units, switching from one Service Unit instance to another is a very frequent operation. In Linux, entering the kernel every time is prohibitively expensive. Because of this, the system privileges user level scheduling and context switching using a set of kernel threads pinned to CPU cores. This approach has the additional advantage that it does not require changes to the kernel.

Figure 2.3 presents the Linux implementation. A Service Boosters runtime over Linux maintains a POSIX thread for each available CPU core scheduling Service Units.

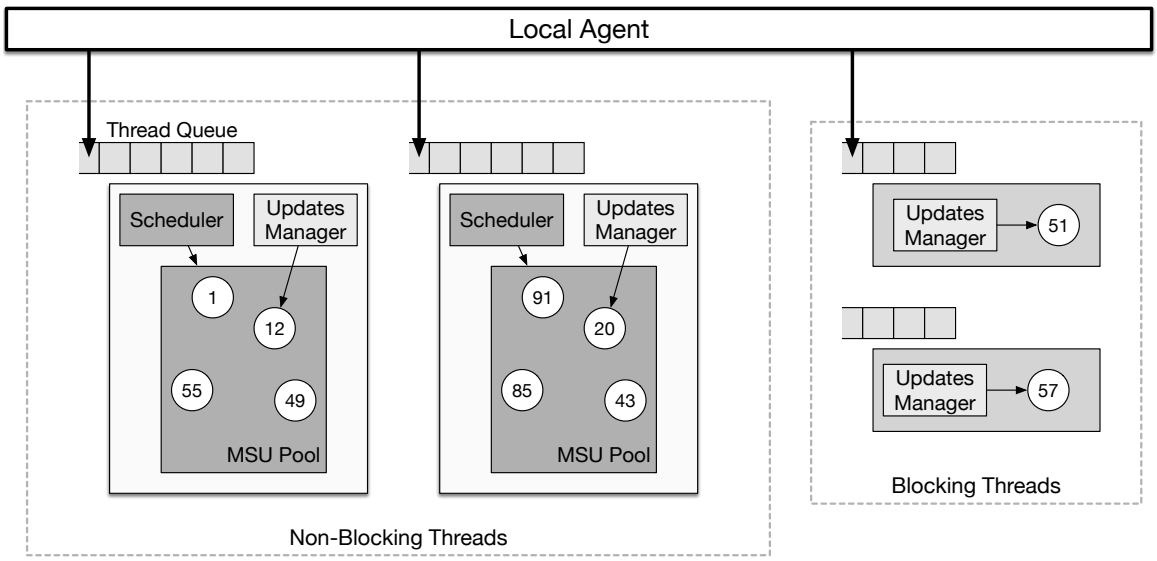


Fig. 2.3. Linux implementation of Service Boosters. Pthreads are used as kernel threads to run pools of Service Unit. Blocking units run on non-pinned CPU cores, while non-blocking units run on pinned CPU cores.

On each core, Service Boosters instantiates a worker thread running a local scheduler. The scheduler polls message queues for each of the local Service Unit instances in a round robin fashion — the default policy — and delivers new messages to the target Service Unit, waiting for the Service Unit instance to finish processing it. Scheduling is partitioned and non-preemptive — workers do not steal messages from other workers and they do not interrupt Service Unit instances while they are processing messages. Partitioned scheduling avoids inter-core coordination in the general case and thus keeps context-switching fast. The Service Boosters framework exposes an API to tune the policy used by workers to process Service Units, and program custom policies (e.g., Earliest Deadline First — EDF).

The system uses two types of threads: pinned ones for non-blocking operations and non pinned ones for blocking operations. The former allows the operator to schedule Service Units without Linux's scheduler interference. The latter allows Linux's CFS to preempt the thread. Pinning maximizes CPU utilization and reduces cache misses that would otherwise occur if Service Units were migrated between cores. In more detail, pinned threads run a scheduler that continuously (1) picks a Service Unit from its pool, (2) executes it by dequeuing one or more item(s) from its data queue(s) and invoking the associated event handler, and (3) repeats. Both Service Unit scheduling and dequeuing messages from Service Units' queues are configurable — round robin by default.

In the current implementation, Service Units that have blocking operations (*e.g.*, disk I/O) are assigned to their own non-pinned threads, such that they are scheduled by the Linux kernel as generic kernel-level threads.

Each worker thread keeps statistics on resource usage of each of its Service Units and global metrics such as their data queue lengths and number of page faults. The current API allows programmers to implement custom metrics (*e.g.*, frequency of

access of a given URL in an HTTP Service Unit). Those statistic are then gathered by the Service Boosters runtime to consolidate a single, local data store.

Finally, each worker thread periodically runs an *update manager* that processes a special *thread queue*. Unlike the Service Unit data queues, the thread queue is solely for control messages and is itself populated by the Service Boosters runtime. It stores requested configuration changes such as creating and destroying Service Units. In effect, the thread queue serves as a buffer of requested changes, and avoids the overhead of locks and other consistency mechanisms that would otherwise be required if the Service Boosters runtime directly manipulated worker threads' data structures.

Each worker thread is responsible for routing output events to the next stage in the DFG. To do so, they tap into a runtime-global routing table which contains information about Service Unit types and implements customizable load balancing policies and routing functions. The default policy is to route messages to the least loaded Service Unit instance of the destination type, enforcing instance affinity to related packets that requires a state (*e.g.*, from the same flow or user session).

2.7. Support for existing applications

Service Boosters aims at improving applications' resilience from the very beginning of their development process. Consequently, the focus of this thesis is on enabling *new* applications using the Service Unit programming model. Nevertheless, section 5.4 provides a proof of concept by splitting an existing user-level TCP stack into Service Units and supports that Service Boosters do not require applications to be written from scratch in order to benefit from its mechanisms. Since Service Boosters does not require the entire software to be partitioned, rewriting existing code can start small by only carving out the most vulnerable component while the rest of the application runs as a single Service Unit.

Finally, note that is possible to — partially or fully — automate the partitioning. Some domain-specific languages are already written in a structured manner that lends itself naturally to this approach. For instance, a declarative networking [**16**] application can be compiled to an Service Units graph that consists of database relational operators and operators for data transfer across machines (see §5.4). Recent work on programmable switches has shown how to automatically split P4 applications into a chain of subprograms [**17**]. Work in the OS community [**18**] has shown that even very complex software, such as the Linux kernel, can be split in a semi-automated fashion.

2.8. Takeaways

Service Boosters enables programmers to declare and annotate the structure of their request processing pipeline. The system offers a declarative interface to be parsed by resource management modules. The subsequent chapters detail techniques exploiting this information to detect and mitigate high tail latency.

CHAPTER 3

FineLame

A common source for high tail latency are requests consuming an unexpected amount of resources and blocking subsequent, well-behaved requests from processing. Such event can happen when requests take an infrequent execution path, hit the worst case complexity of an algorithm, or are issued from malicious senders. In the latter case, these attacks contain carefully-crafted, pathological payloads that target algorithmic, semantic, or implementation characteristics of the application's internals. Because they can be low-volume, we refer to these attacks as Asymmetric Denial-of-Service (ADoS) attacks.

The first step toward taming high tail latency is the ability to quickly detect such requests. Network-based detection techniques are generally ineffective against such requests because they lack identifiable problematic patterns at the network level. To be successful, network tools would not only need to perform deep packet inspection, but would also need to be able to predict which requests will hog resources a priori—a challenge akin to solving the halting problem.

Similarly, existing application-level detection techniques are limited in their efficacy: since these requests can consume arbitrary resources at arbitrary components in the service, which may be written in different programming languages and contain multiple binary third-party packages whose source code is not available or with complex dependencies, manual instrumentation of the application is prohibitively difficult, expensive, and time-consuming.

This chapter presents the design and implementation of a request monitoring and anomaly detection module for Service Boosters, FineLame (Fin-Lahm), that can predict when requests are likely to increase the tail latency of the service. FineLame only requires an understanding of the request processing pipeline, which, in the context of Service Boosters, can be obtained by parsing the application's DFG. In the general case, users only need to annotate their own code to mark the start and end of request processing; in many cases, annotations are not even required as applications lend themselves naturally to this demarcation. Our interaction with Apache Web Server[1] and Node.js[2] versions, for example, involves tracing three and seven functions, respectively, and not a single modification in their source code.

Using entry and exit points in the request processing pipeline, FineLame automatically tracks CPU, memory, storage, and networking usage across the entire application (even during execution of third-party compiled binaries). It does so with low overhead and at an ultra-fine granularity, which allows us to detect divergent requests before they leave the system and before they start contributing to the tail.

[1] 2.4.38

[2] v12.0.0-pre, 4a6ec3bd05e2e2d3f121e0d3dea281a6ef7fa32a on the master branch

Generalizing our approach beyond Service Boosters is a recent Linux feature called extended Berkeley Packet Filter (eBPF). eBPF enables the injection of verified pieces of code at designated points in the operating system (OS) and/or application, regardless of the specific programming language used. By interposing not only on key OS services, such as the network stack, the scheduler, but also user-level services such as memory management facilities, FineLame can detect abnormal behavior in a unified fashion across the entire software stack at run time.

FineLame consists of three synergistic components that operate at the user/kernel interface. The first component allows attaching application-level interposition probes to key functions responsible for processing requests. These probes are based on inputs from the application developers, and they are responsible for bridging the gap between application-layer semantics (*e.g.*, HTTP requests) to its underlying operating system carrier (*e.g.*, process IDs). Examples of locations where those probes are attached include event handlers in a thread pool. The second component attaches resource monitors to user or kernel-space data sources. Examples of such sources include the scheduler, TCP functions responsible for sending and receiving packets on a connection, and the memory manager used by the application. To perform anomaly detection, a third component deploys a semi-supervised learning model to construct a pattern of normal requests from the gathered data. The model is trained in the user space, and its parameters are shared with the resource monitors throughout the system, so that anomaly detection can be performed in-line with resource allocation.

By allowing developers to set probe points, FineLame effectively channels critical semantic information from development time to deployment and runtime. The system also enables a clear decomposition of roles between application programmers and performance engineers responsible for positioning probes. In the context of this thesis, FineLame effectively enables a principled transfer of application-level semantic to the execution environment.

Our evaluation shows that FineLame requires low additional instrumentation overhead, requiring between 4-11% additional overhead for instrumenting web applications ranging from Apache, Node.js, and Service Boosters. Moreover, when evaluated against real application-layer attacks such as ReDOS [**19**], Billion Laughs [**20**], and SlowLoris [**21**], FineLame is able to detect the presence of these attacks in near real-time with high accuracy, based on their deviation from normal behavior, and before they contribute to the tail latency of the system.

3.1. Motivation

We begin by showing via an example server-side application the operation of an ADoS attack and how it increases tail latency, the limitations of current detection mechanisms, and design goals for our system.

3.1.1. Background on ADoS attacks. Denial-of-service (DoS) attacks have evolved from simple flooding to pernicious asymmetric attacks that intensify the attacker's strength by exploiting asymmetries in protocols [**22, 23, 24**]. Unlike traditional flooding attacks, adversaries that perform asymmetric DoS (ADoS) are

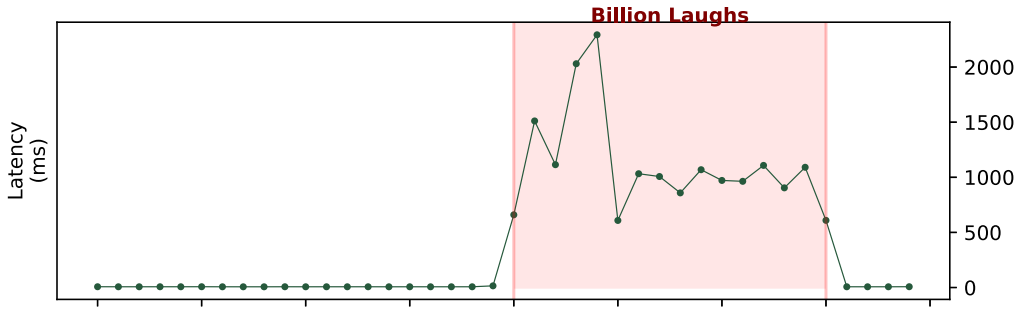


Fig. 3.1. Billion Laughs (XML Bomb) Attack. Under a normal load of about 500 requests per second, legitimate users experience a median of 6.75ms latency. After a short period of time, we start a malicious load of 10 requests per second (shaded area). XML bombs can take up to 200ms to compute (*vs.* a median of about 60*ns* for normal input). As a results, legitimate requests get serviced much slower, experiencing up to 2s latency. Setup details covered in (§3.3).

typically small in scale compared to the target victims. These attacks are increasingly problematic; the SANS Institute described "targeted, application-specific attacks" [**22**] as the most damaging form of DoS attack, with an average of four attacks per year, per survey respondent. Such attacks typically involve clients launching attacks that consume the computational resources or memory on servers. Types of asymmetric DoS vary, and are often targeted at a specific protocol. ADoS vulnerabilities are widespread and often affect entire software ecosystems [**25**]. They are representative of difficult situations that contribute to high tail latency in cloud services. We detail a few of them below.

Regular-expression DoS (ReDoS) [26, 27, 28]. ReDoS attacks target programs that use regular expressions. Attackers craft patterns that result in worst-case asymptotic behavior of a matching algorithm. An example pattern is `(a+)+`, which does not match any string of the form `a*X`, but requires the system to check 2^N decomposition of the pattern to reach that conclusion, where N is the length of the target string.

XML Bomb [20]. An XML bomb (or Billion-Laughs attack) is a malicious XML document that contains layers of recursive data definitions[3], resulting in quadratic resource consumption: a 10-line XML document can easily expand to a multi-gigabyte memory representation and consume an inordinate amount of CPU time and memory on the server. Fig. 3.1 illustrates the impact of XML bombs on the latency of requests on a susceptible server. Under normal operation, a load of 500 legitimate requests per second are served in less than 10 milliseconds each; under a low-volume attack of 10 XML bombs per second, the latency jumps up to more than two seconds. An XML bomb affects any serialization format that can encode references (*e.g.*, YAML, but not JSON).

[3]For example, the first layer consists of 10 elements of the second layer, each of which consists of 10 elements of the third layer, and so on.

Improper (de-)serialization [29, 30, 31]. This class of attacks encompasses those where malicious code can be injected into running services. These vulnerabilities are, unfortunately, common in practice, and they allow malicious users to, for instance, inject a `for (;;) {}` loop to stall a process indefinitely.

Event-handler Poisoning (EHP) [32]. Attacks like the preceding can be additionally amplified in an event-driven framework. In event-handler poisoning, attackers exploit the blocking properties of event-driven frameworks so that, when a request unfairly dominates the time spent by an event handler, other clients are further blocked from proceeding. Any slowdown, whether it is in the service itself or in its recursive layers of third-party libraries can contribute to this head-of-line blocking.

3.1.2. Design Goals. The attacks in the previous section highlight several goals that drive FineLame's design (§3.2) and implementation (§3.2.4).

In-flight Detection. Actions often need to be taken while requests are "in the work" — for example, when requests are actively contributing to increasing the tail latency of the system. Detection needs to catch such requests *before* they leave the system, by monitoring resource consumption at a very fine temporal and spatial granularity.

Resource Independence. Requests contributing to high tail latency may target arbitrary system-level resources (CPU, memory, storage, or networking), and may even target multiple resources. A desirable solution needs to be agnostic to the resource and able to handle any instance of inordinate consumption.

Cross-component Tracking. Given the complex structure of modern cloud applications, requests contributing to the tail of the latency distribution can also cross component boundaries and consume resources across several microservices on the execution path.

Language Independence. Applications today combine several ready-made libraries, which are written in multiple programming languages and often available only as compiled binaries. In addition, different microservices in the application graph could be written in different programming languages. Thus, detection should remain agnostic to the application details such as the programming language, language runtime, and broader ecosystem (*e.g.*, packages, modules).

Minimal Developer Effort. Detection needs to impose minimal burden to developers, while allowing performance engineers to design mitigation techniques without having to study application internals. Rather than presenting developers with an overabundance of configuration knobs, the detection system should direct precious human labor at sprinkling applications with key semantic information utilized at runtime.

3.1.3. Assumptions. To be more concrete, FineLame assumes the following about requests causing high tail latency and the broader environment.

Workloads. We consider that consumers of the application (i) can send arbitrary requests to a service hosting a vulnerable application, and (ii) is aware of the application's structure and vulnerabilities, including exploits in its dependency tree. We do not distinguish between legitimate and malicious clients who intersperse harmful

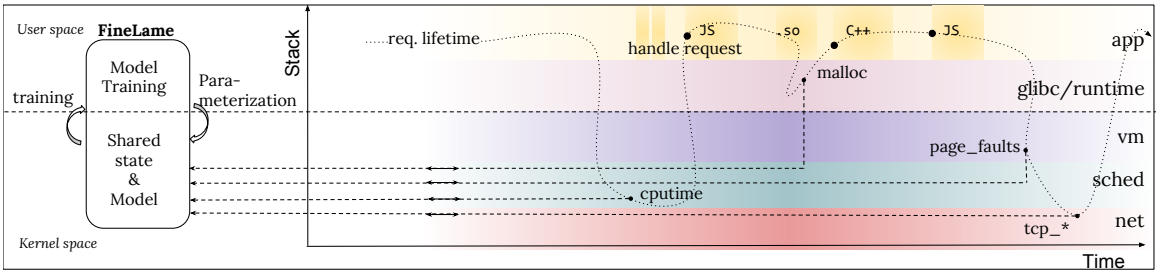


Fig. 3.2. FineLame overview. Key elements: (1, right) user and kernel data-collection probes at points where an HTTP request interacts with resource allocation; (2, mid-left) a data structure shared between user and kernel space, that aggregates and arranges collected data; (3, left) a userspace training component that instantiates model parameters, fed back to the probes. Information flow between 1–3 is bidirectional.

requests that attack resources with one or more benign requests. Specifically, any subset of hosts can send any number of requests that may or may not attack any subset of resources. The goal of FineLame is to detect any requests that is about to contribute to increasing tail latency.

We do not limit resources of interest to CPU; requests can abuse memory, file descriptors, or any other limited resource in the host system. That means that a single client can attempt to consume 100% of the CPU indefinitely, or multiple clients can abuse many of the system's resources.

Environment. We assume (i) vulnerable but not actively malicious code, and (ii) that FineLame sees at least some benign traffic. If all traffic abuse resources from the beginning, in-flight detection and mitigation become less urgent, as anomalies become the norm, and the application owners should first revise their deployment pipeline. We also assume that the resource utilization of request processing can be attributed to a single request by the end of each processing phase, even if the processing phases is split into multiple phases across different application components. As keeping a reference to the originating request is a natural design pattern, in all of the services we tested, a unique identifier was already available; in cases where there is no such identifier, one must be added, and we detail how to do so in section 3.2.

3.2. FineLame Design

Figure 3.2 depicts the overall design of FineLame. Conceptually, FineLame consists of three main components:

- *Programmer annotations* that mark when a request is being processed. FineLame requires only a few annotations, even for complex applications, to properly attribute resource utilization to requests.
- *Fine-grained resource monitors* that track the resource utilization of in-flight requests at the granularity of context switches, mallocs, page faults.
- A *cross-layer anomaly detection* model that learns the legitimate behavior and detects attacks as soon as they deviate from such behavior.

Programmers can use FineLame by annotating their application with what we call *request-mappers*. These annotations delineate, for each component and processing phase, the start and end of processing, as well as the request to which resource utilization should be attributed. For example, in an event-driven framework, the beginning and the end of each iteration of the event handler loop should be marked as the start and the end of a request's processing, respectively.

At runtime, when FineLame is installed on the host environment, FineLame attaches small, low-overhead *resource monitors* to particular points in the application or operating system. The aforementioned request-mappers enable FineLame to determine the request to which the resource consumed by a thread or process should be credited. In section 3.2.4, we detail our out-of-the-box FineLame library of request-mappers and resource monitors for several popular cloud frameworks. Our library tracks the utilization of a range of key OS-level resources; however, programmers can further extend it with user-level resource monitors to track application-specific resources (*e.g.*, the occupancy of a hash table).

Finally, FineLame's monitoring data is used to perform lightweight, inline anomaly detection. Resource monitors first feed data to a machine learning model training framework that computes a fingerprint of well-behaved behavior. Parameters of the trained model are installed directly into the resource monitors, which evaluate an approximation of the model to automatically detect anomalous behavior on-the-fly. The end result of FineLame is a high-accuracy, fine-grained and general detection engine for requests about to increase tail latency.

3.2.1. Request-mapping in FineLame. Conceptually, there are three operations in request mapping:

- `startProcessing()`: This annotation denotes the start of a processing phase. Any resource utilization or allocations after this point are attributed to a new unique request.
- `attributeRequest(reqId)`: As soon as we can determine a unique and consistent request identifier, we map the current processing phase to that request. For instance, when reading packets from a queue, if the best consistent identifier for a packet is its 5-tuple, resource tracking would start as soon as the packet is dequeued, but would only be attributed to a consistent request ID after Layer-3 and Layer-4 processing are completed. In general, `attributeRequest(reqId)` is called directly after `startProcessing()` and depending on the specific of the application, the two can sometimes be merged (§ 3.2.4).
- `endProcessing()`: Finally, this operation denotes the completion of processing, indicating that subsequent utilization should not be attributed to the current request.

In order for the resource monitors to properly attribute utilization to a request, FineLame requires programmers to annotate their applications using the above three *request mapping* operations. Ideally, the annotations should cover as much of the code base as possible; however, not all resource utilization can be attributed to a single request. In such cases, programmers have flexibility in how they perform mapping: for

true application overhead—rather than request processing overhead—utilization can remain unattributed, and for shared overhead (*e.g.*, garbage collection), utilization can be partitioned or otherwise assigned stochastically.

Every request is given an identifier that must be both unique and consistent across application components and processing phases. This identifier is used to maintain an internal mapping between OS entity (process or thread) and the request. Example identifiers include the address of the object representing the request in the application, a request ID generated by some application-level tracing solution [**33, 34, 35, 36, 37, 38, 39**], or a location in memory if the request is only processed once. From the moment a `startProcessing` annotation is called to the moment the `endProcessing` annotation is called, FineLame will associate all the resources consumed by the OS entity to the request.

An optimization of this technique can be implemented when the application lends itself naturally to such mapping between OS entity and request. For instance, event-driven frameworks or thread-pool based services usually have a single or small number of entry points for the request, to which FineLame can readily attach request-mappers via eBPF without source code modification. We found this optimization to be the common case, and FineLame does not require any modification to the application we explore in section 3.3.

3.2.2. Resource monitoring in FineLame. Resource tracking between `startProcessing` and `endProcessing` annotations are done via a recent Linux kernel feature called eBPF. We first provide some background on the operation of eBPF, and then discuss how we utilize it to perform extremely fine-grained resource monitoring of in-flight requests.

3.2.2.1. *Background on eBPF.* The original Berkeley Packet Filter (BPF) [**40**] has been a long-time component of the Linux kernel networking subsystem. It is a virtual machine interpreting a simple language traditionally used for filtering data generated by kernel events. Notable use cases are network packets parsing with Tcpdump [**41**] and filtering access to system calls in the seccomp facility. In version 3.0 a just-in-time compiler was implemented, allowing for a considerable speedup of the processing of BPF programs by optimizing them on the fly.

In version 3.15, Alexei Starovoivtov significantly extended BPF (dubbing the new system "eBPF"). The new version has access to more registers and an instruction set mimicking a native RISC ISA, can call a restricted subset of kernel functions, and can share data from kernel-space to user-space through hash-like data structures. While eBPF is a low-level language, users can write programs in higher languages such as C (and even Python with the BCC project [**42**]) and generate eBPF code with compilers such as GCC and LLVM.

Generated programs are verified before being accepted in the kernel. The verifier imposes a set of strict constraints to eBPF programs to guarantee the safety of the kernel. Common constraints include the absence of floating point instructions, a limit of 4096 instructions per program, a stack size capped at 512 Bytes, no signed division, and the interdiction of back-edges in the program's control flow graph (*i.e.*, no loops).

Name	Description	Event	Type
`tcp_idle_time`	TCP inactivity	`tcp_cleanup_rbuf`	kernel probe
`tcp_sent`	TCP Bytes sent	`tcp_sendmsg`	kernel probe
`tcp_rcvd`	TCP Bytes received	`tcp_cleanup_rbuf`	kernel probe
`cputime`	CPU time consumed	`scheduler_tick`, `finish_task_switch`	kernel probe
`malloc_memory`	*malloc* bytes	`glibc_malloc`	user probe
`page_faults`	Page faults events	`exceptions:page_fault_user`	kernel tracepoint

Tab. 1. Default FineLame resource monitors.

The ability of eBPF programs to be attached to both kernel and user-space functions and events, their extremely low overhead, and their ability to share data with user space without the need for any IPC or queuing mechanism make eBPF a prime candidate for implementing our resource monitors.

3.2.2.2. *Resource Monitor Architecture.* FineLame's resource monitors are attached to various user- and kernel-space data sources (*e.g.*, the scheduler or TCP stack) and use the mapping described in section 3.2.1 to associate resource consumption to application-level workflow (*e.g.*, HTTP requests). A resource monitor requires the following information: the type and name of the data source, and potentially the path of its binary.

Our current prototype of Service Boosters uses the features listed in Table 1. When executed, most resource monitors operate under the following sequence of actions: i) verify whether a request mapping is active for the current PID and exit if not; ii) collect the metric of interest (usually through the arguments of the function triggering it) and store it, time-stamped, in a shared data structure; and iii) perform anomaly detection on the request if the model's parameters are available (see section 3.2.3).

The time a request spends executing instructions on a processor is represented by *cputime*. We instrument both the *scheduler_tick()* and the *finish_task_switch()* kernel functions, which are called at every timer interrupt and context switch, respectively, to either start a timer when a thread executing a registered request is scheduled for execution or collect the amount of CPU time consumed by the task swapped out. We instrument the *tcp_sendmsg()* and *tcp_rcleanbuf()* to collect *tcp_sent* and *tcp_rcvd*, the amounts of bytes sent and read from a TCP connection, respectively. To compute *tcp_idle_time*, which represents the period of inactivity from the sender on a TCP connection, we measure the time elapsed between two occurrences of *tcp_cleanup_rbuf()*. To monitor the heap memory consumption occasioned by the processing of a request, we monitor the glibc *malloc* function. Applications where memory management is partly handled by the runtime (such as in Python) can be monitored in a similar fashion. Likewise, the model can be generalized to garbage collected languages. Finally, we monitor the page fault events in the application by attaching a resource monitor to the *exception: page_fault_user* kernel tracepoint. We observed in our evaluation that *cputime* was the best discriminant for CPU based attacks, while *tcp_idle_time* the best for slow attacks (such as Slowloris and RUDY).

The above default, general-purpose resource monitors are sufficient for a large set of existing applications; however, it can be extended to all the kernel events available for tracing and probing, as well as user-level functions (to monitor application-level metrics). If any application-level metrics are required (such as data structure occupancy, counters, and so on), programmers can augment our resource monitors with custom eBPF programs attached to arbitrary probe points in either kernel- or user-space.

3.2.3. Detection in FineLame. Detection algorithm. For fast detection, FineLame is designed to enable anomaly detection as close as possible to the resource allocation mechanism. Without a method for in-flight anomaly detection *in addition to* mechanisms for in-flight resource tracing, detection and mitigation of in-flight requests would not be possible.

This detection problem can be reduced to quantizing the abnormality of a vector in n-dimensional space. Once a sufficient amount of data has been gathered to compute a fingerprint of the legitimate requests' behavior, we can train an anomaly detection model. The model can span all the metrics collected by the resource monitors, allowing us to detect abuse on any of the resources of the system as well as cross-resource (multi-vector) attacks.

For the unsupervised version of this problem, the most popular methods take one of two approaches: distance-based or prediction-based. The former family of models aims to cluster known, legitimate data points and compute the distance of new data points to those clusters—distance that is used to quantify the anomaly. The latter family assumes the existence of a set of input data points that are correct, and learns a function representing those points. When a new point enters the system, the model computes the value of the learned function; the prediction error is then used to quantify the degree of anomaly.

Because of the training complexity, prediction complexity, and required training data, many existing solutions in both distance-based and prediction-based categories are impractical to execute at fine granularity. For instance, the popular algorithm DBSCAN [**43**] is not suitable for our purpose, as it requires us to evaluate the distance of new data points to all the possible "core" data points in the model. The amount of data points considered (and therefore the size of the model) is usually linearly proportional to the size of the training set. Some accurate approximations of DBSCAN have been proposed [**44**], but even with a small number of clusters, almost all of the training dataset still needs to be part of the model. Likewise, the performance of prediction-based models made on neural networks, such as Kitsune [**45**], is highly dependent on the depth and width of the model. The amount of parameters of such networks grows exponentially with the number and size of the hidden layers.

Given the above concerns, we chose to implement anomaly detection in FineLame with k-means, a technique that allows us to summarize the fingerprint of legitimate requests with a small amount of data. In k-means, the objective function seeks to minimize the distance between points in each cluster. The model parameters are then the centroids and distribution of the trained clusters. In a typical use-case scenario, FineLame is configured to perform only request monitoring for a certain amount of

```
FPAS          # FPA scaling factor
pid_to_rid    # OS carrier to request
req_points    # Request profiles
model_params  # K-means parameters
dp_dists      # Distances to centroids
thresholds    # Alerts cut-off bar

fun resource_monitor(context):
 pid = bpf_get_current_pid()
 rid = pid_to_rid.get(pid)
 if (rid):
    ts = get_timestamp()
    metric = context.get_arguments()
    dp = req_points.get(rid)
    if (dp):
       dp.update(metric, ts)
    else:
       dp = init_dp(rid, metric, ts)
       req_points.insert(dp)
    μ, σ = model_params.get()
    if (μ && σ):
       metric_scaled = metric << FPAS
       metric_scaled -= μ
       if metric_scaled < 0:
           metric_scaled *= -1
           metric_scaled /= σ
           metric_scaled *= -1
       else:
           metric_scaled /= σ
       min_dist, closest_k
       #pragma loop unroll
       for k in K:
           current_dist =
             dp_dists.get(dp, k)
           new_dist =
             metric_scaled+current_dist
           dp_dists.update(dp, new_dist)
           if (new_dist < min_dist):
               min_dist = new_dist
               closest_k = k
       t = thresholds.get(closest_k)
       if new_dist > t:
           report(rid, dp, s)
```

Required data structures

Is there a mapping?

Update request profile

If anomaly detection parameters are available, scale and standardize the data in the FPA space

Update distance to clusters

Perform anomaly detection

Fig. 3.3. FineLame anomaly detection. Pseudocode for Service Boosters 's inline anomaly detection.

time, after which it trains k-means on the monitoring data gathered in user-space from the resource monitors shared maps. In practice, we found that a k value equal to the number of request types in the application yields a reasonable estimation of the different behaviors adopted by legitimate requests, while being a number low enough such as to contain FineLame's overhead.

Model training and deployment. Gathering the training data is done by a simple look-up from the user-space agent to the shared eBPF maps holding the requests resource consumption data. Using those profiles, the user-space agent standardizes the data (center to 0 and cast to unit standard deviation). Subsequently, the agent trains k-means to generate a set of centroids representing the fingerprint of well-behaved traffic. The parameters of the model, to be shared with the performance monitors, are then the cluster centroids, as well as the mean μ and standard deviation σ of each feature in the dataset, and a threshold value τ statistically determined for each cluster.

As described above, the performance monitors have limited computing abilities and do not have access to floating point instructions. Thus, they are designed to perform fixed point arithmetic in a configurable shifted space, and require 's to shift the model parameters in this space before sharing them. Using two precision parameters a and b, each datapoint is transposed in a higher space 10^a, and normalized such that the resulting value lies in an intermediate space 10^{a-b}, retaining a precision of $a - b$ digits. This means that during the normalization operation each parameter value x undergoes the following transformation: $x = \frac{(x * 10^a) - (\mu * 10^a)}{\sigma * 10^b}$.

Once standardized, the clusters' centroids as well as each feature's mean and standard deviation are shared with the resource monitors through eBPF maps. Upon availability of those parameters, the resource monitors update not only the resource consumption of existing requests, but also their *outlier scores*, a measure we use to quantify the degree of anomaly of a request. Due to the constraints imposed on eBPF programs—specifically, taking a square root is complex as we do not have access to loops—we choose the normalized L1 distance to the closest cluster as the outlier score. While being a crude measure, the L1 is equivalent to more complex norms as resource vectors are of finite dimension. It preserves information about which resource is abused, and it lets us set statistical thresholds to determine cut-off points used for flagging abnormal requests. The algorithm for this entire process is shown in Figure 3.3.

Finally, we note that because FineLame is primarily designed toward the detection of requests contributing to high tail latency, we allow the anomaly detection engine to maintain signed values for outlier scores. This means that requests that have not reached their expected legitimate amounts of resource consumption, and that would look abnormal in an absolute value setting, are not flagged as such. This is important because it highlights the fact that FineLame is not geared toward volumetric DoS attacks that aim to bring the system down with a vast amount of low consumption requests.

3.2.4. Use Cases and Implementation. To demonstrate the generality of FineLame and the minimal developer effort required to use it, we apply FineLame

Application	Request mapping probes	SLOC
`Apache`	5	41
`Node.js`	9	64
`DeDoS`	2	21

Tab. 2. Intrusiveness of FineLame, quantified.

to three web platforms: Apache [**2**], which is estimated to serve ∼40% of all active webpages; Node.js [**3**] a popular server-side JavaScript-based web server; and Service Boosters. Our prototype of FineLame is available on `https://github.com/maxdml/Finelame`. Table 2 quantifies the programming effort required to write request-mappers for those three applications to use FineLame.

Apache web server. Primarily written in C, Apache's request processing is implemented by Multi-Processing Modules (MPM). In the latest versions of Apache (2.x), requests are served by multiple processes which can have multiple worker threads themselves; each thread handles one connection at a time.

When a request enters the system, an application-level (`conn`) object is created by the `core_create_conn` function to contain it before the request is dispatched to a worker thread. Subsequently, the request is processed by either the `ap_process_http_sync_connection` or the `ap_process_http_async_connection` functions, which take the `conn` object as argument. From FineLame, we attach one request-mapper to `core_create_conn`, and two requests-mappers to the HTTP processing functions, one over a *uprobe* called upon entering the function, the other over a *uretprobe* called when returning from it. We exploit the `conn` object to generate a unique identifier for each request and map it to the underlying thread worker, so that resource monitors can later gather resource consumption data on the request's behalf. The mapping is undone when the function returns and the request exits the system. When a worker thread executes a new request, the request-mapper updates the mapping with the new request's ID. This solution requires no modification to the Apache source code, and 41 lines of eBPF code over 5 probes.

Node.js required more slightly more instrumentation due to its asynchronous model, which offloads work to a worker pool (implemented with *libuv* [**46**]). The instrumentation required eBPF probes to be attached to seven user-space functions within the libuv library. As in Apache, we found a data structure — `struct uv_stream_t` — that could (i) be used to generate a unique identifier, and (ii) was carried consistently across the disparate components of the framework.

Request-mappers were applied to the seven libuv functions as follows:

- `uv_accept`: a new request is initialized, and is associated with the `uv_stream_t` structure that handled communication with the client.
- `uv__read` and `uv__write`: the request associated with the client's stream is assigned to the current thread for the duration of the function.
- `uv__work_submit`: the request assigned to the current thread is associated with a work-request submitted to the worker pool.

- `uv__fs_work`, and `uv__fs_done`: the request associated with the work-request is assigned to the current (worker) thread.
- `uv_async_send`: the request is unassigned from the current thread.

Again, this solution requires no changes in Node.js source code, only knowledge of which functions are processing requests. The request-mappers summed up to 64 lines of eBPF code.

Service Boosters is, by design, the canonical use case for FineLame. Applications declare Service Units in their DFG and the framework provides an application-layer protocol that FineLame can use to attach request mappers to each Service Unit entry point.

3.3. Evaluation

In this section, we present our evaluation results of FineLame. Our evaluation use ADoS attacks and is centered around the following aspects of the system:

- **Overhead.** The overhead of FineLame compared to no monitoring, or in-application instrumentation
- **Accuracy.** The ability of FineLame to accurately detect requests about to contribute to high tail latency and never seen yet by the application

3.3.1. Experimental setup. We present the setup on which we evaluate both the overhead and accuracy aspects of FineLame. In all cases, the server applications are running on a 12 cores Xeon Silver 4114 at 2.20GHz , while our legitimate and attack clients are running on an Intel Xeon E5-2630L v3 at 1.80GHz. Both server and client machines have a total of 62G of RAM, and have hyper-threading and DVFS disabled.

We use version 2.4.38 of Apache, and configure it to use 50 worker threads. We use version $12.0.0 - pre$ of Node.js with the default configuration of 4 worker threads for libuv. Both Apache and Node.js are configured to serve a set of Wikipedia [**47**] pages. Node.js parses a regular expression provided in the request's URI to find the path of the file to serve. It's parser, *liburi*, is vulnerable to the ReDoS attack. All the applications impose a timeout of 20 seconds on connections. We deploy a simple webserver in Service Boosters which can process three types of requests: serve a Wikipedia article, process a randomly generated XML file uploaded in a POST request, and parse a regular expression. The server is decomposed into several software components: socket reading, HTTP parsing, file serving, XML parsing, regular expression parsing, and response writing. The XML parser is implemented with *libxml2*, which is vulnerable to the Billion Laughs attack.

Our good traffic is generated by Tsung [**48**] and explores evenly all the servers' exposed endpoints; bad traffic is generated by an in-house **C** client for the ReDoS and Billion Laughs attacks, and pylorys [**49**] for the Slowloris attack. Tsung generates load under an exponential distribution centered on a configurable mean, while our attack client is configured to send a fixed load.

3.3.2. Overhead of FineLame. Figures 3.4, 3.5, and 3.6 presents the overheads incurred by FineLame's instrumentation on Apache, Node.js and Service Boosters.

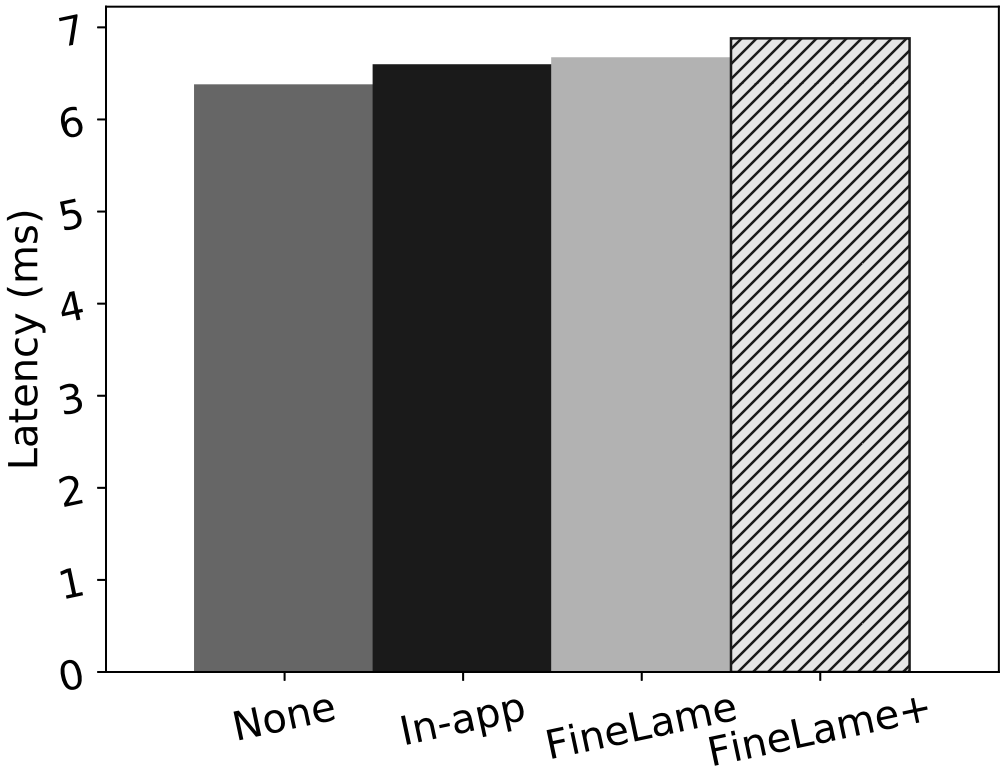


Fig. 3.4. FineLame overheads with Service Boosters

In all of our experimental setups, we evaluate the legitimate client latency experienced when the server is not instrumented, when it is instrumented by FineLame, and when FineLame's resource monitors are also performing anomaly detection (FineLame +). The load is as described earlier in sec 3.3.1, and explore all the instrumented paths in the applications. We also evaluate the cost of instrumenting the Service Boosters framework itself to evaluate FineLame overheads compared to a traditional user-space solution. The bars plot the median of the clients latency, and all our experiments are run thrice for a period of 100 seconds. In the case of Node.js the instrumentation cost adds 8.55% overheads and adding anomaly detection 9.21%. In the case of Apache, FineLame adds 11.38% and 11.72% overheads respectively. In the case of Service Boosters, the baseline latency is higher than with the two previous services, due to the fact that the application is not only serving files but also parsing POST requests, and also the framework is less optimized than the two battle-tested Apache and Node.js. Instrumenting directly the framework comes with an overhead of 2.9%, while FineLame comes with 4.23% overheads, 6.3% if also performing anomaly detection.

In general we observe that the overheads incurred by FineLame are higher when the baseline processing time of the service is low, and does not grow linearly with

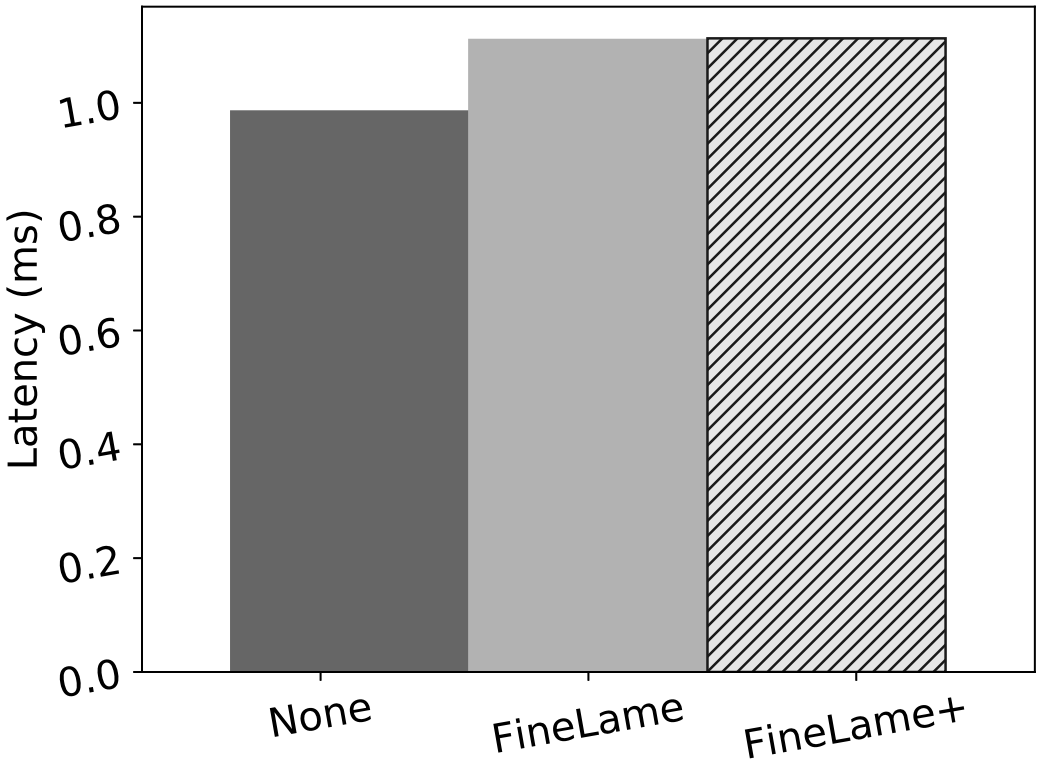


Fig. 3.5. FineLame overheads with Apache

the complexity of the application. In addition, we found that performing anomaly detection in addition to monitoring resource consumption almost comes for free.

3.3.3. Performance of FineLame. Our performance evaluation of FineLame is centered around its ability to detect requests about to contribute to high tail latency before they do so, in this case detect attacks requests before they exit the system, while providing accuracy competitive with non-approximated user-level algorithms.

3.3.3.1. *Attacks.* Our experiments aim to quantify the impact of attacks on quality of service. Consequently, we tune attacks strength such that they will not bring down the server but rather degrade the quality of service provided to legitimate users.

ReDoS: This attack consist of specially crafted regular expressions which are sent to the server for processing. The strength of the attack grows exponentially with the number of malicious characters present in the expression. Because the application processing units are busy handling those requests, legitimate requests get queued for a longer period of time, and ends-up being responded to more slowly.

Billion Laughs: The attack consists of XML files filled with several levels of nested entities. The parsing cost is exponentially proportional to the depth of the document. The impact is similar to the ReDoS attack.

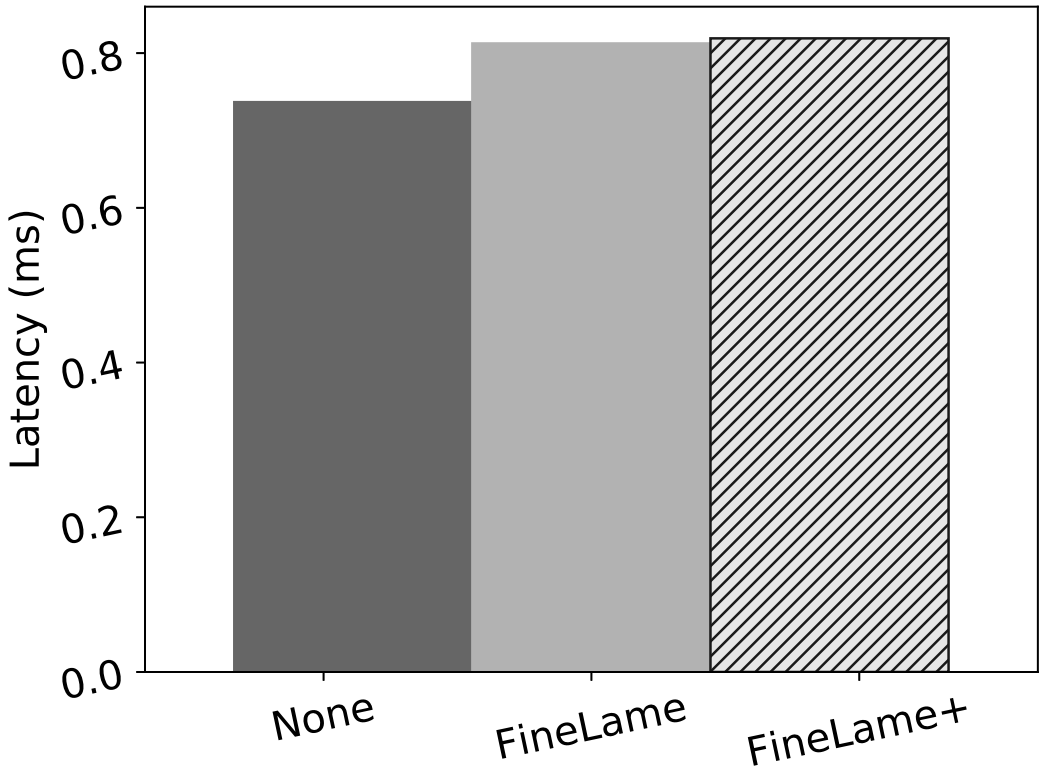


Fig. 3.6. FineLame overheads with Node.js

SlowLoris: The attack consists in maintaining open connections to the server, keeping them alive by sending individual HTTP headers at a regular interval smaller than the server's timeout, but never completing the request—we assume that the attacker is able to probe the service and discover this timeout. As a result, the server's connection pool gets exhausted, and it can't answer new requests. This technique can also implement a dormant attack which cripples the ability of the server to handle surges of legitimate traffic, by denying a fraction of the total connection pool.

3.3.3.2. *Anomaly Detection Performance.* **Evaluation metrics** As is common with anomaly detectors, the output of FineLame is a score which quantifies the abnormality of a request. This score is then either used as a raw metric for mitigation algorithms, or compared against a threshold τ to be transformed into a binary variable where 0 means negative (no anomaly), and 1 means positive (attack). With τ set, and using our knowledge of the ground truth, we can determine the accuracy of each of the detector's outputs as true/false positive/negative. The choice of τ is crucial, as too low a value can result in a large amount of false positive, while too high a value can induce a large amount of false negative. For our experiments, we set τ to be the outermost point for each cluster in the training set, *i.e.*, the most consuming legitimate request we've seen so far for the cluster. The challenge associated with deriving a large τ from the training traffic is that attacks can now take longer to

detect—and might not be detected at all if they are too weak. This latter case does not concern us, because to bring down the system with weaker attacks, an attacker would be forced to change its method from asymmetric to volumetric. The benefit of a higher τ is that it helps decreasing the False Positive Rate (FPR, $\frac{FP}{FP+TN}$), a desirable behavior for operators using the system. For our experiments, we present the True Positive Rate (TPR, $\frac{TP}{TP+FN}$), True Negative Rate (TNR, $\frac{TN}{TN+FP}$) and F1 ($\frac{2TP}{2TP+FP+FN}$). TPR evaluates the system's ability to detect all the attack requests. TNR evaluates its ability to evaluate legitimate requests as such. The F1 score is the harmonic mean of the TPR and the recall. It evaluates both the TPR and the precision of the system.

In addition to its post-hoc instrumentation abilities and low programmer burden, the main contribution of FineLame is it's detection pace. We evaluate the *Detection Speedup* (DS) of the system, which we define as being the delta between the time of last detection and the time to first detection, over the lifetime of the request. We expect DS to increase as users set more strict thresholds (lower values of τ), but found that even with τ set to the outermost point in each training cluster, FineLame is able to detect attacks up to more than 97% faster.

Experiments All our experiments are run for a duration of 400 seconds, split into 3 phases. The first phase sees only legitimate traffic flowing through our target applications, and last 200 seconds. FineLame is configured to only have the performance monitors gather data for the first 180 seconds, after which point it triggers the training of the anomaly detection model and share its parameters. Attacks start at time 200, and last for 150 seconds. A final period of 50 seconds sees the attack stop, and only good traffic is sent to the application. We perform two CPU exhaustion attacks, Billion Laughs and ReDoS, as well as a connection pool exhaustion attack, SlowLoris. For all experiments, we compare the TPR and TNR of FineLame to a non approximated user-space implementation of k-means (that is, with floating point arithmetic) to confirm that the system is competitive with more complex user space solutions. We set $k = 3$, the maximum number of request types that the application we setup can accept, and use $a = 10$ and $b = 6$ factor to retain 4 digits in fixed point arithmetic.

Tables 3 and 4 present the detection speed and performance of FineLame.

Attack	**Strength**	**TPR**		**TNR**	
		FL	KM L2	FL	KM L2
ReDoS	28.7×	100%	100%	99.995%	99.999%
	57×	100%	100%	99.993%	99.994%
	113.7×	100%	100%	99.997%	99.999%
Billion Laughs	4.7×	100%	100%	100%	100%
	34.8×	100%	100%	99.998%	99.998%
SlowLoris	5 sockets	100%	100%	100%	100%

Tab. 3. FineLame TPR and TNR for Apache, Node.js and Service Boosters.

Attack	Strength	F1		DS		
		FL	KM L2	median	75*th*	max
ReDoS	28.7×	99.88%	99.98%	80.9%	81.2%	83.2%
	57×	99.81%	99.83%	90.4%	90.5%	91.0%
	113.7×	99.29%	99.76%	90.9%	95.1%	95.3%
Billion Laughs	4.7×	100%	100%	83.1%	85.5%	87.7%
	34.8×	99.53%	99.76%	97.0%	97.1%	98.2%
SlowLoris	5 sockets	100%	100%	75%	n/a	n/a

Tab. 4. FineLame F1 and detection Speedup for Apache, Node.js and Service Boosters.

ReDoS: In our first experiment, we attack Node.js with three strengths of ReDoS requests. In the two first experiments, the workload is made of 98% of benign requests and 2% of malicious regular expressions blocking the event loop of the server (about 500 and 10 r/s, respectively). In the third experiment, with the strongest attack, we reduce the attack rate to 1 r/s, such that the attack does not bring down the server. Legitimate requests are served in about 0.8ms on average under normal conditions, but get delayed in proportion of the intensity of the ReDoS requests when the attack starts. During the first attack, bad requests are served in 23ms on average, a 28.75× increase compared to normal requests. Good requests are also penalized and are served in about 4ms. During the second attack, bad requests are served in 45.6ms on average, a 57× increase compared to normal requests. Legitimate requests are affected and incur an average latency of 13.5ms. During the third attack, bad requests are served in 90.9ms on average, a 113.6× increase. Legitimate requests incur an average latency of 6ms. Due to its ability to credit requests' resource consumption at the granularity of context switches, in both experiments, FineLame is able to detect attack requests before they exist the system, at least 80.9% earlier for 50% of the bad traffic, and up to 95.3% earlier. The user-space, non-approximated evaluation of k-means using the L2 norm for measuring distances, perform only marginally better.

Billion Laughs: In this experiment, we attack Service Boosters with two different strengths of Billion Laughs (XML bomb) requests. The good traffic follows a diurnal pattern, oscillating between 250 and 750 requests per second. Under normal conditions, legitimate requests are served in 6.87ms on average. In the first experiment, we send 15 malicious requests per seconds (about 2% of the peak legitimate traffic, and 6% of the lower phase), which are served in 29.28ms on average, a 4.26× increase in response time. In the second experiment, we decrease the number of bad requests to one per second (about 0.1% and 0.4% of the peak and low traffic, respectively), and increase their intensity such that they are served in 203ms in average (an order of magnitude increase compared to the first case), which represents a 29.55× increase in load compared to legitimate requests in normal conditions. For the weaker attack, FineLame is able to detect malicious requests 78.83% faster than the user-space solution, at least 50% of the time, and up to and 97% faster for the strongest attack.

SlowLoris: In this experiment, we configure Apache to handle requests with 25 worker threads, and timeout on reading HTTP headers after 20 seconds. We configure the attack client to maintain 5 connections to the server opened at all times, refreshing it every 5 seconds. Effectively, this drives the *tcp_idle_time* of the malicious request high and makes them standout from the legitimate ones. This attack is "all or nothing", in the sense that it will not impact the legitimate requests until the connection pool gets exhausted. FineLame's is able to detect the abnormal idle time about 75% faster than the application $(1 - \frac{5}{20} * 100)$, which would have otherwise to experience the timeout before reporting the request.

3.4. Related work

Performance monitoring Magpie [**50**] instruments an application to collect events from the entire stack and obtain request profiles *post-mortem*. X-trace [**51**] is a tracing framework that preserves causal relationship between events, and allow the offline reconstruction of request trees. X-ray [**52**] builds on taint-tracking to provide record and replay system to summarize the performance of application events offline. FineLame's allows us to perform anomaly detection while the request is still in the system and consequently supports advanced resource harvesting and scheduling techniques for Service Boosters. Retro [**53**] provides a tracing architecture for multi-tenant systems that enables the implementation of resource management policies. It focuses on performance degradation caused by competing workloads, rather than the detection of degradation within a single application.

Programmer Annotations Prior work proposes an annotation toolkit that programmers can use in their code to specify resource consumption requirements [**54**] and detect connections that violate the provided specification (and then attempts to mitigate by rate limiting or dropping them). Service Boosters decouples the skills required to break down a request processing pipeline — owned by programmers — from the skills required to understand resource management techniques — owned by performance engineers.

Volumetric Attack Detection There is a large body of work addressing volumetric DoS attacks [**55, 56, 57, 58, 59, 60, 61**], including attacks that target the network [**62, 63, 64**]. As described earlier (§3), these systems do not protect against *asymmetric* DoS attacks, a concern shared by both industry [**65, 66**] and academia [**27, 28, 32**].

Application-based Detection Prior works on application-layer DoS detection either depend heavily on repeated outliers, or are often deeply tied to a specific application. Techniques include comparing the entropy of offending and legitimate traffic [**67, 68**], sampling traffic flows [**69**], and sketch-based feature-dimensionality reduction [**70**]. While these techniques work well for volumetric attacks, they have self-assumed limitations when the attack traffic is low—the primary focus of this paper.

DSHIELD [**71**] is a system that assigns "suspicion scores" to user sessions based on their distance from legitimate session. While similar in nature to FineLame's anomaly detection technique, it relies on the operator knowing all the possible classes of requests that the server can process. FineLame anomaly detection engine learns

on legitimate requests so that it does not depend on *a priori* knowledge of execution paths or vulnerabilities.

BreakApp [**72**] is a module-level compartmentalization system that attempts to defend against DoS attacks, among other threats stemming from third-party modules. While BreakApp's capabilities increase with more and smaller modules, FineLame works even with monolithic applications entirely developed as a single module.

BreakApp's mitigation uses simple queue metrics (*i.e.*, queue length at the module boundary *vs.* replica budget), whose cut-off parameters are statically provided by the programmer; FineLame uses a more advanced learning model, which parameters are adjusted at runtime.

Rampart [**73**] focuses on asymmetric application-level CPU attacks in the context of PHP. It estimates the distribution of a PHP application function's CPU consumption, and periodically evaluates running requests to assess the likelihood they are malicious. It then builds filters to probabilistically drop offenders—repeated offenders increase their probability of being filtered out. While FineLame profiles legitimate requests resource consumption, it is not limited to CPU-based attacks. It also works with applications with components built with many different languages.

In-kernel Detection Recent work has shown good results for mitigating ADoS attacks by exploiting low level system metrics. Radmin [**74**] and its successor Cogo [**75**] train Probabilistic Finite Automatas (PFAs) offline for each resource of a process they want to monitor, then perform anomaly detection by evaluating how likely the process' transition in the resource space is. Training the PFAs requires days in Radmin, and minutes in Cogo, while FineLame can train accurate models in seconds or hundreds of microseconds. We expect this capability to be helpful in production systems where the model has to be updated, *e.g.*, to account for changes in an application's component. In addition, Cogo reports detection time in the order of seconds, while FineLame's inline detection operates at the scale of the request's complexity—milliseconds in our experiments. Lastly, Radmin/Cogo operate at the granularity of processes/connections. FineLame assumes a worst-case threat model where malicious requests are sent sporadically by compromised clients, and thus operate at this granularity. Per-request detection has the added benefit to enable precise root cause analysis, further enhancing the practicality of FineLame.

Prevention-as-a-Service A recent vein of work proposed "Attack prevention as a Service", where security appliances are automatically provisioned at strategic locations in the network [**76, 77**]. Those techniques are largely dependent on attack detection (to which they do not provide a solution), and thus are orthogonal to our platform, which operates directly at the victim's endpoint.

3.5. Takeaways

FineLame demonstrate how a selected subset of application-level semantic, here an understanding of the critical stages of the request processing pipeline, can support advanced techniques for detecting the source of high tail latency. Service Boosters is designed to expose the request processing pipeline and make it easy to deploy FineLame at scale. Specifically, Service Boosters allows the decomposition of roles

between programmers who declare their DFG and can provide request mappers, performance engineers who write request monitors, and machine learning experts who can build and train anomaly detection models.

In the next chapter of this thesis, we present Perséphone, another Service Boosters module that, leveraging annotations in the DFG, can improve tail latency for high-performance applications operating at the microsecond scale. Perséphone classifies incoming requests and adaptively allocate them resources with the goal of avoiding head-of-line blocking from slower requests contributing to high tail latency.

CHAPTER 4

Perséphone

Data center networks and in-memory systems increasingly have (single) microsecond [**78**] latencies. These latencies are critical for today's complex cloud applications to meet SLOs while fanning out to hundreds of datacenter backend servers [**79, 80**]. At microsecond-scale, the distribution of request processing times can be especially extreme; for example, Redis can process GET/PUT requests in 2μs [**81**] but more complex SCAN and EVAL requests can take hundreds of microseconds or milliseconds to complete. As a result, a single long-running request can block hundreds or thousands of shorter requests. For example, the total load time of a Facebook timeline or an Amazon purchase page depends on hundreds of small, parallel requests being fanned-out from a web front-end to backend services [**80, 79**].

To bound tail latency, especially for short requests, modern datacenter servers run at low utilization to keep queues short and reduce the likelihood that a short request will block behind long requests. For instance, Google reports that machines spend most of their time in the 10-50% utilization range [**82**]. Unfortunately, this approach wastes precious CPU cycles and does not guarantee that microsecond datacenter systems will always meet SLOs for short requests.

In recent years, the community has increasingly relied on bypassing the kernel's network and storage stack to eliminate many sources of long tail latency [**83, 84, 81, 4**]. Though recent kernel-bypass schedulers have improved utilization with shared queues [**85**] and work-stealing [**86, 87**], these techniques only work for uniform and lightly-tailed workloads. For workloads with a wide distribution of response times, Shinjuku [**88**] leverages interrupts for processor sharing; however, Shinjuku's interrupts impose non-negligible delays for single digit microsecond requests and are too expensive to run frequently (i.e., our experiments saw ≈2us per interrupt and preempting as frequently as every 5μs had a high penalty on sustainable load). Furthermore, Shinjuku's non-standard use of hardware virtualization features makes it difficult to use in the datacenter [**85**] and public clouds, *e.g.*, Google Cloud, Microsoft Azure, AWS, *etc.*.

Recent congestion control schemes [**89, 90**], similarly, optimize network utilization and reduce flow completion times by implementing *Shortest-Remaining-Processing-Time (SRPT)*, which is optimal for minimizing the average waiting time. Unlike CPU scheduling, though, switch packet schedulers have a physical 'preemption' unit which is the MTU in the worst case, they process packet headers that include the actual message size, and leverage traffic classes that can prioritize packets based on the size of the flow they belong to, which makes scheduling decisions and policy enforcement easier. A CPU scheduler cannot know in advance for how long each request will occupy the CPU and there is no upper limit on execution time, which

makes the implementation of *SRPT-like* policies, or generally policies that prioritize short requests hard to implement at the microsecond scale.

The unifying factor between congestion control schemes, such as Homa [**90**] and CPU schedulers, such as Shinjuku, that deal with heavy-tail flow and request distributions, respectively, is that they both temporarily multiplex the shared resource. This chapter takes a different approach to CPU scheduling for heavy-tailed service time distributions by taking advantage of parallelism and the abundance of cores on a modern multicore server through application-aware [**7**] spatial isolation of CPU resources.

First, we observe that using the Service Boosters architecture, a kernel bypass scheduler can easily identify the type of incoming requests because users can annotated the application's DFG and classify them at ingress. For many cloud applications, the messaging protocol exposes the required mechanisms to declare request types: Memcached request types are part of the protocol's header [**91**]; Redis uses a serialization protocol specifying commands [**92**]; Protobuf defines Message Types [**93**]; Next, we observe that requests of the same type often have similar processing types, so, given the ability to identify types, we can track past processing times for each type to predict future processing times. Finally, we carefully leave cores idle to prevent short requests from queuing behind indefinitely longer ones.

Inspired by prior research in networking [**94**], our approach goes against the grain for OS schedulers, which commonly prioritize work-conservation. We show that by making a minor sacrifice in the maximum achievable throughput, we can increase the achievable throughput under an aggressive latency SLO and as a result increase the overall CPU utilization of the datacenter.

To implement this approach, we need to tackle two challenges: (1) predict how long each request type will occupy a CPU and (2) efficiently partition CPU resources among types while retaining the ability to handle bursts of arrivals and minimizing CPU waste. To this end, Service Boosters ships Perséphone, a new application aware, kernel-bypass scheduler. Perséphone lets applications define *request filters* and uses these filters to dynamically profile the workload. Using these profiles, Perséphone implements a new scheduling policy, *Dynamic, Application-aware Reserved Cores* (DARC) that uses work conservation for short requests only and is not work conserving for long requests. DARC prioritizes short requests at a small cost in throughput – 5% in our experiments – and is best suited for applications that value microsecond requests. For other applications, existing kernel-bypass scheduler work well, though we believe there is a large set of datacenter workloads that can benefit from DARC.

We prototype Perséphone using DPDK and compare it to two state-of-the-art kernel-bypass schedulers: Shinjuku [**88**] and Shenango [**87**]. Using a diverse set of workloads, we show that Perséphone with DARC can drastically improve requests tail latency and sustain up to 2.3x and 1.3x more load than Shenango and Shinjuku, respectively, at a target SLO. In addition, these improvements come at a lower cost to long requests than Shinjuku's preemption technique, highlighting the challenges of traditional OS scheduling techniques at microsecond scale.

Tab. 1. Unlike most existing kernel-bypass OS schedulers, DARC is not work conserving. It extracts request types from incoming requests, estimates how long a request will occupy a CPU before scheduling it and reserves workers for short requests to minimize dispersion-based head-of-line blocking.

Policy	Exploit typed queues	Non Work conserving	Non preemptive	Example System
d-FCFS	✗	✓	✓	IX [**4**] Arrakis [**81**]
c-FCFS	✗	✗	✓	ZygOS [**86**] Shenango [**87**]
TS	✓	✗	✗	Shinjuku [**88**]
DARC	✓	✓	✓	Perséphone

4.1. The case for idling

For workloads with wide service time distribution, long requests can block short requests even when queues are short because long requests can easily occupy all workers for a long time. We refer to this effect as *dispersion-based head-of-line blocking*. To better understand how dispersion-based blocking affects short requests, we look beyond request latency and study *slowdown*: the ratio of total time spent at the server over the time spent doing pure application processing [**95**].

Slowdown better reflects the impact of long requests on short requests. For heavy-tailed workloads, short requests experience a slowdown proportional to the length of the tail. More concretely, consider the following workload, similar to Zygos' "bimodal-2" [**86**], a mix of 99.5% short requests running for 0.5μs and 0.5% long requests executing in 500μs. A short request blocked behind a long one can experience a slowdown of up to 1001, while a long request blocked behind a short request will see a slowdown of 1.001. As a result, a few short requests blocked by long requests will drive the slowdown distribution and increase tail latency.

Using this workload, we simulate four scheduling policies, including DARC, listed in Table 1. *Decentralized first come, first served* (d-FCFS) models Receive Side Scaling, widely used in the datacenter today [**96, 97**] and by the seminal IX operating system [**4**]. With d-FCFS, each worker has a local queue and receives an even share of all incoming traffic. *Centralized first come, first served* (c-FCFS) uses a single queue to receive all requests and send them to idle workers. c-FCFS is usually used at the application level — for example, web servers (e.g., NGINX) often use a single dispatch thread — and captures recent research on kernel-bypass systems [**86, 87**], which simulate c-FCFS with per-worker queues and work stealing. *Time Sharing* (TS) is used in the Shinjuku system [**88**], with multiple queues for different request types and interrupts at the microsecond scale using Dune [**98**]. We simulate TS with a 5μs preemption frequency and 1μs overhead per preemption, matching Shinjuku's reported $\approx$ 2000 cycles overhead on a 2GHz machine.

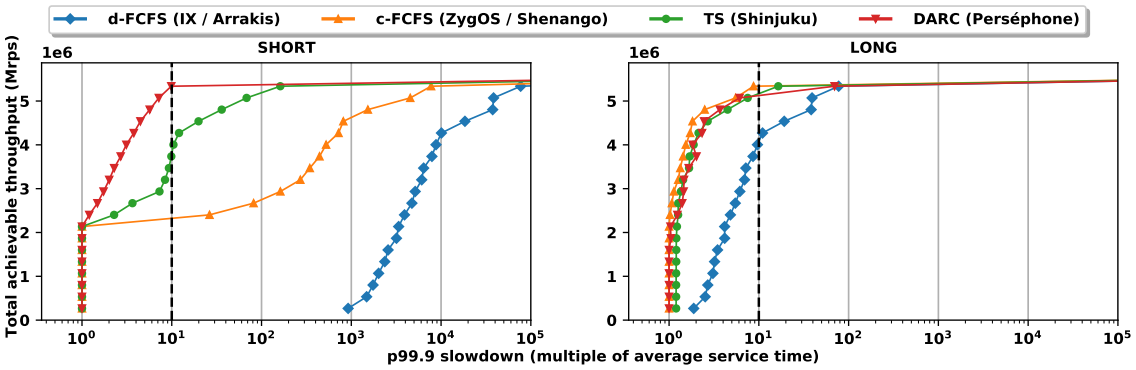


Fig. 4.1. Simulated achievable throughput as a function of 99.9th percentile slowdown for the policies listed in Table 1 on a 16 cores system and a workload composed of 99.5% short requests (0.5μs) and 0.5% long requests (500μs). For a target SLO of 10 times the average service time *for each request type*, c-FCFS and TS can only handle 2.1 and 3.7 Millions requests per second (Mrps), respectively. DARC can sustain 5.1 Mrps for the same objective. The Y axis represents the total achievable throughput for the entire workload.

Figure 4.1 shows our simulation results assuming an ideal system with no network overheads. We use 16 workers, simulate 1 second of requests distributed under Poisson, and report the observed slowdown for the 99.9th percentile of each type of requests — so as to capture the impact of the 0.5% long requests on the tail — at varying utilizations, up to a maximum of 5.3 million requests per second (Mrps).

d-FCFS performs poorly; it offers an uncontrolled form of non work conservation where workers sit idle while requests wait in other queues. Additionally, d-FCFS has no sense of request types: workers might process a long request ahead of a short one if it arrived first. c-FCFS performs better because it is work conserving but short requests will block when all workers are busy processing long requests. To meet a target SLO of 10x slowdown for *each type of requests*, c-FCFS must run the server at 2.1 Mrps, 40% of the peak load. Shinjuku's TS policy fares better than c-FCFS and d-FCFS, being both work conserving and able to preempt long requests: it maintains slowdown bellow 10 up to 3.7 Mrps, 70% of the peak load. However, this simulation accounts for an optimistic 1μs preemption overhead and overlooks the practicality of supporting preemption at the microsecond scale (*Cf.* §4.5).

The DARC approach: Our key insight is that prioritizing short requests is critical to protect their service time, an observation the networking community has already made when designing datacenter congestion control schemes [**89, 90, 94**]. However, using traffic classes and bounded buffers is challenging for CPU scheduling since schedulers do not know how long a request may occupy a CPU and preemption is unaffordable at single-digit microsecond scales. We observe that *leaving certain cores idle for readily handling potential future (bursts of) short requests is highly beneficial* at microsecond scale. For a request that takes 1 μs or less, even preempting as frequently as every 5μs introduces a 6x slowdown. Instead, given an understanding of each request's potential processing time, a request type aware, not work conserving

policy can reduce slowdown for short requests by estimating their CPU demand and dedicating workers to them. These workers will be idle when short requests do not arrive, but when they do, these requests are guaranteed to not be blocked behind long requests.

As seen in Figure 4.1, DARC can meet the 10x slowdown SLO target for both type of requests at 5.1 Mrps. This represents *2.5x and 1.4x more sustainable throughput* than c-FCFS and Shinjuku's preemption policy. At this load, short requests experience 9.87μs p99.9th tail latency, *3 and 1 orders of magnitude* smaller than c-FCFS and TS with 7738μs and 161μs, respectively. To achieve this, DARC asks programmers for a *request filter* to identify types, estimates their CPU demand, and reserves 1 worker for short requests at a small penalty of 5% achievable throughput. Counter-intuitively, although DARC wastes cycles idling, it *reduces the overall number of machines needed to serve a workload* because servers can run at much higher utilization while retaining good tail latencies for short and long requests.

4.2. DARC scheduling

The objective of DARC is to improve tail latency for single-digit microsecond requests in cloud workloads without preemption. Like recent networking techniques that co-design the network protocol and switches priority queue management [**89, 90, 94**] to favor small messages, we protect short requests at backend servers by extracting their type, understanding their CPU demand, and dedicating enough resources to satisfy their demand.

In this section, we describe the challenges associated with implementing these techniques as a CPU scheduling policy, then present the DARC scheduling model, how to compute reservations and when to update them. Table 2 describes the notation used throughout this section.

Challenges. Protecting short requests in a dynamic way through priority queues and non work conservation is difficult because we need to (1) predict how long each request type will occupy a CPU and (2) partition CPU resources among types while retaining the ability to handle bursts of arrivals and minimizing CPU waste.

The first challenge stems from the granularity of operation DARC is targeting, microsecond scales, and from the need to react to changes in workload. We tackle this challenge with a combination of low-overheads workload profiling and queuing delay monitoring, using the former to build a fingerprint of requests' CPU demand and the latter as a signal that this fingerprint might have significantly changed. This section describes the technique and Sec. 4.3 its implementation.

The second challenge can be detailed in two parts: burst-tolerance and CPU waste. First, though reducing the number of cores available to a given request type forbids it from negatively impacting other types, it also reduces its ability to absorb bursts of arrivals [**99**]. We solve this tension by enabling *cycles stealing* from shorter types to longer ones, a mechanism in which short requests can execute on cores otherwise reserved for longer types — but not the opposite. The rationale for stealing is that shorter requests comparatively cause less slowdown to long requests. Note that cycle stealing is a similar concept to work stealing [**86, 87**] but is different in practice, as

performed from the DARC dispatcher rather than from application workers (so it does not require expensive cross-worker coordination).

Second, and similarly to message types and priority queues in network devices, the number of request types can be different than the number of CPU cores on the machine, so very likely the demand for each request type will be fractional — *i.e.*, a request type could require 2.5 workers on average. As a result, we need to determine a strategy for sharing — or not — CPU cores between certain request types. Sharing cores leads to a tension: regrouping types onto the same cores risks dispersion-based blocking, but always giving entire cores to types with fractional demand can lead to over-provisioning and starving other types. We handle this tension with two mechanisms: grouping types together and providing spillway cores. Grouping lets all request types fit onto a limited number of cores and reduces the number of fractional ties while retaining the ability to separate types based on processing time. Spillway cores allows DARC to always service types with little average CPU demand (typically much less than an entire core) as well as undeclared, unknown requests.

Tab. 2. Notation used to define DARC

Symbol	Description
N	Number of request types
S	Average service time
τ	A request type
$\tau.S$	Type's average service time
$\tau.R$	Type's occurrence
δ	Service time similarity factor for two types

Algorithm 1 Request dispatching algorithm

```
procedure DISPATCH(Γ)
    w ← None
    for τ ∈ Γ do
        if τ.queue == ∅ then
            continue
        else
            workers ← τ.reserved ∪ τ.stealable
            for worker ∈ workers do
                if worker.is_free() then
                    w ← worker
                    break
            if w ≠ None then
                r ← τ.queue.pop()
                schedule(r, w)
```

Scheduling model. DARC presents a single queue abstraction to application workers: it iterates over typed queues sorted by average service time and dequeues them in a first come, first served fashion. Requests of a given type can be scheduled not only on their reserved cores but also steal cycles from cores allocated to longer types — a concept used in Cycle Stealing with Central Queue (CSCQ), a job dispatching policy for compute clusters [**100**]. Algorithm 1 describes the process of worker selection; For each request type registered in the system, if there is a pending request in that type's queue, DARC greedily searches the list of reserved workers for an idle worker. If none is found, DARC searches for a stealable worker. If a free worker is found, the head of the typed queue is dispatched to this worker. When a worker completes a request, it signals completion to the DARC dispatcher.

Algorithm 2 Worker reservation algorithm

```
procedure RESERVE(Types, δ)
    // First group together similar request types
    groups = group_types(Types, δ).sort()
    // Then attribute workers
    S ← Σ_{j=0}^{N} S_j * R_j
    n_reserved = 0
    for g ∈ groups do
        g.S = Σ τ.S * τ.R ∀ τ ∈ g
        d = g.S / S
        P ← round(d)
        if P == 0 then P ← 1
        for i ← 0; i <P; i++ do
            g.reserved[i] ← next_free_worker()
            n_reserved++;
        // Set stealable workers
        n_stealable ← num_workers - n_reserved;
        for i ← 0; i < n_stealable; i++ do
            g.stealable[i] ← next_free_worker()
```

DARC reservation mechanism. The number of workers to dedicate to a given request type is based on the average CPU demand of the type at peak load. We compute this demand using the workload's composition, normalizing the contribution of each request type's average service time to the entire workload's average service time. The contribution of a given request type is its average service time multiplied by its occurrence ratio as a percentage of the entire workload. Specifically, given a set of N request types $\{\tau_i\ ; i = 0 \dots N\}$, the average CPU time demand Δ_i of τ_i with service time S_i and occurrence ratio R_i is:

$$0 \leq \frac{S_i * R_i}{\sum_j^N S_j * R_j}, \leq 1 \tag{4.1}$$

Given a system with W workers, this means that we should attribute $\Delta_i * W$ workers to τ_i.

Because CPU demand can be fractional and given the non-preemptive requirement we set for the system, we need a strategy to attribute fractions of CPUs to request types. For each such "fractional tie", we have to make a choice: either `ceil` fractions and always grant entire cores or `floor` fractions and consolidate fractional CPU demands on *shared cores*. The former risks over-provisioning certain types, at the cost of others, while the latter risks creating dispersion-based blocking by mixing long and short requests onto the same core(s).

Our approach combines the two: first we decrease the number of "fractional ties" by grouping request types of similar processing times, computing a CPU demand for the entire group, and second we round this demand. As a result, for N groups, if f_i is the fractional demand of group i, the average CPU waste for DARC across all N groups is $\sum_i^N 1 - f_i$ if $f_i \geq 0.5$, else it is 0. In practice, during bursts, because we selectively enable work conservation through work stealing for shorter requests, CPU waste is smaller.

Algorithm 2 describes the reservation process. First, we identify similar types whose average service time falls within a factor δ of each other. Next, we compute the demand for each group and accordingly attribute workers to meet it, rounding fractional demands in the process. We always assign at least one worker to a group. DARC grouping strategy can still result in earlier groups — of shorter requests — consuming all CPU cores. For example, a group of long requests with a CPU demand smaller than 0.5 will not find any free CPU core. To provide service to these groups, we set aside "spillway" cores. If there are no more free workers, `next_free_worker()` returns a spillway core. In our experiments (§4.4), we use a single spillway core.

Finally, we selectively enable work conservation for shorter requests and let each group steal from workers not yet reserved, *i.e.*, workers that are to be dedicated to longer request types. This lets DARC better tolerate bursts of shorter requests with little impact on the overall tail latency of the workload.

As we process groups in order of ascending service time, we favor shorter requests, and it is possible for our algorithm to under-provision long requests — but never deny them service thanks to spillway cores. Operators can tune the δ grouping factor to adjust non work conservation to their desired SLOs. Grouping lets DARC handle workloads where the number of distinct types is higher than the number of workers.

Profiling the workload and updating reservations. At runtime, the DARC dispatcher uses *profiling windows* to maintain two pieces of information about each request type: a moving average of service time and an occurrence ratio. These are the S_i and R_i of equation 4.1. The dispatcher gathers them when application workers signal work completions. The dispatcher uses queuing delay and variation in CPU demand as performance signals. If the former goes beyond a target slowdown SLO and the latter deviates significantly from the current demand, the dispatcher proceeds to update reservations and transition to the next windows. During the first windows, at startup, the system starts using c-FCFS, gathers samples, then transitions to DARC. This technique lets DARC cope with changing workloads where a type's profile change (effectively, *misclassification*). During a profiling windows, unknown or unexpected requests can use the spillway core(s) to execute. We discuss the sensibility of this mechanism in Sec. 4.3.3.3.

4.3. Perséphone

We implement DARC within a kernel-bypass scheduler called Perséphone and provided with Service Boosters. Though DARC requires no special hardware or major application modifications, Perséphone must meet the following requirements to support microsecond kernel-bypass applications: (1) the networking stack must be able to efficiently sort requests by type in the data path, (2) the scheduler must be able to quickly make per-request scheduling decisions, and (3) profiling workloads and updating DARC reservation must present low overheads.

Perséphone meets the first requirement with an API for capturing request types, *request filters*. Using request filters, programmers provide a way for the system to classify requests based on types as they enter the system. Perséphone meets the remaining two requirements with a carefully architected networking stack, profiler, and scheduler packaged in a user-level library with the application.

4.3.1. System model. Perséphone is designed for datacenter services that must handle large volumes of traffic at microsecond latencies. Examples include key-value stores, fast inference engines [**101**], web search engines and RESTful microservices. We assume the application uses kernel-bypass for low latency I/O (e.g., with DPDK [**102**] or RDMA [**103**]) and performs all application and network processing through Perséphone.

4.3.2. Request filters. Perséphone relies on user-defined functions, *i.e.*, "request filters" to classify application-level payloads. A request filter accepts a pointer to an application payload (Layer 4 and above) and returns a request type. If the filter cannot recognize a request, Perséphone categorizes it as `UNKNOWN` and places it in a low priority queue. Though most of our target application use optimized protocols such as Redis' RESP [**92**] that allow a filter to look-up for a header field to parse the request type, we opted for generality and allowing users to write arbitrarily complex filters. There is, of course, a performance trade-off: a non-optimized request filter will impact the dispatcher's performance because request filters are "bumps-in-the-wire" on the dispatching critical path. We leave it to users to quantify this trade off based on the performance they wish to obtain from the dispatcher (*i.e.*, how many requests per second should it be able to sustain). While a complete study of filter performance is out of scope for this paper, we found that for standard protocols where the request type's position is known in the header, our dispatcher can process up to 7 millions packets per second on our testbed, a number competitive with existing kernel-bypass schedulers.

4.3.3. Perséphone architecture. Perséphone consists of three components, shown in Figure 4.2: one or many *net Service Units* dequeueing packets from the network card, a *dispatcher* applying request filters and performing DARC scheduling, and *application Service Units* performing application processing (*e.g.*, fetching the value from the key-value store). These components operate as an event-driven pipeline and process packets as follows:

1. On the ingress path, the net Service Unit takes packets from the network card and pushes them to the dispatcher, which

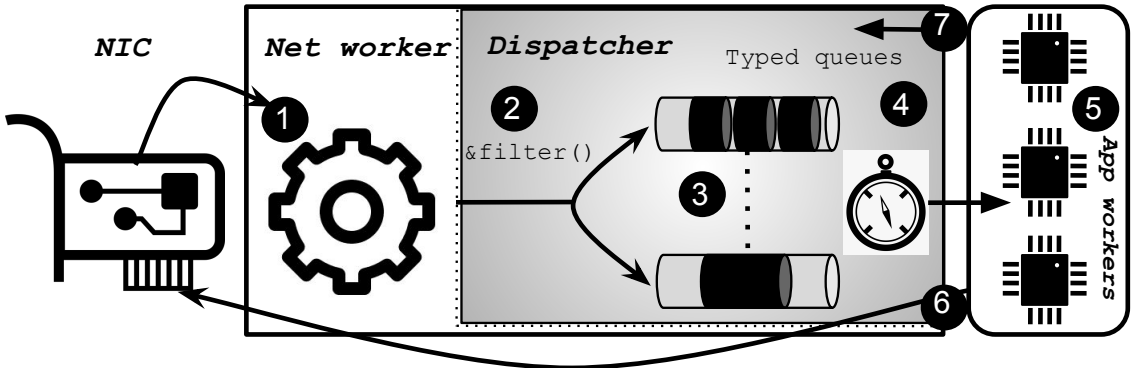


Fig. 4.2. Perséphone architecture. After the net Service Unit processes incoming packets, the dispatcher classifies requests using a user-defined filter. Requests wait in typed queues for DARC to push them to Service Unit.

2. Classifies incoming requests using a user-defined *request filter* and
3. Stores them in *typed queues*, *i.e.*, buffers specialized for a single request type.
4. The dispatcher, running DARC, selects a request from a typed queue and pushes it to a free *application Service Unit*.
5. The Service Unit processes the request, formats a response buffer, and
6. Pushes a pointer to that buffer to the NIC. In addition,
7. Service Units notify the dispatcher of requests' completion.

4.3.3.1. *Networking.* Both the net Service Unit and application Service Units receive a *network context* at initialization. This context gives them unique access to receive and transmit queues in the NIC. Perséphone registers a statically allocated memory pool with the NIC for contexts to quickly allocate new buffers when receiving packets. This memory pool is backed by a multi-producer, single-consumer ring so Service Units can release buffers after transmission. Both the net and application Service Units use a thread local buffer cache to decrease interactions with the main memory pool. For requests contained in a single application-level buffer, we perform zero-copy and pass along to Service Units a pointer to the network buffer. To issue a response on the transmit path, the Service Unit reuses the ingress network buffer to host the egress packet, reducing the number of distinct network buffers in use (with the goal of allowing all buffers to fit in the Last Level Cache space used by DDIO [**104**] — usually 10% of the LLC).

4.3.3.2. *Component Communication.* The dispatcher uses single-producer, single-consumer circular buffers to share requests and commands with application Service Units in a lockless interaction pattern. We use a lightweight RPC design inspired by Barrelfish [**105**], where both sender and receiver synchronize their send/read heads using a shared variable. To reduce cache coherence traffic between cores, the sender synchronizes with the receiver — to update the read head and avoid overflows — only when its local state shows the buffer to be full. In our prototype, operations on that channel take 88 cycles on average.

4.3.3.3. *Dispatcher.* The dispatcher maintains three main data structures: a list of `RequestType` objects, which contains type information such as the type ID and instrumentation data; typed request queues; and a list of free Service Units. In addition, the dispatcher holds a pointer to a user-defined request filter. The list of free Service Units is updated whenever a request is dispatched and each time application Service Units notify the dispatcher about work completion; this is done using a specific control message on the memory channel shared between dispatcher and each Service Unit. Finally, the dispatcher maintains profiling windows, during which it computes a moving average of service times by request type and increment a counter for each type seen so far. DARC uses these profiling windows to compute resource allocation (§4.2) and adjust to changes in the workload's composition. In our prototype, at the median, updating the profile of a request takes 75 cycles, checking whether an update is required takes about 300 cycles, and performing a reservation update takes about 1000 cycles.

To control the sensibility of the update mechanism in face of bursty arrivals, we set a lower bound on the number of samples required to transition — 50000 in our experiments — and the minimum deviation in CPU demand from the current allocation — 10% in our experiments. As a measure of flow control, when the system is under pressure and Service Units cannot process requests as fast as they arrive, the dispatcher drops requests from typed queues are full. This allows to shed load only for overloaded types without impacting the rest of the workload.

4.3.3.4. *Application Service Units.* Upon receiving a pointer toward a request, application Service Units dereference it to access the payload. As an optimization, they can access the request type directly from the `RequestType` object rather than duplicating work to identify needed application logic (e.g., to differentiate between a SET or GET request). Once they finish processing the request, they reuse the payload buffer to format a response and push it to the NIC hardware queue using their local network context. Finally, they signal work completion to the dispatcher.

Tab. 3. Perséphone's API exposes calls to register and update request types and register request filters for ingress traffic classification.

System call	Description
`psp_init(hw_cfg, n_workers, *filter, [types])`	Initialize NIC, CPUs, and Service Units; register request types and configure a request filter
`psp_register_rtype(type)`	Register a new request type
`psp_set_rf(*filter)`	Update the request filter used to classify ingress requests
`psp_register_service_unit()`	Add a new Service Unit

4.3.4. Perséphone's API. Table 3 describes the system call interface exposed by Perséphone. Perséphone exposes a set of API calls to manage the system. Application call `psp_init` to set up the hardware managed by the libOS (e.g., network interfaces, CPUs), the network stack and the request dispatcher. Through this call, users also register a list of expected request types and the filter function to be used at runtime for classification. At runtime, users can call `psp_register_rtype` and `psp_set_rf` to update request types or register new ones and to update the request filter. Finally, the system can be scaled up and down by adding application Service Units using `psp_register_service_unit`.

4.3.5. Example usage. We now walk through a simple example where a programmer sets up Perséphone for an in-memory database and a RESTful service. Both use an application-defined protocol. For the in-memory database, this protocol includes, alongside other necessary input parameters, the type of transaction to be executed. For the REST API, the protocol is HTTP, and the request type the API endpoint (either "help" or "compress"). Snippet 4.3 presents this workflow: the user defines request filters for both applications (l. 5-19) and create two request types (l. 22-24); then, she initializes Perséphone (l. $27 - 30$) and register 16 Service Units (l. 32-35).

```
#include <libpsp.h>

/* Filter functions examples */
#define TXN_ID_POS 42
enum ReqType txn_request_filter(char *payload){
  return *static_cast<ReqType>(
    (int *)payload[TXN_ID_POS]
  );
}

#define COMPRESS_ENDPOINT "/compress"
#define HELP_ENDPOINT "/help"
enum ReqType http_request_filter(char *payload) {
  if (strstr(payload, COMPRESS_ENDPOINT))
    return ReqType::COMPRESS;
  if (strstr(payload, HELP_ENDPOINT))
    return ReqType::HELP;
  return ReqType::UNKNOWN;
}

/* Init Persephone */
ReqType types[2] = {
  COMPRESS_ENDPOINT, HELP_ENDPOINT
};

uint32_t n_workers = 16;
psp_init(
  config_filepath, n_workers,
  &http_request_filter, types,
)

/* Register Service Units */
for (int i = 0; i < n_workers; ++i) {
  psp_register_service_unit(units[i]);
}
```

Fig. 4.3. Configuring Perséphone using the C++ API bindings.

4.4. Evaluation

We built a prototype of Perséphone, in about 2600 lines of C++ code, to evaluate DARC scheduling against policies provided by Shenango [**87**] and Shinjuku [**88**]:

- For a workload with 100x dispersion between short and long requests, Perséphone can sustain 2.35x and 1.3x more throughput compared to Shenango and Shinjuku, respectively.
- For a workload with 1000x dispersion, Perséphone can sustain 1.4x more throughput than Shenango and reduce slowdown by up to 2x over Shinjuku.

- For a workload modeled on the TPC-C benchmark, Perséphone reduces slowdown by up to 4.6x over Shenango and up to 3.1x over Shinjuku.
- For a RocksDB application, DARC can sustain 2.3x and 1.3x higher throughput than Shenango and Shinjuku, respectively.

Tab. 4. Workloads exhibiting 100x and 1000x dispersion.

Workload	Short		Long	
	Runtime (μs)	Ratio	Runtime (μs)	Ratio
High Bimodal	1	50%	100	50%
Extreme Bimodal	0.5	99.5%	500	0.5%

Tab. 5. The TPC-C benchmark models operations of an online store. Payment and NewOrder transactions are most frequent.

Transaction name	Runtime (μs)	Ratio	Dispersion
Payment	5.7	44%	1x
OrderStatus	6	4%	1.05x
NewOrder	20	44%	3.3x
Delivery	88	4%	15.4x
StockLevel	100	4%	17.5xx

4.4.1. Experimental setup. Workloads. We model workloads exhibiting different service time dispersion after examples found in academic and industry references. Often such workloads exhibit n-modal distributions with either an equal amount of short and long requests (*e.g.*, workload A in the YCSB benchmark [**106**]) or a majority of short requests with a small amount of very long requests (e.g. Facebook's USR workload [**107**]). Dispersion between shorter and longer requests is commonly found to be two orders of magnitude or more [**108, 109, 110**]. We evaluate *High Bimodal* and *Extreme Bimodal* (Table 4), two workloads that exhibit large service time dispersion, and *TPC-C* (Table 5), which models requests in the eponymous benchmarking suite [**111**], a standardized OLTP model for e-commerce. Finally, we evaluate DARC using RocksDB, an in-memory database used at Facebook [**112**].

With *High Bimodal* long requests represent 50% of the workload but "only" exhibit 100x dispersion. With *Extreme Bimodal*, long requests are much slower — 1000x slower — but very infrequent (0.5% of the mix). We profile *TPC-C* transactions with an in-memory database and run it as a synthetic workload. Our goal with this *TPC-C* is to evaluate how Perséphone performs with an n-modal request distribution. The workload consists of five request types with moderate service time dispersion — at most 17.5x between infrequent `StockLevel` requests and frequent `Payment` requests. We assume that requests are not dependent on each other. Finally, the RocksDB workload is made of 50% GETs and 50% SCANs requests, executing for 1.5μs and 635μs, respectively, and exhibiting a 420x dispersion factor. This workload strikes a balance between *High Bimodal* and *Extreme Bimodal*.

Performance metrics.

We present two performance views: (i) the slowdown at the tail taken across all requests in the experiment, and (ii) the *typed* tail latency, i.e, a selected percentile over *only* the type's response times' distribution. These views help us to understand the various trade-offs offered by the systems and policies under evaluation. For both metrics, we use the 99.9th percentile and plot them as a function of the total load on the system.

Client. The client is a C++ open loop load generator that models the behavior of bursty production traffic. It generates requests under a Poisson process centered at the workloads' average service time. Each experiment runs for 20 seconds and we discard the first 10% of samples to remove warm-up effects. We ran our experiments for several minutes and found the results similar. To interact with the server, we use a simple protocol where TPC-C transactions ID, RocksDB query ID, and synthetic workload requests types are located in the requests' header. We accordingly register request filters on the server to map these IDs to request types. Request filters add a one-time ≈ 100 nanoseconds overhead to each request.

Systems. In addition to Perséphone, we compare two state-of-the-art systems: Shenango and Shinjuku. Shenango's *IOKernel* uses RSS hashes to steer packets to application cores, which perform work stealing to balance load and avoid dispersion-based blocking, in a fashion similar to ZygOS [**86**]. We also compare to a version of Shenango with work stealing disabled, to evaluate d-FCFS. We choose Shenango over ZygOS due to its more recent implementation and its support for UDP. Shinjuku implements microsecond-scale, user-level preemption by exploiting Dune's virtualization [**98**] at up to 5μs frequency. Leveraging this ability to preempt, Shinjuku implements a single queue policy, where preempted requests are enqueued at the tail of the queue, and a multi-queue policy with a queue per request type and where preempted requests are enqueued at the head of their respective queue. The multi-queue policy selects the next queue to dequeue using a variant of Borrowed Virtual Time [**113**]. Across experiments, DARC updates reservations whenever a request experiences queuing delays of ten times its average profiled service time. Lastly, all systems use UDP networking.

Testbed. We use 7 Cloudlab [**114**] c6420 nodes (6 clients, 1 server), each equipped with a 16-core (32-thread) Intel Xeon Gold 6142 CPU running at 2.60GHz, 376GB of RAM, and an Intel X710 10 Gigabit NIC. The average network round trip time between machines is 10μs. We disabled TurboBoost and set `isolcpu`. Shinjuku and Perséphone run on Ubuntu 16.04 with Linux kernel version 4.4.0. Shenango runs on Ubuntu 18.04 with Linux kernel version 5.0. Shinjuku uses one hyperthread for the net worker and another for the dispatcher, collocated on the same physical core. Shenango runs its IOKernel on a single core, and Perséphone runs both its net worker and dispatcher on the same hardware thread. All systems use 14 worker threads running on dedicated physical cores. For Shenango, we provision all cores at startup and disable dynamic core allocation since we want to evaluate performance for a single application and Shenango otherwise re-assign cores to multiple applications running on the same machine.

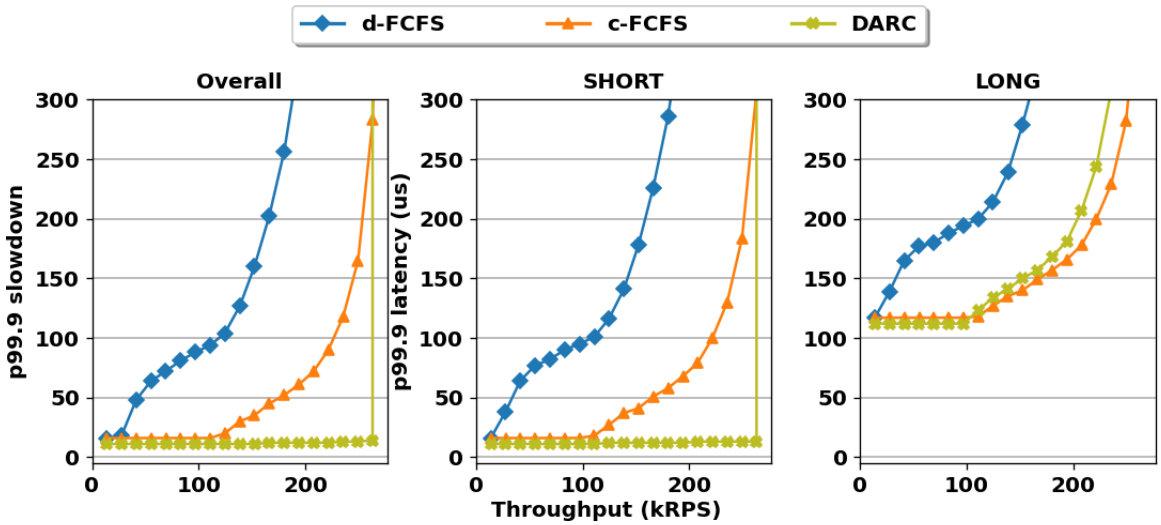


Fig. 4.4. Evaluating DARC on *High Bimodal* (50.0:1.0 – 50.0:100.0) within Perséphone. The first column is p99.9 overall slowdown, the second and third p99.9 latency for short and long requests, respectively. For all columns, the X axis is the total load on the system. DARC improves slowdown over c-FCFS by up to 15.7x, at a cost of up to 4.2x increased latency for long requests.

4.4.2. DARC versus existing policies. To validate that DARC improves performance of short requests compared to c-FCFS and d-FCFS, we run these policies on *High Bimodal* in Perséphone. Figure 4.4 presents our results. c-FCFS improves the tail latency of short requests over d-FCFS by eliminating local hotspots at workers, a result consistent with previous work [**86**]. However, because c-FCFS does not protect the service time distribution of short requests, they experience dispersion-based blocking from long requests. With c-FCFS, short requests experience 309μs p99.9 latency at 260kRPS, driving slowdown for the entire workload to 283x. In contrast, DARC reserves 1 core for short requests and schedules them first, reducing slowdown upon c-FCFS **by a factor of 15.7**[4] and can sustain **2.3x higher throughput** for a SLO of 20μs for short requests. This comes at the cost of up to a 4.2x increase in tail latency for long requests. The average CPU waste occasioned by reserving the core is 0.86 core. Because slowdown is driven by short requests and the two graphs are very similar, we omit short requests in the next sections and focus on overall slowdown and tail latency for long requests.

4.4.3. How much non work-conservation is useful? We empirically validate DARC's reservation mechanism (§4.2) by manually configuring the number of workers dedicated to short requests from 0 to 14. We call this version "DARC-static". It schedules short requests first and let them execute on all the cores. When the number of reserved workers is 0, DARC-static is equivalent to a simple Fixed Priority policy. Figure 4.5 presents the overall slowdown experienced by *High Bimodal* (a)

[4]The network contributes 10μs to response time. At 260kRPS, short requests experience 309μs end-to-end latency with c-FCFS and 18μs with DARC. This means that **server-side slowdown is 37x better with DARC**.

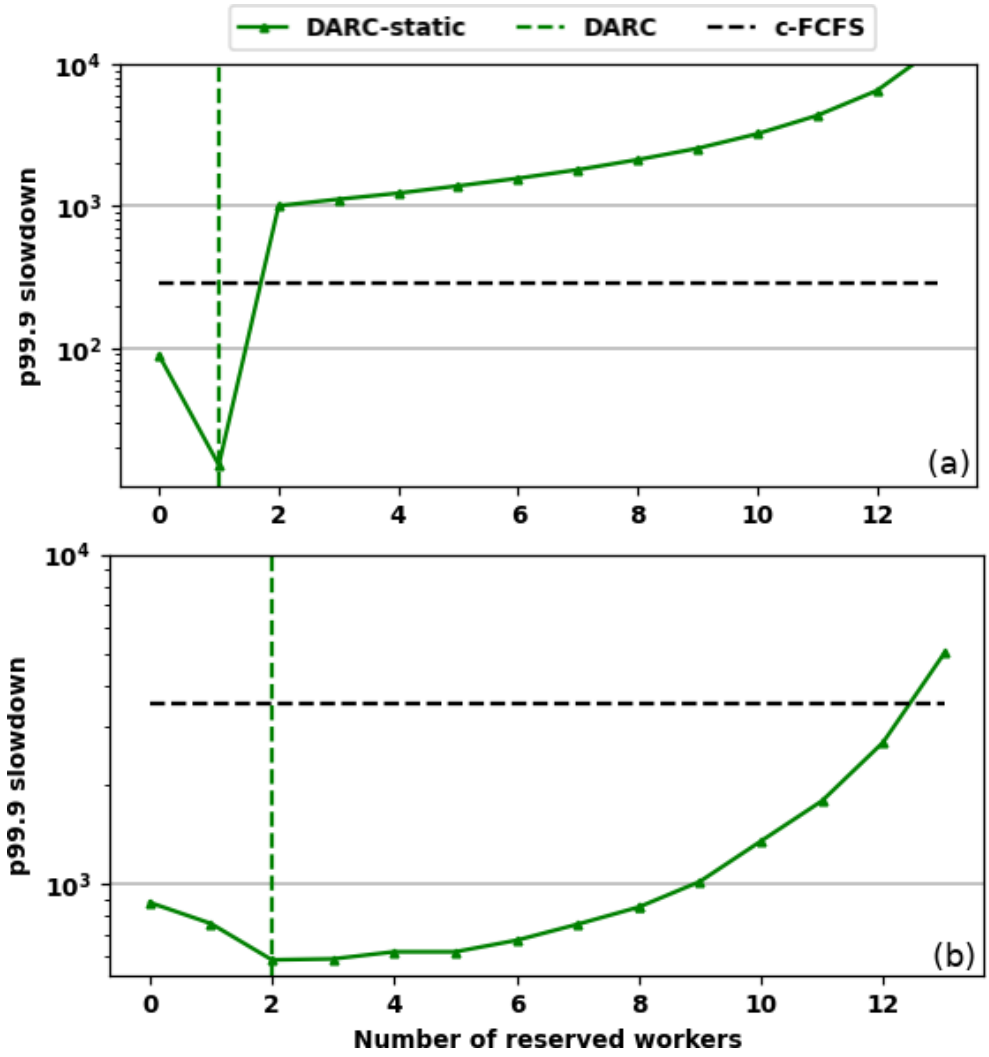
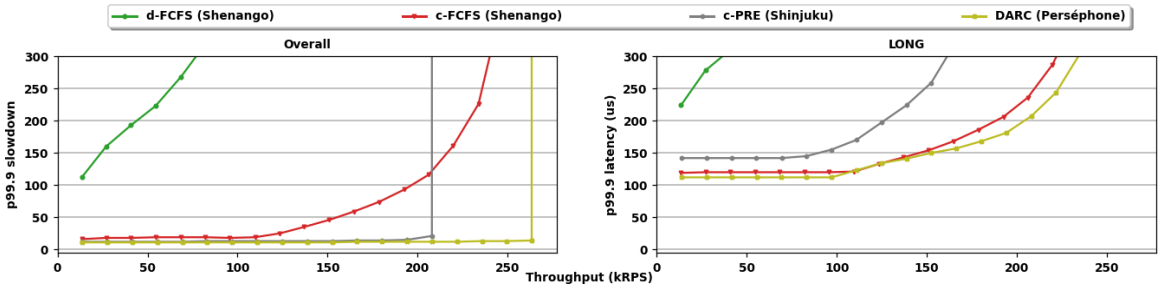


Fig. 4.5. Gradually adjusting the degree of work conservation (" DARC-static") with *High Bimodal* and *Extreme Bimodal* at 95% load. Reserving 1 (a) and 2 (b) cores decreases slowdown by 4.4x and 1.5x, respectively.

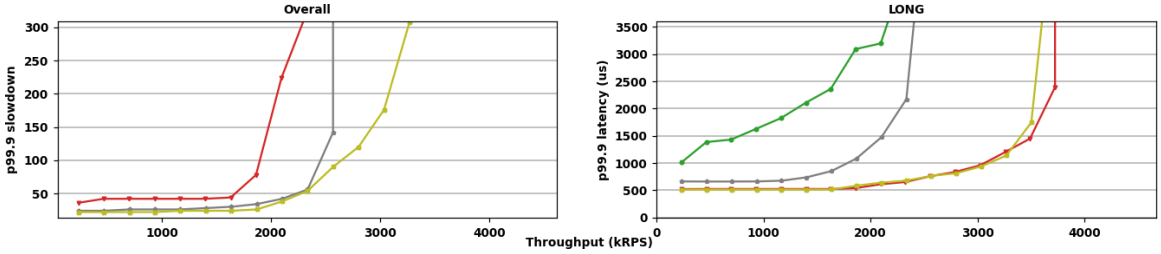
and *Extreme Bimodal* (b) at 95% load. We observe that for the former, the best slowdown — a 4.4x improvement — is achieved with 1 core, and for the latter with 2 cores — a 1.5x improvement. Those settings validate DARC's selection, as described in Sec. 4.4.2 and Sec. 4.4.4.

For comparison, we draw the slowdown line offered by c-FCFS on Perséphone. Reserving too many workers results in long requests being starved. Simple Fixed Priority scheduling results in dispersion-based blocking for short requests.

4.4.4. Comparison with Shinjuku and Shenango. Figures 4.6a and 4.6b show the performance experienced by *High Bimodal* and *Extreme Bimodal* in all three systems. Figure 4.7 presents *TPC-C* performance, and Figure 4.8 RocksDB performance. Shenango implements d-FCFS and c-FCFS. Shinjuku uses its multi-queue policy for *High Bimodal*, *TPC-C*, and RocksDB; and its single queue policy



(a) *High Bimodal* For a 20x slowdown target, DARC can sustain 2.35x and 1.3x more traffic than Shenango and Shinjuku, respectively.



(b) *Extreme Bimodal* For a 50x slowdown target, DARC can sustain 1.4 more load than Shenango. DARC also reduces slowdown by up to 2x over Shinjuku. Note the different Y axis for slowdown and long requests tail latency.

Fig. 4.6

for *Extreme Bimodal* (per the Shinjuku paper [**88**]). We invested significant efforts in tuning Shinjuku for short requests performance and preempting as frequently as possible. We could only sustain 75% for *High Bimodal* (5μs interrupts) and RocksDB (15μs interrupts), and 55% load for *Extreme Bimodal* (5μs interrupts), after which the system starts dropping packets and eventually crashes (despite sustaining close to 4.5 millions 1μs RPS without preemption on our testbed). We found that reducing the frequency of preemption helped sustain higher loads at the expense of shorter requests. *TPC-C* is most favorable to Shinjuku because the services times are higher and dispersion smaller. Shinjuku can handle 85% of this load when preempting every 10μs.

4.4.4.1. High Bimodal. Shinjuku improves the tail latency of short requests over Shenango's c-FCFS by preempting long requests. However, Shinjuku aggressively preempts every 5μs to maintain good latency for short requests and adds a constant overhead — at least 20% in this experiment — to preempted requests. As a result, it can sustain only 75% of the load before dropping requests. In comparison, DARC reserves 1 core for short requests and can sustain **2.35x and 1.3x more load** than Shenango and Shinjuku, respectively, for a target slowdown of 20x. At 75% load, DARC **reduces slowdown by 10.2x and 1.75x over Shenango and Shinjuku**, respectively. Perhaps more importantly, compared to Shinjuku's preemption system DARC consistently provides better tail latency for long requests. We also observe that Perséphone's centralized scheduling offers better performance for long requests than Shenango compared to c-FCFS on Perséphone because Perséphone does not have to approximate centralization with work stealing.

4.4.4.2. Extreme Bimodal. We observe similar trends for this workload. For a target 50x slowdown, both Shinjuku and Perséphone can sustain **1.4x higher throughput** than Shenango. However, past 55% load, the overheads of aggressively preempting every 5μs is too expensive and Shinjuku starts dropping packets. For long requests, preemption overheads are always at least 24% (620μs for 500μs requests). In contrast, Perséphone reserves 2 cores to maintain good tail latency for short requests and can sustain **1.25x more load** while **reducing slowdown up to 2x** over Shinjuku. All the while, Perséphone provides tail latency for long requests competitive with Shenango. For this workload the CPU waste occasioned by DARC is, on average, 0.67 core.

4.4.4.3. TPC-C. For this workload, DARC groups `Payment` and `OrderStatus` transactions (group A), lets `NewOrder` transactions run in their own group (B), and groups `Delivery` and `StockLevel` transactions (group C). DARC attributes workers 1 and 2 to group A, $3-8$ to group B, and $9-14$ to group C. Group A can steal from workers $3-14$, group B from workers $9-14$, and group C cannot steal. There is no average CPU waste with this allocation because groups A and B are slightly under-provisioned and can steal from C. Figure 4.7 presents our findings. DARC strongly favors shorter transactions from groups A and B. Compared to Shenango's c-FCFS, DARC provides up to **9.2x, 7x and 3.6x better tail latency** to `Payment`, `OrderStatus` and `NewOrder` transactions, respectively. These transactions represent 92% of the workload, resulting in up to **4.6x slowdown reduction** at the cost of 5% throughput from the longer `StockLevel` transactions. Because DARC excludes the longer `Delivery` and `StockLevel` transactions from 8 out of 14 workers, those transactions suffer higher tail latency compared to Shenango's c-FCFS. Interestingly, however, due to DARC's priority-based scheduling, `Delivery` transactions experience tail latency competitive with c-FCFS at high load. In addition, though benefiting `Payment` and `OrderStatus` requests, Shinjuku's offers performance similar to c-FCFS for the moderately slow `NewOrder` requests, because it preempts them halfway to protect the shorter requests. Likewise, `Delivery` and `StockLevel` requests suffer from repetitive preemption. In contrast, DARC is able to maintain good tail latency for

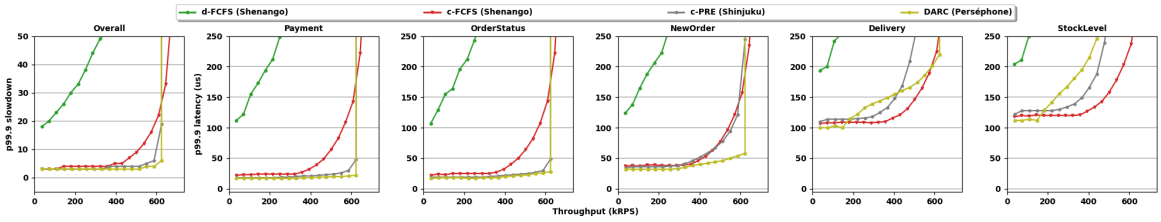


Fig. 4.7. *TPC-C* with Shenango, Shinjuku and Perséphone. The first column is the p99.9 slowdown across all transactions. Each subsequent column is the p99.9 latency for a given transaction. Transactions are presented in ascending average service time. Note the different Y axis for slowdown and latency. At 85% load, Perséphone offers 9.2x, 7x, and 3.6x improved p99.9 latency to `Payment` (b), `OrderStatus` (c) and `NewOrder` (d) transactions, compared to Shenango's c-FCFS, reducing overall slowdown by up to 4.6x (a). For a slowdown target of 10x, Perséphone can sustain 1.2x and 1.05x more throughput than Shenango and Shinjuku, respectively.

`NewOrder` requests, offers a better trade-off for `Delivery` and `Stocklevel` at high load (not show in the graph for the latter), and **reduces slowdown up to 3.1x** compared to Shinjuku.

Given a 10x overall slowdown target, **Perséphone can sustain 1.2x and 1.05x higher throughput** than Shenango and Shinjuku, respectively.

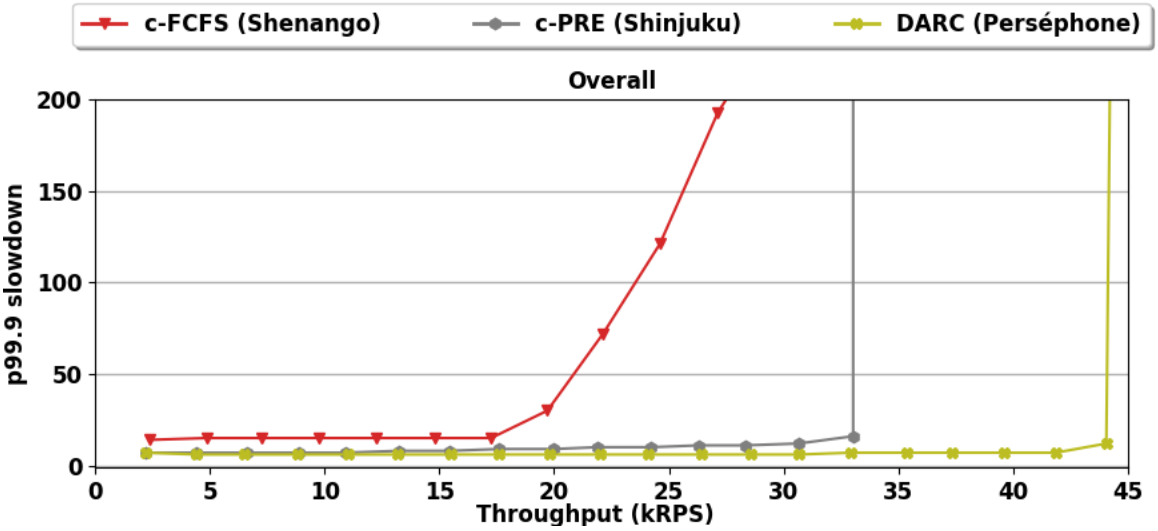


Fig. 4.8. RocksDB slowdown with 50% GETs (1.5μs), 50% SCANs (635μs). For a 20x slowdown target, DARC can sustain 2.3x and 1.3x higher throughput than Shenango and Shinjuku, respectively.

4.4.4.4. *RocksDB.* We use Perséphone to build a service running RocksDB and create a Shenango runtime running a similar RocksDB service. Shinjuku already implements a RocksDB service. The database is backed by a file pinned in memory. We use the same workload as Shinjuku's: 50% GET requests and 50% SCAN requests over 5000 keys. On our testbed, GETs execute in 1.5μs and SCANs in 635μs. Consistently, with previous experiments, we were able to sustain only about 75% of the theoretical peak load with Shinjuku using a 15μs preemption timer and its multi-queue policy. We omit d-FCFS because it offers poor performances. DARC reserves 1 core for GET requests, idling 0.96 core on average. Figure 4.8 presents slowdown for this experiment: **for a 20x slowdown QoS objective, DARC can sustain 2.3x and 1.3x higher throughput than Shenango and Shinjuku, respectively**.

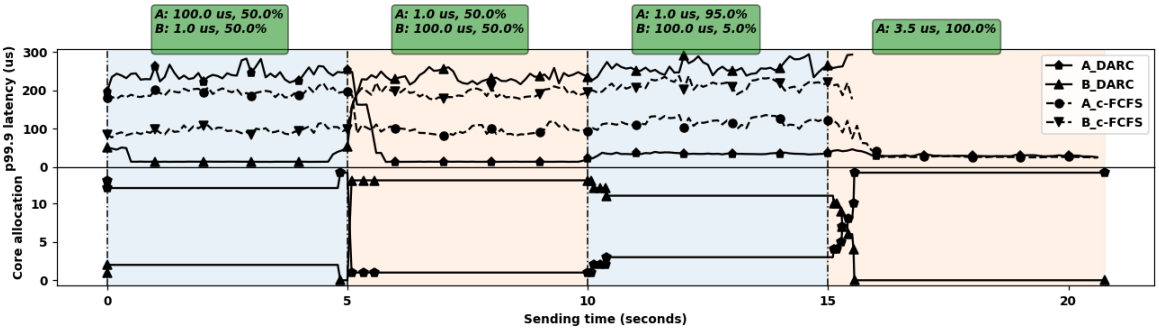


Fig. 4.9. p99.9 latency and guaranteed cores for two request types `A` and `B` during 4 phases, under c-FCFS and DARC. Top boxes describe phases (service times and ratios). During transitions, Perséphone's profiler picks on the new service time and ratio for each type and accordingly adjusts core allocation. Markers for the core allocation row indicate reservation update events.

4.4.5. Handling workload changes. In this section, we demonstrate Perséphone capacity to react to sudden changes in workload composition. For comparison with a baseline, we include c-FCFS performance. The experiment study three phase changes: (1) fast requests suddenly become slow and *vice-versa* (2) the ratio of each type change and (3) the workload becomes only fast requests. Across this experiment, we keep the server at 80% utilization. Each phase lasts for 5 seconds. Figure 4.9 presents the results. Green boxes describe phases. The first row is the 99.9th percentile latency and the second row the number of cores guaranteed to each type (not including stealable cores).

At first, `B` requests can run on all 14 cores — 1 dedicated core and 13 stealable cores — and `A` requests are allowed to run on 13 cores. Latency is slightly higher for `B` requests at the beginning of the experiment because the system starts in c-FCFS before proceeding to the first reservation. In the second phase, we inverse the service time of `A` and `B` to evaluate how DARC can handle miss-classification. During the transition, which takes about 500ms, "`B`-fast" requests observe increased latency — up to 50μs— as "`B`-slow" requests that are still in the system occupy the 13 stealable cores of `A` and "`A`-slow" requests are allowed to run on all cores before the reservation update. The graph shows latency increase before the transition because these `B` requests were already in the system and the X axis is the sending time.

During the second transition, we change the ratio of each type: `A` requests now make up 99.5% of the mix. As a result, their CPU demand increase and DARC reserves them 2 cores. For this new composition, 80% utilization on the server results in increased throughput, and latency becomes slightly higher for both types of requests as all queues grow larger.

Finally, we change the workload to be only made of `A` requests. Despite `A` requests being able to run on all 14 cores, pending `B` requests can be serviced on the spillway core.

4.5. Discussion

Networking model. In the current implementation, the net worker is a layer 2 forwarder and performs simple checks on Ethernet and IP headers. Application workers handle layers 4 and above and directly perform TX. This design intends to maximize our dispatcher's performance — the main bottleneck in Service Boosters — and make it competitive with existing systems. Shenango and Shinjuku separate roles in a similar way. There is no fundamental reason, though, for not having the net worker handle more of the network stack Using a stateful network stack would preclude offloading TX to the workers since shared state between the net worker and application workers would hinder performance. For TCP, this problem is partly addressed by TAS [**115**] and Snap [**85**].

Interrupts at μs scale. Though desirable in theory because it enables a better approximation to SRPT, interrupts at the microsecond scale come with two classes of challenges. The first is about performance. Even with the possibility to interrupt in 1μs, if a 1μs request enters the system to find all cores busy processing longer requests, it would experience a slowdown of at least 2x. In addition, preemption has to be aggressively frequent to minimize the impact of worst case scenarios where a short request enters the system at the same time a preemption check just occurred. As seen in our experiments, this aggressivity has a noticeable impact on performance. The second class of challenges is related to practicality. One has to carefully re-work existing applications to ensure preemption cannot happen during critical sections — memory management, interaction with thread local storage, etc. — or non re-entrant functions. This represents considerable efforts and spurred research in other designs trade-off such as semi-cooperative scheduling [**116**].

4.6. Related work

Kernel-bypass. Bypassing a general-purpose kernel to provide application-tailored routines has been revisited regularly over the past fifty years. Some notable examples include the RC 4000 multi-programming system [**117**], Hydra [**118**], Mach [**119**], Chorus [**120**], SPIN [**121**] and Exokernel [**122**].

More recently, faster networks and stagnating CPU speeds have led researchers to look more closely at user-level network stacks [**83, 115**] to provide high-performance storage systems [**123, 124, 125, 126**], access to disaggregated memory [**127**], user-level network services [**115**] such as eRPC [**84**], and fast I/O processing (*e.g.*, IX [**4**] and Chronos [**128**]). Similarly, user-level scheduling has been explored with ZygOS [**86**] and Shinjuku [**88**], which focus on improving tail latency by implementing centralized dispatch policies and user-level preemption, both of which outperform current decentralized offerings, as is well understood by theory [**129, 99, 130**]. DARC builds on this recent line of work with a more application-customized solution, motivated by recent insights when observing performance gain from sharing application-level information with the dataplane [**7, 131, 132**], and "common case service acceleration", which can improve tail latency for important requests [**133**].

Tab. 6. Summary of different scheduling policies as comparison points to DARC

Policy	App Aware	Non preemptive	Non Work Conserving	Prevent HOL	Ideal Workload	Comments
d-FCFS	✗	✓	✓	✗	Light-tailed	Easy to implement Load Imbalance
c-FCFS	✗	✓	✗	✗	Light-tailed	Ideal policy for the workload
Processor Sharing (Linux CFS, BVT [**134**], MLFQ [**135**])	✗	✗	✗	✓	Heavy-tailed without priorities	Hard to implement
(Deficit) (Weighted) Round Robin	✗	✓	✗	✗	Request flows with fairness requirements	No latency guarantees
Static Partitioning	✓	✓	✓	✗	Different request types with different SLOs	Inflexible with rapid workload changes
Fixed Priority	✓	✓	✗	✗	Request priority indendent of service time	Can lead to priority inversion
Earliest Deadline First	✓	✓	✗	✗	Request priority indendent of service time	Requires clock sync
Shortest Job First	✓	✓	✗	✗	Custom	Can starve long RPCs
SRPT	✓	✗	✗	✓	Heavy-tailed	Optimal for average latency Hard to implement
Cycle Stealing with Central Queue	✓	✓	✓	✗	Mix of short/long requests with the same priority	Can absorb short RPCs bursts
DARC	✓	✓	✓	✓	Heavy-tailed with high priority short requests	Favor short RPCs over longs

Scheduling Policies: Recent works on kernel-bypass and microsecond-scale applications have revived research interest in scheduling policies, specifically for tail-tolerance. We compare DARC with existing policies proposed for process or packet scheduling, and identified the best fit for each. Table 6 summarizes our findings. DARC shares ideas with Fixed Priority (FP) scheduling without suffering from head-of-line blocking and with Cycle Stealing with Central Queue (CSCQ [**100**]), but does not impose limits on stealing for shorter requests. It also shares ideas with Static Partitioning (SP) without being as work conservation avoidant, thus being able to absorb bursts. DARC targets applications with high service time dispersion similarly to Processor Sharing policies, implemented as the Completely Fair Scheduler [**136**], Borrowed Virtual Time [**113**], and Multi-Level Feedback Queue [**135**] in commodity operating systems and variants of these on datacenter operating systems [**88**]. Processor sharing policies, despite being application agnostic, are expensive or impossible to implement in many environment, *e.g.*, the public cloud. DARC is, to our best knowledge, the first application-aware and non-preemptive policy that classifies requests to improve RPC tail latency and can be implemented on a kernel-bypassed system serving microsecond-scale requests. We note that existing work has specifically made use of non work conservation to reduce resource contention in SMT architectures [**137, 138, 139**], though with a focus on instruction throughput rather than tail latency for datacenter workloads.

In-network end-host scheduling. R2P2 [**140**] and Metron [**141**] propose to integrate core scheduling in the network. Loom [**142**] proposes a novel NIC design and OS/NIC abstraction to express rich hierarchies of network scheduling and traffic shaping policies across tenants. Our work is orthogonal since request filters can be offloaded to the network. eRSS [**143**] scaling groups offer the possibility to schedule request groups, which works only on network headers and requires a specific programming model from the NIC. RSS++ [**144**] addresses RSS vulnerability to traffic imbalance but cannot handle variability in application-request processing times. Intel recently introduced Application Device Queues (ADQ) [**145**], a feature for applications to reserve NIC hardware queues. ADQ requires specific network interfaces (currently Intel's Ethernet 800 Series) and does not allow applications to further partition reserved queues by request type.

Network scheduling for tail latency. Prioritizing packets to improve tail latency has been extensively studied in the networking literature [**89, 146, 90, 147, 148, 149, 150**]. As analyzed in [**151**], this line of work uses priority queues in switches to approximate Shortest Remaining Processing Time (SRPT) scheduling and avoid head-of-line blocking caused by FIFO policies. Dedicating more CPU resources to short requests is similar to prioritizing packets belonging to short flows, but whereas network devices schedule at the granularity of packets — bounded by MTU sizes — and preempt long flows by not sending their packets, there is no affordable way to preempt a long request once dispatched at a CPU core within microseconds. DARC efficiently partitions CPU resources among requests by profiling their CPU demand and enabling work-conservation only for short requests, capping resources allocated to long requests and resulting in a similar trade off than Homa [**90**], pFabric [**89**], or HULL [**94**].

Other scheduling effort to improve tail latency. Haque et al. [**152**] exploit DVFS and heterogeneous CPUs to speed up long requests in latency sensitive workloads at the expense of short requests, with the goal of improving overall tail latency. Service Boosters is orthogonal to such optimization, since Service Units define a clear target to configure power and core settings for a given stage of the processing pipeline. Another line of work adapts the degree of parallelism of the pipeline — usually stages running longer requests — to improve overall tail latency [**6, 153**], but this comes at the cost of stages running shorter requests from which resources are taken away.

4.7. Takeaways

This chapter described Perséphone, a new kernel-bypass OS scheduler for Service Boosters that exploits DFG annotation and application-layer protocols to implement DARC, an application aware, not work conserving policy. DARC maintains good tail latency for shorter requests in heavy-tailed workloads that cannot afford the overheads of existing techniques such as work stealing and preemption. DARC profiles requests and dedicate cores to shorter requests, guaranteeing they will not be blocked behind long requests. Our prototype of the Perséphone booster maintains good tail latency for shorter requests and can handle higher loads with the same amount of cores than state-of-the-art kernel-bypass schedulers, overall better utilizing data center resources.

In the next chapter, we demonstrate how Service Boosters can be used to build a resource management booster, DeDoS, that scales Service Units under overload, consuming exactly the resources required to maintain good latency.

CHAPTER 5

DeDoS

This chapter presents DeDoS, a module for Service Boosters that exploits awareness of the request processing pipeline to harvest resources in the data center and mitigate tail latency arising from bottleneck stages.

Often, performance bottlenecks affect a specific stage in the request processing pipeline and requests causing them abuse a fixed resource. For example, the Billion Laughs exploit described in chapter 3 abuses CPU at the deserialization stage by controlling the number of nested entities in the serialized payload. Like the long and infrequent requests in an heavy-tailed workload, asymmetric attacks tend to be relatively low-volume and often do not appear different from the rest of the traffic. We have seen that, given the ability to classify requests like proposed in the previous chapter, one can devise scheduling solutions to protect the rest of the traffic against the long tail of the workload's distribution. Unfortunately, this is not always possible, and operators typically resort to deploying more resources to tame high tail latency. When resource utilization and/or tail latency increases, the service is automatically replicated as virtual machines (VMs) or lightweight containers, on multiple machines to scale "elastically" to the extra load. Replicating *all* of the VM's or container's resources, regardless of which are being consumed, can be enormously costly, making this approach unusable for most service providers. For example, if only a TCP state table is being exhausted (*e.g.*, due to a SYN flood), the replication of an entire VM mitigates the attack, but does so at an enormous overhead (since the TCP state table is a minuscule portion of the system's overall footprint).

This chapter details how, using the Service Boosters architecture, one can build a resource harvesting booster to selectively scale overloaded stages of the request processing pipeline. Ideally, each Service Unit in the pipeline handles some small, focused aspect of an application that may be vulnerable to resource exhaustion. Example components include code for performing TLS handshakes or HTTP requests parsing. The DeDoS booster comes as an adaptive controller making real-time decisions on placing Service Units within physical resources (*e.g.*, machines in a data center) and then adaptively clones, merges, or migrates them in order to meet Service Level Objectives (SLOs).

Service Boosters and the DeDoS booster offer two benefits for improving tail latency. First, fine-grained components make it easier for the operator to harvest all available resources on all machines, exactly as needed. For instance, the system can respond to a TLS renegotiation attack by temporarily enlisting other machines with only spare CPU cycles to help with TLS handshakes. Second, the replication approach is agnostic of the exact source of overload and can thus potentially mitigate unknown contributors of high tail latency. Once the DeDoS booster recognizes that

a component is overloaded or its throughput appears to drop, it can respond by replicating that particular component – without having seen the problematic requests before, and without knowing the specific vulnerability of the application. This allows to generally better utilize the datacenter resources.

The rest of this chapter presents the design of the DeDoS (§5.2) booster and strategies for dynamic adaptation (§5.3). The booster is evaluated through three case studies (§5.5): a web server developed from scratch using Service Boosters' DFG API (similar to the one we built for FineLame's evaluation) and two existing software systems: a user-level transport library written in C ported over to Service Boosters, and a routing software written using a declarative domain-specific language [**16**] that compiled into a Service Boosters dataflow. We evaluate Service Boosters and the DeDoS booster and show that using fined-grained components has low overheads compared to equivalent code executed outside of Service Boosters. Moreover, the DeDoS booster is able to defend against a wide range of asymmetric attacks, maintaining significantly higher throughput for a much longer amount of time in the presence of changing attacks, comparable to traditional replication strategies.

5.1. Motivating example

Consider a 2-tiered web service hosted in a data center, where an HTTP server queries a database server in response to users' requests. The attacker launches a TLS renegotiation attack [**154**] that consumes CPU cycles on the HTTP server. Hence, legitimate requests are being served very slowly, or not at all. In this typical asymmetric attack, the attacker is unable to overwhelm the defender's network bandwidth but succeeds by exhausting other resources (here, CPU cycles).

Our goal is to *automatically handle such performance degradation, even if it has a new vector*, to maintain quality of service (QoS) to the legitimate clients, and to enlist the exact amount and type of resources required, no more.

5.1.1. Strawman solutions. One possible defense against traditional performance degradation such as DoS attacks or increase in load is to performance overload control, usually by *filtering* or *blocking* incoming traffic – either based on source addresses, specific traffic content or other traffic characteristics. However, this relies heavily on request classification, thus is susceptible to false positives and negatives. Moreover, it is difficult to differentiate between legitimate spikes in traffic and actual attacks. A service running on Service Boosters could also rely on scheduling techniques such as Perséphone, but in this chapter we compare with software not running on Service Boosters yet.

Another approach is to increase resource capacity via *replication*. For instance, to handle a TLS renegotiation attack, an operator can launch more web server VMs to sustain more connections. This technique does not depend on accurate detection but it can be inefficient. In the TLS renegotiation example, even though the attack is limited to the key generation logic (and thus stressing CPU usage on the host), naïve replication replicates the *entire* web server, unnecessarily wasting non-affected resources such as memory.

5.1.2. With Service Boosters. Overall, data centers machines are under utilized [**155**], but current software architectures cannot effectively use them. Here, the database servers' CPUs will be mostly idle while the web servers' CPUs are overwhelmed. If the former's CPUs were able to alleviate the load on the latter's by contributing their computational power, the capacity at the bottleneck (TLS handshake) would increase.

Service Boosters allow programmers to design application stacks as smaller functional pieces that can be replicated and migrated independently, effectively declaring the request processing pipeline to the execution environment. This additional flexibility enables a service under overload to use *all* of the available datacenter resources for its defense by temporarily enlisting other machines running different services, resulting in a substantial increase in the service's capacity and achieving better QoS.

For example, in TLS renegotiation, instead of replicating the entire web server, we can instead replicate only the key generation logic. If the database servers have spare CPU cycles, they will be able to accommodate execution of this logic and alleviate the CPU bottleneck caused by the attack. In contrast, naïve replication would not work when the database servers lack the entire set of resource required to run additional HTTP servers. In other forms of asymmetric attacks that exhaust other types of resources (*e.g.*, memory), one can adopt the same approach, in this case, replicating the memory intensive component into other machines that have spare memory.

5.2. The DeDoS booster

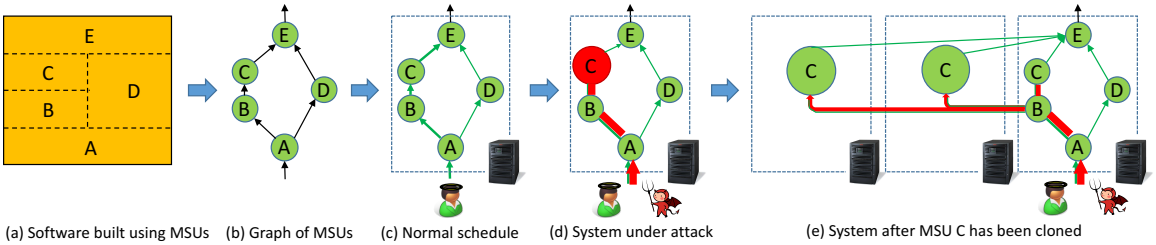


Fig. 5.1. Example use case of the DeDoS booster. The software is built over Service Boosters using Service Units (a), represented as a dataflow graph (b). Service Units are then scheduled on the available machines (c). When a stage of the processing pipeline becomes overloaded (d), the DeDoS booster disperses the attack by generating additional instances on other machines (e).

As described in chapter 2, Service Boosters applications consist of a set of Service Units (Figure 5.1). Each Service Unit is responsible for some particular functionality. For instance, a web server might contain an HTTP Service Unit, a TLS Service Unit, a page cache Service Unit, etc. (Figure 5.1a).

Related Service Units communicate with each other. For instance, HTTPS requests may enter the system at a network Service Unit, be decrypted by the TLS Service Unit, and parsed by the HTTP Service Unit. Collectively, Service Units form a DataFlow Graph (DFG) containing a vertex for each Service Unit and an edge for each communication channel (Figure 5.1b).

The DeDoS booster relies on an external *controller* with visibility over all Service Boosters runtimes on the cluster (Figure 5.1c). Local DeDoS boosters continuously collects runtime statistics about the performance of each Service Unit. If they detect that some Service Unit instances are overloaded (*e.g.*, due to an unknown attack; Figure 5.1d), they can contact the controller to request the creation of additional instances of these Service Unit. The controller is responsible for placing these new instances on machines where abused resources are still available (Figure 5.1e). Thus, the data center can enlist all possible resources to tame high tail latency, not merely the ones that happen to be "in the right place."

5.2.1. Minimal Service Units. When designing an application over Service Boosters, the question of defining the granularity and boundaries of Service Units arises. While smaller Service Units can result in a more precise response during overload — since it allows the DeDoS booster to replicate only the bottleneck functions — *too* small and numerous Service Units can result in unacceptable overheads because of the delay introduced on the execution path. This trade off has already been unveiled in the past with, for example, the fall of mainframe computers and the rise of microservices [**156, 18**].

The general approach Service Boosters advocates is based on the microservices design [**157**]: Service Units split points are appropriate when there are loose couplings between components, functional domains are clearly encapsulated, and individual components are provably stables. Appropriately split Service Units are then dubbed Minimal Service Units (MSUs).

For known performance bottlenecks, it is also advantageous to purposefully demarcate Service Units to most optimally respond to the potential overload. For example, to protect against a SYN flood, the portion of the TCP stack that handles TCP connection state could be isolated into its own MSU.

However, a key benefit of the DeDoS booster is that it does not require *apriori* knowledge of the performance problems it defends against. Hence, in many instances, programs may not be perfectly spliced. In such instances, MSUs may contain features unrelated to the performance bottleneck, resulting in non-optimal resource allocation. However, such duplication will often be preferable and very likely be far better than naïve replication. In general, splitting software components following a microservices-like programming paradigm will yield significant protection against tail latency while incurring limited overheads, provided we correctly schedule each component. Empirical measurements of these overheads in a number of application, constructed using this design pattern, support this assertion (§5.5).

5.2.2. Inter-runtime communication. Communicating Service Unit instances can reside on different machines. A Service Boosters runtime makes this transparent to Service Units by injecting a bit of "glue code" that converts calls into a local function call (if the callee is on the same machine) or a network packet (if the callee is remote). Figure 5.2 pictures how multiple Service Boosters runtimes implemented on Linux (*Cf.* §2.6) communicate.

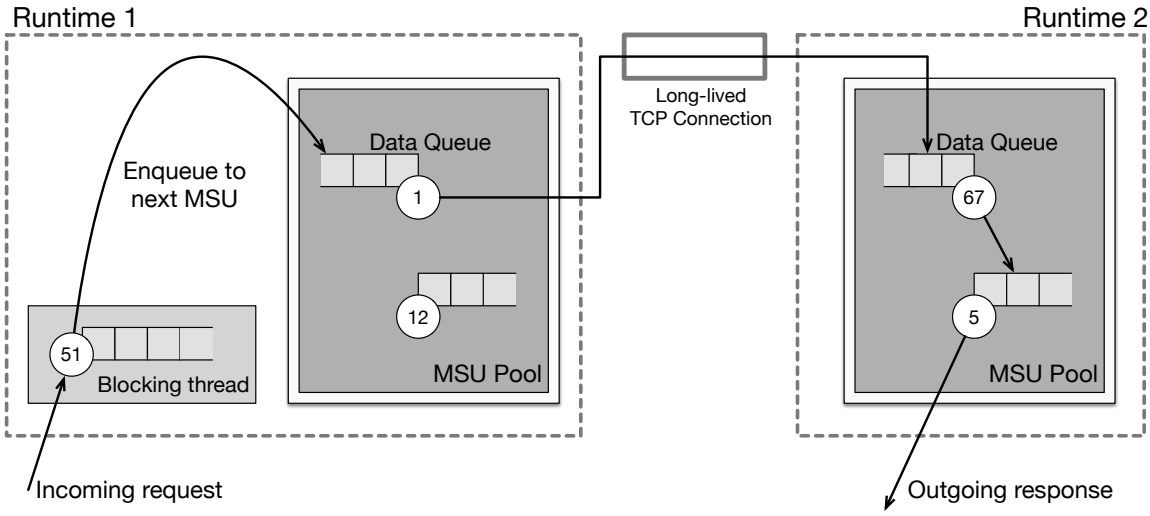


Fig. 5.2. Communication between two Service Boosters runtimes. The systems relies on long lived connections to exchange control and application messages.

5.3. Resource allocation

To ensure that the applications meet their SLAs, DeDoS needs a way to make and enforce resource allocations to MSU instances at runtime. DeDoS performs resource allocation at two layers: each machine schedules MSU instances locally based on their resource needs, whereas the external controller is responsible for decisions requiring a global view, such as scaling-in MSU instances. To enable runtime adaption, DeDoS runs a local agent that continuously monitors their local MSUs, detects local overloads and periodically submits statistics and overload alerts to the controller. In addition, they are responsible for handling the controller's commands.

In this section, we describe the specific resource harvesting design of DeDoS.

5.3.0.1. *Initial assignment.* Initially, deployments use hints from the configuration files to determine how many replicas of each Service Unit should deployed. Over time, using the DeDoS Booster, the system scales to meet the load demand and respect SLAs.

5.3.0.2. *Cloning and merging.* Local agents detect overloads through queuing delay at MSUs. Abnormal queuing delays can happen when MSUs are processing outlier requests such as an ADoS or legitimate but infrequent requests. To perform a scaling decision, the local agent checks whether local resource usage — CPU, DRAM bandwitdh, DRAM amount, and file descriptors pool, are bellow a configurable threshold. If yes, it simply replicates the target MSU locally. Otherwise, it issues a request to the global controller for scaling the target MSU. The controller uses its global view to identify a suitable runtime. In addition, the controller periodically checks if, overall, runtimes are utilizing more than a configurable percentage of a resource (e.g., memory) and some MSU type accounts for a configurable percentage of a runtime's utilization of that resource, in which case the controller will begin to clone MSUs of that type. This latter policy targets the system's bottlenecks by increasing parallelization.

Once the decision to clone is made, the controller picks a satisfying machine for the new instance, favoring locality with the clone's neighbors in the dataflow graph. A local machine is best to minimize network communication. Once a machine has been elected, the controller contacts the machine's local agent to spawn the instance. The controller also updates all the relevant routing tables to enforce the load balancing policy in place for this MSU type. An attempt to clone will fail if it is not possible to place the clone on any available runtime or if the same type of MSU has been recently cloned.

The controller scales-in (removes) cloned MSU instances when they are no longer needed. Two conditions must be met for an MSU to be removed. First, the last runtime where a clone has been placed must report a normal queuing delay in the last monitoring interval; second, the MSU type must not be significantly contributing to more than a configurable percentage of a resource consumption on *any* runtime. An attempt to remove will fail if an attempt to clone an MSU of that type was made in the recent past (to protect against system oscillations), or if some configurable amount of time has passed since the last removal of that type.

We set the following default parameters for DeDoS management policies: the controller clones an MSU type if all runtimes are utilizing more than 40% of the memory or file descriptor (FDs) pool, and the type accounts for at least 50% of its runtime utilization of that resource; for removal, the MSU type must not be contributing more than 40% of the memory or FDs pool on any runtime. Removal fails if an attempt to clone an MSU of that type was made in the last 20 seconds, or if less than 5 seconds elapsed since the last removal of that type or its dependencies. Those parameters are based on our domain expertise of how our testbed performs.

5.4. Case studies

This section demonstrates the feasibility and applicability DeDoS by considering three case studies: a web server; an existing userspace TCP stack; and an application written using a declarative domain specific language [**16**] that has been translated into a Service Units DFG.

Web server. Our first case study showcases a simple web server constructed as five MSUs. Figure 5.3 pictures this appolication. The *socket MSU* accepts incoming requests and steers them toward the *Read MSU*, which performs the TLS handshake, deciphers data, and relays plaintext to the *HTTP MSU*. The HTTP MSU implements NodeJS's HTTP Parser [**158**]. Once the request is parsed, the HTTP MSU issues a call to the database tier to retrieve some object file, then enqueues the request to a *Regex Parsing MSU*, which uses the PCRE engine to parse it. The final HTTP response is sent to the *Write MSU*, which wraps it in a layer of TLS and sends it back to the client. The Service Unit uses OpenSSL version 1.0.1f for TLS support in both Read and Write MSUs.

Importantly, with the exception of the socket MSU that polls on a set of sockets, all of the web server's MSUs are event-driven and non-blocking. To avoid having to migrate socket states between machines, the controller enforces that the socket, Read, and Write MSUs reside within the same Service Boosters instance for a given

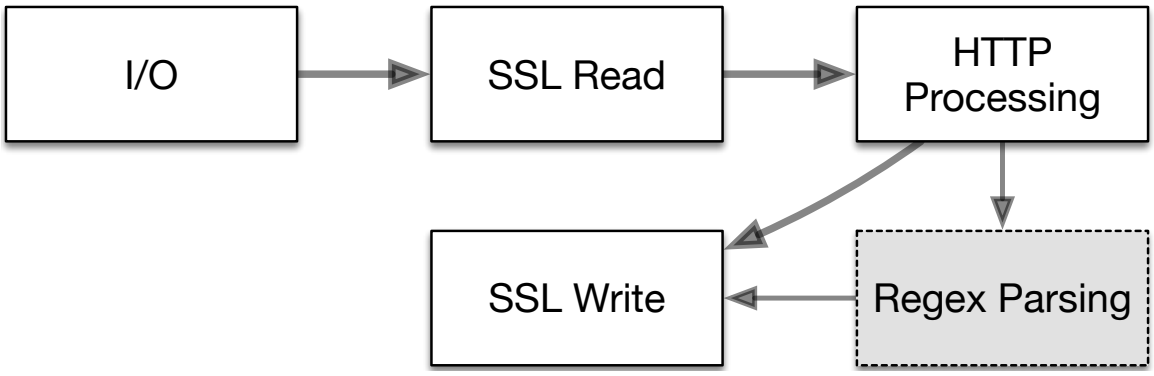


Fig. 5.3. A simple web server ported over Service Units

client connection. HAProxy [**159**] load balances client connections to socket MSUs on DeDoS instances.

The web server leverages DeDoS' fine-grained modular architecture to handle overload. §5.5 demonstrates the system's resilience against three ADoS attacks: TLS renegotiation attacks [**154**], ReDOS attacks [**19**], and HTTP SlowLoris attacks [**21**].

Userspace network stack. The second case study consists of an existing software project ported to run on DeDoS with minimal effort. PicoTCP [**160**] is an open-source userspace TCP stack written in approximately 33,000 lines of C code (as reported by SLOCCOUNT). PicoTCP is well-structured and written in a modular fashion, making it easy to manually determine cut-points (i.e., to form MSUs). Figure 5.4 pictures this application.

Standalone handshaking MSUs are split from PicoTCP. When a SYN flood attack occurs, the TCP Handshake MSU is replicated into multiple copies on the same or different machines. Load-balancing across these clones is achieved by using a consistent hashing scheme within the PicoTCP MSU: based on a hash over the incoming packet's four-tuple (source and destination addresses and ports), Service Boosters performs load-balancing by distributing the handshaking requests (in the form of SYN, SYN/ACK, ACK packets) to the various TCP Handshake MSU instances. Packets belonging to the same three-way handshake (*e.g.*, the client's SYN and ACK) are routed towards the same TCP Handshake MSU, obviating the need to transfer state.

Given the modular nature of the PicoTCP code, separating the TCP stack into separate MSUs was fairly straightforward. The bulk of the efforts lay in wrapping PicoTCP's "main loop" within an MSU to allow Service Boosters's scheduler to execute the MSU according to its scheduling policy. The PicoTCP MSU was augmented to re-inject SYN-ACKs generated by Handshake MSUs into the PicoTCP stack for them to be sent back to the client, and code to restore TCP state received from Handshake MSUs (for successful connections) into PicoTCP's internal TCP state data structure.

In summary only minor modifications were required to transform the monolithic PicoTCP application into a DeDoS-enabled version in which handshake components could be replicated on demand, both within the local machine and on remote DeDoS instances. Overall, less than 0.1% of PicoTCP's original code base changed.

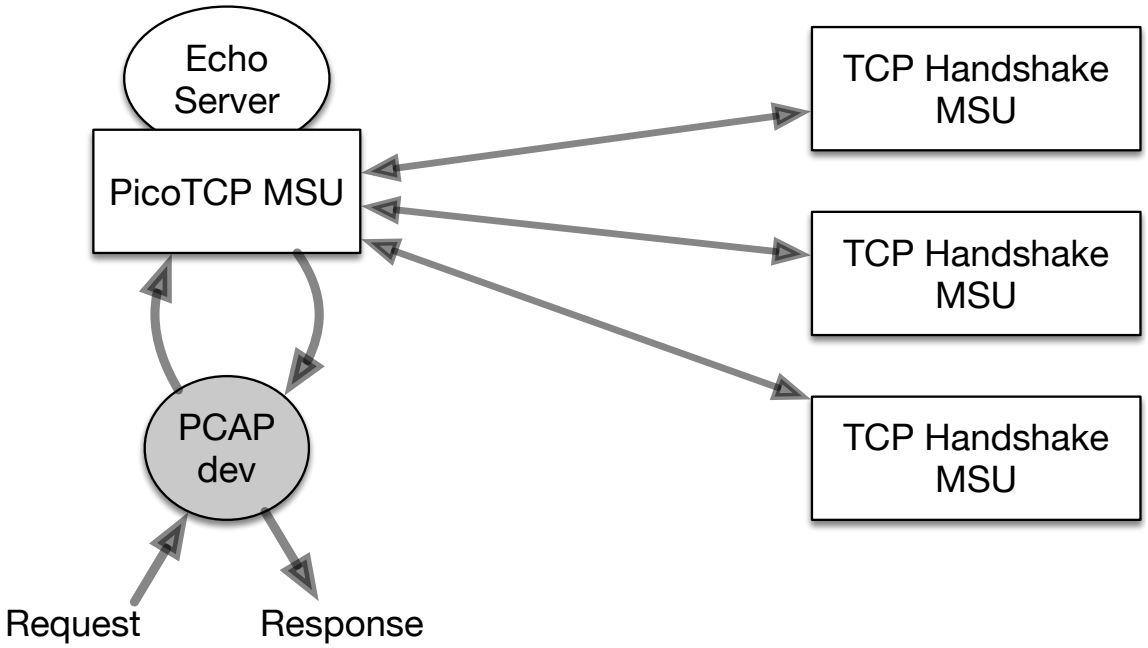


Fig. 5.4. PicoTCP ported over DeDoS's Service Units

Declarative packet processing. The third use case is an application written in a domain specific language that does routing (packet forwarding), written entirely as a declarative networking [**16**] program. Declarative networking programs are written in a variant of Datalog called *Network Datalog* (or NDlog). An NDlog program consists of a set of rules, where each rule is of the form `h :- b1,b2,..., bn`, indicating that a *head tuple* is generated so long as all *body tuples* b1, b2, ..., bn are available. For example, the rule `packet(@Y,A,Data) :- packet(@X,A,Data), Neighbor(@X,A,Y)` results in all packets arriving at X being forwarded to neighbor Y based on some attribute A (e.g., the packet's header data). Declarative networking has been adopted for network forensics, data center programming, and overlay routing.

Since these programs have their roots in the database relational model, they can be compiled into an Service Unit dataflow of relational operators. For example, the body tuples are executed as a series of pair-wise database *join* operations, additional filters in the form of *selection* operators, and the head tuple is generated as a *projection* operator. The generated tuple may be sent to the same or different machine using DeDoS. We find such automatic translation to be a promising method of adopting existing applications to DeDoS.

5.5. Evaluation

This section aims to answer two high-level questions through several sets of experiments: (1) do applications split into Minimal Service Units run with reasonable overheads in normal operation, and (2) how well can the DeDoS booster mitigate

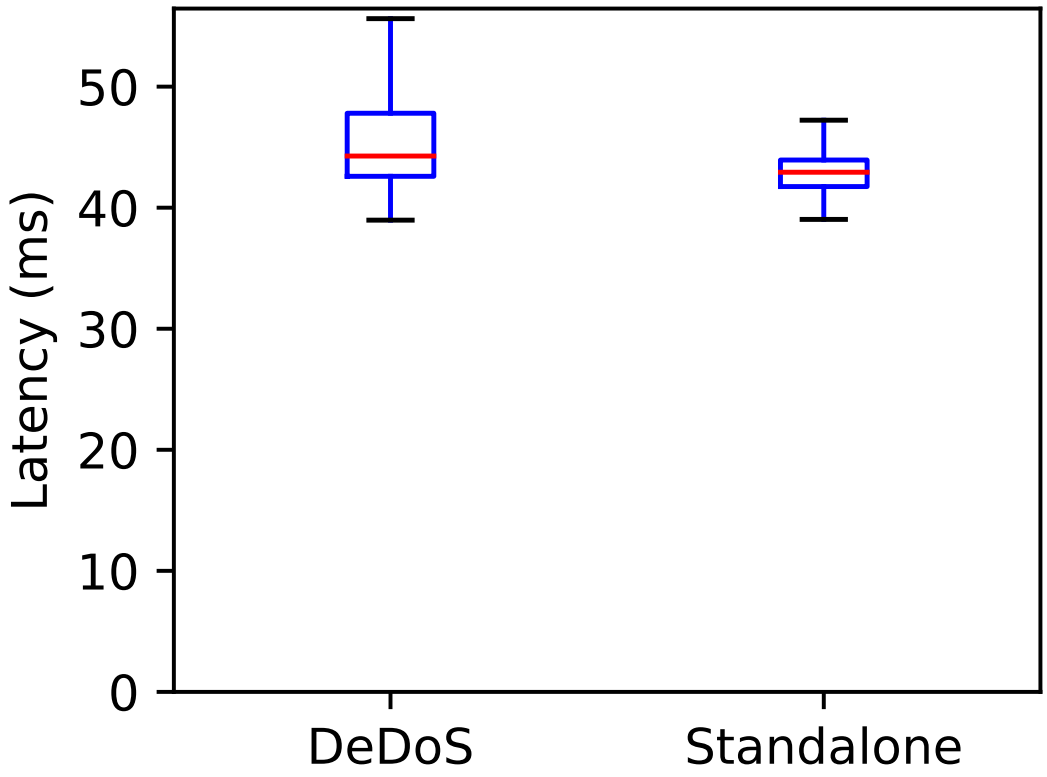


Fig. 5.5. Average latency for DeDoS and standalone

high tail latency arising from adversarial workloads such as ADoS attacks, as compared to whole-system replication? The experimental testbed consists of a cluster of 8 machines connected via a 10 Gbps switch in a star topology. Each machine has 8 1.80 GHz cores (with hyperthreading and DVFS disabled), 64 GB of memory, and runs Linux kernel 4.4.0-62. A video demonstration of DeDoS booster is available online [**161**].

5.5.1. Overheads. This section presents the overheads introduced by the DeDoS runtime during normal operation through the applications described in §5.4 within and outside of DeDoS on a single server machine.

Web server: Figure 5.5 compares the performance of a DeDoS web server to a standalone web server with the same implementation but compiled as a monolithic application outside of DeDoS. The workload consists of HTTPS requests generated by Tsung [**48**] at an exponentially distributed rate for a period of five minutes, with a mean of 2500 requests per second (r/s). Latency is averaged over intervals of one second. Figure 5.5 presents the results. The standalone webserver has mean latency of 43ms, and 1.8ms standard deviation. DeDoS' webserver has a mean latency of 48.5ms and 12.3 standard deviation. This accounts for a mean 10.5% overhead introduced by DeDoS, which is caused primarily by the enqueuing and dequeuing of messages across

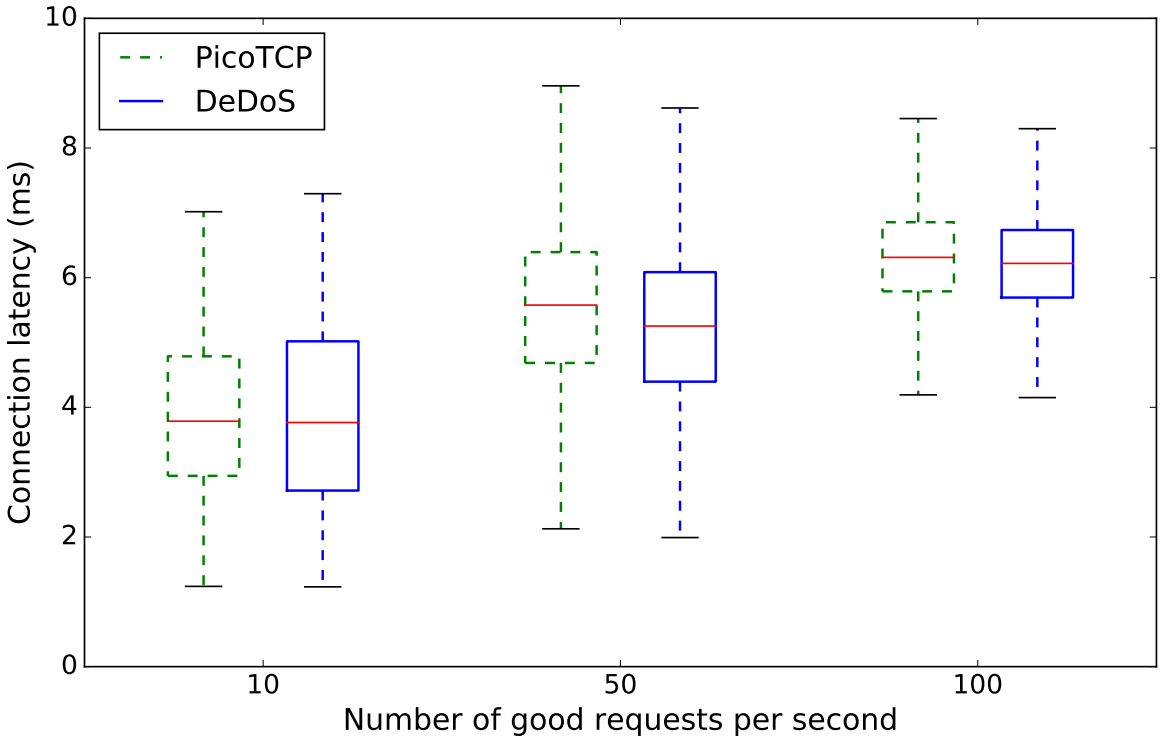


Fig. 5.6. Connection latency for standalone PicoTCP and DeDoS-enabled PicoTCP ("DeDoS").

Service Units. Because in their current implementation, worker threads on DeDoS do not perform work stealing from remote queues, there are some rare instances of MSUs sitting idles while others are building a queue, hence the higher number of outliers with DeDoS.

PicoTCP: The DeDoS-enabled version of uses a single worker thread on a single runtime and has two MSUs: a handshake MSU and the remainder of the PicoTCP stack as a separate MSU. The application is a simple echo server mirroring back incoming requests.

The first metric of interest is *connection latency*, the time required to complete a TCP handshake as measured by the client, measured over a 15-minute period. During this time, the client continuously creates new TCP connections at a steady rate, sends (and receives) 32 bytes of data, and disconnects. Figure 5.6 shows the distribution of connection latency for the vanilla and DeDoS-enabled versions of PicoTCP under different client request rates. The DeDoS-enabled version incurs a modest 5.5% increase in connection latency.

The second metric of interest is the throughput that both TCP stacks can achieve. A number of different clients simultaneously access the echo server. Each client repeatedly sends and receives 1024 bytes, with a 10ms pause between transmissions. There is no significant difference between the throughput that both stacks can achieve. PicoTCP and DeDoS both reached their maximum bandwidth of 57.66 Mb/s and 57.71 Mb/s respectively around 100 simultaneous connections and the throughput remained similar for more than 100 simultaneous connections. (The absolute numbers

Clients	DeDoS Throughput	Standalone Throughput
5	491.6 pkts/s	491.9 pkts/s
10	980.0 pkts/s	981.2 pkts/s

Tab. 1. Average throughput of declarative packet processing.

are low because PicoTCP is a user-space network stack designed for portability instead of performance. It still eventually goes through the kernel. The goal here is to measure the overhead of the DeDoS runtime.)

These results suggest that running applications within the DeDoS runtime does not significantly change the throughput or the latency it can achieve.

Declarative packet processing. Finally, we run an experiment with the declarative packet processing application. We use either five or ten clients that each send 100-byte packets at a rate of 100 packets per seconds; as before, we compare a DeDoS-enabled application to a standalone version that runs the same code, but outside DeDoS. Table 1 shows our results, which confirm our findings from the previous experiments: the throughput with DeDoS is only marginally lower (by less than 1%) than the throughput of the standalone system.

5.5.2. Attack mitigation. The system will receive a set of ADoS — ReDOS, TLS renegotiation, SlowLoris, SYN flood — and a volumetric flood attack to evaluate the efficacy of DeDoS. The first attacks three are run against the webserver and the latter two are on PicoTCP and an NDlog program respectively.

5.5.3. Attacks against webserver. The webserver is deployed on three machines while three other machines run instances of an in-memory database. All web requests access the database, and HAProxy distributes HTTP and database requests. The remaining two machines are used to generate legitimate ("good") and attack traffic respectively. To demonstrate the DeDoS Booster's ability to defend against changing attacks and reclaim resources, under dynamic traffic patterns, the experiment is a two hours long run during which attacks' duration are *randomly* distributed. Good traffic is generated by Tsung and exponentially distributed, simulating diurnal variations by setting Tsung's distribution mean at 1500 r/s, and increasing by slices of 500 r/s up to 3000 r/s, at which point it gradually decreases back down to 1500 r/s. Tsung's requests time out after 1s if they cannot connect. When no attack occurs, clients experience an average latency of 50ms. Attack traffic is generated using an in-house C client which generates malicious ReDOS and TLS renegotiation requests. SlowLoris attacks are generated using an existing Python tool [**49**].

Figure 5.7 shows the main findings for different attacks on the HTTP servers for a single run of the experiment than ran continuously for 2 hours. The top figure shows response times for successful connections averaged every second, while the middle figure presents the connection success rate during this experiment. The bottom figure shows the number of MSU instances of a given type deployed on the system over time. Attacks occur during the period colored in red. We compare Service Boosters and the DeDoS Booster to two other techniques: (1) an approach that does not replicate at all under attack ("standalone"), and (2) an approach that naïvely replicates an

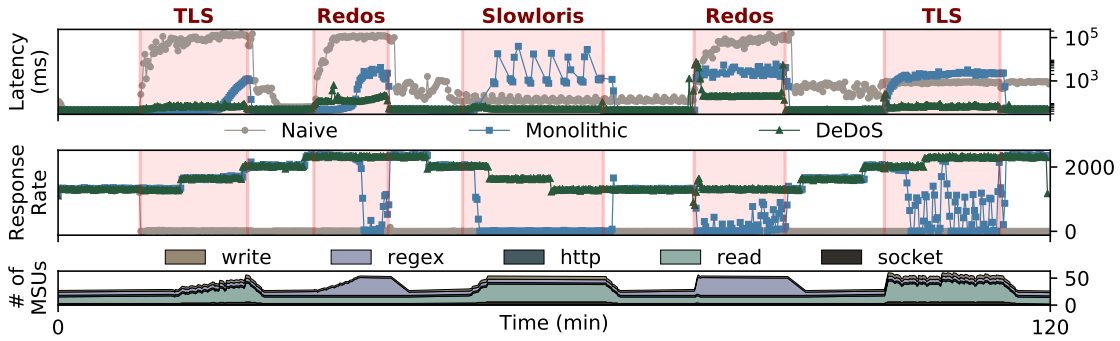


Fig. 5.7. Requests latency and success rate during TLS renegotiation, ReDOS, and SlowLoris attacks on a web server running over Service Boosters, standalone with no defense, and using a naïve replication defense strategy.

entire webserver to one of the database servers when under attack ("naïve"). Initially, the Service Boosters' webserver has 4 Read and 2 Regex MSUs on each of the three starting machines.

During the entire course of the experiment, the DeDoS Booster is auto-piloting without inputs from human users. It can accurately detect and react to the injected attacks based on the resource allocation polices described in §5.3 without *apriori* knowledge of the attacks. DeDoS can consistently *and automatically* decide on an effective mitigation strategy against different types of attacks. Figure 5.7 shows that the Booster consistently outperforms standalone and naïve approaches, and sustains low latency and high response rate while standalone and naïve can only provide limited or sporadic services.

TLS renegotiation attack: This attack consumes the victim's CPU by having malicious connections repetitively triggering TLS handshakes. Here, a single handshake requires about 2.1ms computation time (we use a 2048-bits RSA key), and every malicious request triggers 100 renegotiations before closing. During the first TLS renegotiation attack in Figure 5.7, the attacker increases the strength of the attack from 1 to 100 r/s over a period of 13 mins. At the start of the attack, standalone performs better than DeDoS until CPUs get overwhelmed by attack requests (around 75 r/s); it increases the average latency for good requests to the order of seconds. Naïve replication performs even worse and causes connection success rate to drop to almost 0% once the entire webserver has been replicated to the database machines. This is due to paging that occurs on the database server as a result of the additional memory footprint imposed by the cloned webserver. Even successful connections experience latency on the order of tens of seconds.

During the attack, the DeDoS controller observes abnormal levels of pending requests in the system, and gradually increases the number of Read MSUs from 12 to 39 (1 more on each original machine, plus 8 per database machine). Unlike naïve replication, Read MSUs have a low memory footprint and do not cause paging on the database machines. This results in average latency of 70ms for good requests during attack.

Once the attack stops, the DeDoS controller observes that the conditions explained in Section 5.3.0.2 are met, and reclaims resources by tearing down the cloned MSUs.

The second TLS attack in Figure 5.7 shows the performance of DeDoS under a steady state attack with 100 r/s instead of a gradual increase in attack strength. Under this relatively high attack strength, CPU resources for standalone are quickly overwhelmed, and connection success rate for good clients falls to 50% with 3s latency on average. DeDoS applies the same policies for resource management, maintaining 39 Read MSUs, and while its performance drops momentarily, it manages to serve good clients with an average latency of 70ms.

ReDOS attack: In this attack, each malicious request issues a complex regular expression operation that exploits a PCRE vulnerability [**162**], requiring approximately 100ms of computation time. The first ReDOS attack increases attack strength from 1 r/s to 200 r/s and lasts 9 mins. Similar to TLS renegotiation, standalone initially does better than DeDoS until CPUs are overwhelmed by malicious requests. On the other hand, DeDoS gradually increases the number of Regex MSUs (up to 27 new instances) and maintains 100% success rate, but with an increased average latency of 150ms. There are much less variations in the number of Regex MSU than Read MSU because of the nature of the workload: TLS handshakes are much shorter, and performed over non-blocking I/O, while the regex parsing events cannot be preempted by DeDoS. The second ReDOS attack is performed at a steady rate of 200 r/s over 11mins. Standalone clients almost instantly experience average latency on the order of seconds after the attack is launched. DeDoS, while initially overwhelmed as well, quickly recovers by spawning 27 new Regex MSUs, managing to keep the latency on the order of tens of milliseconds.

HTTP SlowLoris: This attack targets the connection pool of the webserver by exhausting the file descriptors (FDs) available for the process. The attack works by opening a connection to the server, and slowly sending HTTP headers one after the other, at such a pace that the server keeps each connection open for a significantly longer time than usual. The attacker only has to send a number of concurrent requests equivalent to the maximum number of FDs available to the server process to deny service to legitimate clients. The kernels are configured to allow each process to open 2^{13} concurrent FDs (from an initial value of 2^{10}). The attack tool opens up to about 41K concurrent connections to the webserver during 17mins. Standalone is able to withstand the attack until the FDs limit is reached (in about 220 seconds). Then the connection success rate quickly drops to about 3 r/s, and the good requests experience a sharp increase in latency, since the webserver threads are kept busy with processing HTTP headers that are continuously sent from malicious clients. DeDoS, on the other hand, is able to spawn 22 new Read MSUs on each of the database server, increasing its global file descriptors pool, and allowing it to sustain 100% successful connection rate. Due to paging, naïve is unable to respond to a majority of the connections.

5.5.4. Additional attacks. In addition to the attacks discussed on the web server, this section discusses two more attacks and their mitigation using the DeDoS Booster.

SYN flood attack: The SYN flood experiment consists of a number of "good" (i.e., non-attack) clients accessing an echo server built on top of PicoTCP. Each

good client attempts 10 requests per second, where each request establishes a TCP connection, sends and receives 32 bytes of data, and then closes the connection. A TCP connection is considered successful only if the handshake completes within 60 seconds. The SYN flood is launched after one minute of normal traffic, runs for three minutes, and then stops. SYN floods are generated with HPING3, with varying intensity. The experiment continues for an additional two minutes (during which no attack occurs) to observe the recovery period.

As a baseline, we first run the attack against standalone PicoTCP. PicoTCP executes on a single thread within a single machine with the size of the connection buffer set to 1MB, corresponding to 26,214 pending connections. We note that this limit is significantly larger than the 1024 pending connection limit offered by default on Linux. Table 2 shows the percentage of successful handshakes completed during the full attack window.

Attack rate (SYNs per second)	**PicoTCP success percentage**
1000	45.68%
2000	9.04%
3000	7.1%

Tab. 2. Percentage of successful TCP connections for PicoTCP during the attack window.

We observe that a single instance of PicoTCP cannot keep up with increasing attack strengths. For example, at 2000 SYNs per second, less than 10% of clients could complete a three-way handshake.

In contrast, DeDoS can mitigate a SYN flood by cloning MSUs, potentially on other hosts. The DeDoS-enabled version of PicoTCP consists of separate MSUs for performing the handshake and for transferring data. Each instance of a Handshake MSU is provided with a 1MB connection buffer. The PicoTCP MSU (the non-handshake related portion of TCP) runs on a single machine that also hosts the echo server. The booster uses three other physical machines to spawn a maximum of three additional Handshake MSUs per machine. Varying the number of Handshake MSUs (by manually overriding the controller's actions) helps measure system performance during the SYN flood.

The success rate of TCP handshakes during the interval between the first and last instances in which a TCP connection failed, as observed by a good client, reflects the steady state of the attack and avoids the "ramp up" period in which the attack has not yet become effective.

The results show that the DeDoS Booster is able to provide superior service throughout the attack. Figure 5.8 shows the connection latency of good clients during a 2000 SYN/second attack. The second y-axis shows the average percentage of successful TCP handshakes ("success percentage") computed over a two-second interval. PicoTCP (top graph) fails to service good requests as soon as the attack starts—the percentage of successful TCP handshakes almost immediately drops to below 10%. The few connections that are successful experience very high latency (first y-axis).

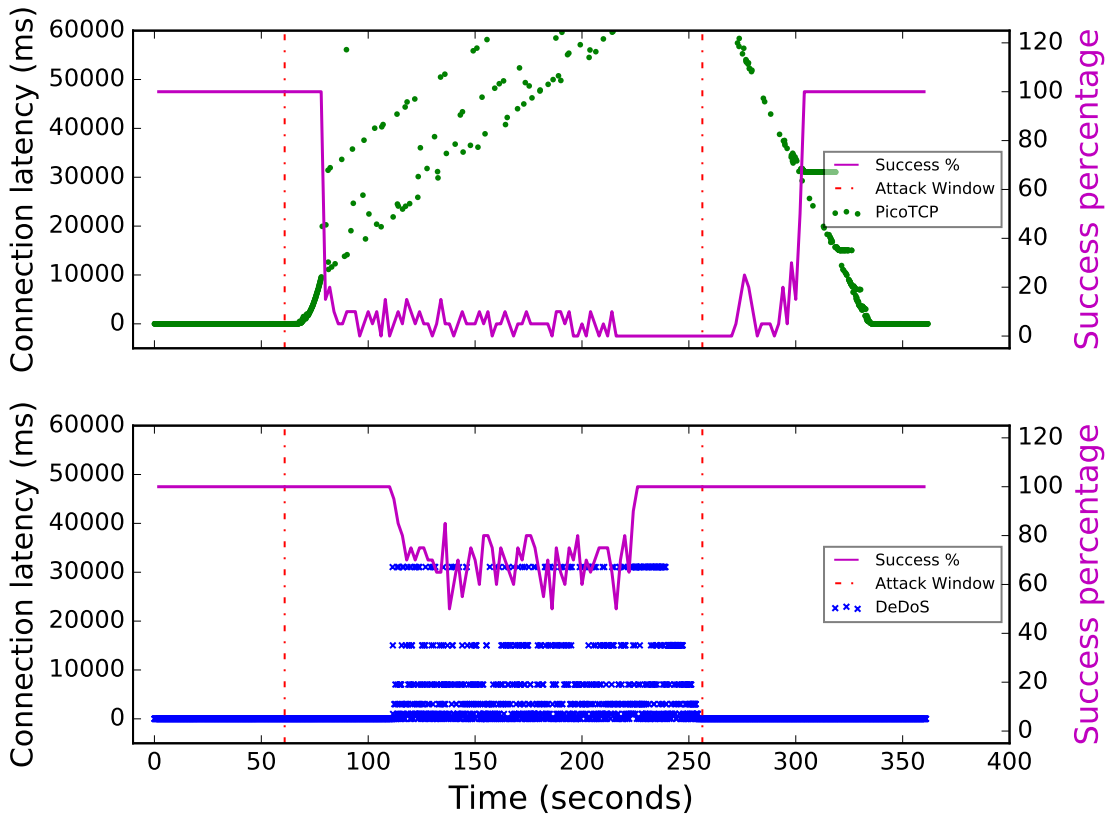


Fig. 5.8. Handshake latency of good clients under a SYN flood (2000 SYNs/sec) with standalone PicoTCP *(top)* and DeDoS with 3 Handshake MSUs *(bottom)*. Vertical lines denote the start and end of the SYN flood. The right y-axis plots the good clients' average percentage of successful TCP handshakes.

In contrast, with three cloned Handshake MSUs running on the same physical host (bottom graph of Figure 5.8), DeDoS is able to achieve an average success percentage of approximately 64% during the steady state of the attack and recovers quickly after the attack ends. (The stratified "bars" in the figure are due to TCP retransmissions and TCP backoff.)

Figure 5.9 shows the scalability of DeDoS and the improved response to various SYN floods with increasing resources. For a given attack strength, DeDoS is able to serve more legitimate requests as the number of handshake MSUs increases. Here, Handshake MSUs are equally distributed across the cores on three physical machines. Notably, we are able to completely mitigate the attack (as measured by successful client TCP connections) for moderate attack rates of 1000 and 2000 SYNs/second with four and seven MSUs, respectively.

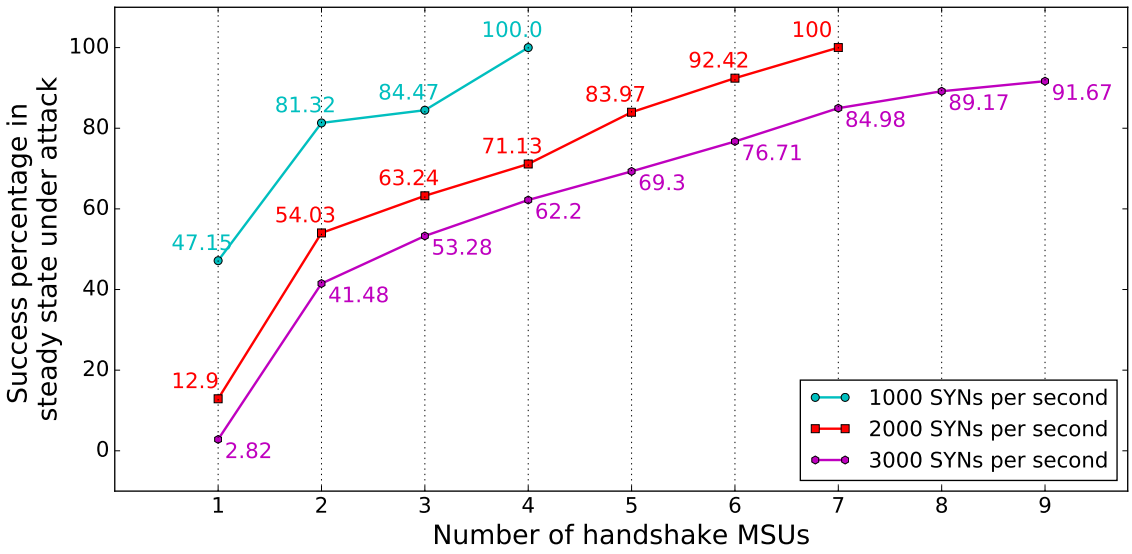


Fig. 5.9. Percentage of successful TCP connections with varying numbers of Handshake MSUs during a SYN flood.

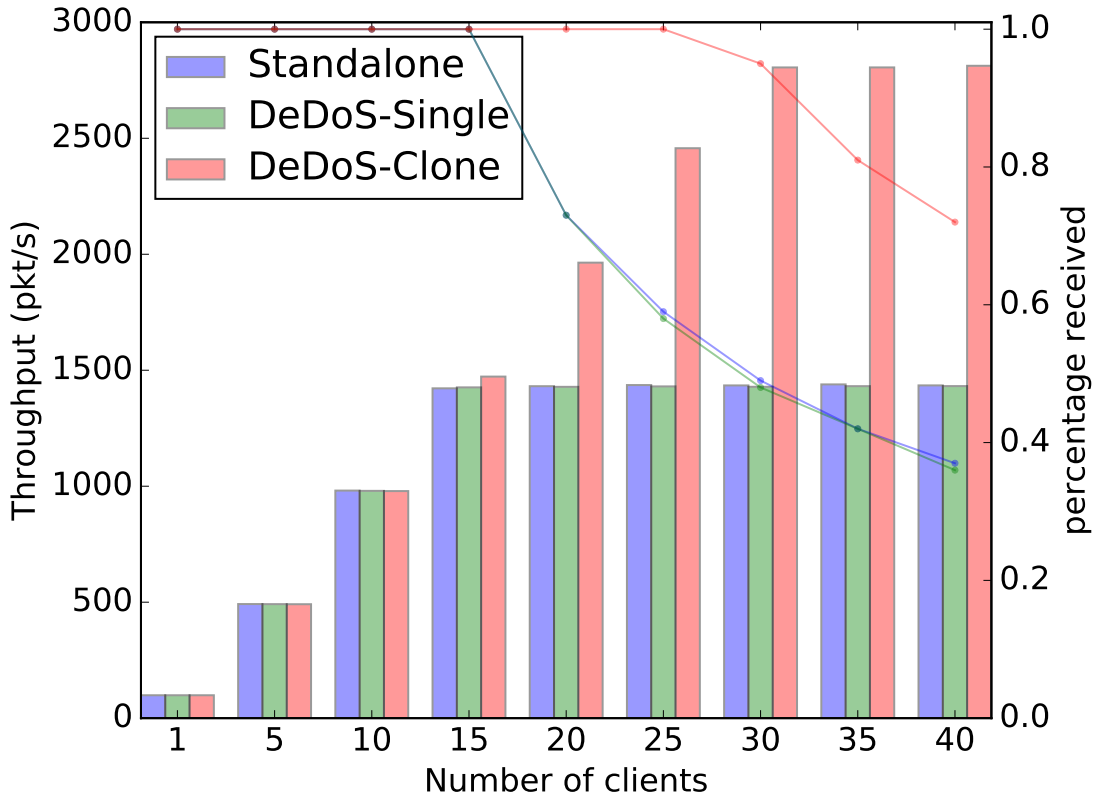


Fig. 5.10. The throughput of the declarative packet processing application under DoS attack.

Declarative packet processing attack: This final experiment uses an attack against the declarative packet processing application described in §5.4. The packet processing application is setup with a large in-memory neighbor table, rendering naïve replication too expensive in this case. The workload consists of a varying number of clients that forward packets via our application. The attack rate increases by using more clients to send more traffic. Figure 5.10 shows the throughput (pkts/s) that can be processed by (i) a standalone implementation, (ii) a DeDoS-enabled application with cloning disabled, and (iii) a normal DeDoS-enabled application. As before, the results show that standalone and DeDoS achieve comparable throughput (which indicates low overhead) but that cloning enables DeDoS to handle roughly twice as many clients during an attack.

5.6. Takeaways

This chapter presented our last contribution to this thesis, a system that, using Service Boosters's primitives, can harvest the exact amount of resources needed to mitigate high tail latency. Given the ability to monitor and manipulate individual Service Units, the DeDoS booster can devise more efficient scaling strategies.

The next chapter will recapitulate our findings and enumerate several interesting challenges for future work.

CHAPTER 6

Future and Related work

This thesis proposed a new execution environment for cloud applications, Service Boosters, offered as a library OS programmers can use to declare and annotate their request processing pipeline. In addition to the base architecture, the thesis contributes a real time anomaly detection engine able to spot requests about to increase tail latency — before they do so. The thesis also contributes two Boosters for request scheduling and resource harvesting, Perséphone and DeDoS. Both boosters exploit the structure and annotation of the DataFlow Graph declared by Service Boosters users to maintain good tail latency. Overall, Service Boosters supports the need for novel abstractions between cloud applications and their underlying execution environment.

6.1. Future work

Several challenges to using Service Boosters and similar component-based designs are not answered in this thesis. These are fundamental limitations from today's software design and should be addressed in future work.

I. Automatically composing a graph of Service Units. Currently, Service Boosters requires users to declare a pipeline of Service Units and program individual event handlers, *somewhat* similarly to Apache Spark's functional interface [**163**]. Ideally, programmers would only have to express the request processing pipeline in a sequential fashion and the Service Boosters runtime would split it into Service Units. Though some existing work tackle this challenge (*c.f.*, the next section of this chapter), the question of how to automatically decompose a sequence of functional units into a graph of components for a given objective (resource utilization, maneuverability, resilience to tail latency, *etc.*) is an open research problem. Specifically, this problem has different answers whether we want to retrofit legacy applications or only support new applications. We envision the answer to be at the intersection of programming languages — through the definition of domain specific languages, compilers and intermediate representations — and distributed systems — through the definition of new network and OS primitives.

II. State management. To be maneuverable at scale, Service Units need to either be stateless, or have access to a high performance shared storage. At the microsecond scale, this means carefully designing this storage to avoid overheads from acquiring the right to modify the storage. Work such as Silo [**164**] and the underlying Masstree B-tree structure [**165**] are interesting techniques for single nodes. Service Boosters also requires efficient distributed techniques to operate at its intended scale. Efforts such as S6 [**166**] are an interesting path forward, albeit limited to object storage. Designing

a satisfying storage abstraction to expose application-semantics to the underlying storage management layer within Service Boosters is still an open challenge.

III. Centralization. The Perséphone and DeDoS Boosters rely on a centralized controller to harvest resources and dispatch requests. This is unlikely to scale well to thousands of Service Boosters runtimes. Creating an efficient hierarchy of controllers that retains good approximation for optimal algorithms is an open research challenge. Specifically, as information needs to travel between controllers and each controller is responsible for a subset of the entire infrastructure, the more controllers, the less accuracy. We envision the solution to be at the intersection between OS optimizations — such as to devise low-overheads information transfer mechanisms — and distributed systems — through the design of concurrency protocols.

IV. Humans in the loop. Managing Service Boosters at scale still requires a lot of human effort to design, maintain, and enhance the request processing pipeline. Ideally, one would like Service Boosters to organically adjust to a target objective functions and automate resource management. Fundamentally, this means designing intelligent systems that can learn, for a given resource management predicate, all the pieces of this predicate: metrics of interest, decision thresholds, and appropriate sequence of actions. We envision the solution to leverage the Service Boosters declarative abstraction and reinforcement learning techniques to learn predicates.

V. Bringing in the network. Applications usually have expectations about the underlying network. Existing communication protocols are often all-include (*e.g.*, TCP) or totally bare (*e.g.*, UDP). As a result, programmers have to either build their own protocol from the ground up, or resort to using over-featured protocols — with an impact on performance. The design of fully modular and customizable network protocols is an open research challenge in the continuity of Service Boosters' aspirations. The question spawns interface design — what API is available for users? — to distributed systems — how to reason about a physical network shared by a large amount of network protocols? We explored some of these questions in the past [**131**] and foresee a large impact from this vein of research.

6.2. Related work

This section goes over systems and ideas related to the Service Boosters architecture. Sections 3.4 and 4.6 presented work related more generally to FineLame and Perséphone.

Breaking down application in Service Units. Previous work has already showcased the benefits of breaking application monoliths in smaller components. Flightplan assists users disaggregating P4 programs in the dataplane [**17**]. Ignis [**167**] shows that you can decompose applications into components that are individually scalable and Lya [**168**] shows that you can get significant insights about application's internal behaviors by tracking events at the module boundary. Service Boosters proposes a framework to capture components behavior and interactions, and expose these to the underlying resource management layer.

Function-as-a-Service and Micro-services platforms. Service Boosters is conceptually related to the trend toward fine-grained granularity decomposition of functions seen in FaaS platforms [**169, 170, 171, 172**]. Though Service Boosters also provides more manageable components, notably for mitigating high tail latency, the abstraction layer it targets is the operating system.

Other efforts exploiting application-awareness to improve tail latency. Robin-Hood [**173**] improves tail latency by provisioning more cache to backends that affect such latency. Minos [**174**] shards data based on size to reduce GETs variability across shards. It would be interesting to integrate such techniques to Service Boosters, specifically to broaden the application-OS abstraction and incorporate more advanced inter-stage load balancing techniques.

Bibliography

[1] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[2] Apache HTTP server project. `https://httpd.apache.org/`.

[3] Node.js server project. `https://nodejs.org/en/`.

[4] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The ix operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Trans. Comput. Syst.*, 34(4):11:1–11:39, December 2016.

[5] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[6] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 161–175, New York, NY, USA, 2015. Association for Computing Machinery.

[7] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 67–81, New York, NY, USA, 2016. Association for Computing Machinery.

[8] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, November 1984.

[9] Statista. Data centers - statistics facts . `https://www.statista.com/topics/6165/data-centers/`, 2020.

[10] Reno Gazette-Journal. Nevada approves Google's $600M data center near Las Vegas, $25.2M in tax incentives. `https://www.rgj.com/story/money/business/2018/11/16/nevada-approves-google-application-600-million-data-center-near-vegas/2026903002/`, 2018.

[11] Domestic Surveillance Directorate. Utah Data Center. `https://nsa.gov1.info/utah-data-center/`, 2020.

[12] Arman Shehabi, Sarah Smith, Dale Sartor, Richard Brown, Magnus Herrlin, Jonathan Koomey, Eric Masanet, Nathaniel Horner, Inês Azevedo, and William Lintner. United states data center energy usage report. 2016.

[13] Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. Recalibrating global data center energy-use estimates. *Science*, 367(6481):984–986, 2020.

[14] David Mytton. Data centre water consumption. *npj Clean Water*, 4(1):1–6, 2021.

[15] Henri Maxime Demoulin, Isaac Pedisich, Nikos Vasilakis, Vincent Liu, Boon Thau Loo, and Linh Thi Xuan Phan. Detecting asymmetric application-layer denial-of-service attacks in-flight with finelame. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 693–708, Renton, WA, July 2019. USENIX Association.

[16] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *Comm. ACM*, 52(11):87–95, November 2009.

[17] Flightplan: Dataplane disaggregation and placement for p4 programs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, April 2021.

[18] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J. Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proc 9th ACM SIGOPS European Workshop*, pages 109–114, 2000.

[19] Regular expression denial of service - ReDoS. 2017. `https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS`.

[20] Common vulnerabilities and exposures (see cve-2003-1564). `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-1564`.

[21] David Senecal. Slow DoS on the rise. 2013. `https://blogs.akamai.com/2013/09/slow-dos-on-the-rise.html`.

[22] John Pescatore. DDoS attacks advancing and enduring: A SANS survey. Technical report, SANS Institute, 2014.

[23] Fabrice J. Ryba, Matthew Orlinski, Matthias Wählisch, Christian Rossow, and Thomas C. Schmidt. Amplification and DRDoS attack defense – a survey and new perspectives. *CoRR*, abs/1505.07892, 2015.

[24] Christian Rossow. Amplification hell: Revisiting network protocols for DDoS abuse. In *Proc. NDSS*, 2014.

[25] Open Web Application Security Project. Owasp top ten project'17, 2018. Accessed: 2018-09-27.

[26] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 3–3, Berkeley, CA, USA, 2003. USENIX Association.

[27] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 361–376, Berkeley, CA, USA, 2018. USENIX Association.

[28] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (redos) in practice: An empirical study at the ecosystem scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 246–256, New York, NY, USA, 2018. ACM.

[29] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. Synode: Understanding and automatically preventing injection attacks on node.js. In *Networked and Distributed Systems Security*, NDSS'18, 2018.

[30] Michael Stepankin. [demo.paypal.com] node.js code injection (rce), 2016. Accessed: 2018-10-05.

[31] N. Seriot. `http://seriot.ch/parsing_json.php`, 2016.

[32] James C. Davis, Eric R. Williamson, and Dongyoon Lee. A sense of time for javascript and node.js: First-class timeouts as a cure for event handler poisoning. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 343–359, Berkeley, CA, USA, 2018. USENIX Association.

[33] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[34] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R Ganger. Stardust: tracking activity in a distributed storage system. In *ACM SIGMETRICS Performance Evaluation Review*, volume 34, pages 3–14. ACM, 2006.

[35] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 9–9, Berkeley, CA, USA, 2006. USENIX Association.

[36] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, pages 20–20. USENIX Association, 2007.

[37] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Technical report, Google, Inc, 2010.

[38] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. *ACM Trans. Comput. Syst.*, 35(4):11:1–11:28, December 2018.

[39] OpenTracing API. Consistent, expressive, vendor-neutral apis for distributed tracing and context propagation.

[40] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 2–2, Berkeley, CA, USA, 1993. USENIX Association.

[41] Vern Paxson Steven McCanne Van Jacobson, Sally Floyd. Tcpdump, a command-line packet analyzer.

[42] bcc on GitHub. `https://github.com/iovisor/bcc`.

[43] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press, 1996.

[44] Junhao Gan and Yufei Tao. Dbscan revisited: Mis-claim, un-fixability, and approximation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 519–530, New York, NY, USA, 2015. ACM.

[45] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. *arXiv preprint arXiv:1802.09089*, 2018.

[46] libuv. A multi-platform support library with a focus on asynchronous i/o.

[47] wikipedia. Wikipedia, the free encyclopedia.

[48] Tsung. `http://tsung.erlang-projects.org/`, 2017.

[49] Gkbrk. SlowLoris attack tool. `https://github.com/gkbrk/slowloris`, 2018.

[50] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: Online modelling and performance-aware systems. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 15–15, Berkeley, CA, USA, 2003. USENIX Association.

[51] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 20–20, Berkeley, CA, USA, 2007. USENIX Association.

[52] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 307–320, Berkeley, CA, USA, 2012. USENIX Association.

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

[54] Xiaohu Qie, Ruoming Pang, and Larry Peterson. Defensive programming: Using an annotation toolkit to build dos-resistant software. *SIGOPS Oper. Syst. Rev.*, 36(SI):45–60, December 2002.

[55] Srikanth Kandula, Dina Katabi, Matthias Jacob, and Arthur Berger. Botz-4-sale: Surviving organized ddos attacks that mimic flash crowds. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 287–300, Berkeley, CA, USA, 2005. USENIX Association.

[56] Tao Peng, Christopher Leckie, and Kotagiri Ramamohanarao. Survey of network-based defense mechanisms countering the dos and ddos problems. *ACM Comput. Surv.*, 39(1), April 2007.

[57] Saman Taghavi Zargar, James Joshi, and David Tipper. A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks. *IEEE Communications Surveys & Tutorials*, 15(4):2046–2069, 2013.

[58] Xiaowei Yang, David Wetherall, and Thomas Anderson. A dos-limiting network architecture. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '05, pages 241–252, New York, NY, USA, 2005. ACM.

[59] Yang Xu and Yong Liu. Ddos attack detection under sdn context. In *INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE*, pages 1–9. IEEE, 2016.

[60] Cristina Basescu, Raphael M Reischuk, Pawel Szalachowski, Adrian Perrig, Yao Zhang, Hsu-Chun Hsiao, Ayumu Kubota, and Jumpei Urakawa. Sibra: Scalable internet bandwidth reservation architecture. *arXiv preprint arXiv:1510.02696*, 2015.

[61] Xin Liu, Xiaowei Yang, and Yong Xia. Netfence: preventing internet denial of service from inside out. *SIGCOMM Comput. Commun. Rev.*, 41(4):–, August 2010.

[62] Min Suk Kang, Soo Bum Lee, and Virgil D. Gligor. The crossfire attack. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 127–141, Washington, DC, USA, 2013. IEEE Computer Society.

[63] Ahren Studer and Adrian Perrig. The coremelt attack. In *Proceedings of the 14th European Conference on Research in Computer Security*, ESORICS'09, pages 37–52, Berlin, Heidelberg, 2009. Springer-Verlag.

[64] Min Suk Kang and Virgil D. Gligor. Routing bottlenecks in the internet: Causes, exploits, and countermeasures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 321–333, New York, NY, USA, 2014. ACM.

[65] SS Jeremy Long. Owasp dependency check, 2015. Accessed: 2017-06-11.

[66] Snyk. Find, fix and monitor for known vulnerabilities in node.js and ruby packages, 2016.

[67] Wei Zhou, Weijia Jia, Sheng Wen, Yang Xiang, and Wanlei Zhou. Detection and defense of application-layer ddos attacks in backbone web traffic. *Future Generation Computer Systems*, 38:36–46, 2014.

[68] Tongguang Ni, Xiaoqing Gu, Hongyuan Wang, and Yu Li. Real-time detection of application-layer ddos attack using time series analysis. *J. Control Sci. Eng.*, 2013:4:4–4:4, January 2013.

[69] Hossein Hadian Jazi, Hugo Gonzalez, Natalia Stakhanova, and Ali A. Ghorbani. Detecting http-based application layer dos attacks on web servers in the presence of sampling. *Comput. Netw.*, 121(C):25–36, July 2017.

[70] Chenxu Wang, Tony TN Miu, Xiapu Luo, and Jinhe Wang. Skyshield: A sketch-based defense system against application layer ddos attacks. *IEEE Transactions on Information Forensics and Security*, 13(3):559–573, 2018.

[71] Supranamaya Ranjan, Ram Swaminathan, Mustafa Uysal, Antonio Nucci, and Edward Knightly. Ddos-shield: Ddos-resilient scheduling to counter application layer attacks. *IEEE/ACM Trans. Netw.*, 17(1):26–39, February 2009.

[72] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. Breakapp: Automated, flexible application compartmentalization. In *Proceedings of the 25th Networked and Distributed Systems Security Symposium*, NDSS'18, 2018.

[73] Wei Meng, Chenxiong Qian, Shuang Hao, Kevin Borgolte, Giovanni Vigna, Christopher Kruegel, and Wenke Lee. Rampart: Protecting web applications from cpu-exhaustion denial-of-service attacks. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 393–410, Berkeley, CA, USA, 2018. USENIX Association.

[74] Mohamed Elsabagh, Daniel Barbará, Dan Fleck, and Angelos Stavrou. Radmin: Early detection of application-level resource exhaustion and starvation attacks. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, RAID 2015, pages 515–537, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[75] Mohamed Elsabagh, Dan Fleck, Angelos Stavrou, Michael Kaplan, and Thomas Bowen. Practical and accurate runtime application protection against dos attacks. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 450–471. Springer, 2017.

[76] Rui Miao, Minlan Yu, and Navendu Jain. Nimbus: Cloud-scale attack detection and mitigation. *SIGCOMM Comput. Commun. Rev.*, 44(4):121–122, August 2014.

[77] Seyed K. Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and elastic ddos defense. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 817–832, Berkeley, CA, USA, 2015. USENIX Association.

[78] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, March 2017.

[79] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *Proc. of NSDI*, pages 385–398, 2013.

[80] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 205–220, New York, NY, USA, 2007. Association for Computing Machinery.

[81] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 1–16, USA, 2014. USENIX Association.

[82] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. The datacenter as a computer: Designing warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 13(3):i–189, 2018.

[83] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. Mtcp: A highly scalable user-level tcp stack for multicore systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, page 489–502, USA, 2014. USENIX Association.

[84] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 1–16. USENIX Association, 2019.

[85] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, and et al. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[86] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 325–341, New York, NY, USA, 2017. ACM.

[87] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, pages 361–377, Berkeley, CA, USA, 2019. USENIX Association.

[88] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for msecond-scale tail latency. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 345–359, USA, 2019. USENIX Association.

[89] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. Pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 435–446, New York, NY, USA, 2013. Association for Computing Machinery.

[90] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 221–235, New York, NY, USA, 2018. Association for Computing Machinery.

[91] memcached. Memcached protocol. `https://github.com/memcached/memcached/blob/master/doc/protocol.txt`. Accessed: 2021-03-24.

[92] Wei Dai. Redis Protocol specification. `https://redis.io/topics/protocol`. Accessed: 2021-05-05.

[93] Google. Protocol Buffers - Google's data interchange format. `https://github.com/protocolbuffers/protobuf`. Accessed: 2020-12-09.

[94] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 253–266, San Jose, CA, April 2012. USENIX Association.

[95] Mor Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.

[96] Drew Gallatin, Netflix Technology Blog. Serving 100 Gbps from an Open Connect Appliance. `https://netflixtechblog.com/serving-100-gbps-from-an-open-connect-appliance-cdb51dda3b99`. Accessed: 2020-12-04.

[97] Marek Majkowski, The Cloudflare Blog. How to achieve low latency with 10Gbps Ethernet. `https://blog.cloudflare.com/how-to-achieve-low-latency/`. Accessed: 2020-12-04.

[98] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 335–348, USA, 2012. USENIX Association.

[99] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–14, New York, NY, USA, 2014. Association for Computing Machinery.

[100] Mor Harchol-Balter, Cuihong Li, Takayuki Osogami, Alan Scheller-Wolf, and Mark S. Squillante. Cycle stealing under immediate dispatch task assignment. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '03, page 274–285, New York, NY, USA, 2003. Association for Computing Machinery.

[101] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3146–3154. Curran Associates, Inc., 2017.

[102] DPDK. Data plane development kit. `https://www.dpdk.org/`. Accessed: 2020-10-27.

[103] Tom Shanley. *InfiniBand network architecture*. Addison-Wesley Professional, 2003.

[104] Intel Data Direct. I/o technology (intel ddio) a primer, 2012.

[105] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new

os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 29–44, New York, NY, USA, 2009. Association for Computing Machinery.

[106] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[107] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[108] Pulkit A. Misra, María F. Borge, Íñigo Goiri, Alvin R. Lebeck, Willy Zwaenepoel, and Ricardo Bianchini. Managing tail latency in datacenter-scale file systems under production constraints. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[109] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. On the diversity of cluster workloads and its impact on research results. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 533–546, Boston, MA, July 2018. USENIX Association.

[110] Zhen Cao, Vasily Tarasov, Hari Prasath Raman, Dean Hildebrand, and Erez Zadok. On the performance variation in modern storage stacks. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 329–344, Santa Clara, CA, February 2017. USENIX Association.

[111] TPC. Tpc-c. `http://www.tpc.org/tpcc/`. Accessed: 2020-10-20.

[112] Facebook. Rocksdb. `https://rocksdb.org/`. Accessed: 2020-11-17.

[113] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (bvt) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, page 261–276, New York, NY, USA, 1999. Association for Computing Machinery.

[114] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of cloudlab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, July 2019. USENIX Association.

[115] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. Tas: Tcp acceleration as an os service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[116] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Lightweight preemptible functions. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 465–477. USENIX Association, July 2020.

[117] Per Brinch Hansen. The nucleus of a multiprogramming system. *Commun. ACM*, 13(4):238–241, April 1970.

[118] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. *Commun. ACM*, 17(6):337–345, June 1974.

[119] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. 1986.

[120] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, et al. Overview

of the chorus distributed operating system. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, 1992.

[121] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 267–283, New York, NY, USA, 1995. Association for Computing Machinery.

[122] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 251–266, New York, NY, USA, 1995. Association for Computing Machinery.

[123] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, page 429–444, USA, 2014. USENIX Association.

[124] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash ≈ local flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 345–359, New York, NY, USA, 2017. Association for Computing Machinery.

[125] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable rdma rpc on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[126] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 17–33. USENIX Association, 2017.

[127] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[128] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[129] Adam Wierman and Bert Zwart. Is tail-optimal scheduling possible? *Operations research*, 60(5):1249–1257, 2012.

[130] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 456–468. IEEE Press, 2016.

[131] Henri Maxime Demoulin, Nikos Vasilakis, John Sonchack, Isaac Pedisich, Vincent Liu, Boon Thau Loo, Linh Thi Xuan Phan, Jonathan M. Smith, and Irene Zhang. Tmc: Pay-as-you-go distributed communication. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, APNet '19, page 15–21, New York, NY, USA, 2019. Association for Computing Machinery.

[132] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network stack specialization for performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 175–186, New York, NY, USA, 2014. Association for Computing Machinery.

[133] A. Mirhosseini and T. F. Wenisch. The queuing-first approach for tail management of interactive services. *IEEE Micro*, 39(4):55–64, 2019.

[134] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose schedular. In David Kotz and John Wilkes, editors, *Proceedings of the 17th ACM Symposium on Operating System Principles, SOSP 1999,*

Kiawah Island Resort, near Charleston, South Carolina, USA, December 12-15, 1999, pages 261–276. ACM, 1999.

[135] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018.

[136] Ingo Molnar. [patch] modular scheduler core and completely fair scheduler. `https://lwn.net/Articles/230501/`. Accessed: 2020-12-01.

[137] Alexandra Fedorova, Margo Seltzer, and Michael D Smith. A non-work-conserving operating system scheduler for smt processors. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA*, volume 33, pages 10–17, 2006.

[138] Emilia Rosti, Evgenia Smirni, Giuseppe Serazzi, and Lawrence W Dowdy. Analysis of non-work-conserving processor partitioning policies. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 165–181. Springer, 1995.

[139] Evgenia Smirni, Emilia Rosti, Giuseppe Serazzi, Lawrence W Dowdy, and Kenneth C Sevcik. Performance gains from leaving idle processors in multiprocessor systems. In *ICPP (3)*, pages 203–210. Citeseer, 1995.

[140] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 863–880, Renton, WA, July 2019. USENIX Association.

[141] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV service chains at the true speed of the underlying hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 171–186, Renton, WA, April 2018. USENIX Association.

[142] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and efficient NIC packet scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 33–46, Boston, MA, February 2019. USENIX Association.

[143] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. Elastic rss: Co-scheduling packets and cores using programmable nics. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, APNet '19, page 71–77, New York, NY, USA, 2019. Association for Computing Machinery.

[144] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. Rss++: Load and state-aware receive side scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, page 318–333, New York, NY, USA, 2019. Association for Computing Machinery.

[145] Intel. Adq. `https://www.intel.com/content/www/us/en/architecture-and-technology/ethernet/application-device-queues-technology-brief.html`. Accessed: 2020-11-10.

[146] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can JUMP them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 1–14, Oakland, CA, May 2015. USENIX Association.

[147] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Phost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, New York, NY, USA, 2015. Association for Computing Machinery.

[148] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 455–468, Oakland, CA, May 2015. USENIX Association.

[149] Ali Munir, Ghufran Baig, Syed M. Irteza, Ihsan A. Qazi, Alex X. Liu, and Fahad R. Dogar. Friends, not foes: Synthesizing existing transport strategies for data center networks. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 491–502, New York, NY, USA, 2014. Association for Computing Machinery.

[150] Yuanwei Lu, Guo Chen, Larry Luo, Kun Tan, Yongqiang Xiong, Xiaoliang Wang, and Enhong Chen. One more queue is enough: Minimizing flow completion time with explicit priority notification. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[151] Aisha Mushtaq, Radhika Mittal, James McCauley, Mohammad Alizadeh, Sylvia Ratnasamy, and Scott Shenker. Datacenter congestion control: Identifying what is essential and making it practical. *SIGCOMM Comput. Commun. Rev.*, 49(3):32–38, November 2019.

[152] Md E. Haque, Yuxiong He, Sameh Elnikety, Thu D. Nguyen, Ricardo Bianchini, and Kathryn S. McKinley. Exploiting heterogeneity for tail latency and energy efficiency. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 625–638, New York, NY, USA, 2017. Association for Computing Machinery.

[153] Myeongjae Jeon, Yuxiong He, Hwanju Kim, Sameh Elnikety, Scott Rixner, and Alan L. Cox. Tpc: Target-driven parallelism combining prediction and correction to reduce tail latency in interactive services. *SIGPLAN Not.*, 51(4):129–141, March 2016.

[154] SSL renegotiation DoS. 2011. `https://www.ietf.org/mail-archive/web/tls/current/msg07553.html`.

[155] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[156] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proc. SOSP*, 1997.

[157] Sam Newman. *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.", 2015.

[158] Joyent Inc. and other Node contributors. NodeJS HTTP Parser. `https://github.com/nodejs/http-parser`.

[159] Willy Tarreau. HA-Proxy load balancer. `http://haproxy.com/`, 2018.

[160] picoTCP, 2018. `http://www.picotcp.com/`.

[161] DeDOS demonstration at SIGCOMM 2017. `https://www.youtube.com/watch?v=KX4EPnUzDqk`, 2017.

[162] Common vulnerabilities and exposures (see cve-2015-8386). 2018. `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8386`.

[163] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.

[164] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, New York, NY, USA, 2013. Association for Computing Machinery.

[165] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 183–196, New York, NY, USA, 2012. Association for Computing Machinery.

[166] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 299–312, Renton, WA, April 2018. USENIX Association.

[167] Nikos Vasilakis, Ben Karel, Yash Palkhiwala, John Sonchack, André DeHon, and Jonathan M. Smith. Ignis: Scaling distribution-oblivious systems with light-touch distribution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1010–1026, New York, NY, USA, 2019. Association for Computing Machinery.

[168] Nikos Vasilakis, Grigoris Ntousakis, Veit Heller, and Martin C. Rinard. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '21, page 1–12, New York, NY, USA, 2021. Association for Computing Machinery.

[169] OpenWhisk. `https://developer.ibm.com/openwhisk`, 2018.

[170] AWS lambda. `https://aws.amazon.com/lambda`, 2018.

[171] Azure functions. `https://functions.azure.com`, 2018.

[172] Google Cloud Functions. `https://cloud.google.com/functions`, 2018.

[173] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. Robinhood: Tail latency aware caching – dynamic reallocation from cache-rich to cache-poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 195–212, Carlsbad, CA, October 2018. USENIX Association.

[174] Diego Didona and Willy Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 79–93, USA, 2019. USENIX Association.