

**A Multilayer Approach Towards Optimizing Spatial and
Temporal Computing Systems**

by

Erika Hunhoff

B.S., Computer Science, George Fox University, 2015

B.S., Mathematics, George Fox University, 2015

M.S., Computer Science, University of Colorado Boulder, 2021

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2025

Committee Members:

Dr. Eric Keller, Chair

Dr. Joe Izraelevitz

Dr. Gerd Zellweger

Dr. Sangtae Ha

Dr. Yueqi Chen

Hunhoff, Erika (Ph.D., Computer Science)

A Multilayer Approach Towards Optimizing Spatial and Temporal Computing Systems

Thesis directed by Dr. Eric Keller

Optimizing computing systems towards efficiency and performance is a fundamental problem of computing systems. This problem continues to be challenging, as new and emerging computing systems exhibit complex performance characteristics influenced by the spatial and temporal details of execution placement and schedule. The execution schedule and placement of a computation is defined and shaped at many levels, potentially including the application structure, a framework, a runtime, the operating system, and the hardware architecture. Thus, it is necessary to optimize for spatial and temporal characteristics throughout a computing stack.

This dissertation presents works that optimize select computing systems using abstractions that manage spatial and temporal characteristics. Each work addresses a different level in the computing stack. The first work, positioned at the intersection between applications and frameworks, proposes an improvement to state management in serverless systems to improve spatial locality at targeted temporal intervals. The second work, positioned at the intersection between runtimes and operating systems, provides fine-grained, temporally-dynamic resource limits for containers. The third work, and primary effort, includes the design and implementation of a library that provides temporally-dynamic spatial replication of data structures. This library, LD-NR, is positioned to support both applications and operating systems. The third work also includes the design and implementation of a distributed operating system targeting emerging extended non-uniform memory access architectures. The fourth work, positioned at the hardware level, extends and refines a programming interface for a type of neural processing units (NPUs) with an explicitly spatial and temporal design.

Acknowledgements

I am grateful to have worked on interesting projects during my PhD, but even more honored to have worked with so many inspiring, technically excellent, and kind individuals. A big thank you to my advisor, Dr. Eric Keller! I learned a great deal about how to ask questions and make decisions from my time in your lab. Thank you to my former advisor, Dr. Eric Rozner. The years I was a part of your lab taught me habits that set me up for success. Thank you also to Google for the Google PhD Fellowship, which likewise helped shape my trajectory. Thank you to all of my coauthors; your work is reflected throughout this dissertation, and I really enjoyed working together. Thank you to my committee for your time and attention. Thank you also to all the folks I got to know while interning at both VMware Research Lab (especially Dr. Gerd Zellweger and Dr. Marcos Aguilera!) and AMD Research & Development (especially Dr. Kristof Denolf and Dr. Joseph Melber!). These internships are highlights of my professional career.

Nearest and dearest to my heart, thank you to my family and friends for supporting me and brightening my life. Thank you, Mike, for walking with me through this long journey. I am proud to be your wife, and excited for our next adventures! Thank you to my self-appointed PhD fan club (A.K.A. my parents) for their support throughout my life. Thank you to my friends. Notably, thank you to Mara. Maybe with another ten years of friendship we will manage to have a video chat that is shorter than two hours (but I kind of hope not!). Thank you Cydney, for inspiring me to be bold in finding my path. Thanks to the furry menagerie (dogs Sable and Maya; loyal cat Wampus) for unconditional love and chaos. I cherish the time I've been able to spend with family, friends, neighbors, and pets.

Contents

Chapter

1	Overview	1
2	Introduction	3
3	Background	7
3.1	An Abstract View of Container-Based Cloud Services	7
3.2	Container-Based Cloud Services in the Wild	9
3.2.1	Serverless	9
3.2.2	Container Orchestration Frameworks	11
3.3	Scalable Operating Systems and NUMA	13
3.3.1	Scalable Operating Systems	14
3.3.2	NUMA and Extended NUMA Systems	15
3.3.3	Challenges	17
3.4	Neural Processing Units (NPUs)	19
3.4.1	Accelerators and NPUs	19
3.4.2	Challenges for NPU Programming	20
4	freshen: Proactive Serverless Function Resource Management	22
4.1	Introduction	22
4.2	Background and Motivation	24

4.3	Design and Implementation	26
4.3.1	When to freshen	27
4.3.2	Opportunities to freshen	27
4.3.3	Implementation	29
4.4	Evaluation	33
4.5	Related Work	34
4.6	Discussion and Conclusion	35
5	Escra: Event-driven, Sub-second Container Resource Allocation	36
5.1	Introduction	36
5.2	Related Work	39
5.3	Introducing Escra	40
5.4	Escra Architecture	44
5.4.1	Application Deployer & Container Watcher	44
5.4.2	Kernel Hooks	45
5.4.3	Controller	46
5.4.4	Resource Allocator	48
5.5	Implementation	51
5.6	Evaluation	51
5.6.1	Experimental Setup	52
5.6.2	Performance - Cost-Efficiency Trade-off	53
5.6.3	Static Allocation vs. Escra	55
5.6.4	Takeaways	57
5.6.5	Serverless	58
5.6.6	OpenWhisk vs. Escra + OpenWhisk	59
5.6.7	Takeaways	61
5.6.8	Escra MicroBenchmarks and Overheads	61

5.7	Discussion and Future Work	62
5.8	Conclusion	63
6	LD-NR and DiNOS: NUMA-Aware Replication for Dynamic Systems and an Operating System for a Shared Memory Rack	64
6.1	Introduction	64
6.2	Background and Motivation	67
6.2.1	Background on Node Replication	68
6.2.2	Motivating Dynamic Replication	70
6.2.3	Impact of NR’s Static Replica Configuration	71
6.3	Live-Dynamic Node Replication (LD-NR)	76
6.3.1	Design Overview	77
6.3.2	NUMA- and Log-Aware Replica Cloning	78
6.3.3	Dynamic matching of threads to replicas	79
6.3.4	Avoiding Stalls with Affinity Management	80
6.4	A Rackscale Operating System using LD-NR	81
6.4.1	Context	81
6.4.2	Design	82
6.5	Scheduling and Allocation at Rackscale	85
6.6	Implementation	89
6.7	Evaluation	90
6.7.1	LD-NR Hashmap	90
6.7.2	DiNOS and LD-NR	94
6.8	Related Work	99
6.9	Conclusion	100
7	Extensions to IRON: Efficiency, Expressivity, and Extensibility in a Close-to-Metal NPU Programming Interface	102

7.1	Introduction	102
7.2	Architectural Overview of AMD XDNA™ NPUs	104
7.2.1	AMD XDNA™ NPU Architecture	104
7.2.2	Implications for Programming and Optimization	105
7.3	IRON: A Performance Toolkit for NPUs	106
7.3.1	IRON Overview	106
7.3.2	Lifecycle of an IRON Design	107
7.3.3	Outline of an IRON Design	107
7.3.4	Data Movement in IRON	109
7.4	Efficiency, Expressivity, and Extensibility	109
7.5	Contributions to IRON	110
7.6	Implementation	116
7.7	Evaluation	116
7.7.1	Efficiency	116
7.7.2	Expressivity	118
7.7.3	Extensibility	121
7.8	Related Work	122
7.9	Conclusion	123

Bibliography**124**

Tables**Table**

4.1 Overheads of Function Triggers	26
5.1 Comparison of Escra and Autopilot Performance and Slack	54
7.1 Example IRON Designs	117
7.2 Features of Advanced IRON Designs	120

Figures

Figure

2.1 Computing Stack Layer Overview	4
2.2 Visual Overview of Works	5
3.1 Illustration of Kubernetes Resource Stranding	13
4.1 freshen Opportunities	24
4.2 Serverless Orchestration Application Function Chain CDF	24
4.3 freshen Timing	27
4.4 File Retrieval Overheads	33
4.5 Cloud File Transfer Latency	33
4.6 Edge File Transfer Latency	33
5.1 Escra Architecture	41
5.2 Escra CPU Tracking	42
5.3 Escra Components	45
5.4 Escra Microservice Latency and Throughput	55
5.5 Escra Microservice CPU Slack	55
5.6 Escra Microservice Memory Slack	55
5.7 Escra Serverless Latencies	60
5.8 Escra Serverless Resource Utilization for ImageProcess	61
5.9 Escra Serverless Resource Utilization for GridSearch	61

6.1	Illustration of a Hashmap Replicated with NR	69
6.2	Illustration of NR Replication Strategies	72
6.3	Performance in Throughput and Memory Utilization of NR Hashmap	74
6.4	Illustration of LD-NR Design	76
6.5	Illustration of Example LDNR-vMem Replication in DiNOS	82
6.6	Memory allocation in DiNOS*	86
6.7	DiNOS* Macro Scheduler Tables	87
6.8	LD-NR Hashmap Benchmarks	91
6.9	LD-NR Liveness Experiment	93
6.10	Memcached Scalability Experiment	95
6.11	Dynamic Page Table Replication with DiNOS	96
6.12	Memcached Throughput of DiNOS* Macro Schedulers	98
7.1	Simplified AMD XDNA™ NPU Architecture	104
7.2	Example IRON Design Comparison	108
7.3	Example <code>taplib</code> Visualizations	114
7.4	IRON Single Lines of Code (SLOC) Comparison	118
7.5	IRON Halstead Vocabulary and Effort Comparison	119

Chapter 1

Overview

For computing systems at all levels — from hardware to top-level applications — complexity and inefficiency can occur due to spatial and temporal characteristics. A spatial characteristic arises when the location where a computation is performed influences efficiency and performance. A temporal characteristic arises when the time a computation is executed influences efficiency and performance. These effects become apparent when an abstract computing task is mapped to concrete spatial placements and temporal schedules for execution within a computing system. It is necessary to consider all levels in the computing stack when considering spatial and temporal optimization, as all levels may influence placement and schedule. For instance, the application layer may define how an abstract task is decomposed into smaller units; a framework may define how and when these units are submitted to a system; a runtime defines the system resources available and exposed to a task; the operating system mediates use of hardware resources; finally, the hardware architecture defines the capabilities of physical resources.

This dissertation presents works focusing on four different levels in the computing stack, from the top down. Each work utilizes knowledge of spatial and temporal characteristics to mitigate inefficiencies or increase the usability of the target computing system. Starting with the intersection of applications and frameworks, the first work addresses state management in serverless systems. State management in serverless systems is difficult to optimize due the conceptual, and often spatial, divide between computation and state in serverless systems. The proposed mechanism uses temporal intervention within the serverless function life cycle to predictively optimize the

placement of state. The second work implements fine-grained, temporally-dynamic resource limits for containers, ensuring that the resources granted to a container reflect the needs of the workload running within the container. The third work is the design and implementation of a library, LD-NR. LD-NR optimizes access to data structures in non-uniform memory access (NUMA) systems. LD-NR works by managing a temporally-dynamic set of data structure replicas, each of which is spatially placed within a NUMA domain. The capabilities of LD-NR are then leveraged to design and implement a distributed operating system, DiNOS. DiNOS allows processes to scale memory and compute beyond host boundaries when deployed on emerging, extended NUMA architectures. The fourth work extends and refines a close-to-metal programming interface for neural processing units (NPUs), an architecture with explicit spatial and temporal characteristics. The goal of the fourth work is to increase programmer efficiency and programming interface extensibility without compromising control of low-level hardware features.

In sum, this dissertation contains four works, positioned at varying levels of the computing stack, that use custom abstractions to optimize computing systems with spatial and temporal characteristics.

Chapter 2

Introduction

The process of mapping a computing task to available resources within a system is a fundamental component of efficient computation. This process remains a challenge because computing resources and computing tasks have grown in scale, complexity, and specialization. Architecturally, the complexity of computing resources has increased with the introduction of hyperthreads, multicores, sockets, chiplets, and on-die accelerators. Per-host complexity is compounded by cloud computing frameworks which slice, compose, and virtualize hosts using containers and virtual machines (VMs). Additionally, advances in resource disaggregation blur the lines between discrete components. Architectural complexity is exacerbated by demanding and complex workloads. These workloads include large-scale AI/ML tasks, data science queries over large data sets, web services with varying scale, and latency-sensitive remote procedure calls (RPCs). Figure 2.1 illustrates common layers in modern computing stacks, along with an example of each layer. These layers exist to support complex workloads over complex architectures.

A practical effect of this complexity is spatial and temporal non-uniform performance characteristics. A spatial characteristic arises when the location where a computation is performed influences efficiency and performance. A temporal characteristic arises when the time a computation is executed influences efficiency and performance. Reasoning about spatial and temporal non-uniform characteristics is difficult, since these effects are determined by the combination of many factors.

All levels of a computing stack can (1) introduce new spatial and temporal characteristics, and

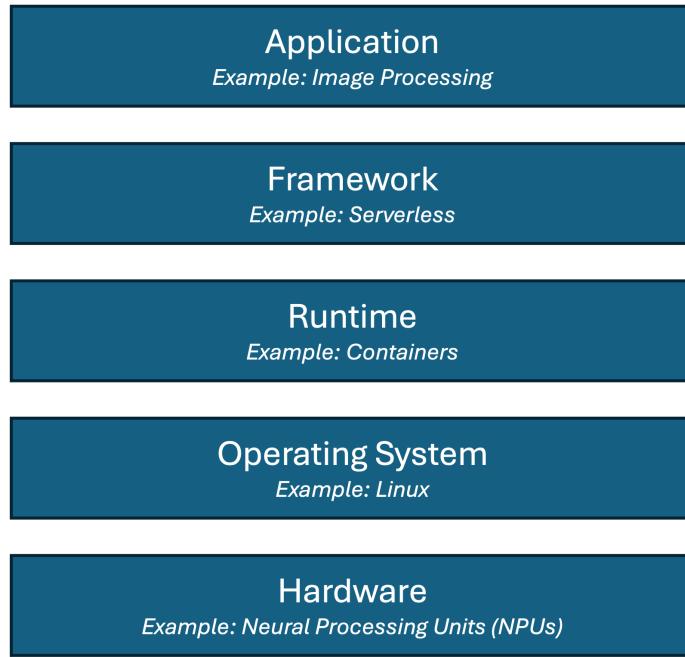


Figure 2.1: Common layers in computing stacks, with an example of each.

(2) influence how characteristics introduced at other levels are exhibited. This can be illustrated using the application layer (the top layer in Figure 2.1). An application typically contains the representation of the abstract computing task. For instance, an application may be composed of a single large task, many small tasks, or anywhere in between. Towards (1), some applications are constructed as distributed applications (e.g., to allow horizontal scaling) with logical partitioning of application state (regardless of whether the application is also physically partitioned). This can introduce new, application-level non-uniform spatial and temporal characteristics, specifically through the introduction of tasks such as load balancing and algorithms to maintain consistency. Towards (2), the representation of an application's work as many small pieces or a monolithic sequential task can influence how work is mapped to abstractions provided by lower levels (for instance, whether pieces of the application can be parallelized by operating system threads). In this way, the application level can influence how characteristics introduced at lower levels are exhibited. Examples for both (1) and (2) exist for other layers in the computing stack.

There are several mechanisms used to simplify the problem of optimizing computing systems with spatial and temporal characteristics. First, the computing stack can be decomposed into a

series of layers (e.g., Figure 2.1). Second, each layer can be represented and accessed by a set of abstractions exposed through a programming interface. Using these techniques, it is possible to pursue optimization and usability through abstractions and programming interfaces layer by layer.

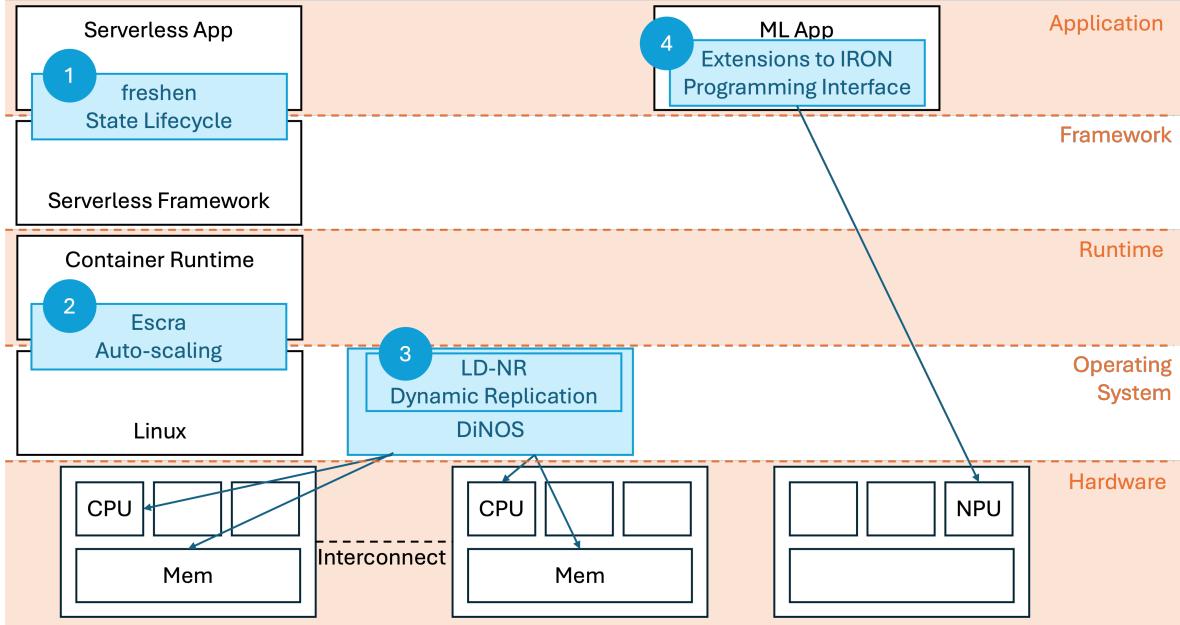


Figure 2.2: A visual representation of the four works presented in this dissertation, and how each work fits into the layers of the computing stack.

This dissertation utilizes this philosophy to present a multilayered approach towards optimizing computing systems with spatial and temporal characteristics. Using a top-down approach, each work is positioned at a different place within the computing stack. Figure 2.2 gives a visual summary of the placement of each work. Each work specifies a system of interest, identifies the effects of spatial and/or temporal non-uniform characteristics in that system, and presents new or refined abstractions. The goal of each proposed abstraction is to (1) provide performance benefit while (2) minimizing complexity for the user.

The first two works target container-based cloud computing systems with the aim of increasing efficiency (performance and resource utilization). The first work, *freshen* in Figure 2.2, identifies how the serverless life cycle introduces spatial and temporal non-uniformity and proposes a new phase in the serverless programming life cycle towards efficient application state management. The second work, *Escra* in Figure 2.2, identifies temporal dynamism in container-based cloud systems

and details how temporal non-uniformity is amplified by rigid container resource limits. We then design, implement, and evaluate a runtime providing Event-driven, Sub-second Container Resource Allocation (Escra).

The third contribution, LD-NR/DiNOS in Figure 2.2, focuses on management of data structures in non-uniform memory access (NUMA) systems, particularly emerging and extended NUMA architectures. LD-NR is a library for NUMA-aware and temporally-dynamic replication of sequential data structures. DiNOS is a distributed operating system targeting a rackscale, extended NUMA architecture; DiNOS uses dynamic replication provided by LD-NR to manage process page tables.

The fourth work, extensions to IRON in Figure 2.2, extends and refines a close-to-metal programming interface for neural processing units (NPUs). The NPU architecture considered in this work has architecturally defined spatial and temporal characteristics, as processing elements are spatially arranged and the memory hierarchy is primarily composed of scratchpad memory (i.e., without hardware support for coherence). This work focuses on how abstractions can reduce complexity for the programmer while still exposing close-to-metal control of the device.

While the details of these works greatly differ, the mechanics — where and when data is placed and accessed, where and when compute is scheduled, the relationships between data and compute — are similar. These works do not compose to form an end-to-end system but in sum they illustrate how optimization of spatial and temporal considerations can be used for end-to-end system optimization, even in multilayered and complex computing environments.

Chapter 3

Background

This section covers information on the application, framework, runtime, and operating system layers as they pertain to container-based cloud services, the operating system and hardware layers as they pertain to NUMA and extended NUMA systems, and the hardware layer as it pertains to NPUs. This information was selected to be directly useful in interpreting the works presented in subsequent chapters. Specifically, Sections 3.1-3.2 present an overview of containerized cloud services using common abstractions, which is useful for understanding Chapters 4-5. Section 3.3 provides overviews on operating systems (Section 3.3.1) and NUMA systems (Section 3.3.2), which is useful for understanding Chapter 6. Lastly, Section 3.4 outlines backgrounds on NPUs and reconfigurable accelerators, which is useful for understanding Chapter 7. In each of these sections, spatial and temporal characteristics are highlighted.

3.1 An Abstract View of Container-Based Cloud Services

This section provides an overview of how container-based cloud services are constructed using four of the layers presented in Figure 2.1: the application, the framework, the runtime, and the operating system.

Application The application layer is, in containerized workloads, the only layer that the customer must provide. We define the application as the logic needed to perform a computational task. Included in this definition is the state needed to run the application in a consistent environment. In containerized workloads, environment state generally consists of a container image, which

may be provided by the cloud provider or by the application owner. In order to run without error, the application must abide by the bounds the container layer imposes. That is, an application cannot be deployed outside of the container so the application cannot use more resources than the encapsulating container is configured to provide. The resources limited by a container often include CPU and memory, but may also include access to other hardware resources such as NICs or accelerators.

Runtime Containers, and the container runtime, allow the cloud service to control many aspects of how application workloads are executed. Containers define boundaries used to enforce isolation in multi-tenant systems. Containers, and container images, also provide useful tooling for managing the state necessary to run applications such that the application experiences a consistent deployment environment despite diversity at the operating system and hardware level. We choose to use the word *container* for this section, rather than *execution environment*, for brevity and because container technologies such as Docker [65] and LXC [143] are popular and well established. However, in practice, there are a variety of implementations of the abstract idea of the container, including firecracker [2], gVisor [239], and even light-weight language-based isolation techniques such as found in WebAssembly [232]. Containers are often deployed by a container orchestration service (e.g., Kubernetes [126]) or by other cloud infrastructure. Containers are limited by the bounds imposed by the operating system. That is, a container cannot use more memory, CPU, or any other resources than the operating system is able to give it. Depending on the particular cloud service, the application owner or the cloud provider is responsible for instantiation, replication, and the definition of resource limits for a container.

Operating System The operating system is responsible for facilitating use of hardware to fulfill resources requested by containers. The capabilities that the operating system layer can provide are governed both by the structure of the operating system (e.g., what is the programming interface that allows containers to ask for resources) and by the underlying hardware (e.g., the operating system cannot mediate access to resources that it does not manage).

3.2 Container-Based Cloud Services in the Wild

This section provides an overview of two common container-based cloud services, serverless and container orchestration, and highlights several spatial and temporal characteristics found in such systems.

3.2.1 Serverless

Serverless¹ is a type of cloud offering characterized by a high level of abstraction. The goal of serverless is to remove all “serverful” considerations of cloud deployment, including provisioning, instantiation, scaling, and error handling. In serverless, the customer provides a function; the cloud provider is then responsible for invoking the function in response to an event. Some examples of serverless offerings include AWS Lambda [20], Google Cloud Functions [81], and Azure Functions [24].

3.2.1.1 Scalability in Serverless Serverless uses a specific method of scaling at the application and container layers: replication. Each invocation of a function is typically run in its own container. A key benefit that sets serverless apart from most other cloud deployment models is that serverless supports scale-to-zero. If a function is not invoked, no resources are used (or paid for) by the customer. While the application scales at a per-invocation granularity, each container instance (as enforced by the operating system) is generally constrained to a statically defined set of resource limits. When a container is spawned to fulfill an application request, the limits typically do not change during the life of the container.² Furthermore, the CPU to memory ratio of the container limits are often fixed [73]. To limit the application overall, the cloud customer may set aggregate resource limits such as the maximum number of concurrent invocations. As container instances themselves are managed across machines via the serverless framework, often through use

¹ Throughout this work, the term serverless is used interchangeably with the term Function-as-a-Service (FaaS).

² Some serverless frameworks in industry [24] and research [112] use per-application containers, where the container typically scales at the granularity of the number of function instances it is currently serving. In this case, the container limits are dynamic in a stepwise fashion.

of a container orchestration systems, the aggregate resource limits are enforced by the framework.

3.2.1.2 Challenges in Serverless

Traditionally, serverless systems offer a scalability model that is simple to articulate: each invocation is given the same amount of resources to use. However, implementing this model can impose performance challenges, often due to spatial and temporal non-uniformity.

Cold Starts Cold starts — the time it takes to spawn a container and prepare it to run a function — add latency overhead compared to deployment options such as RPC servers [112]. This problem can be classed as an example of temporal non-uniform performance characteristics, because function latency can be greatly influenced by whether a container is available for reuse (e.g., a warm start) or whether there is a cold start. Cold starts also have a spatial aspect, as cloud deployments may also utilize regions, so whether a cold start occurs or not may depend on the region where a function is invoked. Techniques such as container reuse can reduce cold starts, at the cost to the cloud provider to keep containers alive when not in use. This is an active area of research [39, 69, 142, 201, 172].

Orchestration Tooling to orchestrate how and when functions are invoked is often needed to scale well. For instance, prioritizing completion of a chain of functions may be more effective than naively executing functions in the order they are invoked [214]. The order of invocation versus the order of execution can be viewed as a temporal scheduling characteristic. While there is some commercial support for serverless function orchestration in industry [22, 23], this is also an active area of research [214, 147, 76].

Application Structure and State In serverless, “serverful” considerations are no longer the responsibility of the application developer. However, the application developer is responsible for ensuring their application is tolerant to the runtime semantics provided by the serverless offering. Serverless frameworks often guarantee at-least-once execution semantics, which generally requires idempotent function code. Frameworks also do not guarantee the lifetime of particular container instances, so application state must be stored elsewhere in order to be persistent. These requirements

add complexity to serverless applications and require careful reasoning from developers.

In particular, state management in serverless not only adds complexity, but can incur a steep performance penalty. If application state is static, it can be packaged with a function. In this case, the temporal characteristic of accessing this state is dependent on whether an invocation is a cold start or not. However, if the application state is dynamic, it must be stored and fetched between serverless function containers and a storage service. The divide between where a function executes and where state is stored is a logical spatial divide. Since serverless systems run in the cloud, there is often no guarantee of physical locality between data services and container instances. The price of sending data over the network can be high compared to local accesses. Optimizing data transfer within the serverless ecosystem is also an active area of research [121, 182, 162], and this challenge is part of the motivation for freshen (Chapter 4).

In sum, the scaling mechanism of serverless, the runtime semantics, and the logical divide between instances and persistent state all introduce spatial and temporal characteristics to serverless systems.

3.2.2 Container Orchestration Frameworks

Container orchestration frameworks are used to run applications which are comprised of one or more containerized components. Examples of container orchestration frameworks include Kubernetes [126], Mesos [101], Borg [224], and more. While an application may be monolithic in form — a single process that runs in a single container — an application may also be composed of many pieces. In microservice architectures, for example, an application is decomposed into microservice components and information is exchanged between them using RPC-like requests. An advantage to the microservice architecture is each microservice may be scaled and replicated independently.

3.2.2.1 Scalability in Container Orchestration Services The tools and toggles used to control scalability in container orchestration services can be complex. At a high level, there

are two methods of scaling: scale up and scale out. Scale up (or vertical scaling) involves adding additional resources, i.e., making a container larger or smaller. Controlling scale up involves setting container and application limits as defined by a policy (e.g., max, min, and/or dynamic policies). Scale out (or horizontal scaling) involves adding additional components, i.e., additional containers that are replicas or shards. Scale out involves making decisions on the degree to which containers are added to the application.

Both scalability mechanisms have the potential to add spatial and temporal non-uniformity to a contained-based computing stack, either through the time it takes to enact a scale adjustment (i.e., an increase or decrease), how these operation map to physical resources, or due to coordination between components.

3.2.2.2 Challenges in Container Orchestration Services While there are many mechanisms to enable scaling in container orchestration services, there are drawbacks and limits to each.

Dynamism Generally a component of scale out, setting resource limits for applications is difficult as resource needs are often temporally dynamic. Too little, and applications are throttled or killed; too much, and cloud consumers pay for resources they do not use. The ability to even set these limits dynamically relies on cooperation from the operating system, the containers, and the container orchestration framework. The prevalence of under-utilized resources in the cloud is well documented [61, 80], indicating that this difficulty is pervasive. This challenge is part of the motivation for Escra (Chapter 5).

Inherited Limits Another component of scale out that can be problematic is that the maximum resource limits for containers is inherited from the operating system, and the operating system inherits those limits from hardware. An example of this inherited limitation is that a single Kubernetes pod cannot be larger than any of the worker nodes in the Kubernetes cluster. This not only sets a limit on the maximum pod size that is likely well below the aggregate resources of the cluster, but it can also lead to fragmentation problems as illustrated in Figure 3.1. This can be considered a hard spatial limit of the system. There is a variety of work on extending

this limitation through use of remote memory [44, 194, 66]; these solutions introduce a level of NUMA characteristics to the system. NUMA characteristics are described further in Section 3.3.2. Addressing scalability across components for both CPU and memory is an active area of research, and is part of the motivation for DiNOS (Chapter 6).

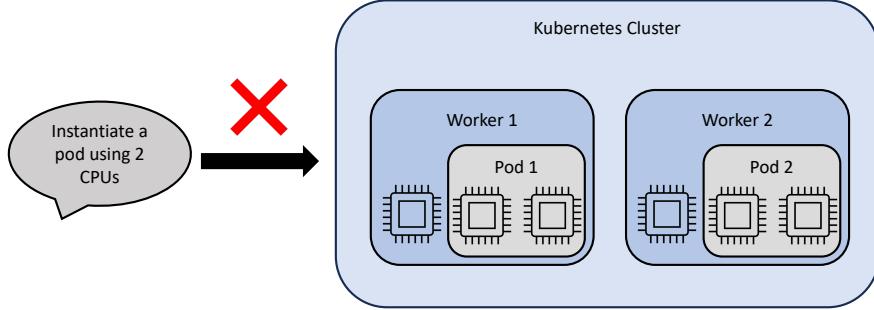


Figure 3.1: This figure illustrates a two node Kubernetes cluster that is unable to schedule the requested pod even though the aggregate resources in the cluster are sufficient to do so. Pod migration is insufficient to address resource fragmentation across hosts in this case.

Complexity and Performance Similar to the application structure overheads incurred in serverless, applications that use scale out often see increased complexity. This increased complexity can make it difficult to identify and fix performance problems, and also difficult to manage the cost of applications. For scale out in particular, separating components also involves data movement that may not be present in systems that use scale up. This data movement incurs performance overhead due to network latency and the cost of data serialization and deserialization. This was recently documented when Amazon Prime Video moved from a microservice to a monolithic architecture, citing cost, resilience, and the ability to scale as motivations for the change [123]. This type of performance overhead can be thought of as spatial-temporal overheads, as the spatial placement and timing of operations determines the extent of potential overheads.

3.3 Scalable Operating Systems and NUMA

Fundamentally, operating systems exist to fulfill the needs of applications by efficiently and performantly managing access to hardware. As application requirements and hardware capabilities have both grown in complexity — the layers on either side of operating systems in the computing stack — operating systems must adapt. Research advancing operating system design is vast.

However, within the context of spatial and temporal performance characteristics, there are two particularly relevant subdomains: operating system scalability and NUMA systems. This section presents background on the development of scalable operating systems (Section 3.3.1) and NUMA architectures (Section 3.3.2). Section 3.3.3 provides an overview of several challenges with regard to spatial and temporal performance characteristics.

3.3.1 Scalable Operating Systems

The creation of scalable operating systems have long been the focus of researchers. We divide this work into two broad categories: multiprocessor operating systems and distributed operating systems.

Multiprocessor Operating Systems A multiprocessor operating system is an operating system that targets a single host with more than one core; as time has passed, the numbers of cores a single host may contain has increased. The management of a many-core system requires that the bookkeeping done by the operating system be thread safe, performant, and scalable in order to provide these qualities to applications. As such, the arrangement of internal kernel state, and the algorithms for accessing and maintaining this state, are key to multiprocessor operating system design.

Different works have explored running multiple library operating systems on different cores of the same host [71, 114], avoiding contention between cores through care with data structures [36], and creating an object-oriented abstraction constructed to use the cache efficiently [74]. Barreelfish [30] presented the multikernel (or replicated kernel, an idea also used in Popcorn Linux [29]). Notably, this approach treats the multiprocessor as a distributed system. NrOS [33] balances message passing (from Barreelfish [30]) with replicated data structures [42] placed in shared memory. Disco [38], Cellular Disco [85], and Cerberus [205] use a hypervisor to run commodity VMs, creating a virtual cluster out of a single host. In Disco, Celluluar Disco, and Cerberus, VMs may run multiple processes but are limited in scale to avoid connection within the guest operating system [38, 85, 205].

Distributed Operating Systems A distributed operating system is an operating sys-

tem that manages the hardware of multiple components. Distributed operating system research first gained popularity in the 80s and 90s [182, 28, 212, 146] but despite the multitude of designs, widespread deployment of distributed operating systems did not occur. At the time, the performance cost of transparent distribution over networks was prohibitively high, especially compared to the growth of CPU speeds [199, 113].

Recently, advances in networking, distributed systems, resource disaggregation, and cloud computing have revived interest in distributed operating systems. Helios targets heterogeneous hardware using bespoke programs run on satellite-kernels [167]. LegoOS separates resources into compute, memory, and storage components managed by splitkernels, with the specific goal of supporting hardware resource disaggregation [203]. dReDBox focuses on racks with custom interconnects and optimization-based scheduling [118]. MIND allows transparent scaling across compute and memory blades [134]. A common trend among these systems is to provide a single system image across components, so that the distributed system appears as a single system to applications. Helios [167], LegoOS [203], fos [234], and MIND [134] all take this approach.

Another technique appropriated from distributed systems is to treat operating system functionality as services [113]. For instance, fos (factored operating system) was inspired by internet services and was designed for both multicore and cloud (distributed) deployment. With fos, operating system services may be replicated and distributed [234, 235]. DOSC also reduces kernel functionality using services but with the goal of enabling an object-oriented computing environment [117].

3.3.2 NUMA and Extended NUMA Systems

A non-uniform memory access (NUMA) system is defined by the characteristic that some memory regions are faster to access than other memory regions, depending on what component is the accessor. Groups of memory regions and accessors that are near each other (where *near* implies low latency accesses) are said to be in the same NUMA domain (or NUMA node). If the access is a far access (where *far* implies high latency access), it is referred to as a cross domain access.

In this way, NUMA systems are expressly spatial (through the arrangement of NUMA domains) with temporal characteristics (access time). The manager of a NUMA system – often the operating system – must juggle both these inherent characteristics and runtime considerations, such as thread migration and paging, which add additional temporal characteristics to NUMA systems.

NUMA architectures are now standard. As such, NUMA support is provided in mainstream operating systems (e.g., Linux, Windows, ESXi). There is a large body of research on both NUMA architectures themselves and how to manage them. This section provides details on both extended NUMA systems and techniques for managing NUMA systems.

Extended NUMA Systems The traditional view of a NUMA system is a single host with multiple sockets, with the CPUs of the system divided symmetrically between the sockets. However, other architectures exist which exhibit NUMA characteristics. We refer to these other architectures as extended NUMA architectures. Extended NUMA architectures include distributed shared memory systems (e.g., [134, 41, 32, 166, 17]), where memory from multiple regions may be shared across hosts. Extended NUMA systems also include remote or far memory systems. These systems have the goal of either expanding the memory of a host or using the memory of other hosts (e.g., [89, 12, 194, 245]), or sharing data between hosts (e.g., [66, 67, 41, 170]). Still other extended NUMA systems are formed when hosts use memory pools such as those provided by CXL [84, 141, 227], programmable switches [134], or other interconnects [118].

Key to using extended NUMA architectures is minimizing access latency. Works focused in this area include remote access via cache coherence protocols implemented at the hypervisor level [169], RDMA [66, 41], kernel bypass [244], and/or optimized messaging [169, 170]. Recent works explore using Compute eXpress Link (CXL) [58] to provide a shared memory pool [84, 141, 227]. There also exist devices that provide shared memory across heterogeneous components [51].

Managing NUMA Systems At the operating system level, NUMA-aware allocation and scheduling policies can be influenced through mechanisms such as pinning (i.e., developers or admins choose the NUMA nodes to run the application), affinity (i.e., threads try to first allocate local NUMA memory), and NUMA migration (i.e., the operating moves pages to the NUMA node

accessing them [53]). The cost (in latency) for a remote access can also be conveyed to operating system algorithms through the assignment of weights. This is useful for tuning a system with a deeper NUMA hierarchy or an asymmetric NUMA topology [54].

There are also tools that can be used by an application (including an operating system) to aid in managing state effectively in NUMA systems. As an example, node replication (NR) [42] is an algorithm providing replication of arbitrary sequential data structures using a shared-memory log. By replicating the data structure in multiple NUMA domains, NR is able to reduce the number of slower, cross-domain accesses required for operations on the data structure. A notable use-case for NR is the node replicated operating system (NrOS) [33], which uses NR to replicate key data structures (e.g., process page tables).

Systems using disaggregated memory can be considered a subset of extended NUMA architectures. A multitude of approaches exist for managing and accessing disaggregated memory, including: single address operating systems [98], distributed file systems [140], application runtimes [228, 43], specialized memory components often with a centralized manager [203, 245, 134], and global object store abstractions [132, 161, 34, 35]. Some of these works focus on pools of non-volatile memory (NVM) which are predicted to be both large and fast [34, 72].

3.3.3 Challenges

In this section, we outline three challenges at the intersection of operating systems and extended NUMA systems.

Scheduling and Placement Operating systems often control the spatial and temporal placement of tasks and data within a system. As such, scheduling and allocation is a crucial challenge for operating systems controlling extended NUMA systems. There are several notable works in this area regarding both operating systems and extended NUMA systems. Shinjuku is a specialized operating system targeted towards fast preemption to support microsecond scale applications [116]. RackSched presents a two-level scheduler that uses network co-design to extend microsecond scale scheduling to rackscale systems [246].

Schedulers are not only found in operating systems, but are also present in cluster management systems. Cluster managers such as Borg [224], Kubernetes [126], Mesos [101], and Docker Swarm [211] provide automatic placement, supervision, and restart of processes across the cluster according to user requirements.

One approach to scheduling is to encode the constraints and state as an optimization problem, through such technologies as declarative cluster management (DCM) [210]. This can be quite effective in managing complex scheduling policies, such as for the Kubernetes scheduler [210]. While cluster managers expose customizable and complex scheduling and allocation interfaces to the user, general purpose operating systems typically do not. Allowing a flexible interface for scheduling and allocation policies in a rackscale operating system is an area of emerging interest. For instance, dReDBox encoding constraints as an optimization problem; however, it does not discuss the policies implemented [118].

Dynamism The traditional view of NUMA systems tends to be both static and symmetric. While there are some exceptions to this (e.g., the enzian system targets a heterogeneous system [51], as does helios [167]), current techniques accepted for optimizing NUMA may not transfer to systems that are dynamic. Specifically, for extended NUMA systems, the spatial NUMA characteristics of the underlying systems might change over time. This challenge is articulated in more detail in Section 6.2, and is part of the motivation for LD-NR/DiNOS (Chapter 6).

Programming Interface and Abstractions General purpose operating systems typically offer familiar abstractions such as processes and threads that map well to the common view of host. However, extended NUMA systems with distributed components (rackscale, resource disaggregation, etc.) do not map as cleanly to the traditional host model. For instance, the process, thread, and memory allocation schemes in Linux were not constructed to allow a programmer to give fine-grained hints to the operating system about dependencies within a program that may be sensitive to NUMA and even more sensitive to extended NUMA. It is possible that abstractions developed for distributed frameworks such as Hadoop [19], Dryad [110, 241], Spark [242], and Ray [161] may be a better fit for operating systems targeting distributed extended NUMA systems.

It is also possible that new abstractions may allow programmers to better optimize applications running on extended NUMA architectures.

3.4 Neural Processing Units (NPUs)

Previous sections outline the current state and future of computation at large scales (e.g., cloud computing) and composite architectures (e.g., extended NUMA systems, especially rackscale). In contrast, this section provides background on and outlines challenges associated with a specific architectural component: the neural processing unit (NPU). Section 3.4.1 provides background on accelerators generally and places NPUs in the accelerator landscape. Section 3.4.2 outlines several challenges involved in programming NPUs.

3.4.1 Accelerators and NPUs

NPUs, sometimes also referred to as tensor processing units (TPUs) or data processing units (DPUs), belong to a class of components referred to as accelerators. Accelerators such as graphics processing units (GPUs), application-specific integrated circuits (ASICs), and NPUs are desirable because they often offer vast and/or specialized computational capabilities. For instance, the AMD XDNA™ 2 architecture provides up to 50 trillion operations per second (TOPS) [16]. Different accelerators offer different types and amounts of computational capabilities and power requirements. For example, on a series of machine learning benchmarks, AMD XDNA™ NPUs exhibited higher performance per watt compared to CPUs, with improvement ranging from 4.3-33× [192]. Accelerators are subsequently desirable when integrated into systems with constrained power or cost margins and when used to accelerate algorithms requiring computationally dense tasks suited to the architecture of the accelerator. For instance, AMD XDNA™ NPUs are used to accelerate Windows Copilot and also to accelerate image and video processing [192].

Within the wide category of accelerators, there is a subclass of accelerators referred to as coarse-grained reconfigurable architectures (CGRAs). While the precise taxonomy of CGRAs is an active area of research, CGRAs are more programmable than ASICs but less versatile than CPUs;

CGRAs also typically offer performance between an ASIC and a CPU [149]. NPUs are generally thought to fall within the category of CGRAs [136, 149].

3.4.2 Challenges for NPU Programming

There are several challenges associated with NPUs. First, the hardware design of NPUs is both opportunity and challenge. However, this section focuses on challenges revolving around *using* existing NPU hardware designs. While the goal of harnessing the capabilities of NPUs is easy to articulate, it is not always easy to achieve in practice [190]. These challenges arise directly from the spatial and temporal characteristics of NPUs, found most directly in the micro-architecture of the NPU itself [149].

Architectural Characteristics and Complexity NPUs have an explicitly spatial and temporal architecture. Spatially, computing elements are arranged into a two-dimensional grid [192, 149]. Temporally, the memory hierarchy is based on scratchpad regions, without hardware-supported coherence [192]. Together, these traits require that a design that is intended to run on an NPU (sometimes referred to as a *kernel* or *operator*) must include: 1) synchronization mechanisms to coordinate components, 2), dataflow specifications to ensure data is where it needs to be when it needs to be, and 3) runtime operations, to coordinate the programming of the device (e.g., the loading of program data, programming of the network on chip, etc.). As noted elsewhere, from both a hardware and software perspective, the characteristics of CGRAs and designs targeting CGRAs are fundamentally different than the sequential architectures of CPUs and sequential programming infrastructure [149]. It remains an open area of research to determine how and when these fundamental differences should affect how programmers view NPUs.

Programming Interfaces for NPUs One component of constructing a computing framework targeting NPUs is the programming interface. Due to the architectural characteristics and complexity of NPU architectures, the development of bespoke abstractions and programming constructs is a key part of this development. The previous challenge outlined the work it takes to create an NPU design; this challenge asks who should do the work, the programmer or the pro-

gramming framework. Generally, abstractions in programming interfaces can be high level (where the framework is responsible for many details) or low level (where the programmer is responsible for many details). The level of abstraction is a spectrum; it is an open research problem to determine which points on this spectrum are most useful.

High level programming interfaces abstract key architectural characteristics, and compilers supporting high level interfaces perform mapping and optimization tasks transparent to the programmer. This lowers complexity for programmers, but often requires the framework to do the heavy lifting of optimization. These automatic optimizations then tend to be focused on specific domains, such as machine learning. Examples of high level frameworks for machine learning pipelines targeting NPUs include AMD Ryzen™ AI Software [15], Intel® OpenVINO™ [175], and Qualcomm® Neural Processing SDK [187]. It is an open area of research to allow both fine-tuning of programs written using high level interfaces and generalize compiler optimizations across computing domains and architectures.

Low level, or close-to-metal, programming interfaces provide clarity about the target architecture. As a consequence, they tend to require programmer expertise and encourage designs that are specialized to an accelerator rather than portable across accelerators. IROn is an example of a close-to-metal programming interface targeting NPUs [151, 155, 158]. Chapter 7 focuses on improving efficiency, expressivity, and extensibility within this framework through extensions to the IROn programming interface. It is an open area of research to increase usability of close-to-metal programming interfaces without compromising the ability to finely-tune kernels.

Chapter 4

freshen: Proactive Serverless Function Resource Management

This chapter is a reprint, with minor edits, of our published paper in the Workshop on Serverless Computing 2020 (WoSC6): *Proactive Serverless Function Resource Management* [104].

This work presents a change to the serverless function life cycle abstraction to mitigate inefficiencies that arise due to temporal non-uniformity. Specifically, application state maintained by the execution environment of a serverless function may become stale over time; this work proposes a mechanism to keep this data fresh.

4.1 Introduction

Serverless computing is an emerging paradigm in which cloud providers seamlessly scale developer-provided functions as demands change. Although seemingly simple, serverless functions have been shown to support a wide variety of workloads, from chat bots, video processing, machine learning, HCI, to even general compute. As serverless ecosystems mature, functions will be integrated into a set of larger and larger microservices and will also be relied upon to directly interface with users. As such, the execution latency of serverless functions becomes an important consideration.

However, the simplicity of today’s serverless deployments may increase execution times. Consider a simple function, λ_1 , which downloads a machine learning model from a server, analyzes an input image, and performs additional processing before writing a result to a datastore. Without care, overheads exist. The function must first create a connection to the server hosting the model and then download the model from the server. This behavior could happen anew for subsequent

instantiations of λ_1 , even if running sequentially in the same warmed container. When writing the result, another connection must be established before the data is sent. Again, this overhead could reoccur for successive invocations of λ_1 . These per-invocation overheads (e.g., establishing connections, refetching the model, incurring TCP slow start, etc.) quickly add up, which is problematic because many functions have short execution times.

To deal with such issues, developers can utilize *runtime reuse*. In runtime reuse variables can be *runtime-scoped* inside the language runtime executing within the container the serverless function runs in.¹ Runtime-scoped variables can be accessed across subsequent serverless function instantiations within a given runtime and container. Revisiting our example, network connections can be reused within a runtime when defined as a runtime-scoped variable to avoid per-instantiation connection overheads.

We argue runtime reuse is insufficient to overcome many of the redundant overheads described earlier. Even with runtime reuse, fetched data could be out-of-date, connections may revert their congestion windows to small initial values or even time out, or application-level state could be stale from the last invocation. To combat these issues, we propose a new primitive called *freshen*, which can be *proactively* invoked by the serverless infrastructure. A *freshen* hook is implemented in the runtime, allowing developers or providers to establish or warm connections, proactively fetch data, or otherwise perform actions to reduce overheads when the serverless function runs. The *freshen* hook is designed to be run before its corresponding function is instantiated, and we contend this is possible because there are many opportunities to predict a function’s instantiation before it is invoked.

This paper provides motivation and background in Section 4.2, a preliminary design in Section 4.3, and potential benefits of *freshen* in Section 4.4. Related work is detailed in Section 4.5. Finally, Section 4.6 contains discussion and conclusion.

¹ We use “container” to generally refer to VMs or containers

4.2 Background and Motivation

This section provides background on runtime reuse and highlights scenarios where inefficiencies may remain. Then, we motivate ways to predict function instantiations.

Serverless runtime reuse While all providers allow runtime reuse, here we explain how an open-source platform, OpenWhisk, enables reuse. OpenWhisk runs functions within Docker containers, listening as a daemon on port 8080. After the container is initialized, the `init` hook starts the language runtime within the container and also loads the actual function code. When the `run` hook is invoked, the function is scheduled to run. Thus, the persistent runtime instantiated during `init` can be thought of as a program that listens for the `run` hook, executes the function, and returns the result.

Without runtime reuse, variables are scoped for use within a single invocation, termed *invocation-scoped*. In contrast, *runtime-scoped* variables can be reused across function instantiations in a given runtime. Common use cases for runtime-scoped variables are persisting network connections (so connection quotas are not exhausted) and fetching frequently-accessed data during the first function invocation and then storing in the runtime for the container’s lifetime.

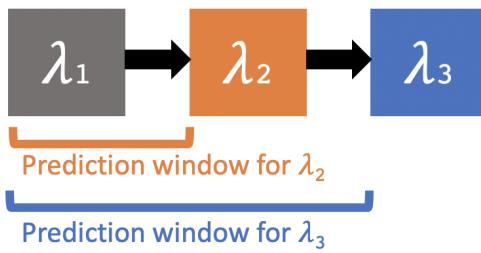


Figure 4.1: Opportunities for freshen within a function chain

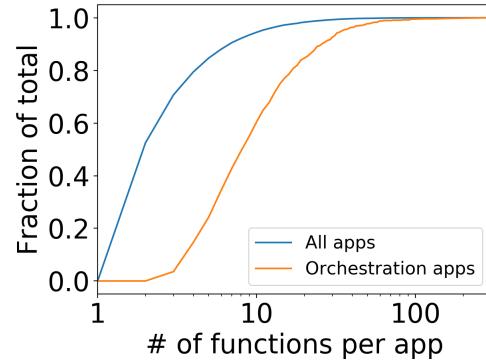


Figure 4.2: Orchestration apps have more functions in chains

Runtime reuse inefficiencies While runtime reuse can increase application efficiency, numerous issues may arise. First, the runtime may not be initialized, such as when a cold start occurs. Studies have shown inefficient container reuse across function invocations, which increases cold start frequency [214]. Other works indicate some serverless infrastructures disallow container

sharing between functions, which can increase cold starts when container resources are limited [229]. Second, there may be cases when the runtime is initialized, but data held within the runtime is stale. For example, an object stored within the runtime may need to be re-retrieved because a newer version is available. Network connections may have timed out or have reset their TCP state (e.g., congestion window, round trip times, etc.). Linux congestion control reduces the congestion window (CWND) on inactive connections. Last, approaches to reduce connection (re)establishment overheads may not apply. Linux `tcp_no_metrics_save` allows metrics like RTT and ssthresh to be cached between TCP connections to the same destination, but does not apply to important parameters such as CWND. TCP Fast Open requires sender/receiver support and limits data sent in initial handshakes to small amounts. As a result, even with runtime reuse several inefficiencies remain that can be addressed with proactive calls to `freshen`.

Regaining efficiency via prediction To alleviate the above concerns, we introduce a `freshen` hook into the runtime, which can be called before a function is run. The `freshen` hook allows arbitrary execution of code intended to speed up function execution times. `freshen` can warm pre-existing network connections, ensure locally-cached items are up-to-date, or even proactively retrieve an object. `freshen` is most effective when functions are predicted, and this is possible in several cases. First, in function chains (as in Figure 4.1) explicit knowledge of the chain could be used to predict impending function invocations. Function chains are often explicitly provided (as in orchestration frameworks like AWS Step Functions) or can be derived via tracing or service mesh techniques [153]. To better understand prediction opportunities, we briefly study function chains in orchestration frameworks. Figure 4.2 shows a CDF of the number of functions within a single serverless application for orchestration applications on Azure (data from [201]), compared to the number of functions within a single application over all applications. Orchestration frameworks specifically support function chains, and hence applications utilizing orchestration frameworks typically consist of more functions: 8 functions in the median orchestration case versus 2 functions in the median case of all. With a median function runtime of \sim 700ms [201], prediction opportunities could be as high as \sim 5.6s in the case of a linear chain (e.g., Figure 4.1).

Trigger Service	Delay (s)
Step Functions	0.064
Direct (Boto3)	0.060
SNS Pub/Sub	0.253
S3 bucket	1.282

Table 4.1: Trigger overhead

Additionally, functions within chains may be triggered by other services, such as storage, pub/sub, or direct invocations. Table 4.1 shows the median delay, over 20k runs, between invoking a function via the listed service and the actual subsequent triggered function start time in AWS. Cold starts are carefully avoided, and the methodology in [214] is used to obtain overheads by measuring timestamps just before the function trigger and at the start of the triggered-function. The table shows latencies range from 60ms to 1.28s, allowing time to call and execute freshen for the next function within the chain.

4.3 Design and Implementation

This section addresses when freshen could run (Section 4.3.1), what freshen could do (Section 4.3.2), and how freshen could be implemented (Section 4.3.3). Throughout, we refer to an example serverless function λ (Pseudocode 1) to illustrate how freshen could warm a connection and prefetch data. λ fetches data (**DataGet**) over a connection, performs some calculation on the fetched data and λ 's parameters, writes an output value to an external resource (**DataPut**), and returns whether the write was successful.

Pseudocode 1 Sample Serverless Function λ

```

1: Runtime Constants: CREDS, ID1, ID2
2: procedure  $\lambda$ (args)
3:   data := DataGet( CREDS, ID1 )
4:   ...
5:   result := ...
6:   ...
7:   ret := DataPut( CREDS, ID2, result )
8:   return ret

```

4.3.1 When to freshen

The serverless framework would attempt to run freshen before the serverless function (best case) or simultaneously (worst case). freshen could cause function execution to block until it is complete, or run synchronously with the function in a separate thread, as shown in Figure 4.3. Simultaneous execution could lead to race conditions and code complexity, but allows most aggressive resource warming; the feasibility of this approach is a subject of future work.

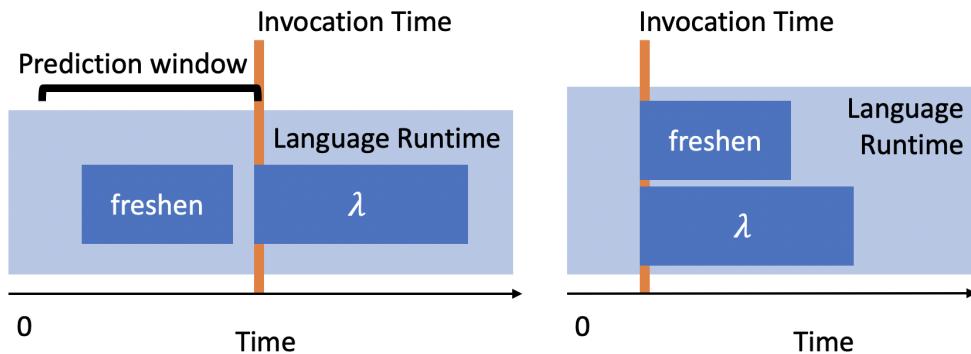


Figure 4.3: Predicted and unanticipated timing of freshen

4.3.2 Opportunities to freshen

freshen could perform a variety of actions, including TCP connection establishment, TCP connection warming, state maintenance of other connection-oriented protocols, and proactive data fetching.

Connection establishment and checks If a serverless function uses a resource with an underlying TCP connection, the function developer can either establish a runtime-scoped connection to take advantage of runtime reuse and create an ephemeral, invocation-scoped connection. In both cases, freshen could help reduce function latency. If the connection is runtime-scoped, freshen would send a TCP keepalive to ascertain connection liveness; if the connection is not alive, freshen could reestablish the connection. If the connection is invocation-scoped, freshen could proactively establish the connection before the function attempts to create it.

freshen could only perform connection establishment for connections with constant arguments (e.g., constant IP and port). We posit this is often the case as serverless functions typically interact

with known services such as storage.

Connection warming `freshen` could also take steps to warm TCP connections used by the serverless function such as setting the CWND. This could be facilitated via a new system call, `warm_cwnd`, which would determine an appropriate value of CWND based on current network conditions and anticipated workloads. The CWND can be estimated via techniques like packet pair probing to determine the current bandwidth [119] or analyzing the CWND of recent similar TCP connections to the same destination. Repetitive invocations can be used to anticipate workload characteristics, which could guide the warming function on whether warming is appropriate. The `warm_cwnd` function can set initial congestion windows or alter congestion windows on longer-running, inactive connections. Since `warm_cwnd` is implemented as a system call, final determination of actual CWND values, as well as permissions on whether such values can be altered, resides within the provider who is running the underlying host infrastructure.

Other connection-oriented protocols `freshen` can establish and warm other connection-oriented protocols and protocols that run on top of TCP such as TLS as long as the credentials are constant. However, for TLS establishment and other user-space protocols, the serverless provider would require some knowledge of the libraries used in order to create provider-generated `freshen` hooks for those resources. Developers who write their own `freshen` hooks, as detailed in Section 4.3.3, would have access to such knowledge.

Proactive data fetching Consider the λ in Pseudocode 1: if the data fetched with `DataGet` is retrieved using constant credentials and resource identifiers, it is possible to prefetch the data before λ is invoked.

Prefetching leads to the concept of a `freshen`-maintained cache of prefetched data. If the function is invoked frequently within the same runtime and accesses a read-only data resource, it may only be necessary to fetch the data once every n seconds instead of every time the function is run, reducing network traffic. The time-to-live (TTL) of values within the `freshen` cache could be set by a default value, by `freshen` configuration values specified by the function developer, or by modifying the `DataGet` library to configure the TTL value on a per-resource level. In the more

general case, associated timestamps or version numbers could be used to determine the freshness of items in the runtime freshen cache, and data could be updated the next time freshen or the serverless function is called.

4.3.3 Implementation

In the simplest implementation of freshen, the function developer would write freshen for each serverless function that requires optimization. This would provide the most opportunity for customized optimization. freshen may also improve code organization by encapsulating and standardizing maintenance of dynamic resources. As an interesting alternative to developers writing freshen, freshen code could be inferred by the serverless framework itself for common resources and for popular serverless languages (e.g., JavaScript, Python).

Code generation would be complex but here we rely on several observations about serverless functions and frameworks to reduce the scope of the problem:

- If freshen were unable to be inferred, the serverless framework could continue unmodified with no major performance loss. Hence, failure to infer is not fatal.
- Source code is available for static analysis for such tasks as identification of read-only data fetched using constant parameters.
- Function code is run repeatedly, so dynamic tracing of functions to identify commonly accessed resources is possible (similar to the tracing used in [100]).
- The latency cost of the network operations freshen seeks to optimize are much slower than CPU speeds so some overhead for freshen inference is permissible.
- Implementing inference only for libraries used to access other cloud services offered by the serverless provider has the potential to lower latency for a majority of functions without having to infer freshen behavior for unknown resources.

One option for implementing *freshen* for scripting languages is to use added runtime-scoped state and dynamically-inserted wrapper functions. The purpose of the runtime-scoped state is to track and coordinate *freshen* resources between the *freshen* call and the actual function invocation. The purpose of the dynamically-inserted wrappers is to intercept access to freshened resources. We will illustrate a simplified example of what an inferred *freshen* could resemble for the λ in Pseudocode 1.

The runtime-scoped state would minimally be a collection of ordered *freshen* resources. A *freshen resource* is any object or resource that the *freshen* code may interact with, such as a socket or a data object. In our example, the *freshen* resources are kept in an ordered runtime-scoped list called *fr_state*. In Pseudocode 1, the **DataGet** operation which *freshen* can fetch or prefetch, will be assigned index 0 since it is the first resource accessed by λ . **DataPut**, which *freshen* can warm, is assigned index 1. Each entry in *fr_state* could contain a variety of metadata, such as a *state* (e.g., *running*, *finished*, etc.), a *result* (e.g., the prefetched data), a *TTL* for the result, and a *timestamp* recording the last time that entry was freshened. For simplicity, we only consider *state* and *result* in the following pseudocode.

Pseudocode 2 Freshen Function for λ

```

1: Runtime State: fr_state
2: procedure Freshen
3:   fr_state[0] := running
4:   fr_state[0].result := DataGet( CREDs, ID1 )
5:   fr_state[0] := finished
6:   fr_state[1] := running
7:   DataPut.warm( CREDs )
8:   fr_state[1] := finished
9:   return

```

Pseudocode 2 illustrates an example *freshen* function for λ . As mentioned, **DataGet** is assigned to index 0 and **DataPut** is assigned to index 1. The states *running* and *finished* surround the **DataPut** and **DataGet** calls of *freshen*, and are used to coordinate the execution of *freshen* with the execution of λ . Pseudocode 3 is the annotated version of Pseudocode 1. The function wrappers appear at lines 3 and 7. The function wrappers used are **FrFetch** (for *freshen fetch*) and **FrWarm**

(for *freshen warm*).

Pseudocode 3 Annotated Sample Serverless Function

```

1: Runtime Constants: CREDS, ID1, ID2
2: procedure  $\lambda(\text{args})$ 
3:   data := FrFetch( 0, DataGet( CREDS, ID1 ) )
4:   ...
5:   result := ...
6:   ...
7:   ret := FrWarm( 1, DataPut( CREDS, ID2, result ) )
8:   return ret
```

Psuedocode 4 and 5 are the implementations of those wrappers. The main function of each wrapper is to synchronize *freshen* actions with λ 's use of that resource. If the resource has already been freshened, the wrapper returns either the prefetched data (line 4 in Pseudocode 4) or nothing where *freshen*'s only job is to warm the resource (line 4 in Pseudocode 5). In Pseudocode 5 it is assumed that there is already some knowledge of how to warm **DataPut** (e.g., the call to **DataPut.warm()** in line 7 of Pseudocode 2). If *freshen* has started freshening the resource (indicated by the state **running**), both wrapper functions wait for the *freshen* thread to finish before returning (line 6 in Pseudocode 4 and line 6 in Pseudocode 5). Finally, if *freshen* either did not run or is executing slower than λ , the wrapper can perform the *freshen* action itself (line 10 in Pseudocode 4 and line 10 in Pseudocode 5). Not included for brevity in Pseudocode 2 are the checks to see if the resources have already been freshened by wrapper functions invoked by λ .

Pseudocode 4 Freshen Fetch Function

```

1: Runtime List: fr_state
2: procedure FrFetch(id, code)
3:   if fr_state[id] == finished then
4:     return fr_state[id].result
5:   else if fr_state[id] == running then
6:     FrWait( id )
7:     return fr_state[id].result
8:   else
9:     fr_state[id] = running
10:    fr_state[id].result = Execute( code )
11:    fr_state[id] = finished
12:    return fr_state[id].result
```

Pseudocode 5 Freshen Warm Function

```

1: Runtime List: fr_state
2: procedure FrWarm(id, resource)
3:   if fr_state[id] == finished then
4:     return
5:   else if fr_state[id] == running then
6:     FrWait( id )
7:     return
8:   else
9:     fr_state[id] = running
10:    resource.warm()
11:    fr_state[id] = finished
12:   return

```

Billing and accounting Since freshen runs to benefit the serverless application, the serverless application owner should pay for it. However, as outlined above, freshen would ideally be triggered based on predictions by the serverless framework. What happens if the platform mispredicts a function call? Confidence in prediction could be used to dictate if freshen is called or not. Metrics kept inside a container, or communicated to the serverless global scheduling entity, could be used to stop freshen from running if predictions have been too inaccurate. Service categories chosen by the application developer could also control freshen behavior. Aggressive freshen invocation would be appropriate for latency-sensitive applications; freshen could be disabled for latency-insensitive functions. Last, we note providers may be incentivized to offer freshen because it provides a method to monetize warmed containers that are otherwise sitting idle.

Preventing abuse and misconfiguration A danger if the application developer were allowed to implement their own freshen is that the application developer would try to implement their entire function in freshen. This is undesirable and unprofitable for the developer for several reasons: freshen has no access to function arguments, the application developer is paying for the compute and network resources regardless, and the application would have to handle spurious invocations (mispredictions) gracefully.

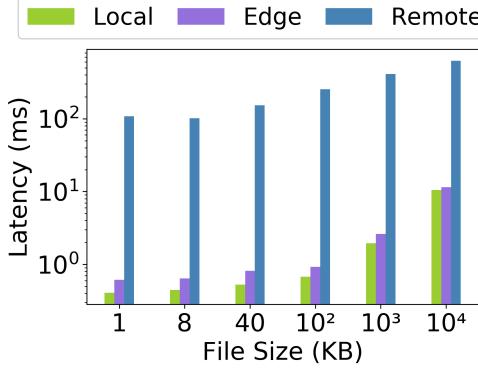


Figure 4.4: File retrieval overheads to save with freshen

4.4 Evaluation

This section explores the advantages a freshen hook could provide. First, the benefits of file caching are evaluated. Then, improvements from connection warming are illustrated.

File caching evaluation Figure 4.4 demonstrates the potential benefits of proactive file retrieval (file caching). In this benchmark, an OpenWhisk serverless function queries a server for a file of one of six different sizes (x-axis) over a TCP connection. The time measured (y-axis, log scale) is the duration from connection to when the file has been completely received. The file server is located in one of three locations: local on-host (green), edge on-site (purple), and remote off-site (blue). On-site resides on the same 10 Gbps LAN and off-site averages 50ms away. The experiment was conducted using CloudLab [70] with 20 iterations. The results show how much execution time freshen could save a serverless function if freshen is proactively run. Maximum benefits range from 11-622ms.

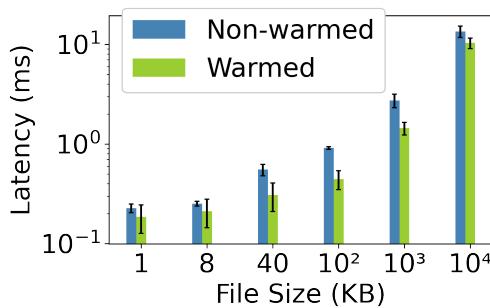


Figure 4.5: Warming to cloud

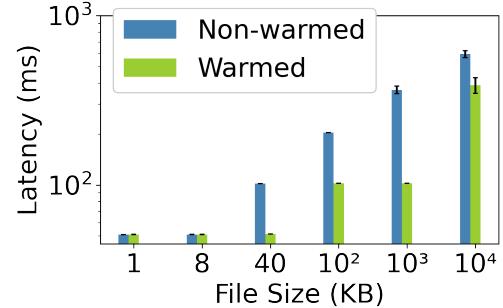


Figure 4.6: Warming to edge

Warmed connection comparison To demonstrate the benefits of freshen warming a TCP connection, we run an OpenWhisk serverless function on CloudLab which sends different file sizes to a server. We measure the time of a client initiating a file transfer to the response from the server indicating completion. To understand the potential benefits, we emulate a warmed TCP connection by sending a large file before sending our desired file size. The server is located at two locations: on the same cloud or at the edge (~ 50 ms away). The experiment was conducted over 20 iterations. The cloud case is presented in Figure 4.5 and the edge case is presented in Figure 4.6. With smaller file sizes, the performance of warmed and non-warmed is similar. As file sizes grow, the benefit of warmed connection ranges from 51.22% to 71.94%. The edge performance is better because network delay, and not system overheads, dominate totals.

4.5 Related Work

Much research reduces cold start costs. These works are partitioned into two categories: those that are compatible with existing serverless architectures and those that propose significant changes to serverless architecture. Of those compatible with existing serverless architectures, techniques include cold start avoidance (runtime reuse), light-weight isolation mechanisms [172], intelligent host scheduling [209], and caching of resources ranging from libraries [172] to virtual Ethernet infrastructure [160]. Our work has a different focus, optimizing warm starts, but is compatible with these techniques. Catalyster [68] snapshots static application state; our work addresses dynamic state and is complementary. AWS Lambda Extensions address static and dynamic resources, but do not provide opportunities for prediction [236]. Works that focus on avoiding cold starts by predicting function execution [201, 90, 209] motivate our design because freshen would be most effective when function invocations are predicted. Of works that propose fundamental changes to serverless architecture such as running more than one function within the same isolation context [7] or adding distributed application state and/or message passing abilities between serverless functions [7, 206], the motivation for freshen remains but implementation strategies would vary.

Last, Containerless [100] avoids the cost of strong isolation mechanisms by transforming

JavaScript serverless functions into Rust via dynamic tracing. Their dynamic tracing design, as well as analysis of the resulting traces, could help inform how `freshen` could be inferred by providers.

4.6 Discussion and Conclusion

Discussion There exists many opportunities for future work. First, the system should be fully deployed and thoroughly evaluated. Quantifying how `freshen` affects variability in application behavior would be an important component of this evaluation. Prediction success must be additionally quantified, especially in the case of non-deterministic function chains. In addition, the framework must be analyzed for misuse and resource limiting [120] and hardened as necessary. Impact on developer burden and the extent to which providers can automatically generate `freshen` must also be further studied. Integration with microservices or other primitives [8] is interesting future work. Finally, integrating `freshen` into serverless architectures that provide different isolation scopes is an additional area for future study (e.g., Azure offers chain-level isolation).

Conclusion This paper proposes a new primitive to serverless language runtimes called `freshen`. With `freshen`, developers or service providers specify functionality to complete before a given function executes. This proactive framework allows for overheads associated with serverless functions to be mitigated at execution time, which improves function responsiveness. We argue predictive opportunities exist to enable `freshen` to be run with ample time. A high-level design and implementation are presented, along with preliminary results to show potential benefits of the scheme.

Chapter 5

Escra: Event-driven, Sub-second Container Resource Allocation

This chapter is a reprint, with minor edits, of our published paper in 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS): *Escra: Event-driven, Sub-second Container Resource Allocation* [57]

This work enables dynamic, fine-grained, sub-second adjustments to container resource limits.

Allowing the container to flexibly adjust over time helps to mitigate temporal non-uniformity, for instance, through eliminating unnecessary throttles that may occur when an application needs resources above the current container limit.

5.1 Introduction

Containerized infrastructure is quickly becoming a preferred method of deploying applications. The light-weight nature of containers coupled with rich orchestration systems enable a new way to design automated operations that are integrated with development workflows. In these deployments, per-container resources limits are used to prevent interference between containers and unchecked resource usage.

Setting container resource limits is a trade-off between application performance and efficient use of underlying system resources. When resource limits are set low to prioritize efficient resource use, applications will experience an increased number of CPU throttles and out-of-memory (OOM) events. Throttles slow processing and OOMs kill containers; both result in degraded application performance. When resource limits are set high to prioritize application performance, resources are underutilized which increases deployment cost [56, 44]. Developers pay the cost when cloud

providers charge tenants based on resources reserved [196, 13, 25]. Cloud providers pay the cost in cases where developers are charged by usage, such as in serverless computing [20, 24, 49, 107].

Due to this trade-off, setting accurate limits is important. In practice, it is also difficult [11, 185, 196, 61, 173].¹ Using profiling to characterize application resource requirements will only result in accurate estimates if there is a representative workload. As workloads are often dynamic, the resources needed will change over long timescales (diurnal patterns, gradual changes in application popularity, etc.) and short timescales (bursts, failures of coupled systems, etc.). Since creating an accurate estimate of resource requirements is so complex, developers and operators often resort to over-provisioning resources. This results in underutilized deployments, a trend often observed by datacenter operators [220, 152, 91, 61, 80].

Recent work has addressed some of these challenges by leveraging machine learning to predict future needs and then automatically scaling container resource limits based on those predictions [196, 185]. These works eliminate the developer burden of setting resource limits but are constrained to using coarse-grained intervals (e.g., several minutes) to set resource limits. Coarse-grained intervals are required because the system has to learn enough information to be able to predict resource use. This is a poor fit for some workloads with short-lived containers, such as in serverless systems [201, 21, 103, 50]. Coarse-grained intervals also increase the odds of misprediction since the dynamics of applications can change throughout an interval. Thus, these works still contend with the performance and efficiency trade-off.

In this paper, we argue the performance and efficiency trade-off can be avoided by using a *fine-grained, event-based resource allocation* scheme. To this end, we introduce *Escra*: a fine-grained, event-based resource allocation infrastructure for single containers and distributed resource allocation capable of managing resources of multiple containers across multiple nodes. We find resource allocation can easily adapt to sub-second intervals within and across hosts, allowing datacenter operators to cost-effectively scale and assign resources without performance penalty. This scheme

¹ The aggregate CPU utilization at Twitter is <20% but the reservations reach up to 80%. Memory utilization is only slightly better at 40-50% but the reservations still greatly exceed the usage [61].

has numerous benefits. Instead of a container being killed when it reaches an OOM event, an *event-based* system can catch the event and scale the container dynamically. Instead of making conservative allocations in order to avoid performance degradation over coarse-grained time intervals, a *fine-grained* system can always aim to right-fit allocations to current resource demands and can quickly react to instances of CPU throttling.

Escra consists of a logically centralized controller that administers resource allocations to containers across servers. Each server is instrumented with kernel hooks and runs an agent process that applies resource decisions and reports container usage to the controller. A *Distributed Container* abstraction enforces resource isolation by enforcing per-application resource limits, similar to resource quotas found in other container orchestration systems [127, 174, 126, 88]. In these systems, resource quotas are enforced at the admission control stage. However, unlike resource quotas, a Distributed Container enforces resource limits both at deployment and throughout the lifetime of a container, allowing containers belonging to the same tenant to share compute resources across hosts on the order of milliseconds. Runtime limit enforcement enables Escra to fully utilize the per-application limit even when some containers are using less than their initial deployment allocation. The contributions of our work are as follows:

- We expose fine-grained telemetry data from Linux’s Completely Fair Scheduler (CFS) [223]. This allows Escra to quickly track and react to actual resource needs, resulting in both high performance (low latency and high throughput) *and* low cost (minimal slack²).
- We implement event-based memory scaling and periodic memory reclamation. Escra uses memory scaling to increase container memory upon an OOM event, rather than allowing the container to be killed. Periodic memory reclamation increases application memory efficiency.
- We show Escra is effective by comparing slack, latency, and throughput performance to recently proposed systems. We reduce application latency by up to 96% while increasing

² Slack: a container’s CPU or memory limit minus its CPU or memory usage

throughput up to 3.2x over a state of the art container orchestrator. These low latency and high throughput rates are achieved while simultaneously reducing the median CPU and memory slack by over 10x and 2.5x, respectively. We show the overhead from the central controller is minimal.

- We show Escra reduces slack and both CPU and memory reservations in serverless applications without increasing application latency, potentially reducing cost to both the developer and the infrastructure provider.

5.2 Related Work

Current container orchestration systems (Kubernetes [125], Borg [224], Mesos [101]) set static container resource allocations. Here we present recent works that instead dynamically scale containers and discuss the limitations of these systems.

Vertical Pod Autoscaler (VPA) VPA is a Kubernetes project that implements automated container scaling through a threshold-based scaling mechanism [88]. VPA sets a target resource utilization and an upper and lower bound on that utilization. When the container usage hits the upper threshold, VPA scales the container up. When the lower bound is hit, VPA scales the container down. VPA also has the capability to enforce per-application limits via resource quotas [127]. A resource quota is a hard resource limit on the aggregate compute usage across all or a subset of deployments or services in a Kubernetes namespace.

Limitations of VPA VPA sets the upper and lower limit scaling bounds far apart. Since scaling a container requires a container restart, VPA only scales a container at most once per minute. The loose scaling-bound limit and infrequent container scaling results in high slack which translates to decreased cost-efficiency.

Autopilot Autopilot is a proprietary Google project that addresses the low cost-efficiency of static container deployments [196]. Autopilot runs a control loop that collects both per-second and five minute aggregated usage data from each container, analyzes it, and then makes a prediction

on whether or not a container needs to be scaled. Autopilot uses machine learning predictions to scale container limits as frequently as every five minutes.

Limitations of Autopilot While Autopilot provides an automated mechanism to set limits, it does so at coarse-granularity which causes cost-efficiency and performance issues for two reasons. First, Autopilot’s heavy-weight algorithm and periodic control loop prevent it from quickly responding to changes in workloads. As a result, resource predictions are forced to at least match the maximum predicted usage over the next allocation period (Autopilot uses a default 5-minute period). This leads to unnecessary slack. Second, because Autopilot relies solely on prediction, it is unable to correct inaccurate predictions even when resources are available. Inaccurate predictions can cause unnecessary OOMs and CPU throttles.

Firm Firm also uses machine learning to improve containerized application performance and cost-efficiency [185]. While Firm does attempt to minimize CPU reservations, the primary objective of Firm is to reduce service-level objective (SLO) violations. Firm minimizes SLO violations by intelligently multiplexing compute resources to optimize the critical path of an application. Firm is similar to Autopilot because it does not require a pod restart to scale container CPU resources and can update container limits automatically.

Limitations of Firm Firm does not implement seamless or automatic *memory* scaling, requiring users to set static limits. Firm shares the limitations of Autopilot regarding performance and cost-efficiency issues as both frameworks feature a coarse-grained, ML-based feedback loop.

5.3 Introducing Escra

Escra is a container resource allocation system that achieves high performance, cost-efficiency, and strong isolation. Escra automatically scales containers in a fine-grained manner, while providing strong isolation via a new abstraction called a Distributed Container. A Distributed Container allows containers belonging to the same tenant to dynamically share resources across multiple compute nodes while capping the overall aggregate resource usage for a given application or tenant at runtime.

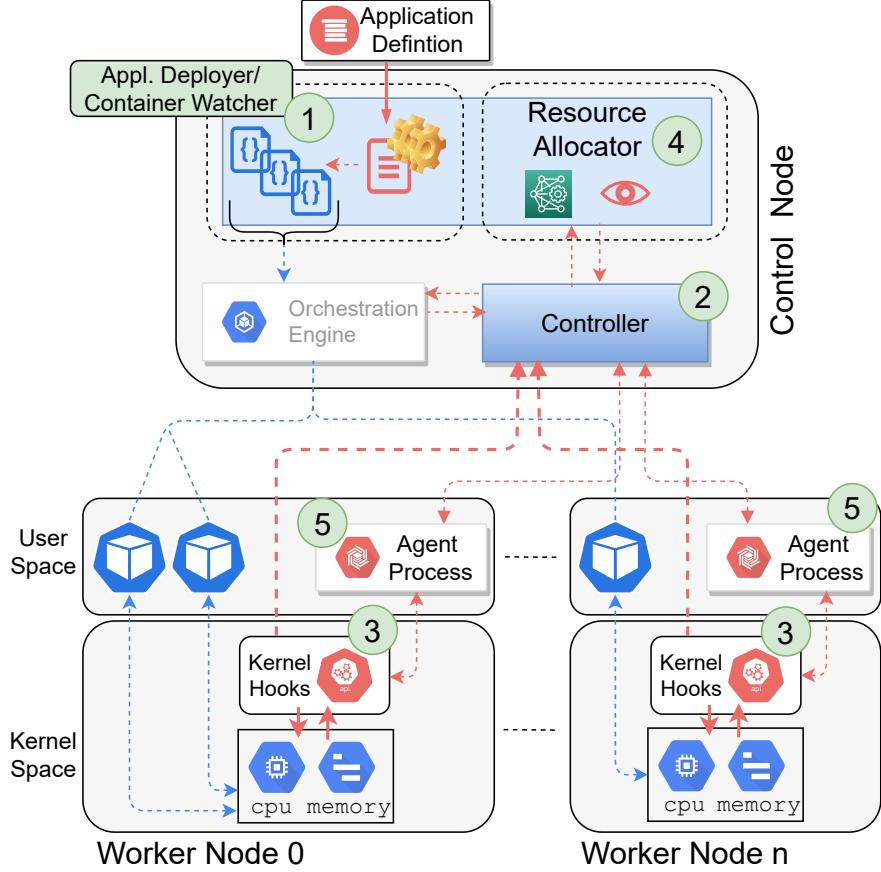


Figure 5.1: Escra Architecture. A single control node manages and controls a set of containers distributed across multiple worker nodes.

Figure 5.1 shows a high-level view of the four key components in the Escra architecture. The Application Deployer and Container Watcher (①) take a set of YAML files describing a set of Kubernetes deployments, services, and containers. The Application Deployer interfaces with the Kubernetes API to deploy containers. The Container Watcher monitors Escra containers and enables newly deployed containers to start streaming fine-grained telemetry to the Controller. The logically centralized Controller (②) handles the unique, fine-grained telemetry sent from the kernel via kernel hooks on workers (③). These kernel hooks obtain fine-grained scheduler data that is not available in user-space. A centralized controller model can be capable of scaling, as evidenced by production systems for datacenters [226] and geo-distributed network services [216]. The Resource Allocator (④) ingests telemetry from the Controller and makes per-container resource allocation decisions. Finally, similar to Kubernetes's per-node kubelet [125], an Agent is run on each host

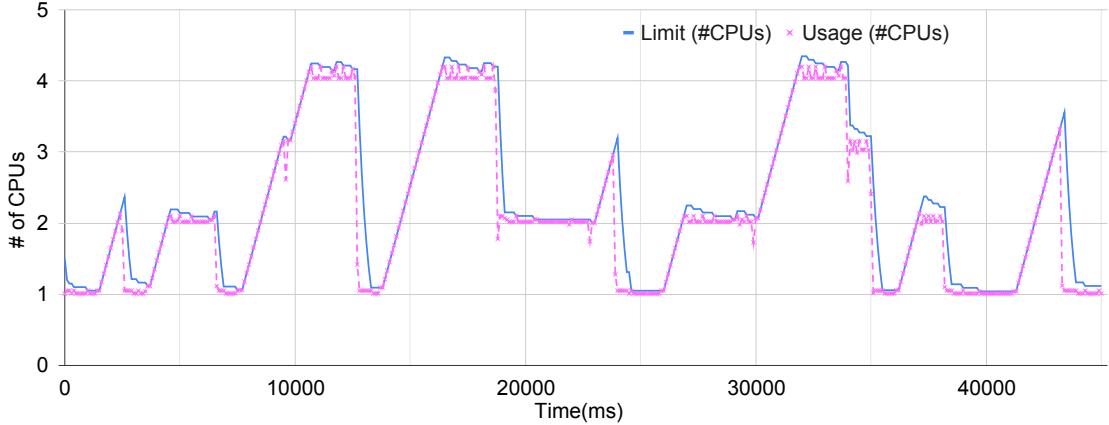


Figure 5.2: Escra’s CPU tracking ability under a dynamic workload

(⑤). The Agent handles resource updates sent from the Controller and can dynamically scale both CPU and memory container limits without restart on the order of 100s of microseconds. In this section, we describe Escra’s unique ability to make scaling decisions on a fine-grained timescale and in an event driven manner. A complete description of Escra’s architecture follows in Section 5.4.

To illustrate the benefits of fine-grained container resource allocation, we deployed and loaded a container with sysbench [124], saturating 1-4 CPUs at any one time. The trace of the application execution with Escra is shown in Figure 5.2. Escra tracks the exact resource needs on a rapid time-scale by reacting to container throttles and OOM events and adjusting resources based on information collected during each CPU scheduling period and at OOM events. The implication of this fine-grained right-sizing is that Escra (1) significantly reduces slack and (2) simultaneously improves performance as applications are being allocated the resources they need rather than being throttled or killed due to OOMs. The remainder of this section provides further insights into how Escra achieves fine-grained resource allocation.

Per-period CPU Telemetry and Dynamic Reallocation Fine-grained telemetry data is required to minimize slack via fine-grained resource allocation. Our initial analysis of systems that aggregate CPU and memory data (cAdvisor [40], Prometheus [183], Kubectl [125], etc.) found they suffer from inefficiencies stemming from reliance on coarse-grained timescales. Allocating resources quickly is not useful if allocations are based on usage data that is stale or aggregated at insufficient levels. Our goal is to obtain near-instant usage information so Escra never operates on stale data.

In order to obtain fine-grained CPU data, Escra uses kernel hooks into Linux’s Completely Fair Scheduler (CFS). Upon deployment of each container, the Agent process creates a kernel socket for the container to use to report its metrics to the Controller. To implement fine-grained telemetry, containers report their per-period runtime statistics to the Controller at the end of each period. The telemetry data consists of the cgroup ID of the container, whether the container was throttled in the last period, and the amount of unused runtime in that period.

The Resource Allocator ingests raw container metrics from the Controller and uses two windowed statistics to track unused runtime and the number of throttles. The Resource Allocator uses these statistics to update per-container limits as often as every 100ms. The goal is to proactively update limits in order to keep the container limits just above container usage at all times. We update container CPU quotas using RPCs to the Agent process running on the host of the container, similar to [185].

Reactive Memory Reclamation and Reallocation upon OOM Events Escra monitors container memory usage and can seamlessly scale memory limits via two custom system calls that hook into Linux’s memory cgroup structure.³ One unique opportunity of fine-grained allocation is the ability to react to OOM events. To achieve this, a kernel hook is added in Linux’s memory allocation function, `try_charge()`, to catch a container after it exceeds its memory limit and right before it gets OOMed. This hook combats inaccurate predictions within autoscalers. For example, VPA [88] and Autopilot [196] scale containers at most once a minute and once every five minutes, respectively. There is a chance a container could OOM between allocation decisions. Our kernel hook allows a container to request more memory from the Controller before the container is killed. While this is a reactive mechanism for memory scaling, the request lookup penalty is orders of magnitude faster than a container crash and restart.

One beneficial aspect of this OOM-preventing kernel event is that the Resource Allocator can determine how to allocate additional memory resources depending on the state of the node and the application. If there is available memory on the node, the Allocator can simply scale the

³ Docker supports seamless container scaling [65], but Kubernetes does not.

needy container up. If the node is under memory pressure, the Controller can launch an aggressive memory reclamation process that reclaims memory from other containers on the node with high slack. Not only will this free up memory for the container in need, but it also increases node utilization, reduces slack, and improves cost-efficiency.

Proactive Periodic Memory Reclamation In order to reduce memory slack, the Escra Controller periodically contacts the Escra Agent on each worker node, asking the Agent to reduce the memory limits of each container on the same node as the Agent. The Agent checks the usage and the limit of each container it manages. If the limit of a container exceeds the usage of the container by more than δ bytes, then the Agent shrinks the container memory limit such that the memory limit minus the memory usage equals δ bytes. Each Agent then reports back the total reclaimed memory from its containers to the Escra Controller. The Resource Allocator can then give the reclaimed memory to other containers experiencing memory pressure.

5.4 Escra Architecture

This section describes the architecture of Escra, our container orchestrator built with Kubernetes, that implements (i) automated container limit settings, (ii) seamless container scaling, (iii) fine-grained resource allocation, and (iv) dynamic, per-tenant resource sharing and collective resource limits enforced at runtime. Escra implements these features using fine-grained telemetry, event-based memory scaling, aggregated application-wide resource limits, and a centralized Controller and Resource Allocator.

5.4.1 Application Deployer & Container Watcher

The Application Deployer ingests a Distributed Container configuration as a set of YAML files (Figure 5.1, ①) describing a set of containers, and maximum CPU and memory limits. The maximum CPU and memory limits represent the limit on the aggregate usage of all containers in the application (Figure 5.3, ②). Prior to deploying the containers via Kubernetes, the Deployer sends the global application limits to the Controller. This informs the Resource Allocator (Figure 5.1,

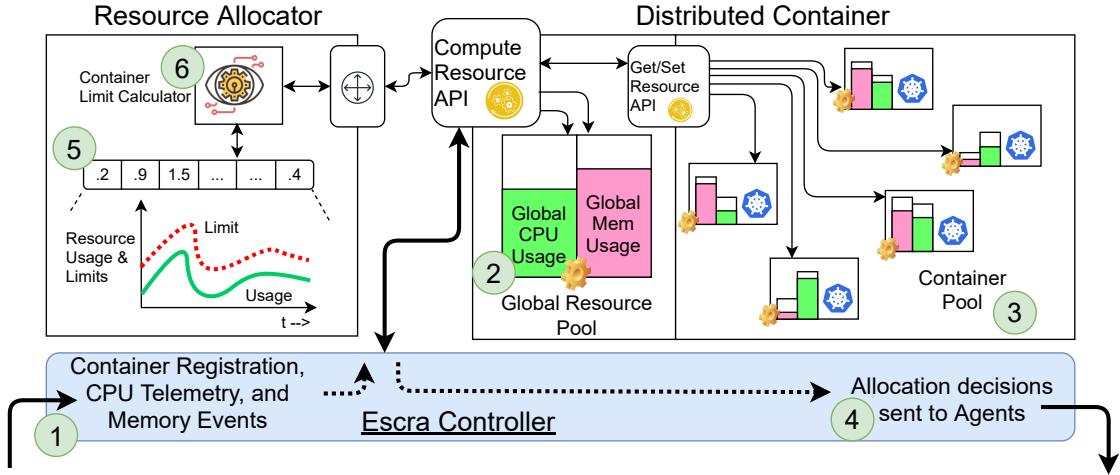


Figure 5.3: Escra Controller, Resource Allocator, and Distributed Container

④) of the total maximum usage of the containers in the deployment. Once the Deployer sends the application limits to the Controller, the Controller is ready to accept network connections from each container.

Initial limits are set to bootstrap containers when they first deploy but these limits will be changed by the Controller at runtime. The Deployer initializes the CPU and memory limit of each container to:

$$\frac{\text{global_cpu_limit}}{\# \text{ containers}} \quad (5.1) \quad \frac{\text{global_mem_limit} * \sigma}{\# \text{ containers}} \quad (5.2)$$

where σ is a configurable parameter representing the percentage of the global application memory limit to be withheld for containers that experience OOM events.

The Container Watcher integrates with Kubernetes to detect container creation. Upon detection, the Watcher notifies the Agent (Figure 5.1, ⑤) located on the same host as the newly created container.

5.4.2 Kernel Hooks

Escra uses kernel hooks to enable fine-grained telemetry and trap OOMs. After an Agent is notified that a new container has deployed, the Agent invokes a custom syscall that carries out three tasks, each implemented via kernel hooks (Figure 5.1, ③). First, the syscall creates a TCP

kernel socket to message the Controller (Figure 5.1, ②) and informs the Controller of the existence of the container. The per-container TCP kernel socket will persist for the life of the container. Once the Controller registers the new container, it updates the container’s CPU and memory limit based on the global application limits and current application resource use.

Next, the syscall modifies the container’s underlying Linux CPU and memory cgroup structures to enable fine-grained telemetry and event handling. For CPU, the syscall hooks into Linux’s Completely Fair Scheduler to extract runtime data to stream to the Controller. At the end of each period, the hook writes the container’s cgroup quota, unused runtime (the `runtime` variable in the CFS Bandwidth kernel structure), and whether the container was throttled in the last period into a shared FIFO buffer in the kernel.⁴

After the hook finishes writing data to the buffer, the runtime of the cgroup is refilled and the next period begins. Per-container kernel threads consume statistics from the FIFO queue and send the queued CPU statistics over UDP to the Controller. Along with the container quota and remaining runtime, the CPU statistic message also includes a tag letting the Controller know what container the incoming statistic refers to. The hook will report statistics once per-period for the life of the container.

To handle OOM events, the syscall adds a kernel hook in the memory cgroup structure (`mem_cgroup`) for the container. If a container exceeds its memory limit, before it is killed this kernel hook forwards the OOM event to the Controller over the existing TCP kernel socket that was previously used during container initialization. If memory is available in the global application pool, the container can increase its memory limit and continue running.

5.4.3 Controller

The Controller brings all of the system components together and coordinates their interactions. Figure 5.3 shows a more detailed view of the Controller, Resource Allocator, and the

⁴ Note that per-period unused runtime is not available in userspace and while one could interpret similar data from the `cpuacct` cgroup subsystem, `cpuacct` was never designed for accuracy and was initially designed as a way to showcase the capabilities of cgroups [37].

Distributed Container abstraction.

When containers register themselves with the Controller upon deployment, the Controller creates a logical container object and adds it to a pool of the other Escra containers within the application (Figure 5.3, ②). The logical pool of Escra containers is used to maintain an updated view and status (resource usage, limit, etc.) of the containers it is managing.

Once all containers are deployed and registered with the Controller, the Controller becomes responsible for several additional tasks. The Controller is responsible for launching a periodic memory reclamation process, handling fine-grained telemetry data from all containers, and handling memory requests from containers under memory pressure (Figure 5.3, ①). The Controller is also responsible for carrying out allocation decisions made by the Resource Allocator (Figure 5.3, ④). The Controller is *not* responsible for making those CPU and memory allocation decisions.

The Controller launches a periodic reclamation loop on behalf of the Resource Allocator that triggers each Agent to reclaim excess reserved but unused memory from each container in the cluster. The Resource Allocator determines to what extent each container’s memory is resized. Every 5 seconds, the Controller sends a request to each Escra Agent, requesting the Agent to reduce the memory limit of each Escra container, $C(i)$, and send back the amount the container was resized by ψ . This resized value is the amount of memory reclaimed from that specific container. The reclaim process is as follows. The Agent reduces the memory limit on a container if:

$$C(i)_l > C(i)_u + \delta$$

where $C(i)_l$ and $C(i)_u$ are the memory limit and usage of the container, respectively, and δ is a tunable parameter managed and set by the Resource Allocator that represents the memory reclamation “safe margin.” If the condition above is satisfied, the container limit is updated via: $C(i)'_l \leftarrow C(i)_u + \delta$; otherwise, the container limit is left unchanged. We empirically set the safe margin to 50 MiB. The amount of reclaimed memory is measured as:

$$\psi \leftarrow C(i)_l - C(i)'_l$$

where $C(i)'_l$ is the resized container limit and ψ is the amount of reclaimed memory. Therefore,

for each container that is resized, the Agent passes back to the Controller ψ bytes of memory. The Escra Controller forwards ψ bytes to the Resource Allocator which then adds ψ bytes of memory into the global memory pool via: $global_mem_limit \leftarrow global_mem_limit + \psi$. Note that the Controller passes all CPU telemetry data, memory requests, and reclaimed memory updates to the Resource Allocator.

5.4.4 Resource Allocator

The Resource Allocator is the lightweight decision-making component that determines the containers whose resources should be allocated to or reclaimed from. The Resource Allocator is composed of three key components. First, it has a global resource pool for both CPU and memory. For CPU and memory, it keeps track of the maximum application limit (Figure 5.3, ②), the total allocated resources, and the total unallocated (or available) resources (Figure 5.3, ⑥). Second, the Resource Allocator collects fine-grained CPU telemetry data from the Controller and uses a lightweight algorithm to make decisions on whether or not to scale up or scale down individual container CPU limits (Figure 5.3, ⑤). Third, the Resource Allocator consumes *out-of-memory* events sent from the Controller and, based on the globally available memory, increases the memory limit of memory-pressured containers.

If a container is not using up to its allocated resource limit, the Resource Allocator will trigger the Controller to take away those excess resources. However, the Allocator is designed to quickly identify when resources need to be given back to containers and will instruct the Controller to update container limits as needed.

Dynamic CPU Allocation The CPU allocation algorithm consumes CPU telemetry data sent from each container across all nodes in order to share CPU allocations across nodes and remain under the maximum CPU limit (Ω_l). At the end of the container running period t , the Resource Allocator consumes a runtime statistic from the Controller. The runtime statistic for a container i during period t ($C(i)[t]$) includes the container quota ($C(i)_q[t]$) in ms, the amount of unused runtime ($C(i)_q[t] - C(i)_u[t]$) in ms, and whether the container was throttled ($C(i)_{th}[t]$) in the last

period t .

The Resource Allocator uses two sliding windowed statistics that track (i) the excess runtime a container has at the end of each period and (ii) if a container was throttled during the last period. Based on these windowed statistics, the Resource Allocator determines whether a container needs or has excess CPU runtime and updates container quotas. A container quota (or limit) during period t is increased if $C(i)_{th}[t] = 1$ and will be increased for the following period $t + 1$ via:

$$C(i)_q[t + 1] = C(i)_q[t] + \frac{\sum_{t=0}^n C(i)_{th}[t]}{n} * \Upsilon(\Omega_l - \sum_{i=0}^{\lambda} C(i)_q[t])$$

where $\frac{\sum_{t=0}^n C(i)_{th}[t]}{n}$ is the windowed statistic measuring the average number of throttles over the last n container periods, $\sum_{i=0}^{\lambda} C(i)_q[t]$ is the unallocated CPU runtime for the entire application, λ is the number of containers in the application, and Υ is a tunable parameter that affects the rate at which a container CPU quota is scaled.

A container quota during period t is decreased if $C(i)_q[t] - C(i)_u[t] > \gamma$, where γ is a tunable parameter that adjusts when container quotas should be scaled down. A container quota for period $t + 1$ is scaled down via:

$$C(i)_q[t + 1] = C(i)_q[t] - \kappa \frac{\sum_{t=0}^n (C(i)_q[t] - C(i)_u[t])}{n}$$

where $\frac{\sum_{t=0}^n (C(i)_q[t] - C(i)_u[t])}{n}$ is the windowed statistic measuring the average runtime remaining during the last n container periods, and κ is a tunable parameter that affects the rate at which container quotas are scaled down. We empirically found that systems with high variance in CPU usage between periods performed better with a larger Υ and a smaller γ and κ .

Dynamic Memory Allocation This section details the Resource Allocator algorithm for handling *out-of-memory* events received from containers and ensuring the proper sharing of memory resources across an application. The Resource Allocator determines the amount of additional memory to allocate to containers under memory pressure and the amount of memory to reclaim from containers with unused memory.

The Resource Allocator consumes *out-of-memory* events that are sent from a container just before the container is killed for exceeding its memory limit. Upon receiving an *out-of-memory* event from a container $C(i)$, the Resource Allocator checks if there is unallocated memory available in the global resource pool. If there is no available memory (all global memory has been allocated to containers), the Allocator tells the Controller to reclaim unused memory from other containers in the application (described in Section 5.4.3). We implement *out-of-memory* events in Escra this way to avoid killing a container for exceeding its memory limit when available memory in the application exists.

If the Controller is able to reclaim memory from other containers in the application, the Resource Allocator will allocate a fixed number pages of memory to $C(i)$ by invoking the Agent to update the memory limit of $C(i)$. If the Allocator is unable to reclaim any memory from other containers, $C(i)$ is killed by the operating system (as is standard).

Integrating Escra With Serverless Frameworks The fine-grained approach to resource allocation in Escra is well suited to serverless environments due to the high degree of multi-tenancy in serverless systems as well as the short-lived nature of serverless functions. Since functions have short execution times (90% execute in under 1 minute [201]), coarse-grained resource management solutions are insufficient for serverless workloads. Since Escra is fine-grained and designed for use with containers, it is compatible with serverless frameworks that use containers to isolate serverless functions.

We choose OpenWhisk [18], an open-source serverless platform, as an example to illustrate how Escra may be integrated with serverless frameworks. In our configuration, OpenWhisk is deployed via Kubernetes and serverless functions (termed *user actions*) are run in pods. Each pod is deployed as part of the Kubernetes `openwhisk` namespace. Treating OpenWhisk as a single application, one can use the `openwhisk` namespace and invoker `containerPool` memory limit to set global application memory in Escra. We modified pod affinity to ensure OpenWhisk infrastructure was deployed on dedicated infrastructure nodes so there would be no resource contention between OpenWhisk components and user actions. While there is no global invoker CPU limit in

OpenWhisk, one can set memory and CPU to scale linearly, which indirectly sets a global CPU limit. Escra does not delay container creation in OpenWhisk because the connection between a container and the Controller does not block the container from beginning to execute. Escra already interfaces with Kubernetes so no further modifications are needed for a minimal integration that allows all user action pods to benefit from resource sharing and reclamation.

5.5 Implementation

Escra implementation consists of a total of 14.1k SLOC. The Controller and Resource Allocator are written in C++ and utilize gRPC to communicate with the Deployer, Watcher, and Agents (all written in Go). The Deployer sits on top of Kubernetes and integrates with the Kubernetes deployer API via client-go [48] to deploy Escra containers. Docker is used as the underlying container runtime. The Container Watcher integrates with the Kubernetes work-queue API and communicates with the Agent via gRPC as well.

Escra worker nodes run a custom Linux kernel based on Linux kernel 4.20.16. The custom kernel includes a hook in the CFS cgroup subsystem and in the memory management subsystem. The kernel also includes a custom message structure used for CPU telemetry reporting and memory requests to the Controller. The rest of the kernel modifications include approximately 1,500 SLOC spread across six kernel modules that implement limit resizing and CPU telemetry.

5.6 Evaluation

The goal of Escra is to automatically and seamlessly achieve high performance, cost-efficiency, and isolation. As fine-grained allocation is a key capability of Escra, the first goal of our evaluation is to show how much Escra’s highly reactive decision making process is able to improve both performance and cost-efficiency in comparison to common practice (static allocation) and a state-of-the-art system (Autopilot). Our second goal is to show how Escra can reduce the overall reservation requirements for serverless applications, while maintaining application performance; this has the potential to reduce cost for both the application owner and the infrastructure provider.

5.6.1 Experimental Setup

Experiment clusters are created using Cloudlab [70] resources consisting of a control node and worker nodes. Along with the default Kubernetes components, the control node runs the Escra Deployer, Watcher, Controller, and Resource Allocator. Each worker node runs an instance of the Escra Agent.

Microservice Benchmark Applications We first evaluate Escra on a set of four microservice applications running across three worker nodes and one control node. Each node consists of two Intel Xeon Silver 4114 10-core 2.20 GHz CPUs, 192GB of ECC DDR4-2666 memory, and a dual-port Intel X520-DA2 10Gb NIC. We set κ to 0.8, γ to 0.2, and Υ to 20 in the Resource Allocator for all experiments unless otherwise stated.

The microservice applications represent a set of four interactive, real-world benchmarks: (1) *MediaMicroservice* [75] (32 containers): a microservice similar to IMDB [108] where users can search, review, rate, and add films, (2) *HipsterShop* [102] (11 containers): an online shopping microservice consisting of standard browsing and purchasing of various items, (3) *TrainTicket* [222] (68 containers): a microservice that simulates a train ticket booking service consisting of searching, booking, modifying tickets, and (4) *Teastore* [215] (7 containers): a simulated online tea store where users can browse and purchase hundreds of various teas.

For each microservice experiment we load the microservice with one of four workload distributions: a fixed request rate, an exponentially distributed request rate, a bursting request rate, and an Alibaba datacenter trace [10]. The Fixed workload sends requests at a constant 400 requests per second. The Exponential (Exp) workload sends requests in an exponential distribution with $\lambda = 300$. The Burst workload sends a fixed 50 req/sec. with an additional 10 second exponential burst of requests where $\lambda = 600$ every 20 seconds. Finally, the Alibaba workload is sped up by 10x and sends requests at rates anywhere from 56-548 req/sec.

Evaluation Metrics Below is a list of metrics used in this section (derived from [196]):

- **Absolute Slack:** The container CPU or memory limit minus the container CPU or mem-

ory usage.

- **Application Throughput:** Measured in successful requests per sec.
- **Application 99.9th Percentile Latency:** Measured as the 99.9th percentile end-to-end latency.

Autopilot Implementation Autopilot [196] is not open-source so we implemented a recreation of the Autopilot ML recommender to compare against Escra. The Autopilot ML recommender is inspired by a multi-armed bandit problem in which an agent tries to use the best set of arms to maximize the total reward gain over time. Some parameters used in the Autopilot algorithm are manually tuned by their engineers (w_o , w_u , etc.). As they did not specify what values they used for these parameters, we tuned them to values that resulted in the best performance.

Note that Autopilot defaults to updating container limits every 5 minutes. We tested the update period of Autopilot at 60, 30, 10, and 1 seconds and saw finer-grained update periods achieve better performance. The throughput of HipsterShop with Autopilot at 1, 10, 30, and 60 second update periods degrades from 422 req/sec. to 382 req/sec. to 279 req/sec. to 108 req/sec., respectively. While we do not know how practical it is to run Autopilot at that granularity at scale, we show comparisons against 1 second intervals as a best case for Autopilot.

5.6.2 Performance - Cost-Efficiency Trade-off

Intuitively, there exists a resource allocation trade-off between performance and cost-efficiency. One can allocate a large amount of resources to eliminate any possible performance penalty (measured in throughput and latency), but this leads to poor cost-efficiency (measured in terms of slack). In contrast, one can significantly under-allocate resources and improve the cost-efficiency, but this is at the price of reduced performance. We further examine this trade-off in the context of both common practice (static allocation) and state-of-the-art (Autopilot), and illustrate that Escra achieves better performance and cost-efficiency than each system, and that the other systems compromised on one of the metrics.

First, we estimated the resources needed for the MediaMicroservice from the Deathstar Benchmark [75] by profiling each container and measuring maximum CPU and memory usage. We then ran the application in underutilized (limits set at 0.75x the profiled max), best-estimate (set at 1.0x), and safe buffer (set at 1.5x) cases. For each case, we measure the end-to-end performance (latency and throughput) and slack (CPU cores allocated minus cores used, and MiBs allocated minus MiBs used). As expected, performance increased (i.e., latency decreased and throughput increased) with more resources allocated; however, slack (resource wastage) also increased. We find the 1.5x allocation level illustrates a sufficient buffer and use that setting for evaluating the trade-offs in comparison to Autopilot and Escra.

For this evaluation, we deployed each microservice and used the workload generation-based benchmarking tool wrk2 [217] with the four different workloads. Each application is evaluated when managed by 1.5x static limits, Autopilot, and Escra. This setup allows us to measure both latency and throughput, to quantify the performance in each approach, and slack, to quantify the cost-efficiency of each approach. Figure 5.4 shows the resulting *change* in latency and *change* in throughput between Autopilot and Escra and between static limits and Escra for all four applications and workload distributions. Table 5.1 summarizes our results and is broken down in the subsequent sub-sections.

App Comp.	Avg. Δ Latency	Avg. Δ Tput.	Avg. Δ 50% CPU Slack	Avg. Δ 99% CPU Slack	Avg. Δ 50% Mem. Slack	Avg. Δ 99% Mem. Slack
Static vs. Escra	38.0%	25.4%	81.3%	74.2%	55.0%	95.9%
Autopilot vs. Escra	36.1%	54.5%	78.3%	78.6%	26.7%	68.9%

Table 5.1: Average performance increase and average slack reduction for both CPU and memory between static and Escra and between Autopilot and Escra. Escra improves performance, while significantly reducing slack

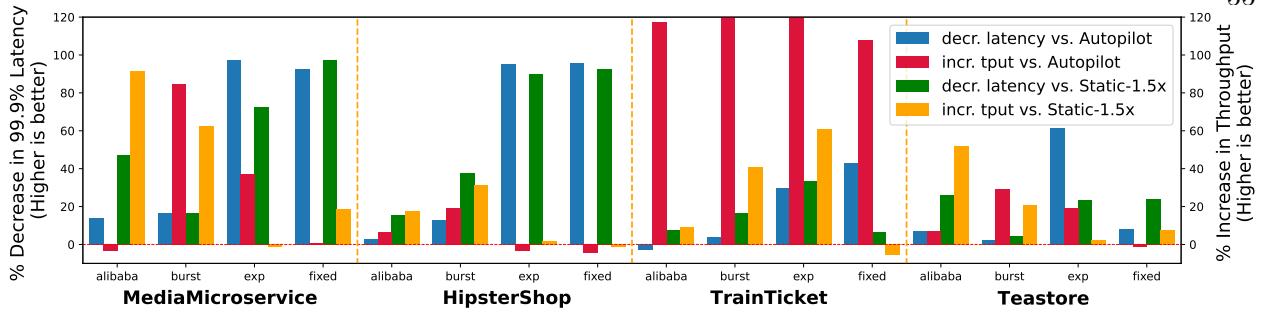


Figure 5.4: Change in 99.9% latency and throughput between Autopilot, the 1.5x measured peak static allocation and Escra. Note: TrainTicket with Burst and Exp workloads experienced a throughput increase of 134% and 324% respectively but are cut off at the top of the figure

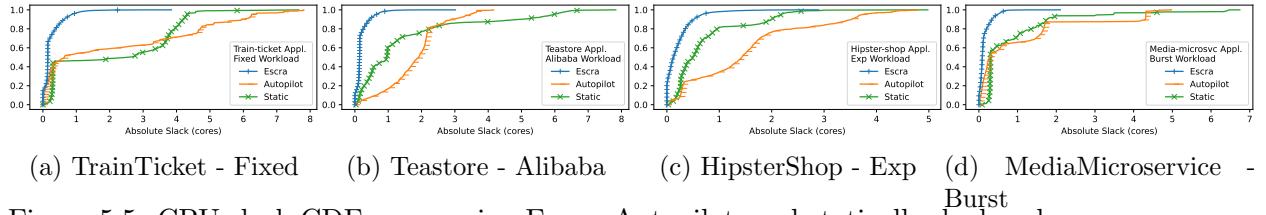


Figure 5.5: CPU slack CDFs comparing Escra, Autopilot, and statically deployed resources across the MediaMicroservice, HipsterShop, TrainTicket, and Teastore microservices with various work-

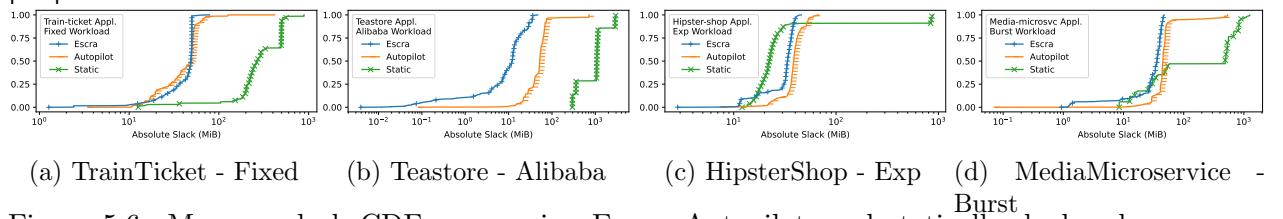


Figure 5.6: Memory slack CDFs comparing Escra, Autopilot, and statically deployed resources across the MediaMicroservice, HipsterShop, TrainTicket, and Teastore microservices with various workloads. The x-axis is log scale

5.6.3 Static Allocation vs. Escra

We first look at the change in both latency and throughput between a statically allocated application and an application deployed with Escra. Table 5.1 show that on average, Escra decreases latency by 38% and increases throughput by 25.4% compared to statically allocated applications. Escra can achieve these performance numbers with an average 50th and 99th percentile CPU slack improvement of 81.3% and 74.2%, respectively. Escra also decreases 50th and 99th percentile memory slack by 55% and 95.9%, respectively.

In an ideal world, we would not see a performance improvement from Escra over a statically deployed application allocated 1.5 times the peak measured resource usage; the static deployment would never experience any throttles or OOMs. However, this result is a testament to how difficult

it is for developers to set resource limits on containers [11, 185, 196, 61, 173]. Not only is it hard to profile containers, since you never know what the workload rate is truly going to be, but also the tools to measure resource usages (especially for CPU) tend to aggregate over seconds to minutes, smoothing out usage spikes [40, 183, 180].

The other reason for the performance difference between Static Allocation and Escra is from the fact that Escra can dynamically share and shift resources between containers at runtime. For example, in a static deployment, when a container is underutilized (C_u) and another container is getting throttled (C_t), C_t cannot use any of C_u 's resources. However, in Escra C_u is scaled down while C_t is scaled up (without exceeding the per-application global limit). Escra's ability to shift resources among containers and enforce a per-application limit at runtime enables an application to fully utilize its allocated CPU and memory. This is a Distributed Container's main difference to Resource Quotas [127, 174]. Resource Quotas are only enforced at container deploy time, so in the case above, C_t cannot scale up because C_u is already deployed and the global limits were enforced on deployment. In the case of VPA [88] (discussed in Section 5.3), the autoscaler would have to constantly kill and restart containers as CPU usages changed.

We break down TrainTicket with Fixed and Teastore with Alibaba experiments in the following paragraphs to help illustrate the ability of Escra to achieve both high performance and cost efficiency.

TrainTicket with Fixed Workload Figure 5.4 shows that TrainTicket with Fixed performs slightly worse with Escra than with static allocation, seeing a 5.5% decrease in throughout. Examining the slack in Figures 5.5a and 5.6a, 50% of the time, the static allocation has over 2.5 cores of CPU slack and 256MiB of memory slack. In contrast, Escra has a 50% CPU slack of 0.14 cores (a 17.9x improvement) and memory slack of 49MiB. This experiment shows the trade-off the static deployment makes, sacrificing significant cost-efficiency for a slight performance increase.

Teastore with Alibaba Workload Escra improves latency and throughput of Teastore by 25.7% and 51.6%, respectively. Figures 5.5b and 5.6b show while Escra is able to increase performance, it can do so while reducing 50th and 99th percentile CPU slack by over 81% and 74%

respectively, while also significantly reducing memory slack.

5.6.3.1 Autopilot vs. Escra

Autopilot aims to reduce slack without sacrificing performance using ML. However, Table 5.1 shows on average, Escra decreases latency by 36.1% and increases throughput by 54.5% compared to Autopilot. Table 5.1 also shows Escra's average 50th and 99th percentile CPU slack improvement over Autopilot is 78.3% and 78.6%, respectively. Escra also decreases 50th and 99th percentile memory slack by 26.7% and 68.9%, respectively. We further examine the results of two of these experiments below to determine how Escra can achieve both high performance and high cost efficiency.

HipsterShop with Exp Workload

In a few cases, Autopilot gets some performance improvements over Escra since it trades for performance gains at the cost of slack. Autopilot increases the throughput of HipsterShop compared to Escra by 3.16%. However, Figures 5.5c and 5.6c show Autopilot over allocates resources, with the median slack greater than 1.43 cores and 20% of allocations over 2.38 cores. For Escra, the median slack is 0.12 cores (an 11.6x decrease) with an 80th percentile CPU slack of 0.35 cores.

MediaMicroservice with Burst Workload

Figure 5.4 shows Autopilot degrades MediaMicroservice with Burst throughput and increases its latency. This indicates that Autopilot fails to quickly react to rapid and significant changes in CPU workloads and memory usages, resulting in low slack but higher latency and lower throughput. For the same application and workload, Escra is able to not only increase latency and throughput performance by 16.6% and 84.3%, but also able to reduce slack over Autopilot. Escra has a 99th percentile slack less than 66% of a core and a 99th percentile memory slack of 46MiB.

5.6.4 Takeaways

Table 5.1, Figure 5.4, and the four cases above show Escra rarely performs worse than static allocation and Autopilot, but when it does, the performance degradation is small and the slack savings are significant. When Escra outperforms the static allocation and Autopilot, Escra does so

with significantly reduced slack, proving that Escra is able to achieve both high performance and high cost efficiency. One of the key reasons for the high performance Escra is that Escra is able to greatly reduce OOMs. In all 32 experiments, Escra experienced zero OOMs, while Autopilot had up to 8 OOMs in a single experiment.

5.6.5 Serverless

This section shows how Escra integrates with OpenWhisk [18] by benchmarking two applications: ImageProcess and GridSearch. We run ImageProcess with one control node, three worker nodes, and two nodes reserved for serverless infrastructure (i.e., OpenWhisk and a data store). The GridSearch application runs with one additional worker node. Each node is composed of two Xeon E5-2650v2 8-core 2.6 Ghz CPUs, 64GB of DDR-3 memory, and a dual-port Intel X520 10Gb NIC. For both applications OpenWhisk is configured to create each user action pod with 1 vCPU for CPU request and limit, and 256 MiB of memory. We set κ to 0.8 and γ to 0.2 for both applications and Υ to 35 for ImageProcess and 20 for GridSearch in the Resource Allocator.

Serverless Benchmark Applications ImageProcess is a single-function application inspired by the image processing application in [240]. The function reads an image from a database, processes image metadata, creates a thumbnail, and writes the thumbnail to the database. Our workload is simple: an ImageProcess request is sent every 0.8 seconds over 10 minutes. We perform four iterations of the experiment for a total of 3k invocations for each test case. At the beginning of each experiment, we ensure there are no ImageProcess pods running (to ensure initial cold starts).

GridSearch is a traditional approach for tuning hyperparameters in classifiers. This batch-like application [106] uses \sim 115 serverless function pods to classify an Amazon product review dataset using scikit-learn [200] and tunes the classifier hyperparameters using the GridSearch algorithm. Each function is charged with completing tasks until all 960 tasks are completed. GridSearch uses the Lithops framework [145] for orchestration. We set the Lithops serverless backend to OpenWhisk and the Lithop storage backed to Redis.

The reason Υ is set to different values for GridSearch verses ImageProcess is due to the

differences in workload characteristics. In GridSearch, each user action is relatively long-lived as each action is a worker that will complete as many tasks as possible. Thus, it was performant to give Υ for GridSearch the same value used for microservices. In ImageProcess, a user action is a short-lived request. As such, container reuse is common and containers may experience periods of idleness between user actions. Increasing Υ allows containers to more quickly be granted the resources they need as they are created and as they transition from idle (unused) to used (running a user action).

Evaluation Metrics Below are the metrics used in the evaluation of the serverless benchmarks:

- **Aggregate Limits:** Since it is common in serverless systems to bill based on total usage, and serverless providers have a strong incentive to pack as many functions as possible per server, instead of CPU/memory usage per pod we focus on the aggregate of container CPU and memory limits.
- **Application Latency:** Application is measured in end-to-end latency per request (ImageProcess) or job (GridSearch).

5.6.6 OpenWhisk vs. Escra + OpenWhisk

Performance We first consider ImageProcess performance for OpenWhisk alone and OpenWhisk + Escra. Figure 5.7a shows that, up to the 80th percentile, OpenWhisk + Escra sees modest performance gains over OpenWhisk alone while the overall 99th percentile latency remains similar for both. The average invocation latency with OpenWhisk + Escra is 1.99 seconds as opposed to 2.12 seconds with OpenWhisk alone. Unlike other applications tested with Escra, ImageProcess requires Escra to handle a variable number of pods as the number of application pods at the start of each benchmark iteration is zero. The similarity in tail latency between OpenWhisk alone and OpenWhisk + Escra indicates that Escra is capable of supporting the dynamic scale-up of application pods needed in serverless environments.

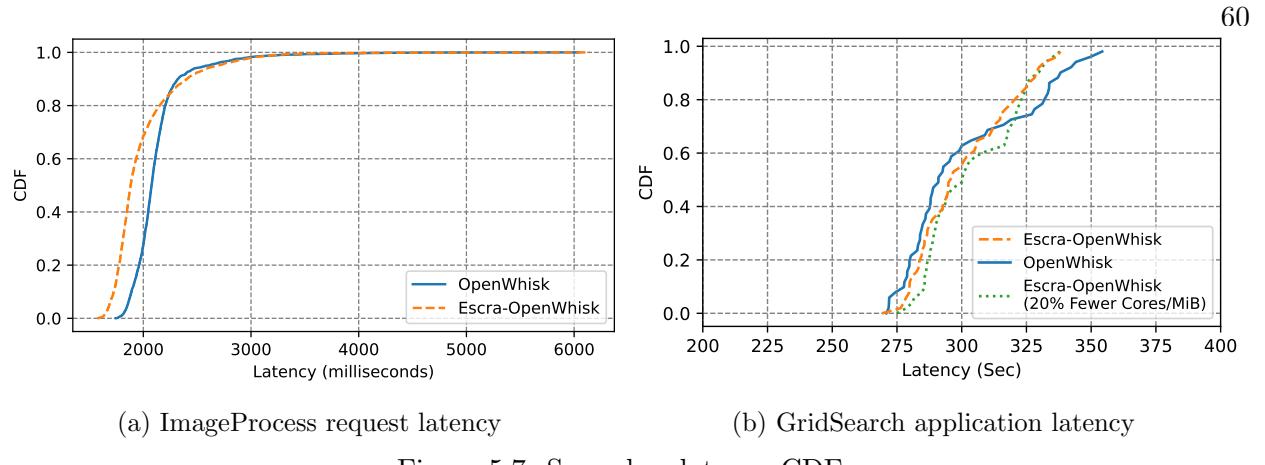


Figure 5.7: Serverless latency CDFs

To obtain a CDF of GridSearch application latency, we ran GridSearch on: (1) OpenWhisk alone, (2) OpenWhisk + Escra with the same amount of resources allocated as in the OpenWhisk alone experiment, and (3) OpenWhisk + Escra with 80% of the application resource limits allocated compared to OpenWhisk alone. We ran the application 50 times for each configuration. Interestingly, we observe the same average latency (~ 300 seconds) when we run GridSearch by allocating equal resources to OpenWhisk and Escra + OpenWhisk (cases 1 and 2) and only 1% higher average (303 seconds) for case 3, showing Escra can allocate fewer resources to an app and maintain similar performance. As is indicated in Figure 5.7b, Escra + OpenWhisk outperforms OpenWhisk alone at 99th percentile and has lower tail latency.

Efficiency Figure 5.8 shows aggregate CPU and memory limits for OpenWhisk and OpenWhisk + Escra for ImageProcess. On average, OpenWhisk + Escra sets the limit at 7 vCPU whereas OpenWhisk static allocation results in a limit of 12 vCPU, resulting in a savings of approximately 5 vCPU for identical workloads. For memory, the difference in the limit averages around 1550 MiB.

According to Figure 5.9, OpenWhisk allocates 113 vCPUs for GridSearch on average. On the other hand, Escra + OpenWhisk was able to reduce the vCPU allocation to 53 vCPUs. For memory, on average, OpenWhisk sets the application aggregate limit to 29087 MiB while Escra + OpenWhisk is able to run the same GridSearch application with an application limit of 22264 MiB. On average, Escra + OpenWhisk saves 60 vCPUs and roughly 7 GiB of memory space.

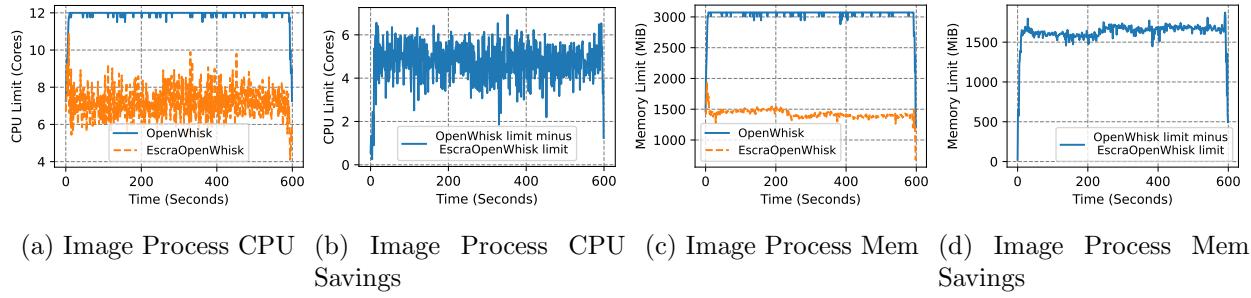


Figure 5.8: Aggregate memory and CPU limits averaged per second over four test iterations for ImageProcess. We highlight the difference (savings) between OpenWhisk limits and OpenWhisk + Escra limits with the savings graphs.

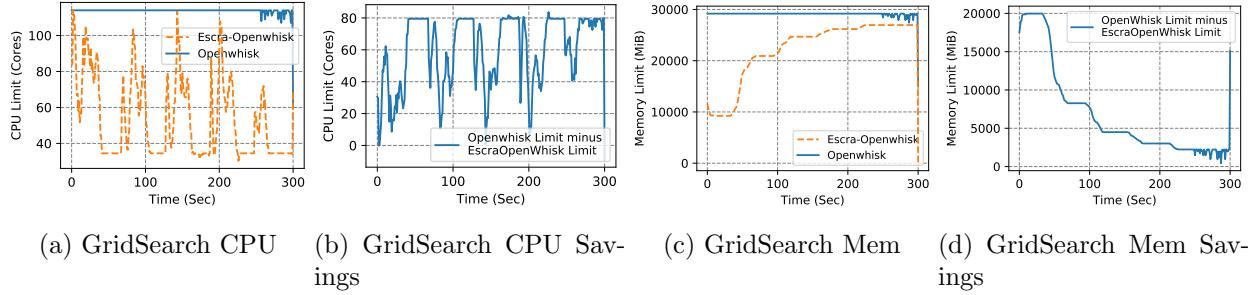


Figure 5.9: Aggregate memory and CPU limits over 5 minutes of running GridSearch. We highlight the difference (savings) between OpenWhisk limits and OpenWhisk + Escra limits with the savings graphs.

5.6.7 Takeaways

As shown in the ImageProcess and GridSearch benchmarks, Escra only minimally effects function latency while providing significant resource savings on static CPU/memory limits. In sum, Escra increased efficiency while maintaining performance. ImageProcess in particular shows that Escra is able to handle a dynamic and rapid increase in number of application pods. The GridSearch results showcases how Escra can help running batch-like, data intensive, long-running applications with fewer resources but without increasing latency.

5.6.8 Escra MicroBenchmarks and Overheads

Why a 100ms Report Period? Escra uses a 100ms CPU telemetry report frequency for two main reasons. First, 100ms complements the default Linux CFS period. Second, we measured the 99% end-to-end latency performance across various report frequencies every 50ms from 50ms to 200ms. Collecting CPU statistics at the end of every period (100ms) and reporting them directly

to the controller resulted in the lowest application latency.

Escra Network Overhead Escra sends usage statistics over UDP to the Controller and the Controller launches RPC calls to the Agent process to update container limits. The peak network overhead measured for 32 containers is 12.06 Mbps. Since the majority of the bandwidth usage comes from the per-container CPU telemetry, we expect the network overhead to scale linearly with the number of containers managed. An investigation into how Escra scales as containers are geographically farther away from the Controller and Resource Allocator (increasing network latency) is left to future work.

Escra CPU Overhead The largest CPU consumers in Escra are the Controller, Resource Allocator, and the kernel threads running on each worker node reporting telemetry data. The Controller consumes the most CPU out of the three since the memory reclamation process relies on the cAdvisor API [40], consuming up to 85% of a core. Replacing the cAdvisor functionality with memory limit/usage system calls would greatly reduce the memory reclamation overhead. Without cAdvisor, the Controller and Resource Allocator together use 5.7% of a core with 68 containers. For a cloud-scale analysis, we assume a separate Escra Controller and Resource Allocator that manage each application. Escra Controllers and Allocators are able to manage 1,192 containers per core. Assuming 20 cores per node, a collection of Escra Controllers and Allocators can manage up to 23,859 containers per node. Note, as more containers are registered with the Controller, the mean time between subsequent container stats increases sublinearly.

5.7 Discussion and Future Work

This section discusses how Escra affects cloud ecosystems and describes some directions for future work.

Multi-tenant Building a fully-fledged cluster management system that takes advantage of Escra remains future work. The contribution of this paper is that fine-grained, event-driven resource allocation is possible and performs well. While Escra can effectively reduce slack and increase performance, it remains an open question in how such benefits translate to a large-scale,

complex, multi-tenant system.

Serverless Our initial implementation of OpenWhisk + Escra is naive in several ways: (1) all containers are treated as the same application; the framework would need to modify this to deploy pods in per-tenant namespaces, and (2) the OpenWhisk invoker remains unaware of the actual CPU and Memory limits being used; it would need to be modified to ingest current usage and limits from Escra. We leave these to future work.

Beyond the efficiency benefits of using Escra in serverless systems, the Distributed Container abstraction may further be useful for billing and accounting in serverless systems [8, 165]. Many commercial frameworks set global limits on serverless applications by setting an invocation limit (i.e., the maximum number of concurrently running functions). With the Distributed Container abstraction, it would be possible to instead limit based on maximum memory or CPU usage. The study of limits and billing using Distributed Containers in serverless systems is a subject of future work.

5.8 Conclusion

This work illustrates how current orchestration systems fail to achieve both high performance and cost efficient container deployments, typically trading performance (throughput, latency) for cost-efficiency (slack) or vice versa. We motivate the need for a fine-grained and seamless container scaling orchestrator and propose a solution: Escra. Escra uses kernel hooks to generate both fine-grained telemetry and OOM handling events that allow a logically-centralized Escra Controller to allocate resources within 100s of milliseconds. As a result, Escra minimizes CPU slack by over 10x compared to our implementation of Autopilot. Escra also reduces application limits in serverless frameworks, saving more than 2x the CPU and memory resources over a standard serverless deployment. Escra’s comparison to static approaches, Autopilot, and OpenWhisk indicates fine-grained container scaling finds the balance between performance and efficiency while maintaining isolation. Escra is open-sourced at <https://github.com/gregcusack/Escra.git>.

Chapter 6

LD-NR and DiNOS: NUMA-Aware Replication for Dynamic Systems and an Operating System for a Shared Memory Rack

This work focuses on management of data structures in non-uniform memory access (NUMA) systems, particularly emerging and extended NUMA architectures. LD-NR is a library for NUMA-aware and temporally-dynamic replication of sequential data structures. By supporting changes to the replica set over time, LD-NR is able to manage spatial NUMA characteristics even for workloads with temporally dynamic characteristics. DiNOS is a distributed operating system targeting a rackscale, extended NUMA architecture; DiNOS uses dynamic replication provided by LD-NR to manage process page tables.

6.1 Introduction

Computing infrastructure is increasingly composed of non-uniform memory access (NUMA) architectures, whereby a processor can access locally attached memory faster than memory attached to other processors. Today, we find NUMA in the form of tiles [204] or chiplets [191, 14] within processors, in the form of multi-socket servers, and even across servers in rackscale systems with shared memory [52, 207, 59].

Despite the prevalence of NUMA, software designed for traditional uniform memory access (UMA) architectures can perform poorly on NUMA architectures due to the extra latency incurred on remote accesses: NUMA characteristics have been documented to impact the performance of applications by up to 20% [213]. Many ideas have been proposed to address this issue, ranging from

thread migration [148, 135, 181] to new NUMA-aware data structures [64, 230, 188, 111, 231] and applications [193, 55]. A general and powerful technique for adapting data structures to NUMA architectures is to replicate data across NUMA domains, so that a processor can access data locally on reads, while using a NUMA-aware replication algorithm to update the replicas on writes [42].

Unfortunately, existing methods to replicate data across NUMA domains support only a **static** set of replicas—typically one per NUMA domain—which must be decided at initialization and subsequently cannot be changed. Static replication works well when the computational needs of the system are also static, but modern systems are often dynamic and so several problems may occur. First, applications and their workloads can vary over time, for instance, in idle periods applications may use a few processors fitting on a single domain while in peak periods the same application may need processors in multiple domains. In this case, replicating data across many domains is sometimes wasteful but replicating data across few domains is not always sufficient to attain the benefits of replication. Second, the scale of applications are increasing, meaning applications may have large states and may require resources from a large number of NUMA domains. Here, it may be too expensive to replicate data across all domains beyond a certain count. Third, resource scheduling is a complex problem and scheduling decisions vary over time, requiring an adaptation in the number of replicas. For example, if the system is running out of memory due to other colocated applications, it may be necessary to reduce the number of replicas. Fourth, the system infrastructure is also becoming dynamic through new disaggregation technologies [59], whereby the number of domains provided by the hardware can vary over time. For example, an operator may assign a new machine to a cluster on a rack to scale an application, where it is desirable to expand the replica set to include the new machine.

We run several experiments to quantify the performance issues that arise when the system has the wrong number of replicas due to dynamic changes (Section 6.2.3). Although the memory overheads of replication are linear to the number of replicas, we find that the relative performance benefits across static replication configurations are heavily influenced by dynamic factors. We illustrate that it is not possible to optimize the memory-performance tradeoffs of replication in a

dynamic system using a static policy. Furthermore, we identify cases where static replication can be detrimental beyond suboptimal performance. For instance, replication with specific workload characteristics can lead to stalls, where operations on the replicated data are unable to complete.

To address these issues, we introduce a new method to replicate data across NUMA domains called Live-Dynamic Node Replication (LD-NR). LD-NR allows the number and location of replicas to change dynamically. This enables a user of LD-NR to adjust the performance-memory tradeoff of replication in response to changes in workload, the number of NUMA domains used by an application, resource scheduling, or even to changes in system infrastructure.

LD-NR tackles three main challenges in dynamically modifying the replica set: (1) determining which replica to use for a given operation since there may not be a local replica, (2) ensuring liveness when some of the replicas may be lagging behind because the replica has not been activated by application threads for a while, and (3) adapting the underlying replication algorithm to provision or release replicas. LD-NR addresses these challenges as follows. First, it uses a new NUMA-aware replica cloning technique to adapt the replication algorithm to add or release replicas (Section 6.3.2). Second, it introduces a NUMA-aware dispatcher that matches threads to replicas while prioritizing locality (Section 6.3.3). Third, it provides a new mechanism for liveness whereby threads from other NUMA domains detect and update lagging replicas (Section 6.3.4).

We evaluate LD-NR by comparing LD-NR to concurrent data structure implementations and to a state-of-the-art NUMA-aware replication library, NR, whose static policies are not able to adapt to system and application state [42]. We find that LD-NR can maintain optimal replication configuration even in systems with dynamicity, and that LD-NR incurs no memory overhead and nominal performance overhead when configured identically to NR. For instance, for a hashmap being accessed by 96 cores, LD-NR achieves $2.6\times$ the throughput over the best performing concurrent hashmap implementation, and provides 97% of the throughput provided by NR with an ideal, static configuration.

To demonstrate the power of LD-NR in a more complex setting, we use it to implement DiNOS, a new operating system for systems with rackscale shared memory. The design of DiNOS

is motivated by the current resource disaggregation trend (e.g., [52, 207, 59]), whereby a rack may have a variable number of compute domains sharing memory with each other. DiNOS uses LD-NR to replicate per-process operating system state such as page tables across the domains where each process is currently running. By doing so, DiNOS reduces the memory overhead of replication (by not avoiding useless replicas) while replicating when it is useful (the process is active in that NUMA domain) and feasible (the system has enough memory). The dynamic replication scheme in DiNOS enabled by LD-NR provides a finer granularity than state-of-the-art page table replication schemes support [186, 178, 1]. This demonstrates the utility of LD-NR for optimizing complex systems in emerging architectures. In conclusion, the contribution of this paper are as follows:

- We identify and quantify issues stemming from the static nature of the existing NUMA-aware replication technique node replication (NR) [42], which supports only a fixed number of replicas (Section 6.2).
- We introduce a new NUMA-aware replication scheme called LD-NR that supports dynamic replication (Section 6.3).
- We illustrate the power of LD-NR to build a new operating system called DiNOS for rackscale environments (Section 6.4).
- We implement and evaluate LD-NR and DiNOS (Section 6.7).

6.2 Background and Motivation

NUMA patterns are an ubiquitous trend of modern computation. The primary method of optimizing NUMA systems is to maximize locality, with the accessor of memory ideally sharing the same NUMA domain as the memory being accessed. Intuitively, replication can increase NUMA locality, since all domains containing a replica of data can be considered NUMA-local to the data. Given this, Node Replication (NR), a novel algorithm and associated general purpose library, was designed to provide strongly consistent NUMA-aware replication for sequential data structures [42].

The dynamic behavior of workloads along with the increasing scale of systems provide a great opportunity for system optimization, and NUMA-aware replication is positioned to be an important component of this optimization. However, when properties of the system do not match the conditions assumed by NR, the benefits of NR dwindle and degrade while overheads remain. This is especially problematic since some properties are shaped by dynamic behavior. To optimize dynamic systems using NUMA-aware replication for data structures, *dynamic replication* is needed.

This section begins by providing the details on NR necessary to understand the underlying assumptions and resulting performance characteristics (Section 6.2.1). We motivate the need for dynamic replication by highlighting four system trends which illustrate the importance of dynamic replication as a tool for system optimization (Section 6.2.2). We then show, with measurements, that the core approach of NR leads to statically decided tradeoffs between throughput and memory utilization as well as susceptibility to stalls (Section 6.2.3).

6.2.1 Background on Node Replication

Illustrated in Figure 6.1, NR is a general-purpose library implementing an algorithm that provides NUMA-aware replication for arbitrary sequential data structures [42]. Central to NR is a shared log (Figure 6.1 `SharedLog`). The shared log allows operations to be appended to and consumed from the log operation buffer (Figure 6.1 `OpBuffer`). Each *operation* is an encoding of an action that accesses or modifies the replicated data structure (Figure 6.1 `get` and `put`). For simplicity, NR assumes application thread assignment is static (e.g., a thread is pinned to a core)—an assumption shared in this work. Each thread must register with a replica, and NR assumes threads register with a NUMA-local replica. Application threads submit operations using the *execute* function provided by NR replicas. Each NR *replica* contains a copy of the replicated data structure (Figure 6.1 `hashmap`) and NR-specific metadata (e.g., Figure 6.1 `LocalTailPtrs`, which record the next operation in the shared log that the replica should consume).

The NR shared log is inspired by shared logs in distributed systems (e.g., CORFU [27]) and is used to provide strong consistency through linearizability of operations across replicas. When a

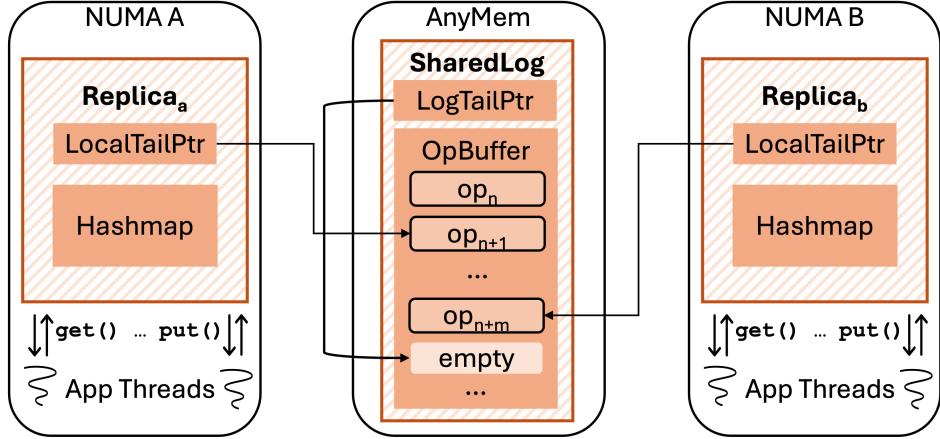


Figure 6.1: A hashmap replicated by NR [42] across NUMA domains A and B. NR components are orange striped.

thread submits an operation, it attempts to become the per-replica *combiner*. Only one combiner per replica at a time is required, so not all threads need to become the combiner. A thread will fail to become the combiner if another thread is already that replica’s combiner. Note that the combiner for a replica only exists in response to an application invoking `execute`. If a replica is not being utilized by any application threads, and thus there has been no combiner activity for some time, we consider that replica *inactive*.

The combiner for a replica handles operations submitted by multiple threads registered to the replica in batches. The combiner for a replica *appends* operations that mutate the data structure to the log, which ensures that the operations will be applied to every copy of the data structure. The combiner of a replica *consumes* operations from the shared log by applying those operations to the replica’s copy of the data structure. The result of an operation o can be computed once all operations submitted to the shared log before the submission of o have been consumed from the shared log by the combiner; at this point, o can be applied safely and consistently to the local replica to obtain the result. As the shared log is stored in an unspecified NUMA domain (in Figure 6.1 denoted as **AnyMem**), actions on the shared log may require cross NUMA domain memory accesses; however, operations on the local copy of the data structure do not. The NR shared log is optimized through techniques such as flat-combining [99]. Complete descriptions of optimization techniques and garbage collection of the shared log can be found elsewhere [42].

In addition to the benefits of NUMA-aware replication, NR also reduces contention since only the combiners (and only one combiner at a time per replica) perform mutating operations on copies of the data structure and the shared log. NR increases the throughput and scalability of accesses compared to lock-free and lock-based data structures [42].

6.2.2 Motivating Dynamic Replication

NR is designed around a core assumption that there will statically be one replica per NUMA domain. We argue that the use of NR for system optimization requires dynamic configuration of the placement and number of replicas, breaking this core assumption. This section outlines four characteristics of modern systems that motivate the need for dynamic replication towards the goal of system optimization. This provides context for Section 6.2.3, which quantifies how NR can be used for optimization through measurements of NR under dynamic conditions.

Applications are dynamic: Workloads are inherently variable [196, 201, 57]. Demands on applications can change, leading the application to allocate or deallocate memory and spawn or release threads. Likewise, usage behavior changes between applications, deployment scenarios, and over time. This leads to variance in the data structure access patterns, such as the read/write ratio and timing. The set of replicas should be configured to meet application needs.

System scales are increasing: While hosts with one to four NUMA domains are common today, the scale and complexity of NUMA systems continues to rise. This trend is driven by demands from cloud and machine learning workloads and is supported by architectural advances such as sub-NUMA domains [191, 14]. Closely related is the more general trend towards disaggregation, which can also result in more complex NUMA topologies through dynamic cross-component scaling supported by advances in high-performance interconnects [109, 59]. With the increasing scale of systems, replication of every data structure on every NUMA domain becomes a significant overhead; dynamic replication is needed to dynamically control that overhead.

System resource scheduling is complex: Increases in complexity and scale for NUMA topologies result in a greater number of permutations of NUMA domains across which compute

and memory may be distributed. A set of resources is assigned to a set of NUMA domains based on physical resource constraints, current utilization, the allocation and scheduling policies of the operating system, and any techniques employed by the application to influence the operating system. An application may not receive its ideal NUMA placement, particularly in a shared system. Application performance and the benefits of disaggregation are diminished when applications are unable to effectively use the NUMA topology reflected in the resources assigned to the application. For instance, if one NUMA domain is short on memory, it may not be feasible to maintain a replica there. If the NUMA topology contains many domains and the data structure is large, a reduced replication factor may be necessary to avoid exceeding the available memory. Dynamic replication is a key optimization tool for applications deployed in large-scale, shared systems.

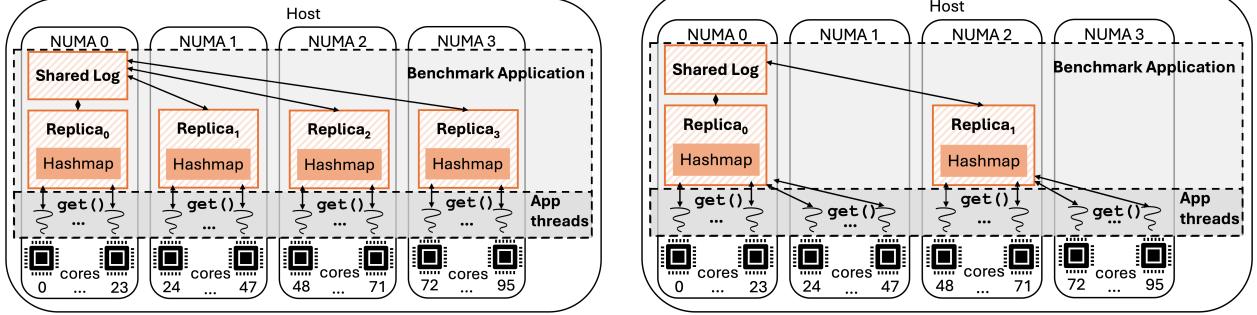
Systems are becoming dynamic: With disaggregation comes the ability for a system to be pieced together dynamically at runtime. This is a higher-level form of system optimization than we are referring to, as it is optimizing the formation of a logical computer from a pool of disaggregated resources. Nevertheless, there is still a need for system optimization within each logical computer. Dynamic replication is needed to enable an adaptable logical computer to utilize replication effectively.

6.2.3 Impact of NR's Static Replica Configuration

NR was designed around the assumption that an application will statically allocate a replica on each NUMA domain application threads run on. This is essential to achieve NUMA awareness and enables the NR algorithm and library to perform effectively. To fully understand NR's design characteristics and sensitivity to changes in the underlying assumptions, this section presents experiments measuring the memory utilization and throughput of NR under various conditions.

6.2.3.1 Experimental Setup

We evaluate the effectiveness of NR with several replication configurations and introduce dynamic workload characteristics. To do this, we write a benchmark application modeled from



(a) Replication strategy **nr4**, one replica per NUMA domain.

(b) Replication strategy **nr2**, two replicas in two NUMA domains.

Figure 6.2: Replication strategies for NRHashMap with 96 threads and a workload of all `get` operations. NR components are orange striped.

experiments presented in related work but modify the structure to generate workloads of data structure operations with dynamically varying characteristics [42, 33, 99]. Our benchmark application uses the Rust implementation of NR to replicate a hashmap, NRHashMap; a code snippet for this hashmap is included in the publicly available artifact [33, 225].

The test machine is a 96-core Intel Xeon Platinum 8260 host that has four equidistant NUMA domains of 24 cores each and runs Ubuntu 22.04.5 LTS with NUMA balancing and hyperthreading disabled. Since the test machine has four NUMA domains, we test NR instantiated with one, two (Figure 6.2b), and four replicas (Figure 6.2a), respectively denoted **nr1**, **nr2**, and **nr4**. Application threads are balanced across replicas and pinned to ensure locality between a thread and its assigned replica. If locality is not possible, application threads are pinned to cores to minimize the number of NUMA domains that access a replica, shown for **nr2** in Figure 6.2b.

Unless otherwise stated, each application thread continuously applies `get` or `put` operations to the NRHashMap in a closed loop, with the choice of `get` or `put` chosen statistically according to a write ratio. The hashmap keys for each operation are generated randomly from a uniform distribution. During initialization, the hashmap is pre-allocated with 67 million key-value pairs with 64-bit keys and 64-bit values. Throughput is aggregated over all threads each measurement second, and the total memory utilization of NRHashMap is measured at the end of each second using Rust allocator statistics [87].

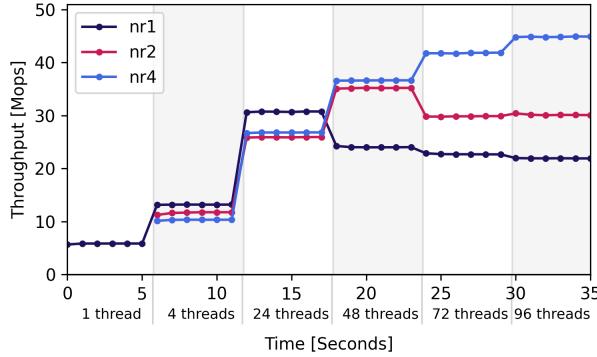
6.2.3.2 Replica Count and Scale

We measured the effects of replication count and number of application threads on throughput (Figure 6.3a) and memory utilization (Figure 6.3d). At each stage in the experiment (delineated by gray blocks and labeled below the x-axis) the number of threads increases. **nr2** and **nr4** start at four threads to ensure at least one thread per replica.

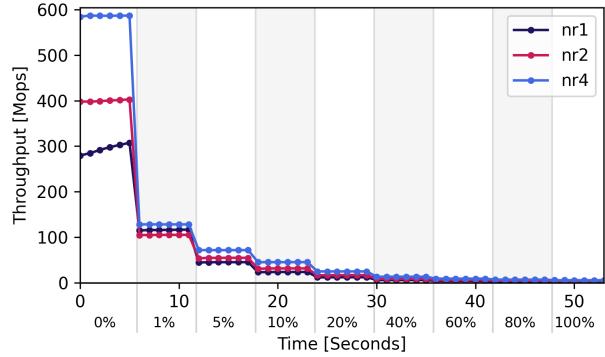
The best replication policy for throughput depends on the number of application threads with respect to the NUMA topology of the hardware. One replica (**nr1**) performs best when all threads t fit on one NUMA domain ($t \leq 24$). Once threads spill over to a second domain ($24 < t \leq 48$), **nr2** becomes the best policy. While **nr4** consistently uses the most memory, **nr4** only provides the best throughput for $48 < t \leq 96$. **nr1** and **nr2** show performance degradation once threads in different NUMA domains begin sharing a replica (> 24 and > 48 , respectively). NR does not support changing the number of replicas outside of instantiation so it is not possible to pick a replication factor that provides the best throughput across all stages in this experiment.

6.2.3.3 Diminishing Returns on Memory Utilization for Given Workloads

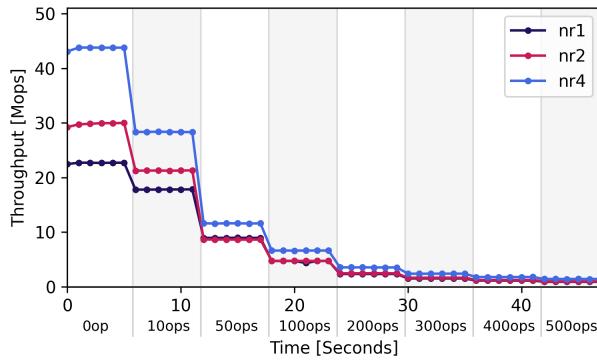
There are factors beyond scale that influence the success of an NR configuration. Here, we consider three of them: write ratio (Figure 6.3b), contention (Figure 6.3e), and operation latency (Figure 6.3c). All three experiments are configured with 96 threads. For write ratio in Figure 6.3b, we increase the percentage of **put** to **get** operations in each stage in the experiment. On average, **nr4** provides the best throughput; however, as write ratio increases the gap between different replication strategies narrows. Figures 6.3e and 6.3c show a similar pattern. Each experiment is configured with 10% write ratio. For contention (Figure 6.3e), each thread executes **n** operations on a thread local hashmap (independent of the NRHashMap) between each NRHashMap operation. In Figure 6.3c, operation latency is artificially increased by modifying the hashmap wrapped by NR to perform **n** extra **gets** or **puts** each time **get** or **put** is called. The modification of the wrapped data structure is transparent to NR. It is natural for throughput to decrease both as we reduce contention and increase operation latency, as the highest theoretical throughput is also decreased



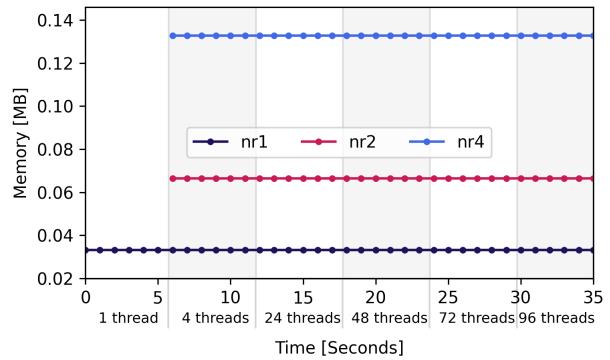
(a) Throughput of NRHashMap with increasing threads.



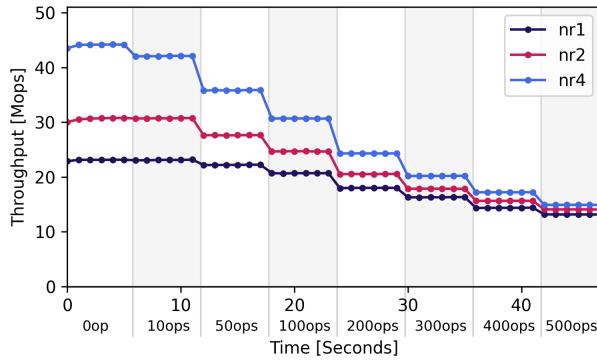
(b) Throughput of NRHashMap as write ratio increases.



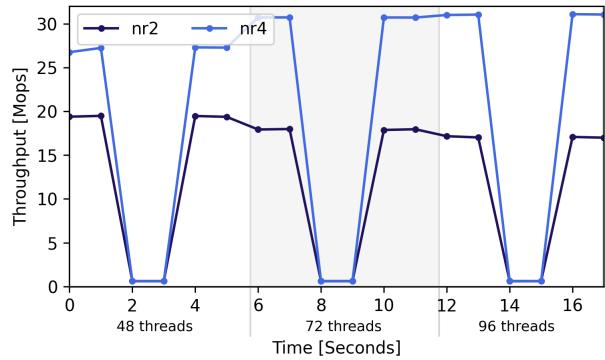
(c) Throughput of NRHashMap as operation latency increases.



(d) Memory utilization of NRHashMap with increasing threads.



(e) Throughput of NRHashMap as contention decreases.



(f) Throughput of NRHashMap with periods of replica inactivity.

Figure 6.3: Performance in throughput (6.3a-6.3c,6.3e-6.3f) and memory (6.3d) of NRHashMap, across varied configurations. Each experiment phase (denoted by a grey or white background) runs for six seconds.

in these scenarios.

All three experiments show reduced benefit of replication while the memory overhead remain constant, illustrating that in dynamic systems static replication can show diminishing returns with equal cost. While the cost in memory is marginal for small data structures replicated across few NUMA domains, memory overhead scales with data structure size and the number of replicas.

6.2.3.4 Replica Inactivity and Stalling

Static registration of threads to replicas at initialization makes applications susceptible to NR stalls. NR can cause blocking if one or more replicas become inactive while other replicas remain active. Recall that a replica is active if threads that are registered to that replica are performing operations, since in the course of processing an operation the replica is updated. While increasing the number of replicas potentially increases peak throughput, it also increases the susceptibility that a replica in that system will become inactive. If a replica is inactive, space exhaustion on the shared log can cause aggregate throughput to drop to zero.

We can illustrate a stall using NRHashMap with a 10% write ratio. In Figure 6.3f, during each six-second interval, all threads assigned to replica₀ are made idle for the middle two seconds, rendering replica₀ inactive during that interval. Threads assigned to other replicas continue to attempt to perform `get` and `put` operations, gradually filling up the shared log with pending operations. Eventually, this triggers the stall and throughput falls to zero. This is true regardless of the replication factor (`nr2`, `nr4` in Figure 6.3f) and the total number of threads (48, 72, and 96 in Figure 6.3f).

Existing techniques to mitigate stalls including a dedicated worker per replica [42] or a periodic timer per replica [33]. However, there are downsides to both options. Dedicating resources can be wasteful when there is low utilization. Periodic timers still result in stalls, just stalls bounded by the timer interval; furthermore, threads interrupted by the timer are distracted from the work they would otherwise be doing. Additionally, both of these solutions rely on information about the number of replicas and which threads are assigned to each replica. This information is easy

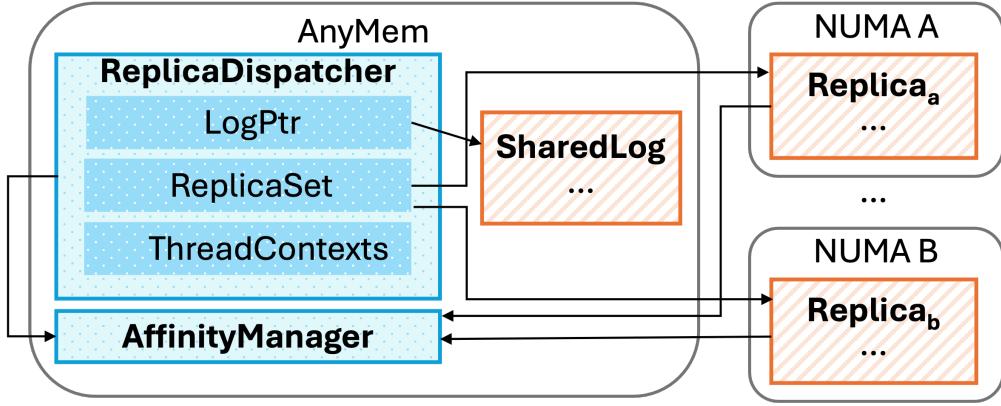


Figure 6.4: Illustration of LD-NR design highlighting new components (dotted blue). Additional details on components similar to NR (striped orange) is shown in Figure 6.1.

to track within the static assumptions of NR, but would be hard to track if those values became dynamic, as in dynamic replication.

6.2.3.5 Summary Our evaluation of NR shows that dynamic factors influence the best replication policy for performance (Figure 6.3a) as well the performance-memory tradeoff of replication configurations (Figure 6.3b—Figure 6.3f). We also note that limitations of available memory can make certain replication configuration infeasible. Furthermore, we show that poor fits between workload characteristics and replication configurations are not only suboptimal, but can cause degradation (Figure 6.3a) and stalls (Figure 6.3f). Hence, we need a mechanism to adapt the replication algorithm to workload and system state.

6.3 Live-Dynamic Node Replication (LD-NR)

We introduce Live-Dynamic Node Replication (LD-NR), an algorithm that provides NUMA-aware replication with dynamic reconfiguration to adapt to changes in system and workloads. We give an overview of LD-NR (Section 6.3.1), and then introduce three new techniques used by LD-NR to support dynamic replication (Section 6.3.2–Section 6.3.4).

6.3.1 Design Overview

LD-NR is designed around two new components: a replica dispatcher and an affinity manager. Figure 6.4 illustrates the components of LD-NR.

Dispatcher The replica dispatcher serves two purposes: it is a container for metadata and a gateway through which application threads submit operations to a replica. LD-NR requires a container because metadata must be preserved independent of the existence of any replica. LD-NR requires a gateway because threads cannot assume the existence of any particular replica and thus need a static construct to mediate their access to replicas. The dispatcher provides the basis for *liveness*, meaning threads can always access and perform operations on the replicated data structure.

When an application calls the dispatcher `execute` function, the dispatcher selects a replica to handle the operation. The thread-to-replica matching process is transparent to the thread. The dispatcher also provides methods for adding and removing replicas. LD-NR does not change the replication factor unless the application directs it to do so. Exposing the mechanism enabling dynamic replication to the application rather than implementing specific dynamic replication policies internal to LD-NR allows LD-NR to be adapted to various data structures, workloads, and architectures.

Affinity manager LD-NR uses an affinity manager to modify and restore the memory allocation policy before and after actions that could allocate memory on behalf of a replica. A core concept of NUMA-aware replication is that each replica is constrained to a particular NUMA domain. In dynamic systems and where the dispatcher performs thread-to-replica matching, the affinity manager is invoked to set an affinity before an operation that might allocate memory (e.g., before a combiner for a replica applies operations to that replica) and then restore it afterwards. Previous replication techniques such as NR did not integrate a similar mechanism because in static systems this is often realized for free, e.g., if a thread only acts on a NUMA-local replica, default memory allocation policies such as first-touch make it very likely that the memory allocated for

a replica will be from the local NUMA domain. The additional capabilities of LD-NR, however, require an affinity manager to retain the property of NUMA-bounded replicas.

Because LD-NR is a general-purpose library not tied to a particular operating system, LD-NR requires the user to provide an implementation of an affinity manager. For instance, in Linux, we can use numactl [171]. Affinity managers are reusable, as they tend to use operating system specific and not application specific memory allocation controls. While the user provides an implementation of the mechanism, LD-NR internally uses the manager as necessary, transparent to the user.

Shared log and replicas LD-NR uses the shared log of NR, allowing LD-NR to replicate arbitrary sequential data structures, linearize operations, and use NR’s optimization techniques. Although the components of the shared log and replicas are similar to those in NR, they required modifications for LD-NR. First, we extend the shared log to keep track of the dynamic replica set which is necessary for garbage collection of operations on the log. Second, we moved replica-private metadata, such as thread registration identifiers and thread contexts, from each replica to the dispatcher to ensure they are available and unique across replicas.

6.3.2 NUMA- and Log-Aware Replica Cloning

When adding a new replica, LD-NR creates a copy of the replicated data structure. However, it is not sufficient to simply clone an existing replica as this cloning must be done in cooperation with the shared log for consistency, and memory must be allocated with the NUMA affinity of the new replica. We develop a NUMA- and log-aware replica cloning technique consisting of the following steps:

1. **Select Model Replica:** When an application initiates the addition of a replica, the dispatcher selects the replica that has made the most progress (i.e., consumed the most operations from the shared log) as a model for the new replica. This ensures that the new replica is not a straggler.
2. **Freeze Replica Model:** The dispatcher then stops the model replica from performing any log operations (consume or append) while cloning is in progress. This freezing is necessary

to ensure the consistency of the data structure and its metadata, as replica cloning is not atomic. Freezing the model replica also stops operations from being garbage collected from the log, which ensures that they will not be missed by the new replica. In the initial implementation of LD-NR, replica freezes are done with coarse granularity – the dispatcher freezes thread matching entirely for all replicas. However, future work could allow unaffected replicas to continue to make progress and even allow a partial freeze where the model replica is still able to append to the log.

3. Clone Model Replica: The dispatcher uses the affinity manager to change the memory allocation affinity of the application thread performing the dispatcher's `add_replica` operation to match the desired NUMA domain of the new replica. The dispatcher then copies the model replica, including both metadata and the data structure.

4. Update Metadata: The dispatcher adds the new replica to the replica set of the shared log (which will allow the new replica to start processing operations) and also adds the new replica to the replica set maintained by the dispatcher (which will allow the dispatcher to match threads to the new replica).

5. Restore State: Finally, the dispatcher restores the memory allocation affinity of the thread performing the add replica operation and unfreezes the model replica. Both the new replica and the model replica are now available for use.

The procedure for removing a replica follows a similar logic but loosely in reverse, starting with removal of the replica from replica sets (to keep threads from acting on the replica) and then freezing the replica, and so on.

6.3.3 Dynamic matching of threads to replicas

Dynamic replication requires dynamic matching of threads to replicas. There are two important considerations in the matching process: 1) ensuring threads can access replicas under all conditions, and 2) ensuring threads access a local replica if one exists. This is critical to achieving the benefits of NUMA-aware replication.

Threads must submit operations to a replica for execution on the replicated data structure.

The dispatcher manages the set of available replicas and chooses a replica to handle the operation. The dispatcher selects the replica with the same NUMA affinity as the thread (which is recorded during registration), if one is available. Otherwise, the dispatcher selects the replica by load-balancing threads without local replicas across available replicas.

Replicas must also keep track of the threads they are responsible for. The replica's combiner uses this information to gather in-flight operations from those threads for batching. In-flight operations from threads not acting as combiners must be handled by exactly one combiner, so it is essential that this bookkeeping be accurate. The dispatcher uses the thread matching algorithm to update the set of contexts each replica must manage; this update occurs whenever a replica is added or removed.

6.3.4 Avoiding Stalls with Affinity Management

In NR, a replica may become inactive if its threads do not execute operations, causing the pending operations in the shared log to build up until there is no free space. At that point, new operations cannot be added to the shared log, which causes active replicas to block. We show how LD-NR uses the affinity manager, the metadata reorganization, and dynamic thread-to-replica matching to solve this problem.

Recall that in both NR and LD-NR, a thread becomes the per-replica combiner to process operations on the replica. The combiner for a replica detects a stall when it is unable to append its operations to the shared log. In LD-NR, we overcome a stall by allowing the combiner that detected the blocking condition to attempt to also become the combiner of the replica that is farthest behind, thus becoming the combiner for two replicas. The attempt to become a second combiner may fail if another combiner is trying to do the same, but regardless, once there is a combiner working on the replica, that replica is no longer inactive. Whichever thread succeeds in becoming the combiner uses the affinity manager to match memory allocation affinity with the inactive replica, and then starts consuming operations from the shared log. This advances the state of the previously inactive replica, ensuring progress. The combiner repeats the process until all

inactive replicas have advanced enough to allow garbage collection to free space in the operation log. This clears active replicas for progress.

6.4 A Rackscale Operating System using LD-NR

We demonstrate the utility of LD-NR by using it to build an operating system for a rack with shared memory. An operating system is a good target for LD-NR because of its synergies with NUMA-aware replication. Operating systems are positioned to control the scheduling affinities of threads accessing a replicated structure, enabling an operating system to act as the ideal user of NUMA-aware replication. Reciprocally, since operating system data structures can become bottlenecks to process performance, an operating system can benefit from scalable, NUMA-aware data structures.

Operating systems are typically responsible for singular hosts, which may be large, but rackscale operating systems using emerging disaggregation technologies may be responsible for extended NUMA topologies composed of the resources of multiple hosts. As noted in Section 6.2.2, dynamic replication is especially important in the context of large or extended NUMA architectures. General-purpose operating systems also experience a great deal of dynamism, as they are expected to gracefully support processes with any workload and any scale up to hardware limits. Section 6.4.1 elaborates on the context of a rackscale operating systems and Section 6.4.2 presents the operating system design.

6.4.1 Context

Recent advances in optimized interconnects such as Compute eXpress Link (CXL) [59] can provide cache-coherent shared memory between interconnected hosts within a rack ($\sim 2\text{--}16$ hosts). Given the utility of such interconnects for disaggregated memory [141] and the trend towards scaling processes and VMs across hosts [47, 44], we posit these interconnects may enable new distributed operating systems [199, 3].

We assume a rackscale operating system is deployed over such a rack. Each host has a local

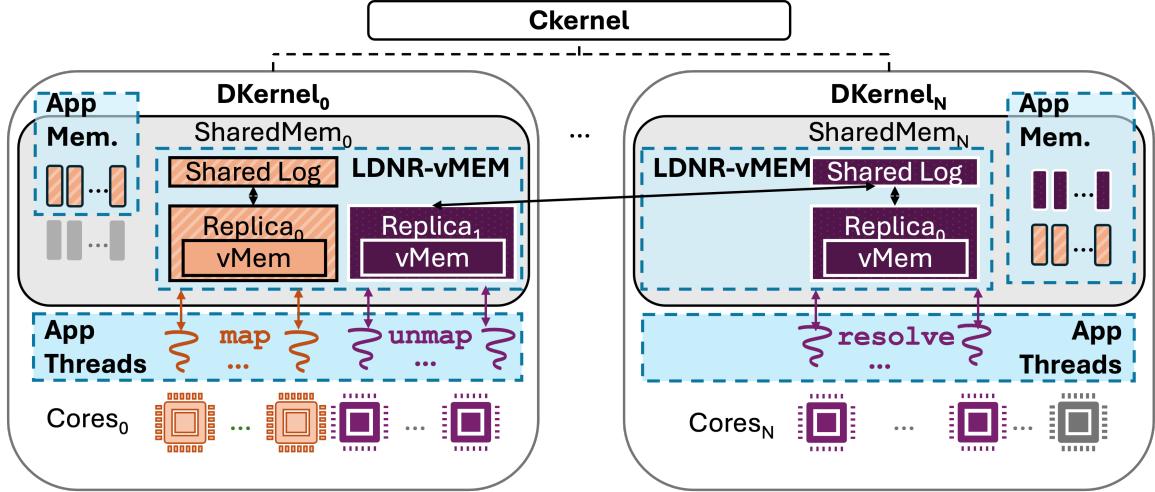


Figure 6.5: Example of LDNR-vMem replication for two processes run by DiNOS.

shared memory region and can access the shared memory regions of other hosts, but at higher latency. In addition to shared memory, hosts also contribute compute (their local cores) to the aggregate computational capability of the rack. Rather than specialize the design of a rackscale operating system to a particular interconnect—a potential avenue for future work—we assume the interconnect provides several features. First, we assume cache coherence across hosts for shared memory regions. Second, we assume each host in the rack can access coherent shared memory via load and store operations. Third, we assume interrupts can be sent between hosts using the interconnect.

6.4.2 Design

This section describes the design of our rackscale operating system, called (Di)stributed (N)ode replicated (O)perating (S)ystem (DiNOS). While DiNOS is primarily designed to exercise LD-NR, key to using a rack effectively is resource management within the rack so this is also a focus of DiNOS’ design. While operating systems for NUMA [33, 38] and operating systems for disaggregation [203] have been considered before, we believe our specific setting (flexible pooling of both memory and compute at rackscale supported by recent advances in interconnects) and focus (NUMA-aware replication) is a unique combination.

The design starting point for DiNOS is NrOS, a multikernel operating system integrating

NR [33]. In a DiNOS deployment, a NrOS-derived DiNOS component is deployed on each host in the rack. There are two primary components to DiNOS.

Ckernel: One host is selected to run a controller instance (a Ckernel) which provides coarse-grained, global resource management. The Ckernel schedules compute by the core and allocates memory by fixed-size, 2 GB contiguous slices. We choose to centralize resource management to ensure the Ckernel can use the holistic rack state for allocation and scheduling decisions [148]. Accordingly, the Ckernel allocator and scheduler are designed in two levels to reflect a rackscale extended NUMA topology. The first level chooses the host within the rack; the second level chooses the particular resource within the host and is aware of per-host NUMA topologies. The global allocator and scheduler in the Ckernel attempts to 1) limit processes to the smallest amount of hosts, and 2) balance resources across the required amount of hosts. The Ckernel is also responsible for other control operations including assignment of process identifiers and management of the in-memory file system.

Dkernels: All other hosts run data instances (Dkernels), which contain minimal operating system state and forward resource requests and control operations to the Ckernel via RPCs implemented over shared memory queues. The purpose of a Dkernel is to support processes using that Dkernel’s physical resources. Each Dkernel contains LD-NR-replicated, per-process operating system state for all processes scheduled to use the Dkernel’s cores. Other Dkernel tasks include mapping shared memory regions, registering with the Ckernel upon boot, and providing interrupt handlers. Section 6.4.2.1 provides more detail on LD-NR in DiNOS and Section 6.4.2.2 discusses support for rackscale processes in DiNOS.

6.4.2.1 LD-NR in DiNOS DiNOS addresses the challenge of managing per-process operating system state across an extended NUMA topology using dynamic replication. By per-process operating system state, we specifically refer to the hardware page tables and the page mappings comprising the virtual address space of the process. This is analogous to NR-vMEM in NrOS [33], but when replicated with LD-NR, we refer to this state as LDNR-vMem. LDNR-vMem supports

operations on a process' virtual address space such as `map` and `unmap`. Sometimes these operations are called explicitly by the process, such as when mapping a physical memory allocation, but often they are called by userspace libraries for tasks such as resizing a process' virtual address space.

In NrOS, NR-vMEM is statically replicated by NR across all NUMA domains for all processes. In contrast, DiNOS uses LD-NR to control where and when LDNR-vMem structures are replicated. DiNOS enacts a fine-grained, per-process, on-demand replication strategy. The LDNR-vMem for each process is only replicated on NUMA domains where it will provide increased locality for the process. Since LDNR-vMem replicas are only accessed in response to actions triggered by process threads of execution, the replicas must be placed only where the process is scheduled. An example of DiNOS LDNR-vMem replication is illustrated in Figure 6.5. The striped, light orange striped process has one LDNR-vMem replica because it has been scheduled on the cores of one Dkernel. Although this process has been allocated slices of memory on a second Dkernel, this does not result in a second replica. The solid dark purple process has two LDNR-vMem replicas because it is scheduled on the cores of two Dkernels.

6.4.2.2 Cross-host Scaling in DiNOS

DiNOS allows a process to dynamically and transparently scale using any CPUs and any memory available in the rack irrespective of host boundaries with a unified process address space across hosts. This provides the benefits of disaggregation (e.g., reduced resource stranding [141], relaxation of bin packing [47]) but requires careful management of resources and per-process operating system state—the foci of DiNOS design.

Specifically, DiNOS manages per-process operating system state using LD-NR and centralizes resource management in the Ckernel. In addition, DiNOS has an extended translation lookaside buffer (TLB) shootdown protocol that uses interrupts across hosts running a process. The Ckernel uses a global resource accounting scheme that provides rack-unique core identifiers and rack-unique shared memory region affinity identifiers. While these considerations were required to make DiNOS function, these techniques are not new to DiNOS so we do not describe them here. Similarly, we

employ known techniques to reduce the incidence of the Ckernel being a performance bottleneck, including coarse-grained resource allocations, multi-level allocation logic, allocation-free shared memory RPC library, page caches on Dkernels, and more.

6.5 Scheduling and Allocation at Rackscale

While the primary purpose of DiNOS is to exercise LD-NR, scheduling and allocation become more crucial at rackscale on an extended NUMA architecture. Thus, we design an alternative form of DiNOS, DiNOS*, that allows the top-level scheduler (hereby referred to as the macro scheduler) to be replaced with more sophisticated mechanisms. Specifically, in DiNOS*, we use declarative cluster management (DCM) [210] to encode allocation problems into decisions that may be solved using of-the-shelf SAT solvers.

In DiNOS*, the scheduler assigns requests for CPU and memory to actual resources available in the rack. DiNOS* uses a centralized scheduler across the rack to make globally optimal decisions. Like DiNOS, the DiNOS* scheduler has two levels: a macro scheduler which uses a constraint solver to make decisions at the host granularity (e.g., which host runs a given thread), and a micro scheduler which uses a simple algorithm to make decisions within a host (e.g., which CPU core within the host runs a given thread). Also like DiNOS, the two-level scheduler of DiNOS* assigns compute at the unit of cores (hardware threads) and memory at the unit of 2 MiB chunks called *memslices*. DiNOS* uses a two-level scheduler to reduce the problem size for the constraint solver: a rack may have 100–10K processes, 100–1K cores, and 100–100K GB of memory, and, at that scale, existing solvers may run for minutes, but DiNOS* need scheduling decisions in milliseconds. Thus, rather than asking the solver to produce a detailed assignment to every core and every memory page in the rack, DiNOS* only uses it to provide assignments to hosts. This approach reduces the solution space for allocation decisions by orders of magnitude. Within a host, resources are interchangeable, and so the second-level micro scheduler simply assigns the first available resource within the host.

Figure 6.6 illustrates the steps for allocating memory. A request for memory ① starts with

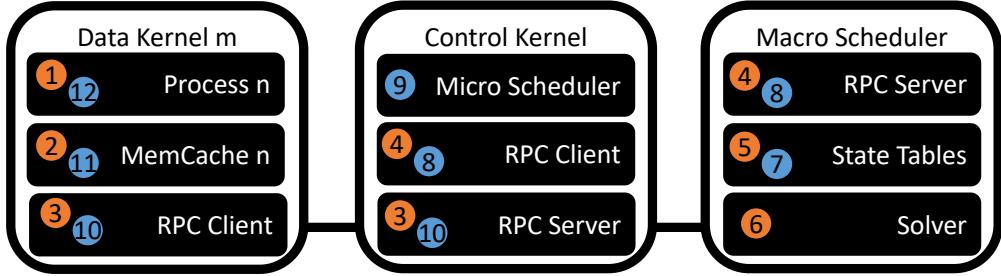


Figure 6.6: The process for allocating memory in DiNOS*.

the `allocate_physical()` syscall or when the Dkernel wants to expand the address space of the process. The per-process cache tries to fulfill the request from its available pages (fast path from ② to ⑫). If it fails, the Dkernel sends an RPC to the Ckernel ③ to request several memslices. The Ckernel forwards the request ④ to the macro scheduler, which adds the request to a pending table ⑤, periodically runs the solver ⑥ to assign a host for any outstanding requests, updates the `pending` and `placed` tables ⑦, and returns the assignment(s) to the Ckernel ⑧. The Ckernel invokes the micro scheduler ⑨, which picks available memslices within the target host. The Ckernel returns the memslices to the Dkernel as an RPC response ⑩. If the request was for less than a memslice multiple, the extra memory is added to the per-process cache ⑪. The requested memory is now available to the process ⑫. Allocating cores is an analogous process. As an aside, the process for allocating memory for the Dkernel (e.g., for the LD-NR-vMEM) bypasses the solver: the macro scheduler simply picks memory local to the log to ensure the log allocations remain consistent. If no such memory is available, the request fails with an out-of-memory error.

Internally, the macro scheduler maintains a model of the rack state in tables and views (Figure 6.7), which keep track of the hosts in the rack, their total and available resources, running processes, pending requests for the next run of the solver, and placed requests. The macro scheduler runs the solver every n milliseconds or m requests (defaults: $n=3$, $m=1$) . The solver also takes as input a set of constraints. The scheduler has some built-in constraints (only assign requests that have not been placed yet, do not assign more resources than available, and balance the load across Dkernels). In addition, it can take process-specific constraints, such as minimizing the number of hosts on which resources are allocated. Some of the constraints are hard (they must always be

Table/View	Description
Hosts	Hosts, total resources
Unallocated	Free resources
Processes	Running processes
Pending Requests	Requests for new resources
Placed Requests	Used resources

<i>Hosts Table</i>	
Field	Description
Host id	Identifier for host
Total cores	# of cores in host
Total memslices	# of memslices in host

<i>Pending Requests Table</i>	
Field	Description
request_id	Identifier of request
process	Process making request
cores	# cores requested
memslices	# Memsllices requested
status	Status of request
controllable_dkernel	Used by solver to make assignment

Figure 6.7: DiNOS*'s macro scheduler keeps a model of the rack in tables and views. The figure shows two such tables.

satisfied), while others are soft (they have a weight to indicate their importance).

Listing 6.1: The `placed_constraint` ensures DCM only tries to assign Dkernels for requests that are not already placed.

```
create constraint placed_constraint as
select * from pending
where status = 'PLACED'
check current_dkernel = controllable_dkernel
```

Listing 6.2: The `mem_cap` constraint ensures that memory is not overprovisioned and adds load-balancing using a pre-defined DCM `capacity_constraint` function. DCMSched also uses a corresponding `core_cap` constraint (not shown).

```
create constraint mem_cap as
select * from pending
join unallocated
```

```

on unallocated.dkernel = pending.controllable_dkernel
check capacity_constraint(pending.controllable_dkernel,
    unallocated.dkernel, pending.memslices,
    unallocated.memslices) = true

```

Listing 6.3: The `app_locality_pending_constraint` (shown) and `app_locality_placed_constraint` (not shown) prioritize locality by attempting to maximize the number of resources (pending and placed, respectively) belonging to a process which are placed on a single Dkernel.

```

create constraint app_locality_pending_constraint as
select * from pending
maximize
  (pending.controllable_dkernel in
   (select b.controllable_dkernel
    from pending as b
    where b.process = pending.process
    and not b.id = pending.id
   ))

```

To define the optimization problem, DCM accepts a number of constraints written as extended SQL. A sample of the constraints used by the macro scheduler are shown in Listings 6.1-6.3. Note that some constraints may be in conflict (e.g., the load balancing aspect of `capacity_constraint` in Listing 6.2 and the locality aspect of `app_locality_pending_constraint` in Listing 6.3). Conflicting priorities may be weighted by applying a multiplier to the property the constraint is seeking to optimize. As DiNOS* is an initial prototype, more complex constraints (such as replication for fault tolerance, partitioning of the rack, and process groups) and fine-tuning (such as policy weighting) are left for future work. Section 6.7.2.3 provides an initial evaluation of the macro scheduler in DiNOS*.

6.6 Implementation

LD-NR is implemented in Rust in approximately 1800 lines of code (LoC) with an additional 1200 LoC of unit tests. While the algorithm behind the shared log is largely unmodified from NR, most LoC are modified compared to the Rust implementation of NR [33] due to changes in types (global instead of replica-local thread identifiers, etc.) and changes to external and internal APIs.

DiNOS is implemented in Rust using NrOS [33] as a base. As the focus on this paper is LD-NR, and DiNOS is presented to illustrate the utility of LD-NR, we omit the discussion of several features of DiNOS in this paper. As such, it is difficult to get an accurate LoC count for the presented version of DiNOS but it easily exceeds 5k LoC. DiNOS is theoretically capable of running unmodified POSIX binaries but the implementation of DiNOS only provides a limited, hypervisor-like API that supports native DiNOS processes and a limited set of POSIX processes packaged within a library operating system (the NetBSD rumprun unikernel [195, 168]). This is an approach inherited from NrOS [33].

The macro scheduler used by DiNOS* is written in Java (\sim 4k LoC, including unit tests and three additional solver implementations used in evaluation). The framework to specify and solve constraints is DCM [210] and the solver is Google OR-tools CP-SAT (version 9.1.9490) [82]. To specify the DCM system state and constraints for DiNOS*, we wrote \sim 75 LoC of SQL.

DiNOS, including DiNOS*, is deployable via emulation using KVM [92] and QEMU [31], with QEMU inter-VM shared memory (ivshmem) [63] regions simulating cache coherent shared memory regions. Minor modification to the QEMU ivshmem server were required for experiments configured with very large shared memory regions. Because the test machine has limited support for sub-NUMA domains, in the prototype of DiNOS, the second-level memory allocator in the Ckernel does not consider host-level NUMA domains.

6.7 Evaluation

We evaluate LD-NR in dynamic situations in two different use cases: a hashmap (Section 6.7.1), and per-process operating system state in a prototype rackscale operating system (Section 6.7.2). We conduct all experiments on the test machine described in Section 6.2.3.1. We answer the following questions:

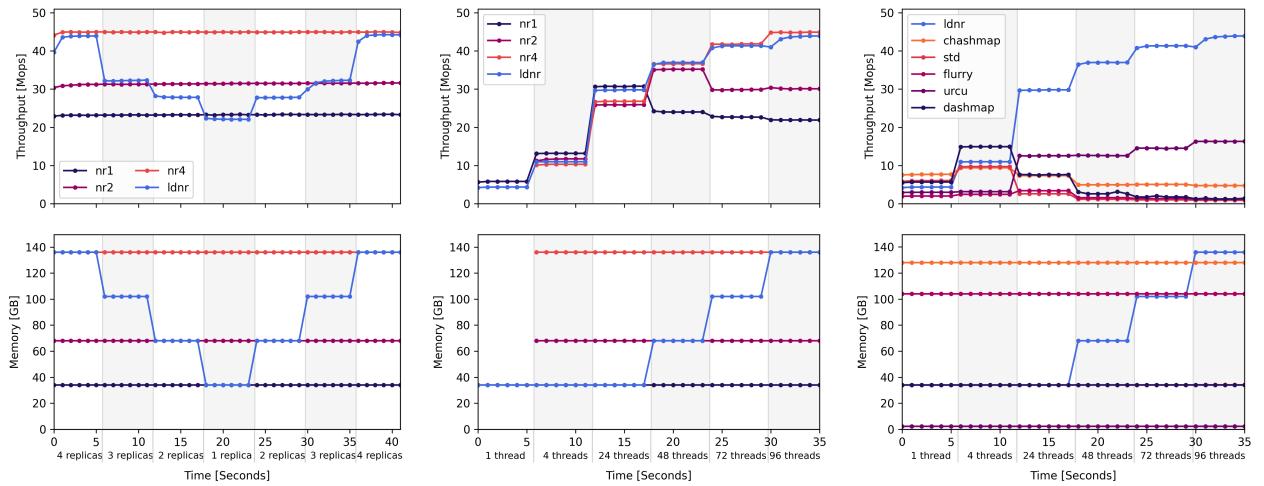
- Can LD-NR provide the benefits of NUMA-aware replication under dynamic conditions?
- Does LD-NR incur minimal overheads compared to an optimally configured static replication scheme?
- Does NUMA-aware dynamic replication benefit data structures and the applications that use them?

Additionally, we briefly evaluate the effectiveness of the macro scheduler in DiNOS* compared to other scheduling heuristics Section 6.7.2.3.

6.7.1 LD-NR Hashmap

We use LD-NR to replicate a hashmap. To answer the first evaluation question, we test the performance of the replicated hashmap under three varying factors: replication count (Section 6.7.1.1), both scale and replication count (Section 6.7.1.2), and replica inactivity (Section 6.7.1.3). To address the second question, we compare the LD-NR-replicated hashmap to an NR-replicated hashmap with static configurations of 1, 2, or 4 replicas (`nr1`, `nr2`, and `nr4` from Section 6.2.3.1). To answer the third question, we compare the LD-NR-replicated hashmap against additional hashmaps (Section 6.7.1.2).

The code to wrap the hashmap for replication and define the encodings of `get` and `put` operations is nearly identical between LD-NR and NR (code snippet for NR included in [33]). Due to the identical hashmap implementations and because the replication libraries use similar optimization techniques (e.g., the NR shared log), this comparison allows us to measure any overheads incurred



(a) Changing replica count to navigate the performance-memory tradeoff of replication.

(b) Scaling up threads for LD-NR compared to NR with static replicas.

(c) Scaling up threads for LD-NR compared to other Rust hashmap implementations.

Figure 6.8: A suite of benchmarks showing the performance of LD-NRhm when the replication policy changes over time (6.8a) and when both scale and replication policy change over time (6.8b, 6.8c).

by LD-NR. If NR is optimally configured, the best performance possible from LD-NR should be close to that of NR; if NR is not optimally configured, LD-NR should provide better performance.

LD-NR requires an affinity manager, which we implement in ~ 30 LoC. This affinity manager is not specific to the hashmap, but is operating system specific as it uses a system call to `mbind` to set and restore the NUMA memory allocation policy for a thread.

6.7.1.1 Performance-Memory Tradeoff We first demonstrate whether LD-NR can explore the performance-memory tradeoff of replication by varying the replication count.

Methodology. We keep other factors steady (96 cores, 10% write ratio) and modify the replication factor over time for LD-NR by removing replicas (at 5.5, 11.5, and 17.5 seconds) and adding replicas (at 23.5, 29.5, and 35.5 seconds).

Analysis. The staggered pattern in Figure 6.8a shows that throughput and memory use change corresponding to the replication factor. LD-NR allows a user to navigate the trade-off of using more memory to achieve better throughput.

6.7.1.2 Scaling with Dynamic Replication We now evaluate whether LD-NR can be used to enact a dynamic replication policy. As noted in Section 6.2.3.2, no single static NR policy provided the best throughput for NRHashMap across application thread counts. We construct a dynamic policy for LD-NR that adopts the best NR configuration at each phase.

Methodology. We measure the throughput and memory usage while scaling up the number of threads. We compare LD-NR with static NR configurations and other concurrent hashmaps implemented in Rust, including: the standard collection `HashMap` in Rust (`std`) [218], an implementation that uses multiple locks over buckets to avoid global contention (`CHashMap`) [46], a port of Java’s concurrent `HashMap` to Rust (`flurry`) [78], the User-space Read-Copy-Update (`urcu`) library’s hash table, which is lock-free [154, 202, 157], and `DashMap` [233].

Analysis. Shown in Figure 6.8b, LD-NR is competitive with the best static policy enacted with NR at each experiment phase. Using the middle two seconds of each phase, LD-NR throughput

is between 74.7% (phase 0, 1 thread) and >100% (phase 3, 48 threads) of the best NR throughput measured for that phase. LD-NR also uses memory conservatively in the early phases of the experiment, especially compared to `nr4`. LD-NR incurs a small overhead compared to NR when configured identically due to the additional checks and logic to handle dynamic configuration changes (e.g., at 96 threads with four replicas each, LD-NR sees 97.6% of the throughput of NR). Compared to other hashmaps in Figure 6.8c, LD-NRhm does not show the best performance with small thread counts (<24), but scales with significantly higher throughput than other concurrent hashmaps ($2.6\times$ better than `urcu`, the hashmap with the second best performance in Figure 6.8c). This underscores the importance of NUMA-aware replication as an optimization technique.

6.7.1.3 Tolerance for Inactive Replicas

LD-NR improves liveness by overcoming stalls when some replicas are inactive (e.g., when their threads do not execute data structure operations). We gauge the effectiveness of this technique by measuring throughput during intentionally triggered stalls.

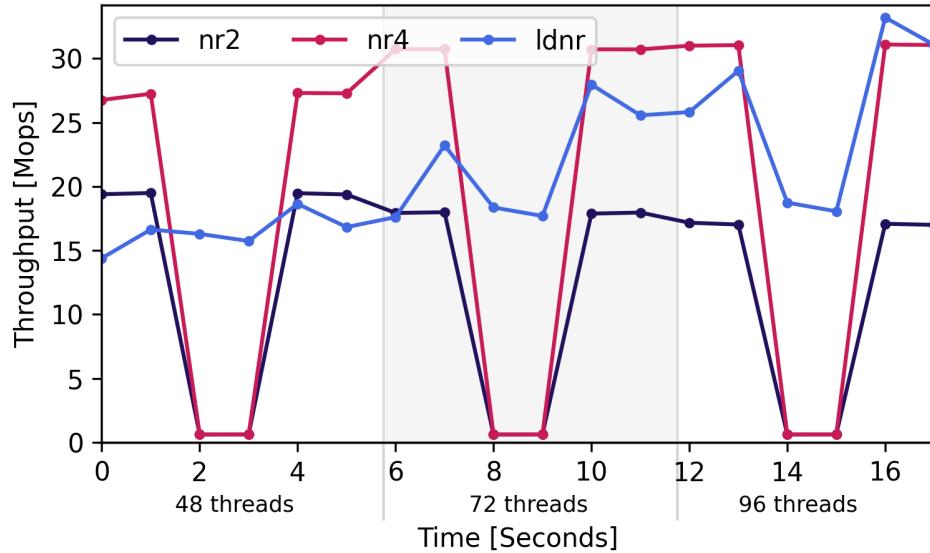


Figure 6.9: When NR encounters stalls, LD-NR continues to provide throughput without application intervention.

Methodology. We use the experimental setup from § 6.2.3.4 to test LD-NR. Recall that the number of threads changes in each six-second phase and during the middle two seconds of each

phase, the threads assigned to replica_0 do nothing, rendering replica_0 inactive and triggering a stall. LD-NR is configured to adapt the number of replicas to the number of active threads in each phase: two replicas for 48 threads, three replicas for 72 threads, and four replicas for 96 threads.

Analysis. Shown in Figure 6.9, LD-NR is able to provide liveness with an inactive replica transparent to the application. At higher scale (96 threads) there is a performance drop, which is natural given that one of the combiners must perform twice the work during the period of inactivity and because fewer application threads are attempting to execute hashmap operations.

6.7.2 DiNOS and LD-NR

We evaluate LD-NR in a more complex setting, DiNOS. We first demonstrate that DiNOS is capable of supporting dynamic cross-host scaling of processes (§6.7.2.1). We then use LD-NR to navigate the performance-memory tradeoff of replication by changing the number of per-process operating system state replicas for a process (§6.7.2.2). In both cases, we select memcached, a common in-memory key-value store, as the process. We measure the throughput of memcached for random `get` requests on a dataset of 64 GiB using 8-byte keys and 64-byte values. Note that memcached `get` operations are not operations submitted to LD-NR, but are operations specific to memcached.

Setup. To run DiNOS, we emulate a rack with shared memory. Each Dkernel and the Ckernel run as a QEMU guest on a single Linux host. We assign resources from a single host NUMA domain (memory, shared memory, and cores) to DiNOS instances. By running guests on different NUMA nodes, we create a performance difference between local and remote shared regions of each guest. The test machine has four NUMA domains so we configure experiments with 1–3 Dkernels so that each Dkernel and the Ckernel are on their own NUMA domain.

6.7.2.1 Cross-Host Scaling in DiNOS When a process needs to scale, it has two primary options: to *scale up* by using more resources on a single host, or to *scale out* as a distributed process across hosts. One motivation for designing a rackscale operating system is to support

processes larger than a single host through cross-host scaling. This experiment evaluates whether DiNOS achieves efficient scale up performance while transparently using resources across hosts.

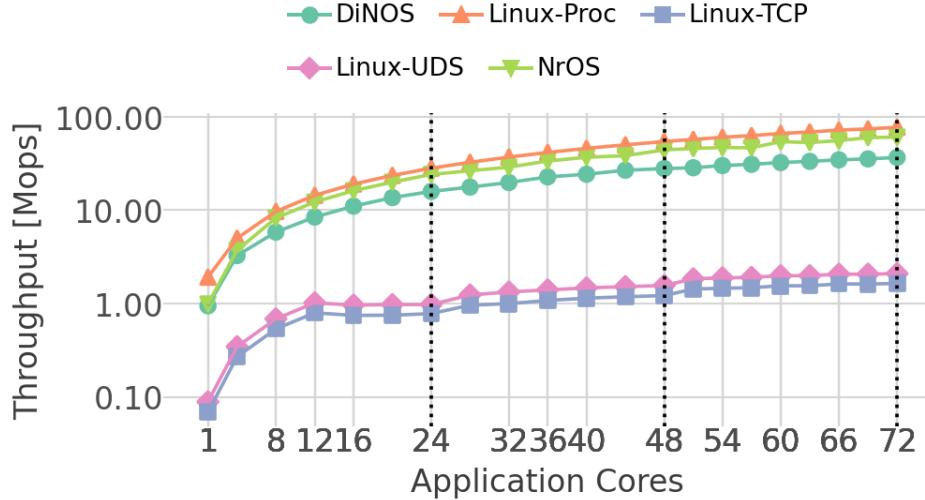


Figure 6.10: Throughput of memcached `get` operations for scale up (DiNOS, Linux-Proc, NrOS) and scale out (Linux-UDS, Linux-TCP) configurations.

Methodology. We configure memcached both as a single process (scale up), and as 1–3 statically sharded processes (scale out). The three scale up memcached deployments are on Linux (Linux-Proc), NrOS, and DiNOS. We consider DiNOS a special case of scaling up since cross-host scaling is transparent to memcached. The two deployments for scale out are Linux-TCP and Linux-UDS, which are deployed with a gateway process that distributes the workload across the memcached shard processes. TCP and Unix domain socket (UDS) specify how the gateway process communicates with the memcached shards. Figure 6.10 shows the throughput (y-axis, millions of operations per second) for different memcached configurations (x-axis, total application cores used by memcached). Dotted lines indicate when an additional Dkernel (or memcached shard) is added, e.g., one shard/Dkernel for $1 \leq t < 24$, two for $24 \leq t < 48$ threads, etc.

Analysis. The single process and single host cases (Linux-Proc, NrOS) illustrate the upper-bound of performance. DiNOS shows comparable performance with these configurations, indicating that DiNOS successfully hides the characteristics of the emulated rack. In contrast, the scale out configurations exhibit performance an order of magnitude lower due to the additional communication required between the gateway process and the memcached shard processes. These results emphasize

the design goals of DiNOS.

6.7.2.2 Replicating DiNOS state with LD-NR

DiNOS uses LD-NR to replicate per-process operating system state, i.e., process virtual address space management (vMEM) including the page tables and metadata. We show that LD-NR enables the operating system to navigate the performance-memory tradeoff.

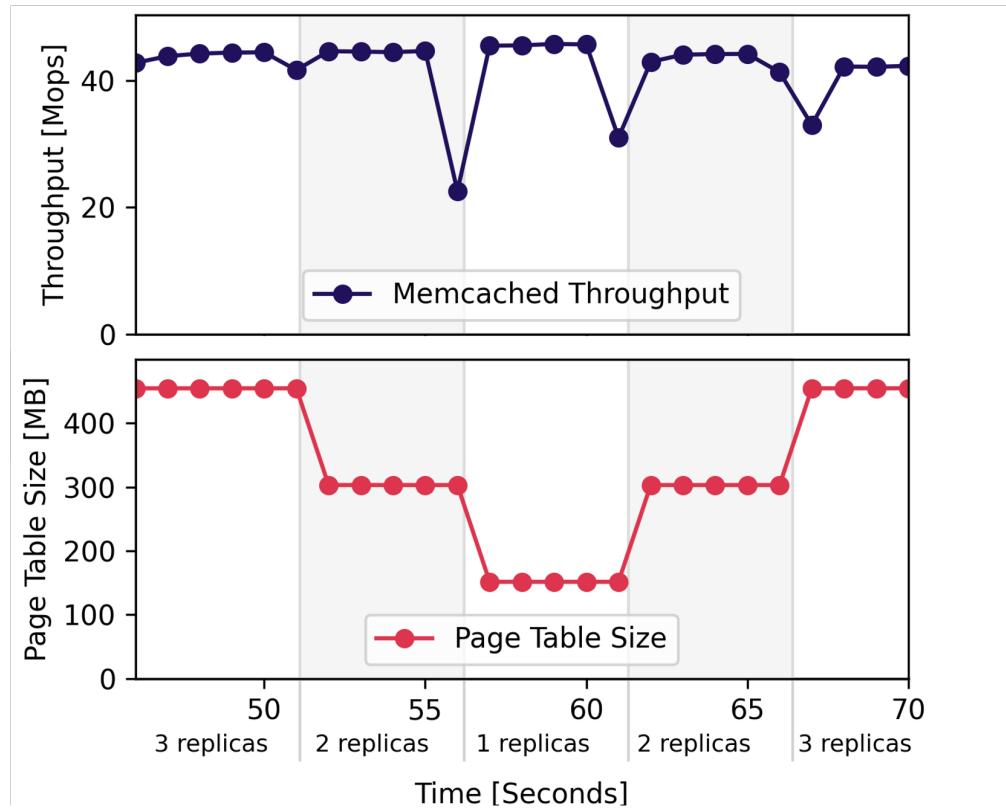


Figure 6.11: Throughput of memcached `get` operations (top) and memory utilization of the replicated page table (bottom) as DiNOS changes DiNOS-vMEM replication factor.

Methodology. We use a static configuration of DiNOS (3 Dkernels with 24 cores each) and static configuration of memcached (72 application threads, a dataset size of 64 GiB). While memcached is running, at specific intervals we manually drop and re-add DiNOS-vMEM replicas, dynamically adjusting the replication factor of memcached's DiNOS-vMEM. Figure 6.11 (bottom) shows the memory requirements of the per-process vMEM supporting the virtual address space of memcached, while Figure 6.11 (top) shows memcached throughput. We show results starting at 45 seconds, to allow memcached throughput to reach a stable state before modifying the replication factor.

Analysis Memcached’s throughput is largely stable, with temporary drops when we adjust the number of replicas due to replica freezes (recall that threads may temporarily have to wait to perform an operation while the dispatcher has frozen thread-to-replica matching, as described in Section 6.3.2). Once the replica has been added or removed, throughput returns to the stable state even when the replica set is reduced because memcached’s vMEM structure is not under enough load to benefit from having three replicas. However, dropping replicas greatly decreases the amount of memory required as memcached’s DiNOS-vMEM structure is large due to memcached’s large virtual address space size. This shows that LD-NR can be used by an operating system to configure a process’ vMEM using the granular controls exposed by both LD-NR and DiNOS.

6.7.2.3 Scheduler Policies with Memcached in DiNOS* This section presents an experiment comparing the performance of a process run by DiNOS* when DiNOS* is configured with different macro scheduler policies. Memcached is once again selected as the process, configured as in previous evaluations with a uniformly distributed workload of `get` operations. A description of the five different macro scheduler policies follows.

- **DCMcap** uses the default macro scheduler policies (no over-provisioning, with load balancing between nodes). Performance of DCMcap tends to see high variance since the solver does not consider factors important for performance, such as locality.
- **DCMloc** is DCMcap plus a locality bias that rewards placements that reduce the number of hosts a process is assigned to.
- **FillCurrent** chooses a Dkernel for a process randomly and then assigns all requests from that process to that Dkernel until the Dkernel runs out of resources. Then, it chooses another Dkernel randomly.
- **Random** chooses a Dkernel at random to fulfill each resource request.
- **RoundRobin** chooses a Dkernel to fulfill each request in a round-robin fashion.

Only the first two schedulers use the constraint solver, but all schedulers use the same in-memory database to view and update the cluster state.

Methodology. DiNOS* is configured with the largest configuration possible in the test environment (3 Dkernels with 24 cores each). Memcached is configured to use 24 cores and 16 MB of memory populated with 8-byte keys and 64-byte values; this makes it possible for memcached to fit within the resources a single Dkernel, allowing for maximum NUMA locality. Each experiment is repeated 10 times per policy and the average throughput and standard deviation is recorded in Figure 6.12.

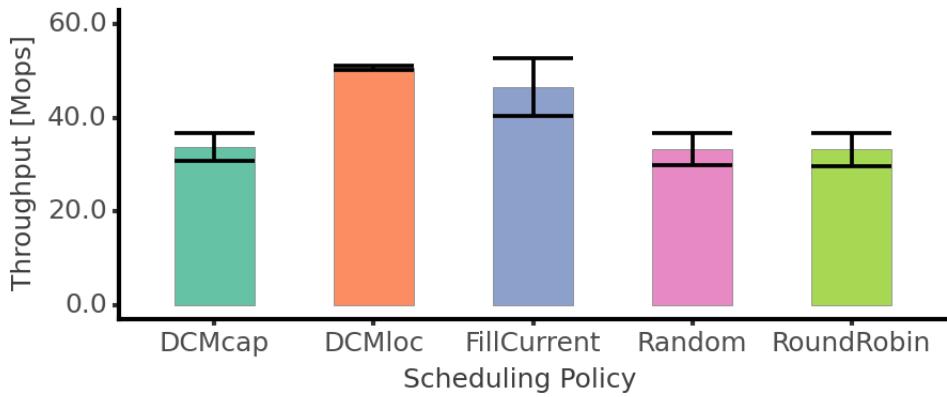


Figure 6.12: Throughput for each DiNOS* macro scheduler policy averaged across 10 iterations of memcached.

Analysis. Scheduling policies that prioritize locality (DCMloc and FillCurrent) achieve the highest average throughput as they reduce the number of remote memory accesses. Policies that do not prioritize locality (DCMcap, Random, and RoundRobin) show degraded average throughput due to more frequent remote memory accesses and distributed core placements among Dkernels. In terms of average throughput, policies with locality show a throughput increase of 38%-53%. This experiment highlights the advantage of prioritizing resource locality when allocating system resources for memory-intensive and latency-sensitive programs such as memcached. As detailed in Section 6.5, it is worth noting that for other processes, policies other than locality may be more effective. These results highlight the benefit of being able to tailor the scheduling policy per application and the importance of customizable scheduling interfaces for rackscale processes.

6.8 Related Work

Optimizing Data Structures for NUMA Many works adapt specific data structures to NUMA architectures, e.g., [188, 111, 231]. Other works build entire NUMA-optimized applications designed for tasks such as graph analytics and queries [243, 177]. In contrast, LD-NR provides one piece of the optimization equation for an application: the method of NUMA-aware replication for data structures. Other works similarly target data structures. Shaol [115] replicates read-only data in array-based structures according to static data access patterns extracted at compile time. LD-NR supports replication based on dynamic factors and is more generic since it supports sequential data structures, modifying operations, and allows a user to enact bespoke replication policies. LD-NR inherited the structure of a general purpose library from NR [42] but NR only supports static replication policy. While NR has previously been extended (e.g., CNR in [33], verification in IronSync [94]), LD-NR is the first to provide dynamic replication.

Optimizations for NUMA Architectures Initial works focusing on NUMA often targeted servers with multiple sockets, but recent works target additional architectures, such as CXL-based architectures [208, 141] and processors with sub-NUMA (tiled) characteristics [150, 204, 191]. SWARM [164] explores replication protocols specialized for disaggregated memory accessed without cache coherence; LD-NR and DiNOS assume cache coherence. However, future research could include redesigning the shared log in LD-NR to function without cache coherence.

Optimizing Operating Systems for NUMA with Replication The synergy between operating system design, NUMA-awareness, and replication is clear. In Linux, AutoNUMA and extensions automatically optimize the NUMA-locality of processes based on dynamic factors [53, 83, 148]. Locus [182] and Harp [144] replicate files for availability and performance. Tornado [74] has a notion of clustered objects supporting partitioning and replication, with local representations created upon first access.

There has been specific focus on page table replication due to its importance for process performance. Carrefour [60] automatically replicates user pages in the kernel. Mitosis [1] and

vMitosis [178] replicate page tables on NUMA systems, with a focus on factors impacting the per-process decision to replicate or not, and provides a mechanism to replicate that tailored for page tables. Similarly, Wasp [186] provides automated policies for enabling or disabling replication of page tables. NrOS [33] uses replication on all NUMA nodes for page tables and other kernel data structures. In contrast to LD-NR, none of the systems allow for dynamically changing the number of replicas beyond full replication on all NUMA nodes or even processors, and no replication at all. Although we use DiNOS with page table replication as a use case, LD-NR is not operating system or page table specific.

6.9 Conclusion

We introduce a new method of replicating data structures across NUMA domains called Live-Dynamic Node Replication (LD-NR). LD-NR addresses a key limitation of prior work—a static inflexible set of replicas—to address the dynamic needs of modern systems (Section 6.2). LD-NR allows a user to adjust the number and location of replicas dynamically in response to changes in workload and system characteristics. We show that LD-NR improves performance of general data structures, and we use LD-NR to build a rackscale operating system called DiNOS that replicates internal data structures.

In future work, LD-NR could be used as a basis to develop new policies for replication in other contexts; LD-NR could also be further extended to disaggregated settings to provide fault tolerance for replicas and the shared log.

LD-NR Discussion LD-NR provides key mechanisms enabling dynamic replication, but some aspects regarding how best to configure dynamic replication remain unexplored. LD-NR provides a thread-to-replica matching scheme that prioritizes balancing threads that do not have local replicas across existing replicas. As NUMA balancing is not always optimal, thread balancing may not always be optimal. Development and analysis of thread matching policies are a potential direction for future work. Generally, optimization of LD-NR to particular NUMA topologies (cache line granularity, integration of bandwidth considerations, etc.) is an area of interest for future work.

DiNOS Discussion Particular to replication in a rackscale operating system, failure is another dimension to the memory performance tradeoff of replicated data structures. One avenue for future work is exploring that tradeoff through operating system replication policy that balances minimization of failure domains with maximizing performance. Another avenue is to change the tradeoff with respect to failure, by enabling LD-NR to recover from shared log or replica failure. Generally, the relationship between allocation, scheduling, migration, and defragmentation in relation to replication policy are an interesting direction for future work.

Chapter 7

Extensions to IRON: Efficiency, Expressivity, and Extensibility in a Close-to-Metal NPU Programming Interface

This chapter is a reprint, with minor edits, of our published paper in the 33rd IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM'25): *Efficiency, Expressivity, and Extensibility in a Close-to-Metal NPU Programming Interface* [105].

This work presents and evaluates extensions to IRON [158], a close-to-metal programming toolkit for neural processing units (NPUs). Unlike the works presented in previous chapters, the goals of this work are not to explicitly manage the spatial and temporal characteristics of the system. Instead, the extensions to IRON presented in this work allow expert programmers to more simply express finely optimized kernels targeting a class of NPUs.

7.1 Introduction

Accelerators such as neural processing units (NPUs) can provide increased performance and higher efficiency compared to general purpose processors; however, utilizing accelerator capabilities effectively is not always simple [190]. One approach is to construct high-level frameworks comprised of operations and compilers that abstract details of the accelerator architecture. Examples of this approach for NPUs are AMD Ryzen™ AI Software [15], Intel® OpenVINO™ [175], and Qualcomm® Neural Processing SDK [187]. These platforms enable developers to build and deploy machine learning models trained in common frameworks such as PyTorch and TensorFlow. A second approach is to construct low-level toolkits and libraries that support creation of bespoke, optimized designs using accelerator-specific characteristics. An example of this approach is IRON,

a close-to-metal, general purpose, open-source toolkit for performance engineers targeting AMD XDNA™ NPUs [151, 155, 158]. Each approach serves a different, but important, purpose.

In low-level toolkits, the programming interface contends with tension between providing tools for programmers to easily express their intent (designer efficiency) and exposing nuanced hardware capabilities required for clarity and precision (complexity). Although some tension is fundamental, specific trade-offs can be navigated through intentional interface design. This work details contributions to IRON that allow designers to represent designs plainly (*principle 1: efficiency*) without compromising designer ability to influence low-level decisions (*principle 2: expressivity*). These contributions also enable designers to create new tools to generate portions of designs (*principle 3: extensibility*). Central to our contributions is a new API implemented above the existing IRON programming interface. This API refines IRON programming constructs, provides a new interface supporting extensible tooling for placement (the matching of design constructs to physical resources), and includes a new supplemental data transformation library, also designed to support custom extensions. The new API coexists with existing IRON APIs, and is integrated into IRON as a core programming interface in the open source repository, <https://github.com/Xilinx/mlir-aie>.

Contributions to IRON are evaluated on the three principles of efficiency, expressivity, and extensibility using a set of 27 IRON designs, ranging from minimal (a data passthrough) to complex (streaming vision pipeline for edge detection). Demonstrating expressivity, it was possible to express all evaluated designs using the new API, with consistent performance between new and previous design implementations. Indicating increased efficiency for designers, we find a ~26% average reduction of single lines of code (SLOC) and average reduction across Halstead metrics [95] across designs. Illustrating extensibility, we craft a custom placement generator and a custom data transformation generator using the new API, and examine the utility of custom extensions. Twenty-four example IRON designs utilize the placement tool. Five of the designs, including a GEMM design with complex runtime data movement tiling patterns, use the data transformation generator.

The remainder of the paper is structured as follows. Section 7.2 provides an overview of NPUs. Section 7.3 summarizes IRON. Sections 7.4-7.7 present the design, implementation, and

evaluation of our contributions. Section 7.8 discusses related work and Section 7.9 contains concluding thoughts.

7.2 Architectural Overview of AMD XDNA™ NPUs

This section provides an overview of AMD XDNA™ NPU architecture (Section 7.2.1) and implications of the architecture for programming and optimization (Section 7.2.2).

7.2.1 AMD XDNA™ NPU Architecture

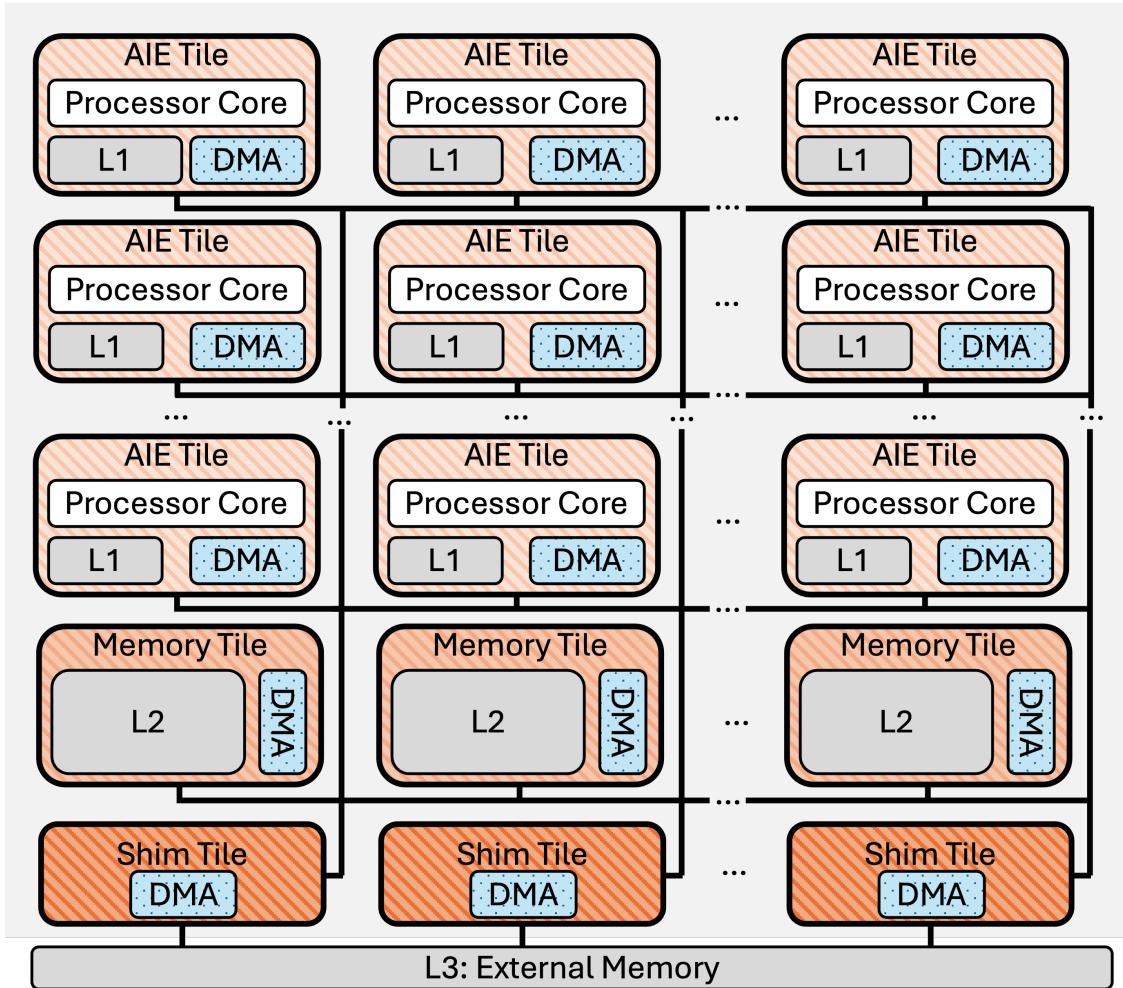


Figure 7.1: Simplified diagram of an AMD XDNA™ NPU such as found in Ryzen™ 7040 processors [192].

An AMD XDNA™ neural processing unit (NPU) consists of several types of tiles arranged

spatially in a two-dimensional grid connected by a streaming interconnect. NPUs are designed for power- and area-efficient inference: the AMD XDNA™ NPU found in Ryzen™ 7040 and 8040 SoCs provides >10 trillion operations per second (TOPS); using the XDNA™ programming stack for DNNs, models run with 4.3-33× performance per watt compared to the on-chip x86 processor [192]. The XDNA™ 2 architecture provides up to 50 TOPS [16].

Figure 7.1 is a simplified architectural diagram; comprehensive descriptions are available elsewhere [192]. AI Engine (AIE) tiles (also found in Versal™ architectures [4, 5]) contain a very long instruction word vector processor core with associated data memory, program memory, program counter, register file, and functional units [192]. Memory tiles provide additional on-chip SRAM scratchpad storage, while shim (or interface) tiles help manage external interfaces [192]. A streaming network-on-chip (NoC) supports point-to-point data transfers between tiles and can be configured for broadcast or selective multicast. Unlike architectures with hardware-managed caches, on NPUs on-chip data is buffered in scratchpads with software-managed data movement that is both explicit and deterministic [192]. Data movement accelerators (DMAs¹) orchestrate asynchronous data movement, and using buffer descriptors (BDs), support reorder, reshape, and repeat patterns.

7.2.2 Implications for Programming and Optimization

While explicitly controlling data movement is an optional optimization possible in most cache-based architectures, the architecture of the NPU **necessitates** explicit data movement for all NPU designs. Explicit data movement can increase the determinism of application latency and throughput and can reduce power consumption [192].

The NPU architecture provides rich features to optimize data movement. Utilizing L2 buffers in memory tiles can reduce the off-chip bandwidth required for some applications. Duplicating data using broadcast supported by the NoC can further reduce off-chip memory bandwidth requirements. On-the-fly data transformations supported by DMAs can reduce the number of DMA operations needed to represent a data movement pattern. This is useful for partitioning data according to

¹ DMA in this context refers to a physical component rather than the act of direct memory access.

an algorithm and for structuring data appropriately for vectorized instructions. These attributes enable a performance engineer or compiler to fine-tune data movement.

The memory hierarchy and arrangement of AIEs in an NPU is spatial. In an NPU design, compute and data must be placed at explicit locations on the device. NPU designs are also temporal. Synchronization is supported in hardware by locks, and the integration of those locks with DMAs operations [192]. To allow a programmer to fine-tune how and when actions occur on the device, the programmer must be able to express both spatial and temporal relationships [149].

7.3 IRON: A Performance Toolkit for NPUs

This section summarizes IRON, an open-source programming toolkit for AMD XDNATM NPUs [158]. IRON provides fine-grained control over spatial and temporal aspects of NPU designs, with an emphasis on data movement.

7.3.1 IRON Overview

The heart of IRON is a multi-level intermediate representation (MLIR) dialect, `mlir-aie` [158]. MLIR is a technology that supports reusable, modular, and domain-specific compiler infrastructure [131], and is a popular choice for targeting diverse architectures, such as accelerators ([79], [219], [221], etc.). `mlir-aie` defines primitives and abstractions for NPUs with a focus on 1) clear and complete representation of hardware capabilities, and 2) simplified representation of critical design components such as data movement, synchronization, and design structure. `mlir-aie` is the core of the IRON low-level programming interface for NPUs and defines an IR that compilers for higher-level dialects may utilize to target NPUs.

Abstractions in IRON provide programming convenience by capturing common patterns but do not hide fundamental characteristics of NPU designs including placement, explicit data movement, resource limitations, and complex access patterns supported by DMAs. The abstractions in IRON are not targeted at a specific application domain, although some patterns supported by IRON are useful for digital signal processing and machine learning.

The core of the IRON Python API is auto-generated from `mlir-aie` via MLIR Python bindings [159]. Bespoke extensions to the bindings hide MLIR implementation details such as conversion between common Python types and MLIR types. IRON-specific extensions are supplemented by `mlir-python-extras`, an open source library providing helpers broadly useful for cleanly and succinctly representing MLIR in Python [139]. The IRON Python API supports metaprogramming by allowing a programmer to mix traditional Python with Python-driven MLIR operation generation.

7.3.2 Lifecycle of an IRON Design

An IRON design can be written in Python using the IRON Python API or in MLIR using `mlir-aie`. If written in Python, the script will emit MLIR assembly. IRON designs use `core` blocks to specify the code to be executed on each processor core used by a design. Code in a `core` block may perform computations, call externally defined kernels, or both. External kernels are separately compiled using tools such as Peano [179]. `aiecc` is the primary compiler driver for IRON and takes as input MLIR assembly, along with any kernel object files, and produces two artifacts: a binary, which includes the programs to run as part of the design, and a sequence of instructions representing (re)configurations of the NPU to be loaded from a host processor during runtime. IRON interfaces with Xilinx® Runtime (XRT) to execute designs [238]. IRON designs are debugged using static analysis of generated MLIR and dynamic analysis through runtime tracing. IRON provides tools to configure tracing mechanisms supported by the hardware [6].

7.3.3 Outline of an IRON Design

Shown in the example IRON design in Figure 7.2a, all constructs that generate MLIR in an IRON design must be created within an MLIR `context` (line 1). At the end of a design, MLIR is emitted using the `module` constructed by the `context` (line 29). The first logical subsection of a design is a declaration of resources required (lines 4-9), including physical components (such as `tiles`) and IRON abstractions that are lowered to physical resources (such as `ObjectFifos`, which are described in Section 7.3.4). Types are expressed using NumPy [96] types (lines 4-5).

```

1 with mlir_mod_ctx() as ctx:
2   @device(AIEDevice.npu1_1col)
3   def device_body():
4     mty = np.ndarray[(MH, MW),
5       ↪ np.dtype[np.int32]]
6     tty = np.ndarray[(TH, TW),
7       ↪ np.dtype[np.int32]]
8     shm0 = tile(0, 0)
9     aie2 = tile(0, 2)
10    fi = object_fifo("in", shm0, aie2, 2, tty)
11    fo = object_fifo("out", aie2, shm0, 2,
12      ↪ tty)
13    @core(aie2)
14    def core_body():
15      for _ in range_(sys.maxsize):
16        a = fi.acquire(ObjectFifoPort.Consume,
17          ↪ 1)
18        b = fo.acquire(ObjectFifoPort.Produce,
19          ↪ 1)
20        for i in range_(TH):
21          for j in range_(TW):
22            b[i, j] = a[i, j] + 1
23        fi.release(ObjectFifoPort.Consume, 1)
24        fo.release(ObjectFifoPort.Produce, 1)
25
26    @runtime_sequence(mty, mty)
27    def sequence(dati, dato):
28      in_task = shim_dma_single_bd_task(fi,
29        ↪ dati, sizes=[1, 1, TH, TW],
30        ↪ strides=[0, 0, MW, 1])
31      out_task = shim_dma_single_bd_task(fo,
32        ↪ dato, issue_token=True, sizes=[1, 1,
33        ↪ TH, TW], strides=[0, 0, MW, 1])
34      dma_start_task(in_task, out_task)
35      dma_await_task(out_task)
36      dma_free_task(in_task)
37
38    print(ctx.module)

```

(a)

Figure 7.2: A matrix-scalar addition IRON design written without (7.2a) and with (7.2b) contributions.

core blocks are declared as decorated functions (line 11). The AIE acquires access to input and output buffers (lines 14-15) of tile dimension (TH, TW). Synchronization is controlled by the ObjectFifos with ownership of those buffers. The AIE performs element-wise scalar addition (lines 16-18) before releasing each buffer (lines 19-20). Loops are denoted with `range_`, a Python wrapper for a looping construct provided by the MLIR `scf` dialect [197].

The runtime sequence (lines 22-28) is used to send a sub-matrix (tile) of data, specified using

```

1 mty = np.ndarray[(MH, MW), np.dtype[np.int32]]
2 tty = np.ndarray[(TH, TW), np.dtype[np.int32]]
3 fi = ObjectFifo(tty)
4 fo = ObjectFifo(tty)
5
6 def core_fn(of_in, of_out):
7   a = of_in.acquire(1)
8   b = of_out.acquire(1)
9   for i in range_(TH):
10     for j in range_(TW):
11       b[i, j] = a[i, j] + 1
12   of_in.release(1)
13   of_out.release(1)
14 my_worker = Worker(core_fn,
15   ↪ fn_args=[fi.cons(), fo.prod()])
16
17 rt = Runtime()
18 with rt.sequence(mty, mty) as (dati, dato):
19   rt.start(my_worker)
20   rt.fill(fi.prod(), dati, sizes=[1, 1, TH,
21     ↪ TW], strides=[0, 0, MW, 1])
22   rt.drain(fo.cons(), dato, sizes=[1, 1, TH,
23     ↪ TW], strides=[0, 0, MW, 1], wait=True)
24 print(Program(NPU1Col1(),
25   ↪ rt).resolve_program(SequentialPlacer()))

```

(b)

sizes and strides calculated from the matrix and tile dimensions, to an `ObjectFifo` with a shim tile endpoint. The output tile is similarly received. The runtime sequence is a mix of declarations (`shim_dma_single_bd_task`), asynchronous operations (`dma_start_task`), and synchronous operations (`dma_await_task`).

7.3.4 Data Movement in IRON

As outlined in Section 7.2, a key architectural component of NPUs is explicit data movement. IRON supports data movement at varying levels of abstraction, but the primary abstraction is the `ObjectFifo`. An `ObjectFifo` is a circular buffer with a configurable amount (depth) of objects and has theoretical roots in dataflow theory [62]. Objects belonging to an `ObjectFifo` share shape and data type and are allocated at the discretion of the compiler. Accesses to `ObjectFifo` objects are mediated with locks managed by the `ObjectFifo`, and access is granted and revoked with `acquire` and `release` operations.

The `ObjectFifo` is designed to support common patterns such as broadcast, sliding windows (if n buffers are acquired, a call to `acquire(n + 1)` will acquire just one more; similarly, if m buffers are acquired, a call to `release(1)` will release just the first buffer, leaving the last $m - 1$ buffers acquired), pipeline balancing (the depth of `ObjectFifos` in a critical path can be increased or decreased), splitting and joining data in L2 (using a `link` operation to join two `ObjectFifos` over a shared memory tile endpoint), and L2 buffering (also using a `link`).

7.4 Efficiency, Expressivity, and Extensibility

Extensions to IRON were crafted in order to increase designer efficiency while maintaining designer expressivity and enabling extensibility.

Efficiency We seek to enable efficient expression of designs by allowing features of a design to be minimally expressed when that feature has not been selected for tuning by the designer. Designs are multifaceted, with many interesting aspects a performance engineer can choose to focus on (placement, dataflow, runtime operations, data tiling, kernels, etc.). Not all designs require a

high level of detail in all aspects. Our contributions aim to allow a designer to efficiently express aspects of a design that are not the designers focus.

Expressivity We seek to gain efficiency – but not at the cost of expressivity. Our contributions to IRON should allow a programmer to express the same constructs as IRON, and designs written using new contributions to IRON should exhibit the same characteristics (including performance) as IRON designs expressed using existing IRON APIs.

Extensibility We seek to integrate pathways into IRON allowing performance engineers, who are not necessarily also compiler engineers, to generate portions of IRON designs using custom algorithms and tools. IRON provides full control over many aspects of a design, which makes it especially suited for use as an experimental sandbox for automation tools being developed for higher-level compiler workflows. Our contributions aim to enable integration of custom extensions for specific aspects of interest such as design space exploration, placement, tiling, and structuring of runtime operations.

7.5 Contributions to IRON

Contributions to IRON are organized around a new API (Section 7.5.0.1) which includes a refined `ObjectFifo` API (Section 7.5.0.2) and new constructs for existing IRON design components (Section 7.5.0.3). The new API supports an extensible placement interface (Section 7.5.0.4) and includes a library for generating on-the-fly data transformations (Section 7.5.0.5).

7.5.0.1 Creating Distance from MLIR The structure of an IRON design is inherited from the structure required by MLIR. There are techniques to control and hide the `context` block (such as those provided by `mlir-python-extras` [139]), but if an MLIR operation is created when the corresponding Python object is created, then the Python objects are subject to the same rules as the corresponding MLIR operations. MLIR requires an operation be complete (i.e., all important information known at construction), and be instantiated according to proper placement in the MLIR `context`. It is possible to insert and reorder MLIR operations in Python. However, we

believe it would add complexity and make human comprehension of the IRON programming stack difficult if these techniques were used in both the compiler driver and the Python programming front-end, so we leave these techniques to `aiecc`.

The constraints of MLIR can be at odds with the goal of increasing the usability of IRON. For instance, one desirable improvement to IRON is reduction of duplicated information. An example of duplicated information in IRON designs are the placements of an `ObjectFifo` endpoint and the `core` block that uses the endpoint (Figure 7.2a lines 8-9, 11); the placement location could be provided once instead of twice. However, to construct the `core` block and `ObjectFifo`, both Python constructors must provide a placement at the time of instantiation – requiring information duplication.

To relax the requirements of position and information completeness, we create a new top-level IRON Python API. Python classes in the new API are inheritors of a `resolvable` interface and defer creation of MLIR operations until their `resolve` function is called. If the Python class does not contain enough information to resolve to an MLIR operation, `resolve` fails. Figure 7.2b shows the design in Figure 7.2a rewritten using the new API and associated contributions.

7.5.0.2 ObjectFifo API Deferred resolution is used to refine the `ObjectFifo` API. `ObjectFifo` declarations (Figure 7.2a lines 8-9) can be succinctly rewritten (Figure 7.2b lines 3-4). The default depth of `ObjectFifos` is set to 2 (due to the prevalence of ping-pong buffers); the `ObjectFifo` name (which becomes the MLIR identifier) is auto-generated; and the `ObjectFifo` endpoints are inferred.

When an `ObjectFifo` is given to an operation or construct for use, an `ObjectFifoHandle` is generated by calling `prod` for a producer handle or `cons` for a consumer handle (Figure 7.2b line 14). `prod` always return the same object, as an `ObjectFifo` can only have one producer. Multiple calls to `cons` generate multiple consumer handles, expressing a broadcast. Since a handle knows if it is a producer or consumer, there is no need to specify the `ObjectFifoPort` in `acquire` and `release`, further reducing redundancy (Figure 7.2a lines 14-15, 19-20 versus Figure 7.2b lines 7-8,

12-13).

Instead of porting `link` to the new API, the `ObjectFifoHandle` class provides `forward`, `split`, and `join` methods which allow data to be forwarded over a tile (useful for L2 buffering), split across a tile (useful for splitting data staged in L2 in one L2-L3 flow to conserve channels), or joined across a tile (same use, but in reverse). At resolution, these methods ensure the generation of necessary `ObjectFifos` and `links`.

7.5.0.3 Worker, Runtime, and Program Constructs We introduce a new construct, the `Worker` (Figure 7.2b line 14). Construction of a `Worker` mirrors that of a thread in many multi-threading libraries. A `Worker` takes a routine to run (`core_fn`) and the context (`fn_args`) needed to run it. Familiarity is one motivation for `Workers`; a second motivation is the explicit separation of task definition (i.e., the `core_fn`, Figure 7.2b line 6) from the configuration for running the task (i.e., arguments to a `Worker`, Figure 7.2b line 14). This pattern provides a clear path for metaprogramming as `fn_args` can contain arbitrary Python values to customize the MLIR generated by a `core_fn`. This type of metaprogramming is often used to adapt computing tasks to variable datatypes, dimensions, or distributions of data.

A new `Runtime` construct represents the runtime sequence (Figure 7.2b lines 16-20). A runtime sequence does not have the freedom to represent arbitrary computations, which can be an unclear distinction to IRON programmers as it is visually similar to the imperative block executed by AIEs in IRON designs. Operations supported by a `Runtime` include `start` to indicate a worker should be configured for execution (Figure 7.2b line 18), `fill` to use a DMA operation to populate an `ObjectFifo` with data from a L3 buffer (Figure 7.2b line 19), and `drain` to collect data from an `ObjectFifo` for a L3 buffer (Figure 7.2b line 20). A last method, `inline_ops`, allows expert designers to insert a Python function generating arbitrary MLIR ops into the runtime sequence; this is roughly analogous to inline assembly in a C program. This technique supports bespoke configuration, and will be used in future work to fully support tracing configurations in the new API.

To compose these components into a design, the new `Program` construct creates a design from a `Runtime` applied to a `Device`. MLIR for the design is produced by `resolve_program` (Figure 7.2b line 21).

7.5.0.4 Interface for Placement The new API supports manually placed programs, meeting the goal of expressivity. However, if a programmer does not provide enough placement information, `resolve_program` will fail. Without additional extension, this requires all IRON designs be fully placed. To meet the goals of both efficiency and extensibility, we construct an interface supporting placement generation.

Objects in the new IRON API that are `Placeable` provide a `place` method that takes a placement tile as an argument. `resolve_program` optionally takes a `Placer` argument (Figure 7.2b line 21). A `Placer` must implement `make_placement`. During the execution of `resolve_program`, `make_placement` is called, allowing the `Placer` to invoke `place` on all `Placeable` design components before MLIR is generated. The placement interface supports partially placed designs as well as placement hints `AnyShim`, `AnyMemTile`, and `AnyComputeTile`. Future work includes support for tagging `Placeable` components with programmer-defined attributes as a method of supporting arbitrary placement hints.

The interface for placement fulfills the goals of efficiency (designers uninterested in manual placement may use and reuse placement generators), expressivity (changing `Placers` or specifying partial placement for a given design is straightforward), and extensibility (a user has a well-defined mechanism to construct new `Placers`). The `SequentialPlacer` in Figure 7.2b line 21 is an example `Placer` and is discussed further in Section 7.7.3.1. The `Placer` API is written in Python but common language-binding tools can be used to wrap existing placement and design space exploration tools written in other languages into `Placers`.

7.5.0.5 Primitives for on-the-fly Data Transformations

One point of complexity stands out in Figure 7.2b: the sizes and strides (lines 19-20). Reason-

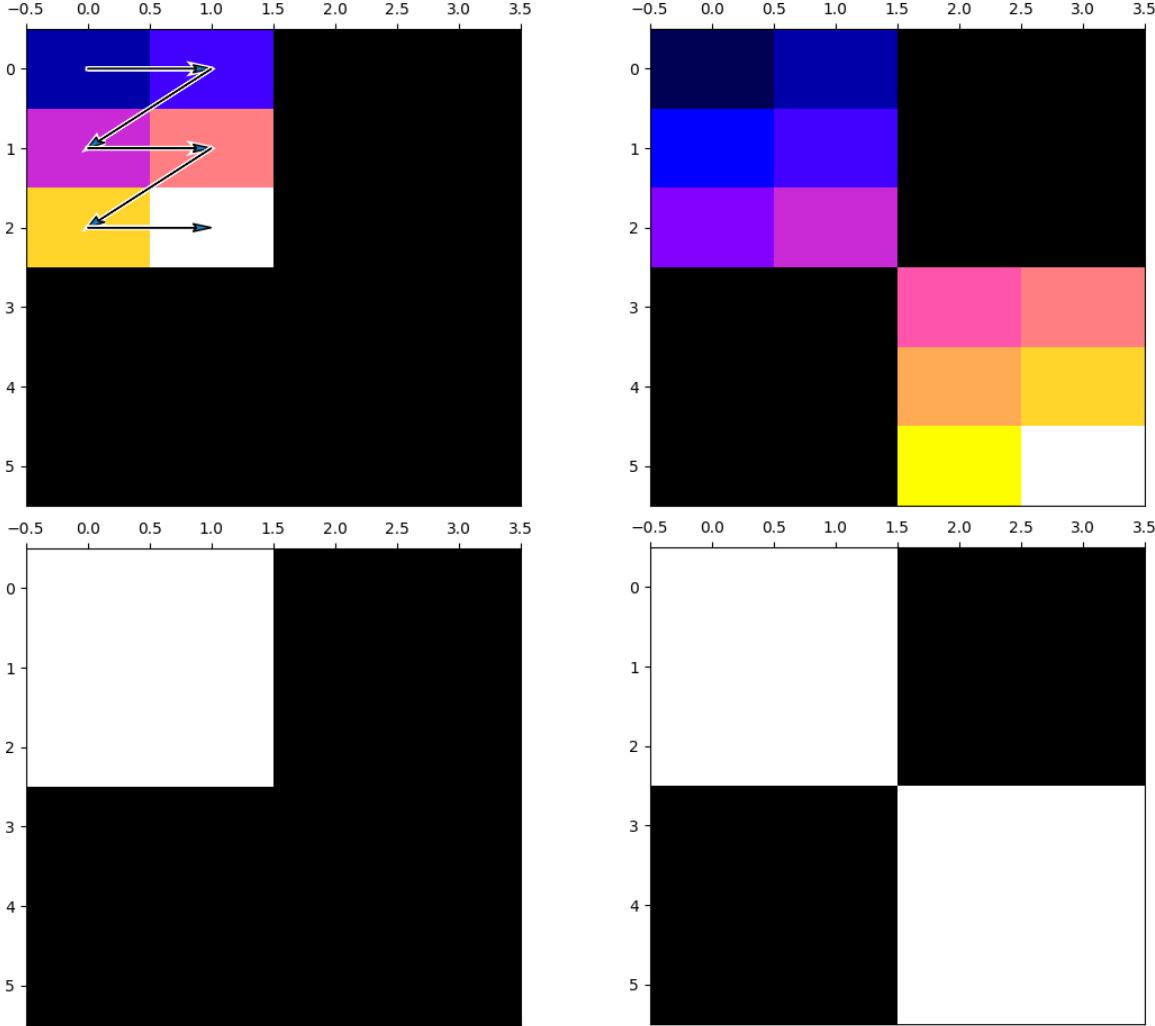


Figure 7.3: Heat maps generated by `taplib` representing access order (top) and access count (bottom).

ing about transformations from just looking at sizes and strides can be difficult. To address this, we create `taplib`, a library containing two primitive abstractions for expressing and reasoning about transformations. A `TensorAccessPattern` (`tap`) is constructed from a set of tensor dimensions, sizes, strides, and an offset into the tensor. A `TensorAccessSequence` (`tas`) is a collection of `tap`s whose tensor dimensions match. Consider an example where a programmer desires to tile a 6×4 tensor into four 3×2 non-overlapping row-major tiles and select the upper-left and lower-right. Using `taplib`, this can be expressed as:

```
tap00 = TensorAccessPattern((6, 4), offset=0, sizes=[1, 1, 3, 2], strides=[0, 0, 4, 1])
```

```
tap11 = TensorAccessPattern((6, 4), offset=14, sizes=[1, 1, 3, 2], strides=[0, 0, 4, 1])
taps0 = TensorAccessSequence.from_taps([tap00, tap11])
```

`taplib` provides mechanisms to reason about `ttaps` and `tases`, including visualizations and animations for 1- and 2-dimensional tensors (Figure 7.3). `taplib` also supports tools for programmatic analysis. The graphics shown in Figure 7.3 were produced using `taplib` access maps. An access count map is a tensor that contains, in each element, the number of times that a `tap` or `tas` accesses the corresponding element in the original tensor. An access order map is a tensor that contains, in each element, the element-wise enumeration of the order of accesses defined by the `tap` or `tas`. Access maps are useful for catching bugs, as a logical error in `tap` definition can be formalized as a `tap` whose access pattern, in count or order, does not match the intention of the programmer. The following code snippets demonstrates verification of `tap00` and `taps0` characteristics using access maps:

```
# Num accessed: Count starts at 0, so highest is 5
assert tap00.access_order().max() == 3*2 - 1
# Count of elements accessed by tap00
assert tap00.access_count().sum() == 3*2
# Num accessed: Highest count for two tiles is 11
assert taps0.access_order().max() == 2*(3*2) - 1
# The tas does not access any element more than once
assert taps0.access_count().max() == 1
```

We define a new concept of equivalence for access patterns using access maps. The strict definition of equivalence requires the tensor dimensions, sizes, strides, and offset be equivalent between two `taps`. We use the term *access equivalent* to describe `taps` which generate identical access order and count maps. Access equivalence is useful in the context of IRON because DMA units within an NPU have differing constraints on the maximum dimension of sizes and strides supported, as well as the maximum size (in bits) supported in each dimension. Thus, two access patterns may be strictly unequal, but be access equivalent, where one yields a valid IRON design and the other does not. `taplib` allows a programmer to express these intricacies precisely. The new IRON API integrates `taplib` (as an example, the `Runtime` `fill` and `drain` methods take a `tap` in lieu of sizes and strides).

7.6 Implementation

Our contributions are implemented in Python, with the new top-level API implemented in $\sim 1,400$ lines of code (LOC), and `taplib` implemented in an additional ~ 560 LOC. The effectiveness of the contributions are largely agnostic to the implementation mechanism; some contributions are suitable for integration into `mlir-aie` as a matter of future work.

7.7 Evaluation

Our contributions aim to increase designer efficiency (evaluated in Section 7.7.1) while maintaining the same level of expressivity (evaluated in Section 7.7.2), and to provide clearly defined mechanisms for further extension (illustrated in Section 7.7.3). Sections 7.7.2 and 7.7.3 compare 27 designs (outlined in Table 7.1) before and after contributions to IRON.

7.7.1 Efficiency

In this section, we decompose the concept of efficiency into two measurements: single lines of code (SLOC) representing brevity and Halstead metrics representing effort. These metrics are imperfect, but we posit the number and variety of evaluated designs imply these results capture representative trends.

7.7.1.1 Brevity SLOC is measured using `pygount` [184] and `radon` [189]. All designs were formatted programmatically using `black` [130]. Figure 7.4 shows percent decrease between design versions while Table 7.1 records the SLOC for each design.² Many designs include some level of argument parsing and verification (identical between both versions), and this code slightly dilutes the percentages. Even so, designs written using our contributions to IRON require 25.53% fewer SLOC on average, with an average decrease of 31 SLOC per design.

7.7.1.2 Effort We use Halstead metrics, a common suite of metrics for evaluating soft-

² MSAdd SLOC is higher in Table 7.1 than in Figure 7.2 because the full design includes additional metaprogramming and configuration.

Name	Description	SLOC before	Be- fore/After
Block Designs			
Copy $\times 3$	Pass data unchanged through the NPU using 1) DMAs only, 2) DMAs + kernel, or 3) DMAs + external kernel.	42/29 52/42	48/39
MTranspo- se	Transpose a matrix using DMAs.	36/28	
VRe- duce $\times 3$	Element-wise vector reduction (<code>add</code> , <code>max</code> , or <code>min</code>).	50/26 50/26	50/26
VSOp $\times 2$	Element-wise vector-scalar operation (<code>add</code> or <code>mul</code>).	49/36	64/55
VVOp $\times 5$	Vector-vector operations (element-wise <code>add</code> , <code>mul</code> , and <code>mod</code> ; vectorized <code>addKern</code> and <code>mulKern</code>).	56/43 56/43 94/76	56/43 94/75
MSAdd	Scalar addition to sub-matrix in matrix	62/43	
MVAdd	Tiled, row-wise, matrix-vector addition.	55/44	
MVMul	Matrix-vector multiplication.	105/69	
VSoft- Max	Softmax implemented using a lookup table approximation, applied to all vector elements.	79/58	
VReLU	Rectified Linear Unit, $ReLU(x) = \max(0, x)$, a common DNN activation function.	76/60	
Conv2d $\times 2$	A 2-dimensional convolution operation; second design fuses kernel with ReLU.	89/77	86/73
Advanced Designs			
GEMM	Matrix-matrix mult. Each matrix is multiplied in blocks with dimensions matching NPU intrinsics, and subtiles distributed to AIEs.	308/279	
BBlock	Bottleneck block used in DNNs such as ResNet [97]. Implemented as a 3 stage pipeline.	332/245	
ResNet- Conv2x	Conv2D_X layers of the BBlocks used in ResNet [97] composed of three BBlocks.	623/414	
Color- Detect	Detect two colors in an image, structured as a 3 stage pipeline.	163/146	
Edge- Detect	Detect edges in an image using a 4 stage pipeline.	198/177	
Color- Thresh.	Tiled data-parallel color threshold detection.	169/81	

Table 7.1: Twenty-seven designs (21 straight-forward block, 6 optimized advanced) used to evaluate contributions to IRON.

ware [9, 26, 86, 95], to analyze the example designs. Halstead metrics use attributes extracted from source code (number of operators, number of operands, and counts of both) to define concepts such as vocabulary and effort [93]. Across all designs for all negative Halstead metrics (calculated using `radon` [189]), the average of the metric is lower for designs written post-contribution to IRON

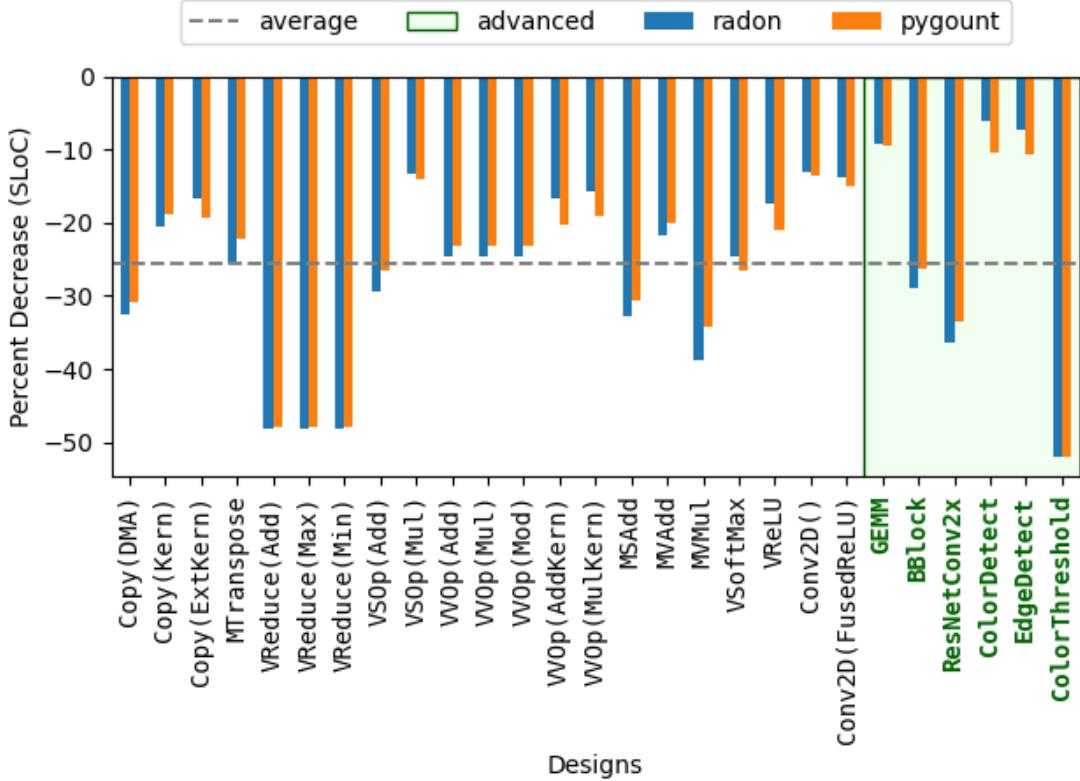


Figure 7.4: Average percent decrease of SLOC per design.

than pre-contribution to IRON. Figure 7.5 shows the calculations for vocabulary (a measure based on unique operators and operands counted in the source code) and effort (a metric designed to reflect the effort it takes to interact with the program as a writer, maintainer, or reader) [95]. The average reduction for vocabulary is 4.59 (an average percent decrease of 23.32%) while the average reduction in effort is 179.97 (an average percent decrease of 24.82%).

7.7.2 Expressivity

This section evaluates whether IRON with contributions is as expressive as IRON before contributions. The example designs, particularly the advanced designs, cover key features and patterns supported by IRON. In Table 7.2, *L2Mem* refers to the use of memory tiles while *SharedMem* refers to the ability of AIEs to access the L1 memory of another AIE (conserving DMA channels). *Tiling* refers to on-the-fly data transformations. *Broadcast*, *Split/Join*, *sliding Window*, and *Skip*

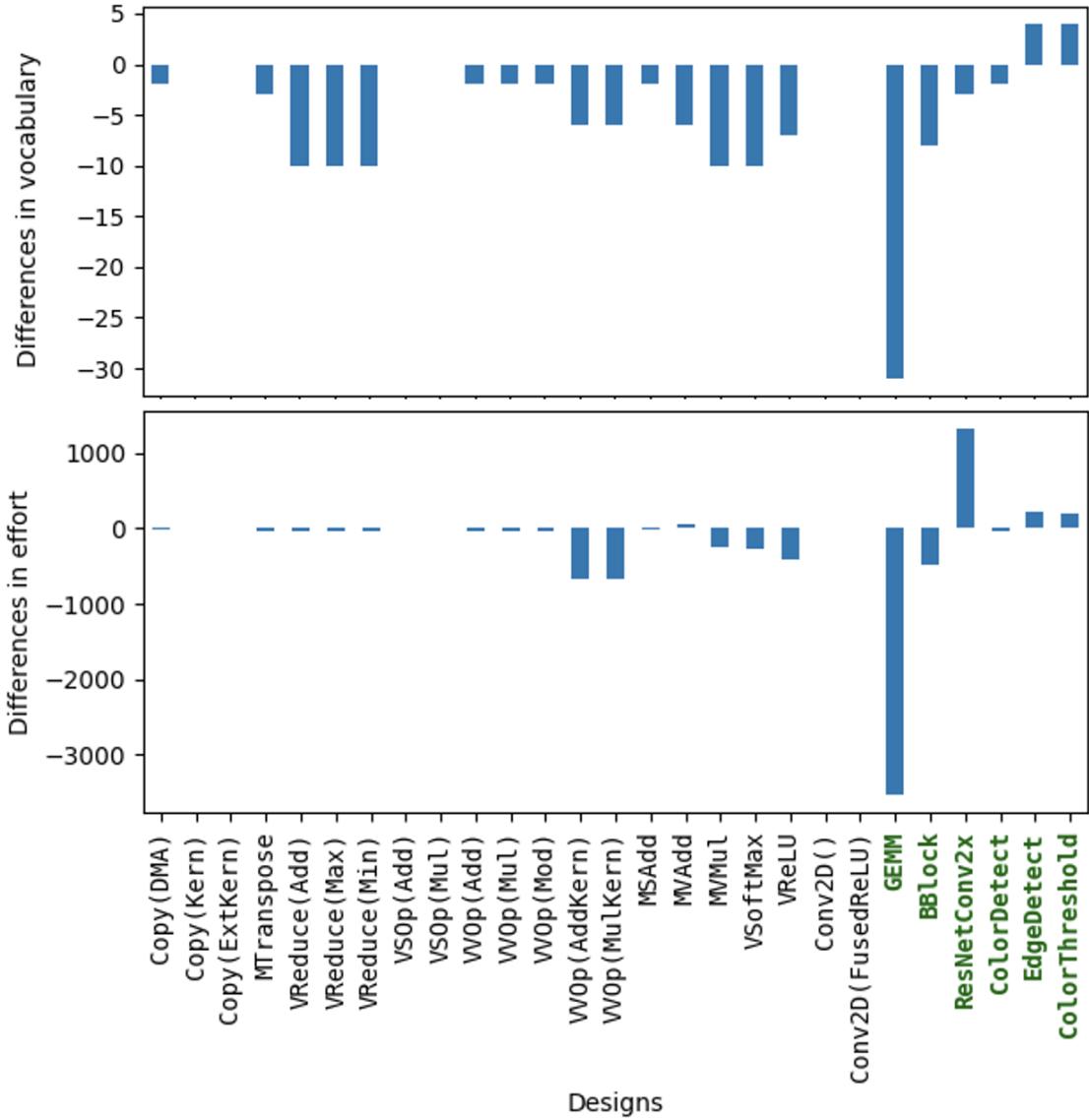


Figure 7.5: The difference between Halstead vocabulary (top) and effort (bottom) per design.

connections are features of data movement. *Pipeline* refers to a design which is task-parallel; other designs are data-parallel. *Metaprog* refers to metaprogramming in the design. *NPUCols* reports the number of NPU columns required by the design. All example designs written in IRON post-contribution produce the same (correct) output as the corresponding original designs; this is the first test of correctness. For further analysis, we employ static analysis of MLIR and measure performance data.

To verify both versions of the same design utilize the features of the NPU identically, the

	<i>L2Mem</i>	<i>SharedMem</i>	<i>Tiling</i>	<i>Broadcast</i>	<i>Split/Join</i>	<i>Window</i>	<i>Skip</i>	<i>Pipeline</i>	<i>Metaprog.</i>	<i>NPUCols</i>
GEMM	✓	✓	✓	✓				✓	1-4	
BBlock	✓	✓	✓	✓	✓	✓	✓		1	
ResNet- Conv2x	✓	✓	✓	✓	✓	✓	✓	✓	✓	3
Color- Detect	✓		✓			✓	✓		1	
Edge- Detect	✓		✓		✓	✓	✓		1	
Color- Thresh.	✓	✓	✓					✓	1	

Table 7.2: Features of advanced designs.

generated MLIR from each version is compared. A naive comparison shows generated MLIR is different for all designs. However, most of these differences are in the ordering of declarations (e.g., tiles declared in different order) and so should not affect the functionality of the design. Controlling for the order of declarations, 20 of 27 designs generate identical MLIR. Three designs (`GEMM`, `MVAdd`, `MTranspose`) use different runtime DMA transfer sizes and strides, but the corresponding access patterns expressed are access equivalent and so are representative of functional equivalency. Four designs (`ColorDetect`, `ResNetConv2x`, `BBlock`, `EdgeDetect`) use one or more `ObjectFifos` with broadcast, and the list of recipients is reordered; reordering does not lead to any differences in functionality.

To corroborate the static analysis, the average latency (100 warm up iterations followed by 1000 iterations) of each design is recorded. The average percentage difference between the versions of each design is approximately 3.36%. 11 of 27 designs using IRON with contributions show (slightly) higher latency than designs using IRON without contributions. The percent difference between the sum of average latencies across designs for each version is less than 0.09%, which is attributed to system noise.

7.7.3 Extensibility

Contributions to IRON include two interfaces designed for extension: the placement interface and `taplib`. We target manual placement and data transformation specification for extension because they can be tedious and error prone tasks, so designers can benefit both from programmability (to algorithmically generate solutions) and the ability to ignore these aspects of a design when desired. In this section, we validate that it is possible to construct extensions using these interfaces.

7.7.3.1 Defining a Placer The purpose of this exercise is to show the `Placer` interface a) can be used to generate functional designs, and b) successfully allows a user to create a `Placer` using only knowledge of IRON and NPU architecture. To this end, we create a `SequentialPlacer`, implemented in just 64 SLOC, that uses only constructs from the new IRON API. The `SequentialPlacer` assigns AIEs in a grid-like pattern, and then assigns memory then shim tiles such as to keep `ObjectFifo` endpoints in the same column. This `Placer` is rudimentary; it respects dimensional constraints of the AIE-array but may yield invalid placements due to other resource constraints. Of the 27 example designs, `SequentialPlacer` is used by 24 for full placement; one for partial placement (`BBlock`); and two are fully hand-placed (`ResNetConv2x`, `GEMM`) as the `SequentialPlacer` yielded invalid designs due to over-allocation of resources. This illustrates that the `Placer` API successfully allows new placers to be constructed and applied to IRON designs.

7.7.3.2 TensorAccessPattern Generators The true utility of `taplib` primitives is that they may be generated to replace hand-crafted sizes and strides. The intent of a programmer can be difficult to ascertain by viewing the arithmetic used to compute series of sizes and strides. This is problematic for readability and maintainability, and puts the burden for creating such arithmetic on the programmer for every new design. To illustrate how `taps` and `tases` may be generated for a class of transformations, we implement an example generator, `TensorTiler2D`. `TensorTiler2D` generates `tases` based on tile size, column- or row-wise element access within a tile, groupings of

tiles (e.g., access more than one tile in a single `tap`), column- or row-wise access of tiles within the group, group repeats (e.g., access the same pattern more than once within the same `tap`), and group steps (e.g., non-contiguous tile groups of specific step sizes in vertical and horizontal directions). `TensorTiler2D` is implemented in only 277 SLOC and is utilized by five designs (`MSAdd`, `MVAdd`, `MVMul`, `GEMM`, `MTranspose`). We largely attribute the differences in vocabulary and effort in the `GEMM` design (Figure 7.5) to the `TensorTiler2D`, which greatly reduces the amount of arithmetic in the `Runtime` of this design.

7.8 Related Work

A class of frameworks targeting NPUs – including AMD Ryzen™ AI Software [15], Intel® OpenVINO™ [175], and Qualcomm® Neural Processing SDK [187] – provide a higher level of abstraction than IRON and focus specifically on machine learning. CUDA [77], OpenCL [163], and HLS [176] are closer to the abstraction level of IRON, but IRON specifically targets accelerators with AIEs. Many works with Python front-ends [45] feature runtime code generation (RCG), often through just-in-time compilation [122, 129, 219]. Others provide domain-specific languages (DSLs) [128] or embedded DSLs (eDSLs) [133, 137, 156]. This work focuses on efficient representation of optimized designs rather than RCG. `Worker` core functions can be considered as written in an eDSL provided by `mlir-python-extras` [139]. Our contributions to IRON instead focus on other aspects of designs, such as data transformations and refinement of constructs representing design structure. [136, 138] proposes a programming model for end-to-end management of AIE architectures, sharing some of our goals (support for performance tuning and metaprogramming) but priorities differ. Our work does not tackle RCG while [136, 138] does not focus on extensible interfaces or measures of designer efficiency. [237] illustrates the complexity of DMA transformations in a scratchpad-based architecture; the tools for expressing DMA transformations provided by `taplib` may be useful for further studies in this area. The use of heat maps and visualization tools for performance engineers is described in [198] and uses similar terminology (access patterns) and techniques (tensor-based heat maps) [198]. However, that work focuses on analyzing the relation-

ship between accesses and movement in cache-based architectures, while access patterns supported by `taplib` in the context of IRON are configurations for data movement. Simulation and modeling described by [198] could be implemented for IRON designs as a future extension.

7.9 Conclusion

This work describes, implements, and analyzes contributions to IRON that increase designer efficiency without compromising expressivity, and provide mechanisms for additional extension. The contributions successfully change source code characteristics of 27 example IRON designs, while preserving functionality, through use of a new API with refined programming constructs and extensible interfaces. We observe an average reduction of $\sim 26\%$ SLOC across example designs due to these changes. While the majority of contributions are specific to IRON, this work provides a data point indicating that fine-tuning the design of programming interfaces for accelerators can have a large effect on characteristics of designs even without changing the abstraction level of the interface.

Bibliography

- [1] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20, page 283–300, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378468. URL <https://doi.org/10.1145/3373376.3378468>.
- [2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), pages 419–434, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7. URL <https://www.usenix.org/conference/nsdi20/presentation/agache>.
- [3] Marcos K. Aguilera, Emmanuel Amaro, Nadav Amit, Erika Hunhoff, Anil Yelam, and Gerd Zellweger. Memory Disaggregation: Why Now and What Are the Challenges. SIGOPS Oper. Syst. Rev., 57(1):38–46, jun 2023. ISSN 0163-5980. doi: 10.1145/3606557.3606563. URL <https://doi.org/10.1145/3606557.3606563>.
- [4] Sagheer Ahmad, Sridhar Subramanian, Vamsi Boppana, Shankar Lakka, Fu-Hing Ho, Tomai Knopp, Juanjo Noguera, Gaurav Singh, and Ralph Wittig. Xilinx First 7nm Device: Versal AI Core (VC1902). In 2019 IEEE Hot Chips 31 Symposium (HCS), Cupertino, CA, USA, August 18-20, 2019, pages 1–28. IEEE, 2019. doi: 10.1109/HOTCHIPS.2019.8875639. URL <https://doi.org/10.1109/HOTCHIPS.2019.8875639>.
- [5] AI Engine. AI Engine: Meeting the Compute Demands of Next-Generation Applications. <https://www.amd.com/en/products/adaptive-socs-and-fpgas/technologies/ai-engine.html>. Accessed 2025-01-05.
- [6] AIE-ML Trace and Profiling. AIE-ML Trace and Profiling. <https://docs.amd.com/r/en-US/am020-versal-aie-ml/AIE-ML-Trace-and-Profiling>. Accessed 2025-01-05.
- [7] Istemci Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance Serverless Computing. In 2018 USENIX Annual Technical Conference (USENIX ATC 18), pages 923–935, Boston, MA, July 2018. USENIX Association. ISBN 978-1-939133-01-4. URL <https://www.usenix.org/conference/atc18/presentation/akkus>.

- [8] Z. Al-Ali, S. Goodarzy, E. Hunter, S. Ha, R. Han, E. Keller, and E. Rozner. Making Serverless Computing More Serverless. In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pages 456–459, July 2018. doi: 10.1109/CLOUD.2018.00064.
- [9] Mahmoud Alfadel, Armin Kobilica, and Jameleddine Hassine. Evaluation of Halstead and Cyclomatic Complexity Metrics in Measuring Defect Density. In 2017 9th IEEE-GCC Conference and Exhibition (GCCCE), pages 1–9, 2017. doi: 10.1109/IEEEGCC.2017.8447959.
- [10] Alibaba Cluster Trace Program. Alibaba Cluster Trace Program. <https://github.com/alibaba/clusterdata>.
- [11] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pages 469–482, Boston, MA, March 2017. USENIX Association. ISBN 978-1-931971-37-9. URL <https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard>.
- [12] 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 EuroSys '20: Proceedings of the Fifteenth European Conference on Computer Systems, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368827. doi: 10.1145/3342195.3387522. URL <https://doi-org.colorado.idm.oclc.org/10.1145/3342195.3387522>.
- [13] Amazon Elastic Container Service. Amazon Elastic Container Service. <https://aws.amazon.com/ecs/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=desc>.
- [14] AMD. Socket SP3 Platform NUMA Topology for AMD Family 19h Models 00h–0Fh. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/design-guides/56795_1_00-PUB.pdf, 2021. Accessed 2025-03-30.
- [15] AMD Ryzen AI Software. AMD Ryzen™ AI Software. <https://www.amd.com/en/developer/resources/ryzen-ai-software.html>. Accessed 2025-01-05.
- [16] AMD Zen 5. AMD Unveils Next-Gen “Zen 5” Ryzen Processors to Power Advanced AI Experiences. <https://www.amd.com/en/newsroom/press-releases/2024-6-2-amd-unveils-next-gen-zen-5-ryzen-processors-to-p.html>, June 2024. Accessed 2025-01-05.
- [17] Cristiana Amza, Alan L. Cox, Shandya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. IEEE Computer, 29(2):18–28, February 1996.
- [18] Apache OpenWhisk. Apache OpenWhisk. <https://github.com/apache/openwhisk>.
- [19] Apache Software Foundation. Hadoop. <https://hadoop.apache.org>. Accessed 2010-02-19.
- [20] AWS Lambda. AWS Lambda. <https://aws.amazon.com/lambda/>. Accessed 2025-07-10.

- [21] AWS Lambda enables functions that can run up to 15 minutes. AWS Lambda Enables Functions That Can Run Up to 15 Minutes. <https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/>.
- [22] AWS Step Functions. AWS Step Functions. <https://aws.amazon.com/step-functions/>. Accessed 2023-12-10.
- [23] Azure Durable Functions. What are Durable Functions? <https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview>? Accessed 2023-12-10.
- [24] Azure Functions. Azure Functions. <https://azure.microsoft.com/en-us/services/functions/>. Accessed 2025-07-10.
- [25] Azure Kubernetes Service (AKS). Azure Kubernetes Service (AKS). <https://azure.microsoft.com/en-us/services/kubernetes-service/#overview>.
- [26] C. T. Bailey and W. L. Dingee. A Software Study using Halstead Metrics. SIGMETRICS Perform. Eval. Rev., 10(1):189–197, January 1981. ISSN 0163-5999. doi: 10.1145/1010627.807928. URL <https://doi.org/10.1145/1010627.807928>.
- [27] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John Davis. CORFU: A Shared Log Design for Flash Clusters. In 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12). USENIX, April 2012. URL <https://www.microsoft.com/en-us/research/publication/corfu-a-shared-log-design-for-flash-clusters/>.
- [28] Amnon Barak and Richard Wheeler. MOSIX: An Integrated UNIX for Multiprocessor Workstations. International Computer Science Institute, 1988.
- [29] Antonio Barbalace, Binoy Ravindran, and David Katz. Popcorn: A Replicated-kernel OS based On Linux. In Proceedings of Ottawa Linux Symposium (OLS), 2014.
- [30] 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 ACM Symposium on Operating Systems Principles (SOSP), pages 29–44, 2009. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629579. URL <http://doi.acm.org/10.1145/1629575.1629579>.
- [31] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATC '05, USA, 2005. USENIX Association.
- [32] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-specific Memory Coherence. In ACM Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 168–176, March 1990.
- [33] Ankit Bhardwaj, Chinmay Kulkarni, Reto Achermann, Irina Calciu, Sanidhya Kashyap, Ryan Stutsman, Amy Tai, and Gerd Zellweger. NrOS: Effective Replication and Sharing in an Operating System. In 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21), pages 295–312. USENIX Association, July 2021. ISBN

- 978-1-939133-22-9. URL <https://www.usenix.org/conference/osdi21/presentation/bhardwaj>.
- [34] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell D. E. Long, and Ethan L. Miller. Twizzler: A Data-Centric OS for Non-Volatile Memory. In 2020 USENIX Annual Technical Conference (USENIX ATC 20), pages 65–80. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL <https://www.usenix.org/conference/atc20/presentation/bittman>.
 - [35] Daniel Bittman, Robert Soulé, Ethan L. Miller, Vishal Shrivastav, Pankaj Mehra, Matthew Boisvert, Avi Silberschatz, and Peter Alvaro. Don’t Let RPCs Constrain Your API. In Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks, HotNets ’21, page 192–198, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450390873. doi: 10.1145/3484266.3487389. URL <https://doi.org/10.1145/3484266.3487389>.
 - [36] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An Operating System for Many Cores. In Symposium on Operating Systems Design and Implementation (OSDI), page 43–57, 2008.
 - [37] Neil Brown. Control Groups, Part 4: On Accounting. <https://lwn.net/Articles/606004/>.
 - [38] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. ACM Trans. Comput. Syst., 15(4):412–447, November 1997. ISSN 0734-2071. doi: 10.1145/265924.265930. URL <https://doi.org/10.1145/265924.265930>.
 - [39] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: Skip Redundant Paths to Make Serverless Fast. In Proceedings of the Fifteenth European Conference on Computer Systems, EuroSys ’20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368827. doi: 10.1145/3342195.3392698. URL <https://doi.org/10.1145/3342195.3392698>.
 - [40] cAdvisor. cAdvisor. <https://github.com/google/cadvisor>.
 - [41] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient Distributed Memory Management with RDMA and Caching. Proc. VLDB Endow., 11(11):1604–1617, jul 2018. ISSN 2150-8097. doi: 10.14778/3236187.3236209. URL <https://doi.org/10.14778/3236187.3236209>.
 - [42] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box Concurrent Data Structures for NUMA Architectures. In Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’17, page 207–221, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450344654. doi: 10.1145/3037697.3037721. URL <https://doi.org/10.1145/3037697.3037721>.
 - [43] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking Software Runtimes for Disaggregated Memory. In Proceedings of the 26th ACM International Conference on Architectural Support for

- Programming Languages and Operating Systems, ASPLOS '21, page 79–92, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446713. URL <https://doi.org/10.1145/3445814.3446713>.
- [44] Blake Caldwell, Sepideh Goodarzy, Sangtae Ha, Richard Han, Eric Keller, Eric Rozner, and Youngbin Im. FluidMem: Full, Flexible, and Fast Memory Disaggregation for the Cloud. In 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS), pages 665–677, 2020. doi: 10.1109/ICDCS47774.2020.00090.
 - [45] Oscar Castro, Pierrick Bruneau, Jean-Sébastien Sottet, and Dario Torregrossa. Landscape of High-Performance Python to Develop Data Science and Machine Learning Applications. ACM Computing Surveys, 56(3), October 2023. ISSN 0360-0300. doi: 10.1145/3617588. URL <https://doi.org/10.1145/3617588>.
 - [46] chashmap. CHashMap: Fast, concurrent hash maps with extensive API. <https://crates.io/crates/chashmap>, 2019. Accessed 2025-04-08.
 - [47] Ho-Ren Chuang, Karim Manaouil, Tong Xing, Antonio Barbalace, Pierre Olivier, Balvansh Heerekar, and Binoy Ravindran. Aggregate VM: Why Reduce or Evict VM's Resources When You Can Borrow Them From Other Nodes? In Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys '23, page 469–487, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394871. doi: 10.1145/3552326.3587452. URL <https://doi.org/10.1145/3552326.3587452>.
 - [48] client-go. client-go. <https://github.com/kubernetes/client-go>.
 - [49] Cloud Functions. Cloud Functions. <https://cloud.google.com/functions>.
 - [50] Cloud Functions Execution Environment. Cloud Functions Execution Environment. <https://cloud.google.com/functions/docs/concepts/exec#timeout>.
 - [51] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, Reto Achermann, Gustavo Alonso, and Timothy Roscoe. Enzian: An Open, General, CPU/FPGA Platform for Systems Software Research. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '22, page 434–451, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392051. doi: 10.1145/3503222.3507742. URL <https://doi.org/10.1145/3503222.3507742>.
 - [52] CCIX Consortium. An Introduction to CCIX. <https://www.ccixconsortium.com/wp-content/uploads/2019/11/CCIX-White-Paper-Rev111219.pdf>, 2019. Accessed 2025-04-17.
 - [53] Jonathan Corbet. AutoNUMA: The Other Approach to NUMA Scheduling. <https://lwn.net/Articles/488709/>, 2012.
 - [54] Jonathan Corbet. Weighted Interleaving for Memory Tiering. <https://lwn.net/Articles/948037/>, 2013. Accessed 2025-07-11.

- [55] Lixiao Cui, Kedi Yang, Yusen Li, Gang Wang, and Xiaoguang Liu. DiffLex: A High-Performance, Memory-Efficient and NUMA-Aware Learned Index using Differentiated Management. In *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, page 62–71, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400708435. doi: 10.1145/3605573.3605590. URL <https://doi.org/10.1145/3605573.3605590>.
- [56] Greg Cusack, Maziyar Nazari, Sepideh Goodarzy, Prerit Oberai, Eric Rozner, Eric Keller, and Richard Han. Efficient Microservices with Elastic Containers. In *Proceedings of the 15th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '19 Companion, page 65–67, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450370066. doi: 10.1145/3360468.3368180. URL <https://doi.org/10.1145/3360468.3368180>.
- [57] Greg Cusack, Maziyar Nazari, Sepideh Goodarzy, Erika Hunhoff, Prerit Oberai, Eric Keller, Eric Rozner, and Richard Han. Escra: Event-driven, Sub-second Container Resource Allocation. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pages 313–324, 2022. doi: 10.1109/ICDCS54860.2022.00038.
- [58] cxl. Compute Express Link. <https://www.computeexpresslink.org/>. Accessed 2025-07-10.
- [59] CXL. About CXL. <https://computeexpresslink.org/about-cxl/>, 2025. Accessed 2025-03-30.
- [60] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 381–394, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318709. doi: 10.1145/2451116.2451157. URL <https://doi.org/10.1145/2451116.2451157>.
- [61] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. *SIGPLAN Not.*, 49(4):127–144, February 2014. ISSN 0362-1340. doi: 10.1145/2644865.2541941. URL <https://doi.org/10.1145/2644865.2541941>.
- [62] Kristof Denolf, Marco Bekooij, Johan Cockx, Diederik Verkest, and Henk Corporaal. Exploiting the Expressiveness of Cyclo-Static Dataflow to Model Multimedia Implementations. *EURASIP Journal on Advances in Signal Processing*, 2007:1–14, 2007.
- [63] The QEMU Project Developers. Inter-VM Shared Memory Device. <https://www.qemu.org/docs/master/system/devices/ivshmem.html>, 2025. Accessed 2025-04-08.
- [64] Dave Dice and Alex Kogan. Compact NUMA-Aware Locks. *EuroSys '19*, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362818. doi: 10.1145/3302424.3303984. URL <https://doi.org/10.1145/3302424.3303984>.
- [65] docker. docker. <https://www.docker.com/>. Accessed 2025-07-10.
- [66] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems*

- Design and Implementation, NSDI'14, page 401–414, USA, 2014. USENIX Association. ISBN 9781931971096.
- [67] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15, page 54–70, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338349. doi: 10.1145/2815400.2815425. URL <https://doi-org.colorado.idm.oclc.org/10.1145/2815400.2815425>.
 - [68] Dong Du et al. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378512. URL <https://doi.org/10.1145/3373376.3378512>.
 - [69] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: Lambdas on a Diet. In Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC '20, page 45–59, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381376. doi: 10.1145/3419111.3421297. URL <https://doi.org/10.1145/3419111.3421297>.
 - [70] 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 Proceedings of the USENIX Annual Technical Conference (ATC), pages 1–14, July 2019. URL <https://www.flux.utah.edu/paper/duplyakin-atc19>.
 - [71] 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. ISBN 0897917154. doi: 10.1145/224056.224076. URL <https://doi.org/10.1145/224056.224076>.
 - [72] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. Beyond Processor-Centric Operating Systems. In Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems, HOTOS'15, page 17, USA, 2015. USENIX Association.
 - [73] Function Configuration, Deployment, and Execution. Function Configuration, Deployment, and Execution. <https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html#function-configuration-deployment-and-execution>. Accessed 2023-12-09.
 - [74] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In Proceedings of the Third Symposium on Operating Systems Design and Implementation, OSDI '99, page 87–100, USA, 1999. USENIX Association. ISBN 1880446391.

- [75] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 3–18, 2019.
- [76] Pedro García López, Marc Sánchez-Artigas, Gerard París, Daniel Barcelona Pons, Álvaro Ruiz Ollobarren, and David Arroyo Pinto. Comparison of FaaS Orchestration Systems. In 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), pages 148–153, 2018. doi: 10.1109/UCC-Companion.2018.00049.
- [77] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel Computing Experiences with CUDA. IEEE Micro, 28(4):13–27, 2008. doi: 10.1109/MM.2008.57.
- [78] Jon Gjengset. flurry: Rust port of Java’s ConcurrentHashMap. <https://crates.io/crates/flurry>, 2024. Accessed 2025-04-08.
- [79] Shikha Goel, Rajesh Kedia, Rijurekha Sen, and M Balakrishnan. EXPRESS: A Framework for Execution Time Prediction of Concurrent CNNs on Xilinx DPU Accelerator. ACM Trans. Embed. Comput. Syst., 24(1), November 2024. ISSN 1539-9087. doi: 10.1145/3697835. URL <https://doi.org/10.1145/3697835>.
- [80] Sepideh Goodarzy, Maziyar Nazari, Richard Han, Eric Keller, and Eric Rozner. Resource Management in Cloud Computing Using Machine Learning: A Survey. In 2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA), pages 811–816, 2020. doi: 10.1109/ICMLA51294.2020.00132.
- [81] Google Cloud Functions. Google Cloud Functions. <https://cloud.google.com/functions/>. Accessed 2018-10-02.
- [82] Google OR-Tools. CP-SAT Solver. https://developers.google.com/optimization/cp/cp_solver. Accessed 2023-01-06.
- [83] Mel Gorman. Foundation for Automatic NUMA Balancing. <https://lwn.net/Articles/523065/>, 11 2012. Accessed 2025-04-16.
- [84] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In 2022 USENIX Annual Technical Conference (USENIX ATC 22), pages 287–294, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-29-65. URL <https://www.usenix.org/conference/atc22/presentation/gouk>.
- [85] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular Disco: Resource Management Using Virtual Clusters on Shared-Memory Multiprocessors. In Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles, SOSP ’99, page 154–169, New York, NY, USA, 1999. Association for Computing Machinery. ISBN 1581131402. doi: 10.1145/319151.319162. URL <https://doi.org/10.1145/319151.319162>.
- [86] Nikhil Govil. Applying Halstead Software Science on Different Programming Languages for Analyzing Software Complexity. In 2020 4th International Conference on Trends in

- Electronics and Informatics (ICOEI)(48184), pages 939–943, 2020. doi: 10.1109/ICOEI48184.2020.9142911.
- [87] Marcus Griep. stats_alloc: An Allocator Wrapper that Allows for Instrumenting Global Allocators. https://crates.io/crates/stats_alloc, 2025. Version 0.1.10. Accessed 2025-03-30.
 - [88] Krzysztof Grygiel and Marcis Wielgus. Kubernetes Vertical Pod Autoscaler. <https://github.com/kubernetes/community/blob/master/contributors/design-proposals autoscaling/vertical-pod-autoscaler.md>.
 - [89] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with Infiniswap. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pages 649–667, Boston, MA, March 2017. USENIX Association. ISBN 978-1-931971-37-9. URL <https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu>.
 - [90] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan Chidambaram, Mahmut T. Kandemir, and Chita R. Das. Fifer: Tackling Underutilization in the Serverless Era. arXiv, 2020.
 - [91] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces. In 2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS), pages 1–10. IEEE, 2019.
 - [92] Irfan Habib. Virtualization with KVM. Linux J., 2008(166), February 2008. ISSN 1075-3583.
 - [93] Halstead Metrics. Halstead Metrics. <https://radon.readthedocs.io/en/latest/intro.html#halstead-metrics>. Accessed 2025-01-07.
 - [94] Travis Hance, Yi Zhou, Andrea Lattuada, Reto Achermann, Alex Conway, Ryan Stutsman, Gerd Zellweger, Chris Hawblitzel, Jon Howell, and Bryan Parno. Sharding the State Machine: Automated Modular Reasoning for Complex Concurrent Systems. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), pages 911–929, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-34-2. URL <https://www.usenix.org/conference/osdi23/presentation/hance>.
 - [95] T Hariprasad, G Vidhyagaran, K Seenu, and Chandrasegar Thirumalai. Software Complexity Analysis using Halstead Metrics. In 2017 International Conference on Trends in Electronics and Informatics (ICEI), pages 1109–1113, 2017. doi: 10.1109/ICOEI.2017.8300883.
 - [96] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. Nature, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL <https://doi.org/10.1038/s41586-020-2649-2>.

- [97] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.
- [98] Germont Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi Single-Address-Space Operating System. *Softw. Pract. Exper.*, 28(9):901–928, jul 1998. ISSN 0038-0644. doi: 10.1002/(SICI)1097-024X(19980725)28:9%3C901::AID-SPE181%3E3.0.CO;2-7. URL [https://doi.org/10.1002/\(SICI\)1097-024X\(19980725\)28:9%3C901::AID-SPE181%3E3.0.CO;2-7](https://doi.org/10.1002/(SICI)1097-024X(19980725)28:9%3C901::AID-SPE181%3E3.0.CO;2-7).
- [99] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’10, page 355–364, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300797. doi: 10.1145/1810479.1810540. URL <https://doi.org/10.1145/1810479.1810540>.
- [100] Emily Herbert and Arjun Guha. A Language-based Serverless Function Accelerator. *arXiv*, 2019.
- [101] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI’11, page 295–308, USA, 2011. USENIX Association.
- [102] Hipster Shop: Cloud-Native Microservices Demo Application. Hipster Shop: Cloud-Native Microservices Demo Application. <https://github.com/Brown-NSG/microservices-demo>.
- [103] host.json reference for Azure Functions 2.x and later. host.json Reference for Azure Functions 2.x and Later. <https://docs.microsoft.com/en-us/azure/azure-functions/functions-host-json#functiontimeout>.
- [104] Erika Hunhoff, Shazal Irshad, Vijay Thurimella, Ali Tariq, and Eric Rozner. Proactive Serverless Function Resource Management. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, WoSC’20, page 61–66, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450382045. doi: 10.1145/3429880.3430102. URL <https://doi-org.colorado.idm.oclc.org/10.1145/3429880.3430102>.
- [105] Erika Hunhoff, Joseph Melber, Kristof Denolf, Andra Bisca, Samuel Bayliss, Stephen Neuenendorffer, Jeff Fifield, Jack Lo, Pranathi Vasireddy, Phil James-Roxby, and Eric Keller. Efficiency, expressivity, and extensibility in a close-to-metal npu programming interface. In *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 85–94, 2025. doi: 10.1109/FCCM62733.2025.00043.
- [106] Hyperparameter tuning grid search example. Hyperparameter Tuning Grid Search Example. <https://github.com/lithops-cloud/applications/tree/master/sklearn>.
- [107] IBM Cloud Functions. IBM Cloud Functions. <https://www.ibm.com/cloud/functions>.
- [108] IMBD. IMDB. <https://www.imdb.com/>.

- [109] Intel. An Introduction to the Intel QuickPath Interconnect. <https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html>, 2009. Accessed 2025-03-30.
- [110] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, mar 2007. ISSN 0163-5980. doi: 10.1145/1272998.1273005. URL <https://doi.org/10.1145/1272998.1273005>.
- [111] Safdar Jamil, Abdul Salam, Awais Khan, Bernd Burgstaller, Sung-Soon Park, and Youngjae Kim. Scalable NUMA-Aware Persistent B+-Tree for Non-Volatile Memory Devices. *Cluster Computing*, 26(5):2865–2881, November 2022. ISSN 1386-7857. doi: 10.1007/s10586-022-03766-1. URL <https://doi.org/10.1007/s10586-022-03766-1>.
- [112] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’21, page 152–166, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446701. URL <https://doi-org.colorado.idm.oclc.org/10.1145/3445814.3446701>.
- [113] Wolfgang John, Joacim Halén, Xuejun Cai, Chunyan Fu, Torgny Holmberg, Vladimir Katardjiev, Tomas Mecklin, Mina Sedaghat, Pontus Sköldström, Daniel Turull, Vinay Yadhav, and James Kempf. Making Cloud Easy: Design Considerations and First Components of a Distributed Operating System for Cloud. In *Proceedings of the 10th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud’18, page 12, USA, 2018. USENIX Association.
- [114] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP ’97, page 52–65, New York, NY, USA, 1997. Association for Computing Machinery. ISBN 0897919165. doi: 10.1145/268998.266644. URL <https://doi.org/10.1145/268998.266644>.
- [115] Stefan Kaestle, Reto Achermann, Timothy Roscoe, and Tim Harris. Shoal: Smart Allocation and Replication of Memory For Parallel Programs. In *USENIX Annual Technical Conference (ATC)*, pages 263–276, 2015. ISBN 978-1-931971-225. URL <https://www.usenix.org/conference/atc15/technical-session/presentation/kaestle>.
- [116] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for μ second-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. ISBN 9781931971492.
- [117] Ramesh K. Karne, Karthick V. Jaganathan, Nelson Rosa, and Tufail Ahmed. DOSC: Dispersed Operating System Computing. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA ’05, page 55–62, New York, NY, USA, 2005. Association for Computing Machinery. ISBN

1595931937. doi: 10.1145/1094855.1094870. URL <https://doi.org/10.1145/1094855.1094870>.
- [118] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends. Rack-scale Disaggregated Cloud Data Centers: The dReDBox Project Vision. In Design, Automation & Test in Europe Conference & Exhibition (DATE'16), pages 690–695, 2016.
- [119] Srinivasan Keshav. Packet-Pair Flow Control. IEEE/ACM transactions on Networking, pages 1–45, 1995.
- [120] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. Iron: Isolating Network-based CPU in Container Environments. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pages 313–328, Renton, WA, 2018. USENIX Association. ISBN 978-1-931971-43-0. URL <https://www.usenix.org/conference/nsdi18/presentation/khalid>.
- [121] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation, OSDI'18, page 427–444, USA, 2018. USENIX Association. ISBN 9781931971478.
- [122] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Runtime Code Generation. In Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Parallel Computing, pages 157–174. ScienceDirect, 2012. URL <https://www.sciencedirect.com/science/article/pii/S0167819111001281>.
- [123] Marcin Kolny. Scaling up the Prime Video Audio/Video Monitoring Service and Reducing Costs by 90%. <https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90>. Accessed 2023-03-22.
- [124] Alexey Kopytov. Sysbench. <https://github.com/akopytov/sysbench>.
- [125] Kubernetes. Kubernetes: Production-Grade Container Orchestration. <https://kubernetes.io/>.
- [126] kubernetes. Kubernetes. <https://kubernetes.io/>. Accessed 2023-11-30.
- [127] Kubernetes Resource Quotas. Resource Quotas. <https://kubernetes.io/docs/concepts/policy/resource-quotas/>.
- [128] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '19, page 242–251, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450361378. doi: 10.1145/3289602.3293910. URL <https://doi.org/10.1145/3289602.3293910>.

- [129] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM ’15*, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340052. doi: 10.1145/2833157.2833162. URL <https://doi.org/10.1145/2833157.2833162>.
- [130] Lukasz Langa and contributors to Black. Black: The Uncompromising Python Code Formatter. <https://github.com/psf/black>. Accessed 2025-01-05.
- [131] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021. doi: 10.1109/CGO51591.2021.9370308.
- [132] Rodger Lea and Christian Jacquemot. The COOL Architecture and Abstractions for Object-Oriented Distributed Operating Systems. In *Proceedings of the 5th Workshop on ACM SIGOPS European Workshop: Models and Paradigms for Distributed Systems Structuring, EW 5*, page 1–8, New York, NY, USA, 1992. Association for Computing Machinery. ISBN 9781450373401. doi: 10.1145/506378.506407. URL <https://doi.org/10.1145/506378.506407>.
- [133] Kevin Lee and Kai-Ting. Wang. PyDSL: A Python Subset for a Better MLIR Programming Experience (Part II). <https://llvm.org/devmtg/2024-10/slides/quicktalks/Wang-PyDSL.pdf>, 2022.
- [134] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. MIND: In-Network Memory Management for Disaggregated Data Centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP ’21*, page 488–504, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450387095. doi: 10.1145/3477132.3483561. URL <https://doi.org/10.1145/3477132.3483561>.
- [135] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC ’15*, page 277–289, USA, 2015. USENIX Association. ISBN 9781931971225.
- [136] Maksim Levental. *An End-to-End Programming Model for AI Engine Architectures*. PhD thesis, University of Chicago, June 2024.
- [137] Maksim Levental, Alok Kamatar, Ryan Chard, Kyle Chard, and Ian Foster. nelli: A Lightweight Frontend for MLIR, 2023. URL <https://arxiv.org/abs/2307.16080>.
- [138] Maksim Levental, Arham Khan, Ryan Chard, Kyle Chard, Stephen Neuendorffer, and Ian Foster. An End-to-End Programming Model for AI Engine Architectures. In *Proceedings of the 14th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART), HEART ’24*, page 135–136, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400717277. doi: 10.1145/3665283.3665294. URL <https://doi.org/10.1145/3665283.3665294>.

- [139] Maksim Levintal. mlir-python-extras. <https://github.com/makslevental/mlir-python-extras>. Accessed 2024-12-28.
- [140] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–15, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450332521. doi: 10.1145/2670979.2670985. URL <https://doi.org/10.1145/2670979.2670985>.
- [141] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166. doi: 10.1145/3575693.3578835. URL <https://doi.org/10.1145/3575693.3578835>.
- [142] Zhen Lin, Kao-Feng Hsieh, Yu Sun, Seunghee Shin, and Hui Lu. FlashCube: Fast Provisioning of Serverless Functions with Streamlined Container Runtimes. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*, PLOS '21, page 38–45, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450387071. doi: 10.1145/3477113.3487273. URL <https://doi.org/10.1145/3477113.3487273>.
- [143] Linux Containers (LXC). Linux Containers. <https://linuxcontainers.org/>. Accessed 2025-07-10.
- [144] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the harp file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, page 226–238, New York, NY, USA, 1991. Association for Computing Machinery. ISBN 0897914473. doi: 10.1145/121132.121169. URL <https://doi.org/10.1145/121132.121169>.
- [145] Lithops. Lithops. <https://lithops-cloud.github.io/>.
- [146] Ami Litman. The DUNIX Distributed Operating System. *SIGOPS Oper. Syst. Rev.*, 22(1): 42–51, jan 1988. ISSN 0163-5980. doi: 10.1145/43921.43924. URL <https://doi.org/10.1145/43921.43924>.
- [147] David H. Liu, Amit Levy, Shadi Noghabi, and Sebastian Burckhardt. Doing More with Less: Orchestrating Serverless Applications without an Orchestrator. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1505–1519, Boston, MA, April 2023. USENIX Association. ISBN 978-1-939133-33-5. URL <https://www.usenix.org/conference/nsdi23/presentation/liu-david>.
- [148] Jianguo Liu and Zhibin Yu. Global-State Aware Automatic NUMA Balancing. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware*, Internetware '24, page 317–326, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400707056. doi: 10.1145/3671016.3671380. URL <https://doi.org/10.1145/3671016.3671380>.
- [149] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. A Survey of Coarse-Grained Reconfigurable Architecture and Design:

- Taxonomy, Challenges, and Applications. *ACM Comput. Surv.*, 52(6), October 2019. ISSN 0360-0300. doi: 10.1145/3357375. URL <https://doi.org/10.1145/3357375>.
- [150] Ye Liu, Shinpei Kato, and Masato Edahiro. Analysis of Memory System of Tiled Many-Core Processors. *IEEE Access*, 7:18964–18977, 2019. doi: 10.1109/ACCESS.2019.2895701.
 - [151] Jack Lo, Joseph Melber, Kristof Denolf, Phil James-Roxby, and Samuel Bayliss. Levering MLIR to Design for AI Engines on Ryzen™ AI. <https://github.com/Xilinx/mlir-aie/blob/main/docs/conferenceDescriptions/micro24TutorialDescription.md>, April 2024. ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24) Tutorial.
 - [152] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the Cloud: An Analysis on Alibaba Cluster Trace. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2884–2892. IEEE, 2017.
 - [153] Jonathan Mace and Rodrigo Fonseca. Universal Context Propagation for Distributed System Instrumentation. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 8:1–8:18, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5584-1. doi: 10.1145/3190508.3190526. URL <http://doi.acm.org/10.1145/3190508.3190526>.
 - [154] Paul E. McKenney, Mathieu Desnoyers, and Lai Jiangshan. URCU-Protected Hash Tables. <https://lwn.net/Articles/573431/>, November 2013. Accessed 2025-04-08.
 - [155] Joseph Melber, Kristof Denolf, and Andrew Schmidt. Leveraging the IRON AI Engine API to program the Ryzen™ AI NPU. <https://github.com/Xilinx/mlir-aie/blob/main/docs/conferenceDescriptions/micro24TutorialDescription.md>, Nov 2024. 57th IEEE/ACM International Symposium on Microarchitecture (MICRO'24) Tutorial.
 - [156] Jackson Melchert, Keyi Zhang, Yuchen Mei, Mark Horowitz, Christopher Torng, and Priyanka Raina. Canal: A Flexible Interconnect Generator for Coarse-Grained Reconfigurable Arrays. *IEEE Computer Architecture Letters*, 22(1):45–48, 2023. doi: 10.1109/LCA.2023.3268126.
 - [157] Maged M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, page 73–82, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581135297. doi: 10.1145/564870.564881. URL <https://doi.org/10.1145/564870.564881>.
 - [158] mlir-aie. mlir-aie: MLIR-based AI Engine Toolchain. <https://github.com/Xilinx/mlir-aie>. Accessed 2025-01-05.
 - [159] mlir python. MLIR Python Bindings. <https://mlir.llvm.org/docs/Bindings/Python/>. Accessed 2024-12-28.
 - [160] Anup Mohan et al. Agile Cold Starts for Scalable Serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association. URL <https://www.usenix.org/conference/hotcloud19/presentation/mohan>.
 - [161] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray:

- A Distributed Framework for Emerging AI Applications. In Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation, OSDI'18, page 561–577, USA, 2018. USENIX Association. ISBN 9781931971478.
- [162] Ingo Müller, Renato Marroquín, and Gustavo Alonso. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20, page 115–130, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367356. doi: 10.1145/3318464.3389758. URL <https://doi.org/10.1145/3318464.3389758>.
 - [163] Aaftab Munshi. The OpenCL Specification. In 2009 IEEE Hot Chips 21 Symposium (HCS), pages 1–314, 2009. doi: 10.1109/HOTCHIPS.2009.7478342.
 - [164] Antoine Murat, Clément Burgevin, Athanasios Xygkis, Igor Zablotschi, Marcos Kawazoe Aguilera, and Rachid Guerraoui. SWARM: Replicating Shared Disaggregated-Memory Data in No Time. In Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP '24, page 24–45, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712517. doi: 10.1145/3694715.3695945. URL <https://doi.org/10.1145/3694715.3695945>.
 - [165] Maziyar Nazari, Sepideh Goodarzy, Eric Keller, Eric Rozner, and Shivakant Mishra. Optimizing and Extending Serverless Platforms: A Survey. In 2021 Eighth International Conference on Software Defined Systems (SDS), pages 1–8, 2021. doi: 10.1109/SDS54264.2021.9732138.
 - [166] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-Tolerant Software Distributed Shared Memory. In 2015 USENIX Annual Technical Conference (USENIX ATC 15), pages 291–305, Santa Clara, CA, July 2015. USENIX Association. ISBN 978-1-931971-225. URL <https://www.usenix.org/conference/atc15/technical-session/presentation/nelson>.
 - [167] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09, page 221–234, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587523. doi: 10.1145/1629575.1629597. URL <https://doi-org.colorado.idm.oclc.org/10.1145/1629575.1629597>.
 - [168] Ruslan Nikolaev, Mincheol Sung, and Binoy Ravindran. LibrettOS: A Dynamically Adaptable Multiserver-Library OS. In Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, page 114–128, March 2020.
 - [169] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out NUMA. In Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, page 3–18, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450323055. doi: 10.1145/2541940.2541965. URL <https://doi.org/10.1145/2541940.2541965>.
 - [170] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. The Case for RackOut: Scalable Data Serving Using Rack-Scale Systems. In Proceedings of the Seventh ACM Symposium on Cloud Computing, SoCC '16, page 182–195, New York, NY,

- USA, 2016. Association for Computing Machinery. ISBN 9781450345255. doi: 10.1145/2987550.2987577. URL <https://doi.org/10.1145/2987550.2987577>.
- [171] numactl. numactl(8) - Linux Man Page. <https://linux.die.net/man/8/numactl>, 2012. Accessed 2025-04-17.
- [172] Edward Oakes et al. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association. ISBN 978-1-931971-44-7. URL <https://www.usenix.org/conference/atc18/presentation/oakes>.
- [173] One year using Kubernetes in production: Lessons learned. One Year Using Kubernetes in Production: Lessons Learned. <https://techbeacon.com/devops/one-year-using-kubernetes-production-lessons-learned>.
- [174] Openshift Quotas and Limit Ranges. Quotas and Limit Ranges. https://docs.openshift.com/online/pro/dev_guide/compute_resources.html.
- [175] OpenVINO™. Intel® Distribution of OpenVINO™ Toolkit. <https://www.intel.com/content/www/us/en/developer/tools/openvino-toolkit/overview.html>. Accessed 2025-01-08.
- [176] Mark Oskin, Frederic T. Chong, and Matthew Farrens. HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Designs. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, page 71–82, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132328. doi: 10.1145/339647.339656. URL <https://doi.org/10.1145/339647.339656>.
- [177] Peitian Pan, Chao Li, and Minyi Guo. CongraPlus: Towards Efficient Processing of Concurrent Graph Queries on NUMA Machines. *IEEE Transactions on Parallel and Distributed Systems*, 30(9):1990–2002, 2019. doi: 10.1109/TPDS.2019.2899595.
- [178] Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K. Gopinath, and Jayneel Gandhi. Fast local page-tables for virtualized NUMA servers with vMitosis. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 194–210, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446709. URL <https://doi.org/10.1145/3445814.3446709>.
- [179] peano. peano: AI Engine Fork of LLVM. <https://github.com/Xilinx/llvm-aie>. Accessed 2024-12-28 December 2024.
- [180] Performance Co-Pilot (PCP) Manual. Performance Co-Pilot (PCP) Manual. <https://pcp.io/docs/index.html>.
- [181] Panayiotis Petrides and Pedro Trancoso. Heterogeneous- and NUMA-Aware Scheduling for Many-Core Architectures. In *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR '17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350358. doi: 10.1145/3078468.3078482. URL <https://doi.org/10.1145/3078468.3078482>.

- [182] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: A Network Transparent, High Reliability Distributed System. *SIGOPS Oper. Syst. Rev.*, 15(5):169–177, dec 1981. ISSN 0163-5980. doi: 10.1145/1067627.806605. URL <https://doi.org/10.1145/1067627.806605>.
- [183] prometheus. Kubernetes Vertical Pod Autoscaler. <https://github.com/prometheus/prometheus>.
- [184] pygount. pygount. <https://github.com/roskakori/pygount>. Accessed 2025-01-05.
- [185] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL <https://www.usenix.org/conference/osdi20/presentation/qiu>.
- [186] Hongliang Qu and Zhibin Yu. WASP: Workload-Aware Self-Replicating Page-Tables for NUMA Servers. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS ’24*, page 1233–1249, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703850. doi: 10.1145/3620665.3640369. URL <https://doi.org/10.1145/3620665.3640369>.
- [187] Qualcomm® Neural Processing SDK. Qualcomm® Neural Processing SDK for AI. <https://www.qualcomm.com/developer/software/neural-processing-sdk-for-ai>. Accessed 2025-01-08.
- [188] Maryan Rab, Romolo Marotta, Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. NUMA-Aware Non-Blocking Calendar Queue. In *2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 1–9, 2020. doi: 10.1109/DS-RT50469.2020.9213639.
- [189] radon. radon. <https://github.com/rubik/radon>. Accessed 2025-01-05.
- [190] Jonathan Ragan-Kelley. The Future of Fast Code: Giving Hardware What It Wants. <https://youtu.be/vU3ryvZY1kk?si=Zk-jRdYFqQ3jcbJp>, June 2024. 44th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI’24) Keynote.
- [191] TIRIAS Research. AMD Optimizes EPYC Memory with NUMA. <https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/AMD-Optimizes-EPYC-Memory-With-NUMA.pdf>, 2018. Accessed 2025-03-30.
- [192] Alejandro Rico, Satyaprakash Pareek, Javier Cabezas, David Clarke, Baris Ozgul, Francisco Barat, Yao Fu, Stephan Münz, Dylan Stuart, Patrick Schlangen, Pedro Duarte, Sneha Date, Indrani Paul, Jian Weng, Sonal Santan, Vinod Kathail, Ashish Sirasao, and Juanjo Noguera. AMD XDNA™ NPU in Ryzen™ AI Processors. *IEEE Micro*, 44(6):73–82, 2024. doi: 10.1109/MM.2024.3423692.
- [193] Probir Roy, Shuaiwen Leon Song, Sriram Krishnamoorthy, Abhinav Vishnu, Dipanjan Sen-gupta, and Xu Liu. NUMA-Caffe: NUMA-Aware Deep Learning Neural Networks. *ACM*

- Trans. Archit. Code Optim., 15(2), June 2018. ISSN 1544-3566. doi: 10.1145/3199605. URL <https://doi.org/10.1145/3199605>.
- [194] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated Far Memory. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 315–332. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL <https://www.usenix.org/conference/osdi20/presentation/ruan>.
- [195] rumprun. Rumprun. <https://github.com/rumpkernel/rumprun>. Accessed 2025-04-08.
- [196] Krzysztof Rzadca, Paweł Findeisen, Jacek Swiderski, Przemysław Zych, Przemysław Broniek, Jarek Kusmirek, Paweł Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: Workload Autoscaling at Google Scale. In Proceedings of the Fifteenth European Conference on Computer Systems, EuroSys ’20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368827. doi: 10.1145/3342195.3387524. URL <https://doi.org/10.1145/3342195.3387524>.
- [197] scf dialect. ‘scf’ Dialect. <https://mlir.llvm.org/docs/Dialects/SCFDialect/>. Accessed 2024-12-28.
- [198] Philipp Schaad, Tal Ben-Nun, and Torsten Hoefer. Boosting Performance Optimization with Interactive Data Movement Visualization. In SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–16, 2022. doi: 10.1109/SC41404.2022.00069.
- [199] Malte Schwarzkopf, Matthew P. Grosvenor, and Steven Hand. New Wine in Old Skins: The Case for Distributed Operating Systems in the Data Center. In Proceedings of the 4th Asia-Pacific Workshop on Systems, APSys ’13, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323161. doi: 10.1145/2500727.2500739. URL <https://doi.org/10.1145/2500727.2500739>.
- [200] scikit-learn. scikit-learn: Machine Learning in Python. <https://scikit-learn.org/stable/>.
- [201] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In 2020 USENIX Annual Technical Conference (USENIX ATC 20), pages 205–218. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL <https://www.usenix.org/conference/atc20/presentation/shahrad>.
- [202] Ori Shalev and Nir Shavit. Split-Ordered Lists: Lock-Free Extensible Hash Tables. J. ACM, 53(3):379–405, May 2006. ISSN 0004-5411. doi: 10.1145/1147954.1147958. URL <https://doi.org/10.1145/1147954.1147958>.
- [203] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 69–87, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL <https://www.usenix.org/conference/osdi18/presentation/shan>.

- [204] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro*, 36(2):34–46, 2016. doi: 10.1109/MM.2016.25.
- [205] Xiang Song, Haibo Chen, Rong Chen, Yuanxuan Wang, and Binyu Zang. A Case for Scaling Applications to Many-Core with OS Clustering. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys ’11, page 61–76, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306348. doi: 10.1145/1966445.1966452. URL <https://doi.org/10.1145/1966445.1966452>.
- [206] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful Functions-as-a-Service. *Proceedings of the VLDB Endowment*, 13(12):2438–2452, Aug 2020. ISSN 2150-8097. doi: 10.14778/3407790.3407836. URL <http://dx.doi.org/10.14778/3407790.3407836>.
- [207] Jeffrey Stuecheli, William J Starke, John D Irish, L Baba Arimilli, D Dreps, Bart Blaner, Curt Wollbrink, and Brian Allison. IBM POWER9 Opens up a New Era of Acceleration Enablement: OpenCAPI. *IBM Journal of Research and Development*, 62(4/5):8–1, 2018.
- [208] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO ’23, page 105–121, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703294. doi: 10.1145/3613424.3614256. URL <https://doi.org/10.1145/3613424.3614256>.
- [209] Amoghvarsha Suresh and Anshul Gandhi. FnSched: An Efficient Scheduler for Serverless Functions. In *Proceedings of the 5th International Workshop on Serverless Computing*, WOSC ’19, page 19–24, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450370387. doi: 10.1145/3366623.3368136. URL <https://doi.org/10.1145/3366623.3368136>.
- [210] Lalith Suresh, João Loff, Faria Kalim, Sangeetha Abdu Jyothi, Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, and Michael Gasch. Building Scalable and Flexible Cluster Managers Using Declarative Programming. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI’20, USA, 2020. USENIX Association. ISBN 978-1-939133-19-9.
- [211] swarm. Docker Swarm. <https://docs.docker.com/engine/swarm/>, 2022. Accessed 2023-10-19.
- [212] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, and Sape J. Mullender. Experiences with the Amoeba Distributed Operating System. *Commun. ACM*, 33(12):46–63, dec 1990. ISSN 0001-0782. doi: 10.1145/96267.96281. URL <https://doi.org/10.1145/96267.96281>.
- [213] Lingjia Tang, Jason Mars, Xiao Zhang, Robert Hagmann, Robert Hundt, and Eric Tune. Optimizing Google’s Warehouse Scale Computers: The NUMA Experience. In *2013 IEEE*

- 19th International Symposium on High Performance Computer Architecture (HPCA), pages 188–197, 2013. doi: 10.1109/HPCA.2013.6522318.
- [214] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. Sequoia: Enabling Quality-of-Service in Serverless Computing. In Proceedings of the Annual Symposium on Cloud Computing (SoCC), 2020.
- [215] TeaStore. TeaStore. <https://github.com/DescartesResearch/TeaStore>.
- [216] Telecom AT&T’s Paradise. Telecom AT&T’s Paradise: 75% of telco’s MPLS tunnel data traffic now under SDN control. <https://www.fiercetelecom.com/telecom/at-t-s-paradise-75-telco-s-mpls-tunnel-data-traffic-now-under-sdn-control>.
- [217] Gil Tene. wrk2: A HTTP Benchmarking Tool Based Mostly on wrk. <https://github.com/giltene/wrk2>.
- [218] The Rust Programming Language. std::collections. <https://doc.rust-lang.org/std/collections/>, 2024. Accessed 2025-04-08.
- [219] Philippe Tillet, H. T. Kung, and David Cox. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL 2019, page 10–19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367196. doi: 10.1145/3315508.3329973. URL <https://doi.org/10.1145/3315508.3329973>.
- [220] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The Next Generation. In Proceedings of the Fifteenth European Conference on Computer Systems, EuroSys ’20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368827. doi: 10.1145/3342195.3387517. URL <https://doi.org/10.1145/3342195.3387517>.
- [221] torch-mlir. Torch-MLIR. <https://github.com/llvm/torch-mlir>. Accessed 2025-01-05.
- [222] Train Ticket: A Benchmark Microservice System. Train Ticket: A Benchmark Microservice System. <https://github.com/FudanSELab/train-ticket>.
- [223] Paul Turner, Bharata B Rao, and Nikhil Rao. CPU bandwidth control for CFS. In Proceedings of the Linux Symposium, pages 245–254, 2010. URL http://www.linuxsymposium.org/LS_2010_Proceedings_Draft.pdf.
- [224] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-Scale Cluster Management at Google with Borg. In Proceedings of the Tenth European Conference on Computer Systems, EuroSys ’15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450332385. doi: 10.1145/2741948.2741964. URL <https://doi.org/10.1145/2741948.2741964>.
- [225] VMWare, Inc. node-replication. <https://github.com/vmware/node-replication>, 2019. Accessed 2025-04-10.
- [226] VMware NSX Data Center. VMware NSX Data Center. <https://www.vmware.com/products/nsx.html>.

- [227] Jacob Wahlgren, Maya Gokhale, and Ivy B. Peng. Evaluating Emerging CXL-enabled Memory Pooling for HPC Systems. <https://arxiv.org/abs/2211.02682>, 2022.
- [228] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. MemLiner: Lining up Tracing and Application for a Far-Memory-Friendly Runtime. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 35–53, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL <https://www.usenix.org/conference/osdi22/presentation/wang>.
- [229] Liang Wang et al. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, 2018. USENIX Association. ISBN 978-1-931971-44-7. URL <https://www.usenix.org/conference/atc18/presentation/wang-liang>.
- [230] Qing Wang, Youyou Lu, Junru Li, Minhui Xie, and Jiwu Shu. Nap: Persistent Memory Indexes for NUMA Architectures. *ACM Trans. Storage*, 18(1), January 2022. ISSN 1553-3077. doi: 10.1145/3507922. URL <https://doi.org/10.1145/3507922>.
- [231] Zhonghua Wang, Kai Lu, Jiguang Wan, Hong Jiang, Zeyang Zhao, Peng Xu, Biliang Lai, Guokuan Li, and Changsheng Xie. NStore: A High-Performance NUMA-Aware Key-Value Store for Hybrid Memory. *IEEE Transactions on Computers*, 74(3):929–943, 2025. doi: 10.1109/TC.2024.3504269.
- [232] webassembly. WebAssembly. <https://webassembly.org/>. Accessed 2023-12-09.
- [233] Joel Wejdenstål. DashMap: Blazing fast concurrent HashMap for Rust. <https://crates.io/crates/dashmap>, 2024. Accessed 2025-04-08.
- [234] David Wentzlaff and Anant Agarwal. Factored Operating Systems (Fos): The Case for a Scalable Operating System for Multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, apr 2009. ISSN 0163-5980. doi: 10.1145/1531793.1531805. URL <https://doi.org/10.1145/1531793.1531805>.
- [235] David Wentzlaff, Charles Gruenwald, Nathan Beckmann, Kevin Modzelewski, Adam Bellay, Lamia Youseff, Jason Miller, and Anant Agarwal. An Operating System for Multicore and Clouds: Mechanisms and Implementation. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC ’10, page 3–14, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300360. doi: 10.1145/1807128.1807132. URL <https://doi.org/10.1145/1807128.1807132>.
- [236] Julian Wood. Building Extensions for AWS Lambda — In Preview. <https://aws.amazon.com/blogs/compute/building-extensions-for-aws-lambda-in-preview/>, 2020.
- [237] Mingchuan Wu, Ying Liu, Huimin Cui, Qingfu Wei, Quanfeng Li, Limin Li, Fang Lv, Jingling Xue, and Xiaobing Feng. Bandwidth-Aware Loop Tiling for DMA-Supported Scratchpad Memory. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT ’20, page 97–109, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380751. doi: 10.1145/3410463.3414637. URL <https://doi.org/10.1145/3410463.3414637>.

- [238] Xilinx Runtime Architecture. Xilinx runtime (XRT) architecture. <https://xilinx.github.io/XRT/2024.2/html/index.html>. Accessed 2025-01-05.
- [239] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The True Cost of Containing: A gVisor Case Study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association. URL <https://www.usenix.org/conference/hotcloud19/presentation/young>.
- [240] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing Serverless Platforms with ServerlessBench. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '20. Association for Computing Machinery, 2020. doi: 10.1145/3419111.3421280. URL <https://doi.org/10.1145/3419111.3421280>.
- [241] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 1–14, USA, 2008. USENIX Association.
- [242] 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, oct 2016. ISSN 0001-0782. doi: 10.1145/2934664. URL <https://doi.org/10.1145/2934664>.
- [243] Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-Aware Graph-Structured Analytics. *SIGPLAN Not.*, 50(8):183–193, January 2015. ISSN 0362-1340. doi: 10.1145/2858788.2688507. URL <https://doi.org/10.1145/2858788.2688507>.
- [244] Yuhong Zhong, Hongyi Wang, Yu Jian Wu, Asaf Cidon, Ryan Stutsman, Amy Tai, and Junfeng Yang. BPF for Storage: An Exokernel-Inspired Approach. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 128–135, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384384. doi: 10.1145/3458336.3465290. URL <https://doi.org/10.1145/3458336.3465290>.
- [245] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-Tolerant Far Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 55–71, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL <https://www.usenix.org/conference/osdi22/presentation/zhou-yang>.
- [246] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association. ISBN 978-1-939133-19-9.