

**EFFICIENT INTER-PROCESS COMMUNICATION MECHANISMS FOR  
DATA-CENTRIC COMPUTING**

A Thesis Proposal  
Presented to  
The Academic Faculty

By

Misun Park

In Partial Fulfillment  
of the Requirements for the Degree  
Doctor of Philosophy in the  
School of Computer Science  
College of Computing

Georgia Institute of Technology

May 2026

© Misun Park 2026

**EFFICIENT INTER-PROCESS COMMUNICATION MECHANISMS FOR  
DATA-CENTRIC COMPUTING**

Thesis committee:

Dr. Ada Gavrilovska  
School of Computer Science  
*Georgia Institute of Technology*

Dr. Nam Sung Kim  
Department of Department of Electrical  
Engineering  
*University of Illinois, Urbana-Champaign*

Dr. Hyesoon Kim  
School of Computer Science  
*Georgia Institute of Technology*

Dr. Francisco Romero  
School of Computer Science  
*Georgia Institute of Technology*

Dr. Alexandros Daglis  
School of Computer Science  
*Georgia Institute of Technology*

Date approved: Jan 13, 2026

Care is a form of intelligence.

To Ben and all the loyal paws and happy wags in the world

## ACKNOWLEDGMENTS

I am deeply grateful to my advisor, Dr. Ada Gavrilovska, who has been a constant source of guidance and inspiration throughout my doctoral studies. I respect her not only as an outstanding researcher, but also as a mentor from whom I learned how to think clearly, act with integrity, and navigate challenges with balance. Her mentorship profoundly shaped both this dissertation and my development as a scholar.

My sincere thanks go to my dissertation committee members, Dr. Alexandros Daglis, Dr. Hyesoon Kim, Dr. Nam Sung Kim, and Dr. Francisco Romero, for their thoughtful feedback and insightful discussions. Their diverse perspectives sharpened both the technical depth and the broader framing of this work. I am particularly thankful to Dr. Nam Sung Kim for his guidance and close collaboration, which significantly influenced the direction of my research.

This research was supported in part by the Applications Driving Architectures (ADA) and PRISM (Processing with Intelligent Storage and Memory) Research Centers, JUMP centers co-sponsored by SRC and DARPA. Additional support from Intel's Transformative Server Architectures (TSA) program enabled several essential aspects of this work.

I also acknowledge the collaborators, reviewers, and members of the research community whose feedback helped shape the ideas presented in this dissertation. Their questions and critiques played an important role in refining both the system design and its evaluation.

Throughout this journey, I have been sustained by the Lord's steady care, for which I am deeply grateful. I am especially thankful for the opportunities I was given to pursue this work and to devote myself fully to learning and research, which I received as a gift rather than something I earned on my own.

Beyond the academic setting, I owe a great deal to those who supported me personally. Carol steadied me during moments of doubt and gave me the courage to continue, helping me grow both personally and intellectually. I am also thankful for my dog, Ben, who stayed

by my side late into countless nights and shared quiet mornings with me, offering constancy regardless of whether I was struggling or thriving.

Finally, I thank my lab mates and friends, who were among the most meaningful gifts of my time at Georgia Tech. From each of you, at every stage of seniority, I learned something valuable, and I hope that I was, in turn, a source of support and encouragement.

## TABLE OF CONTENTS

<b>Acknowledgments</b> . . . . .	v
<b>List of Tables</b> . . . . .	xiii
<b>List of Figures</b> . . . . .	xv
<b>Summary</b> . . . . .	xix
<b>Chapter 1: Introduction</b> . . . . .	1
1.1 Statement of Problem . . . . .	4
1.2 Thesis Statement . . . . .	6
1.3 Contributions . . . . .	6
<b>Chapter 2: Background</b> . . . . .	9
2.1 Cost of Data Movement in Datacenter Systems . . . . .	9
2.2 Modern Applications and the Data Movement Bottleneck . . . . .	11
2.2.1 Data-Intensive Applications . . . . .	11
2.2.2 Memory Pressure and Resource Contention in Fixed-Resource Systems . . . . .	13
2.3 Closing the Memory Copy Bottleneck: Leveraging Emerging User-Space Memory Engines in IPC . . . . .	14
2.3.1 Quantifying the Bottleneck: The Dominance of Memory Copy . . . . .	14

2.3.2	Rethinking IPC with Novel Data Movers . . . . .	15
2.4	Summary . . . . .	17
<b>Chapter 3:</b>	<b>IPC Runtime for Dynamic Resource Allocation in Resource-Fixed Environment . . . . .</b>	<b>19</b>
3.1	The Need for Split-Architecture for ML Serving and Design Challenges . .	20
3.1.1	Split Architecture Components . . . . .	22
3.1.2	Problem Statement: Bottlenecks from Physical Separation and IPC .	23
3.2	Obstacles to Scaling ML Serving at the Edge . . . . .	24
3.2.1	Inefficiency of Conventional Cloud-Native Stacks . . . . .	24
3.2.2	Architectural Hurdles in Decoupled Runtimes . . . . .	25
3.3	System Design for Efficient Resource Sharing and Isolation . . . . .	27
3.3.1	Lightweight Pocket IPC . . . . .	27
3.3.2	Multi-Tenant Isolation . . . . .	29
3.3.3	Resource Amplification . . . . .	29
3.4	Evaluation: IPC Performance and Resource Efficiency . . . . .	30
3.4.1	Experimental Environment and Workload Characterization . . . . .	31
3.4.2	Temporal Efficiency and Scaling Results . . . . .	32
3.4.3	Efficacy of System Mechanisms . . . . .	34
3.5	Summary . . . . .	37
3.5.1	Relationship to the Thesis Statement . . . . .	38
<b>Chapter 4:</b>	<b>Unified IPC Runtime for Novel Data Movers across Hardware and Software Stack . . . . .</b>	<b>39</b>
4.1	Introduction: The Escalating Cost of Data Movement in Modern IPC . . . .	39

4.1.1	The Data Movement Bottleneck and the "Boundary Tax" . . . . .	40
4.1.2	Limitations of CPU-Driven IPC Stacks . . . . .	41
4.1.3	Rocket: A Unified Runtime for Heterogeneous Data Movers . . . . .	42
4.2	Background: Evolution of Decoupled Data Movement . . . . .	43
4.2.1	The Multi-Layered Landscape of Memory Offloading . . . . .	43
4.3	Case Study with Intel DSA . . . . .	44
4.3.1	DSA Primer . . . . .	45
4.3.2	Current Challenges in Software Integration . . . . .	46
4.4	Motivation: Integration Challenges in Offloaded IPC . . . . .	48
4.4.1	Hardware-level Trade-offs . . . . .	48
4.4.2	Cache Interference and Path Divergence: CPU vs. Accelerator . . . . .	50
4.4.3	Summary of the Tradeoffs . . . . .	51
4.5	Design of the Rocket Unified IPC Runtime . . . . .	52
4.5.1	Architectural Overview and Core Principles . . . . .	53
4.5.2	Unified IPC API and Execution Modes . . . . .	54
4.5.3	Runtime Internals and Implementation . . . . .	56
4.6	Implementation Details . . . . .	58
4.7	Evaluation of the Rocket Unified Runtime . . . . .	59
4.7.1	Experimental Methodology . . . . .	59
4.7.2	Impact on Application Performance . . . . .	61
4.7.3	Architectural Sources of Improvement . . . . .	64
4.8	Summary . . . . .	66
4.8.1	Relationship to the Thesis Statement . . . . .	67

<b>Chapter 5: Adaptive IPC Runtime Configuration with ML-Based Tuning Model</b>	<b>68</b>
5.1 From Fixed to Fluid: The Need for Dynamic Policy Orchestration in Unified IPC Runtimes . . . . .	68
5.2 Design Overview . . . . .	70
5.2.1 System-Level Workflow Overview . . . . .	70
5.2.2 Configuration Space Model . . . . .	72
5.2.3 Policy Construction Pipeline . . . . .	72
5.3 Prediction Model . . . . .	73
5.3.1 Quantitative Characterization of IPC Backends . . . . .	73
5.3.2 Runtime Features and Design Hypothesis . . . . .	74
5.3.3 Learning-Based Configuration Engine . . . . .	75
5.3.4 Performance Prediction Model . . . . .	76
5.3.5 Transparent Runtime Integration . . . . .	77
5.4 Runtime Adaptation . . . . .	78
5.4.1 Adaptive Execution Loop . . . . .	78
5.4.2 Implementation Notes . . . . .	79
5.4.3 Deployment-time Configuration and Policy Integration . . . . .	79
5.5 Evaluation . . . . .	81
5.5.1 Regression Model Performance . . . . .	81
5.5.2 End-to-End System Validation . . . . .	82
5.5.3 Decision Analysis of Input Features . . . . .	83
5.5.4 Overhead Analysis . . . . .	85
5.5.5 Case Study: Online Adaptation to Workload Changes . . . . .	86

5.6	Summary	87
5.6.1	Relationship to the Thesis Statement	88
<b>Chapter 6: Related Work</b>		<b>90</b>
6.1	Sharing, Amplification, and Transient Use of Resources	90
6.1.1	Efficient Container Runtimes	90
6.1.2	Resource Amplification: Academic Roots	92
6.2	Data Movement Across Hardware, OS, and IPC Layers	93
6.2.1	A Landscape of Data Movers Across the Stack	93
6.2.2	Empirical Studies and System Adoption of Intel DSA	95
6.2.3	Limitations of Existing Offloading Abstractions	96
6.2.4	Workloads Motivating Efficient Intra-Node Data Movement	97
6.2.5	Synthesis	97
6.3	Runtime Adaptation for Heterogeneous Data Movement	97
6.3.1	The Intractability of Manual IPC Configuration	98
6.3.2	The Case for Learned Control Planes	99
6.3.3	The Latency Requirement in IPC	99
6.3.4	SkyRocket: Lightweight Gradient Boosting on the Critical Path	100
<b>Chapter 7: Conclusion</b>		<b>101</b>
7.1	Dissertation Summary: Reimagining IPC for the Data-Centric Era	101
7.2	Synthesis of Contributions	101
7.2.1	IPC with Resource Management: Pocket	102

7.2.2	IPC with Unified Coordination for Heterogeneous Data Movement: Rocket . . . . .	102
7.2.3	IPC with Adaptive Intelligence: SkyRocket . . . . .	103
7.3	Key Insights and Broader Implications . . . . .	104
7.4	Future Directions . . . . .	105
7.5	Concluding Remarks . . . . .	106
	<b>References . . . . .</b>	<b>107</b>

## LIST OF TABLES

2.1	Representative Data Modality in Modern Applications . . . . .	12
3.1	Comparison of Firecracker configurations: a configuration for a simple function from [47] and a configuration for running TensorFlow-based Yolov3 applications. In the NSDI evaluation, the image size of Firecracker is zero because the boot time was measured as the loading time of a bare mininum kernel, not including any supporting services and not having underlying file systems. . . . .	25
3.2	Testbed Setup. . . . .	31
3.3	Applications used in the evaluation. We include applications that represent different functionality and exhibit different resource requirements. . . . .	32
3.4	Whether it is a backend with dynamic resource transfer or without resource transfer, no meaningful differences in latency have been observed. . . . .	36
4.1	Metrics characterizing data transfer and memory behavior for representative multimodal workloads in application-pipeline IPC scenarios. . . . .	40
4.2	Representative memory-offload mechanisms across system layers. . . . .	44
4.3	Trade-offs of DSA offloading and implications for system design. Each factor highlights a key limitation and how Rocket addresses it through configurable or default design decisions. . . . .	52
4.4	Experimental Setup . . . . .	60
4.5	Pipeline stages per benchmark workload. . . . .	60
5.1	Runtime context factors used by SkyRocket for execution policy inference. . . . .	74
5.2	Context-policy relationships commonly observed in our prototype system. . . . .	75

5.3	Evaluation summary of the regression model. . . . .	81
5.4	Significance test results for decision-related input features. . . . .	84

## LIST OF FIGURES

2.1	Breakdown of Datacenter Tax (Reconstructed from [34]) . . . . .	10
2.2	Modern Application Structure . . . . .	11
2.3	Execution time breakdown of a shared memory IPC microbenchmark. As the data payload increases from 1MB to 4MB, the memory copy operation (hatched area) consumes the vast majority of CPU cycles. . . . .	14
3.1	Existing solutions based on monolithic approaches scale poorly due to resource requirements of ML runtimes. End-to-end request latency in (c) (y-axis: milliseconds, x-axis: the number of concurrent instances). Pocket aims to serve more instances with better performance by sharing and efficiently handling the runtimes' resource demand. . . . .	21
3.2	ML applications and their resource-demanding ML runtimes lead to resource bloat when used with existing cloud-native technologies, which limits scaling. By introducing a shared backend, the ML runtime resource demand is expected to be amortized. . . . .	23
3.3	Comparison of (a) round-trip time (unit: $\mu\text{sec}$ ) and (b) incremental memory footprint per connection for gRPC and shmem-IPC. The regression fit ( $R^2$ ) represents the correlation between the number of concurrent tenants and the resulting server memory demand. . . . .	26
3.4	Design overview of Pocket. Pocket frontends connect to one of the shared backends via lightweight IPC and use it as an ML server. Every request is checked via capabilities list to prevent illegal access, and an additional layer of namespace and private queues for each channel hardens the isolation. A Resource Manager takes care of dynamic resource reallocation. . . . .	28

3.5	Performance and resource efficiency metrics for each application. Overall, with the suggested resource budget configurations, Pocket achieves better performance with on average $5\times$ better resource efficiency, which points to its ability to serve more instances at once with the same amount of resources. . . . .	33
3.6	Execution time breakdown of GPU execution. Even for GPU execution, pocket-dynamic is superior to the other two policies, static(ally reassigning the resource) and none (no resource reassignment). When input preprocessing is the bottleneck for an application, pocket-dynamic has an advantage over static by allocating the resources required for input processing to the frontend. The red dotted line represents the latency when running 20 instances as pocket-dynamic. This reveals how many instances each policy can run concurrently with comparable performance. . . . .	34
3.7	Pocket execution time breakdown. SSDResNet50 is an application that requires input preprocessing from frontend and SmallBERT language processing application requires little preprocessing. Allocating most resources statically to the backend may result in worse latency than dynamic allocation, depending on application. . . . .	35
3.8	Speed up trend with varying resource transfer proportions. X axis represents a proportion to transfer to backend. Y axis is $\frac{latency(monolithic)}{latency(static)}$ or $\frac{latency(monolithic)}{latency(pocket)}$ , which represents the amount of speed up (higher the better). Larger resource transfer leads to more speed up for Pocket. . . . .	35
3.9	To evaluate how Pocket works when serving multiple applications, we create 8 workload mixes. Each one of them is a subset of the 6 models that we use in the earlier experiments. The y axis shows the normalized value of Pocket compared to a monolithic Docker deployment for application latency and total memory usage (total_mem). Lower is better. Overall, Pocket leads to shorter latency results and shows reduced memory usage, compared to the monolithic approach. . . . .	36
4.1	Breakdown of end-to-end latency for intra-node echo RPCs implemented using shared memory (shmem) and gRPC. The figure quantifies the portion of total latency attributed to memcpy as a function of message size. . . . .	41
4.2	DSA programming model. The CPU prepares the task descriptor and submits it to DSA. DSA executes the task and sets the completion flag. The CPU then checks the completion flag to determine if the task is complete. . . . .	47

4.3	Comparison of polling strategies on latency and CPU usage (1MB transfer. <code>lazypoll</code> : polling every $100\mu\text{s}$ ; <code>busypoll</code> : polling with <code>yield</code> but no sleep; <code>UMWAIT</code> : polling with usermode interrupt.) . . . . .	49
4.4	Performance comparison of DSA and CPU <code>memcpy</code> under different memory conditions. Copying to a pinned buffer reduces latency by 95%, and reusing the same buffer achieves a 97% reduction, both relative to cold-buffer access. . . . .	50
4.5	LLC miss rates under single and four-threaded execution, comparing <code>cpu_memcpy</code> , <code>dsa_memcpy</code> , <code>dsa_memcpy(\$inject)</code> (Microbenchmark: summation over all elements in the destination buffer after memory copy). . . . .	51
4.6	Comparison of memory copy execution paths between CPU and DSA. CPU-based <code>memcpy</code> naturally integrates with the cache hierarchy, while DSA-based <code>memcpy</code> bypasses the cache, accessing DRAM directly. . . . .	51
4.7	Overview of Rocket architecture. Its components reflect key design principles such as page fault avoidance, maximizing parallelism, configurable selective offloading with or without cache injection enabled, and efficient synchronization. . . . .	53
4.8	Execution mode structure in Rocket. Each mode differs in synchronization and overlap strategy. . . . .	55
4.9	The latency of <code>memcpy</code> increases linearly with the target buffer size, at approximately $33.4\ \mu\text{s}$ per 1MB on our hardware. . . . .	57
4.10	Impact on execution time breakdown (left), throughput in images/sec (middle), and end-to-end latency improvement over CPU-based baseline (right) across for different Rocket IPC implementations for <code>MobileNetV2</code> , <code>XGBoost</code> , <code>PageRank</code> , <code>MilvusDB</code> , <b>blue ViT (CPU)</b> , <code>ViT (GPU)</code> under varying system load: (a) undersubscribed ( $n=1$ ), (b) matched ( $n=2$ ), and (c) oversubscribed ( $n=3$ ). . . . .	63
4.11	End-to-end latency decomposition across devices and execution modes ( <code>MobileNetV2</code> ). . . . .	64
4.12	Normalized instruction counts, CPU cycles, and bus cycles with DSA-based offloading, relative to the synchronous CPU baseline from <code>MobileNetV2</code> benchmark. . . . .	65
5.1	Design blueprint of Skyrocket . . . . .	71

5.2	Comparison of end-to-end results demonstrating that the async-\$ configuration yields performance equivalent to the optimal setting. . . . .	83
5.3	CDF of end-to-end decision overhead introduced by the SkyRocket configuration engine. The distribution shows that the vast majority of predictions complete within tens of microseconds, indicating negligible impact on overall IPC latency. . . . .	85
5.4	Timeline trace showing SkyRocket’s online adaptation to dynamic workload changes. The system switches between CPU Copy, DSA + \$bypass, and DSA + \$inject based on real-time monitoring of data size (< 5 MB vs. > 10 MB) and LLC pressure, effectively preventing cache pollution and optimizing transfer efficiency. . . . .	86
7.1	Design blueprint of Skyrocket . . . . .	102

## SUMMARY

Modern data-intensive applications, ranging from large-scale analytics pipelines to multimodal AI inference systems, are increasingly constrained by the overhead of transferring and managing large volumes of intermediate data across complex memory hierarchies. While emerging hardware innovations, such as programmable memory engines, offer potential solutions, existing Inter-Process Communication (IPC) mechanisms remain rigid and CPU-centric, failing to adapt to dynamic workload behaviors or to leverage these architectural advances effectively. This thesis explores how IPC services can be redesigned to overcome these limitations, proposing a new class of adaptive IPC mechanisms that integrate architecture-level advances with system-level flexibility.

First, the thesis introduces **Pocket**, a resource-aware IPC system designed to enable efficient split-architecture deployments. Addressing the “boundary tax” inherent in decoupled components, Pocket integrates lightweight resource management directly into the messaging interface. By allowing messages to carry resource expectations, Pocket enables just-in-time resource amplification and receiver-side adaptation. This design eliminates the performance penalties typically associated with isolation, effectively bridging the gap between monolithic efficiency and microservice flexibility.

Second, the thesis presents **Rocket**, a multi-backend IPC runtime that intelligently offloads memory copy operations to hardware accelerators (e.g., Intel DSA) or optimized kernel services. Unlike naive offloading approaches, which can degrade performance due to cache interference or synchronization overheads, Rocket employs backend- and workload-aware strategies to orchestrate data movement. It provides a suite of execution modes and hybrid completion check mechanisms to determine when offloading is beneficial based on data volume, locality, and system noise, thereby maximizing computation-communication overlap in high-throughput pipelines.

Finally, the thesis proposes **SkyRocket**, a runtime-adaptive framework that optimizes

IPC for heterogeneous data flows. Unlike static or naive configurations that apply a uniform strategy regardless of payload characteristics, SkyRocket leverages lightweight workload signatures to drive real-time adaptation. By dynamically switching between execution modes and backend strategies, it achieves adaptive control, ensuring that the IPC mechanism evolves in lockstep with the varying demands of multimodal applications. SkyRocket effectively narrows the performance gap between general-purpose IPC stacks and finely tuned, task-specific solutions.

Together, Pocket, Rocket, and SkyRocket demonstrate a practical and scalable path toward performance-aware IPC systems. By aligning IPC logic with modern hardware capabilities and application-level variability, this dissertation presents a comprehensive design space for efficient data movement in the era of hardware-accelerated, data-centric computing.

# CHAPTER 1

## INTRODUCTION

Modern applications are data intensive and inherently complex. They are typically organized as pipelines whose components consist of microservices and hardware accelerators. Inter-process communication acts as the glue that connects these components, but it is also a critical bottleneck that directly constrains end-to-end performance.

The memory-intensive nature of modern applications stems from both memory capacity and memory bandwidth demands. Application runtimes such as CUDA [1], ROCm [2], Tensorflow [3], and PyTorch [4] are bundled into each instance at deployment time. These runtimes are replicated within each application instance, leading to substantial memory bloat. In resource-fixed environments, this replication becomes a fundamental scaling bottleneck, as memory capacity is rapidly exhausted with the addition of new instances, making runtimes a primary contributor to memory capacity pressure [5].

To mitigate this inefficiency, split architectures can be suggested. In such design an application instance is decoupled into lightweight application logic (i.e. code-level execution logic) and shared, heavyweight runtime backends such as PyTorch. However, deploying applications in this manner fundamentally changes the execution model. A previously monolithic application is divided into tenant-specific instances and shared backend services, introducing frequent communication across isolation boundaries. This shift gives rise to significant communication overheads, often referred to as boundary tax [6, 7, 8], which can offset or even negate the expected gains from resource sharing.

The core challenge is that existing IPC mechanisms are largely oblivious to this setting. Conventional IPC [9, 10, 11] is designed as a generic data conduit and remains agnostic to runtime-varying resource demands and system-level resource availability. It provides no support for conveying execution context or adapting communication behavior based on

backend load, memory pressure, or accelerator availability. As a result, current systems are stuck with inefficient monolithic designs at the expense of memory inefficiency [12].

In parallel, recent multimodal input data, where a single sample consists of multiple heterogeneous modalities such as images, video, audio, and text, often reaches several megabytes per sample [13]. Unlike unimodal inputs, multimodal pipelines typically preserve modality-specific representations across multiple processing stages prior to fusion, significantly increasing the volume of intermediate data [14]. These inputs are processed in batches of tens to hundreds, and the resulting intermediate data are propagated through the pipeline [15, 16]. This process rapidly consumes memory bandwidth and exacerbates system-level bottlenecks. Increasingly, performance bottlenecks are dominated not by compute throughput, but by the cost of transferring and managing large volumes of intermediate data across memory hierarchies [17, 18]. Multi-stage processing pipelines routinely exchange tens or hundreds of megabytes per stage, and the payloads often consist of unstructured, high-dimensional data such as images, video frames, or dense embeddings [19, 20, 21, 22, 23]. As a result, the movement and staging of data within a node have become critical determinants of overall application performance.

Modern platforms support a range of data-movement mechanisms, including CPU-driven copies, kernel-assisted paths [24], memory-controller-assisted transfers [25], and programmable copy engines [26, 27]. Each mechanism exhibits distinct trade-offs in latency, throughput, setup overhead, and sensitivity to memory locality and synchronization. Despite this diversity, conventional IPC reduces data transfer to a single, fixed copy path, most often CPU-centric, and fails to account for these differences. For example, recent hardware innovations, such as programmable memory engines (e.g., Intel’s Data Streaming Accelerator (DSA) [26]), provide new mechanisms to offload memory operations directly from user space, offering the potential to alleviate CPU contention and memory bandwidth pressure. These accelerators promise fine-grained, low-overhead control over data movement, enabling systems to optimize for throughput, latency, or energy efficiency depending

on workload demands.

Despite these advancements, existing IPC mechanisms have seen relatively little change. Traditional IPC systems [10, 9] were designed under assumptions that are increasingly invalid: small, uniform control messages, negligible memory copy costs, and static resource availability. As data volumes and workload heterogeneity grow [19, 20, 21, 22, 23], these assumptions break down. Current IPC designs [11] typically allocate resources rigidly and manage data movement uniformly, without accounting for real-time system load, data structure variability, or the potential trade-offs introduced by hardware-assisted transfers.

Moreover, while memory offloading mechanisms such as DSA offer significant opportunities, they also introduce new complexities. A new question is now what data mover should be used. Also, their effectiveness is highly sensitive to factors such as transfer size, access locality, and synchronization behavior [28, 29]. Without adaptive policies that selectively engage or bypass hardware offloading based on dynamic conditions, systems risk incurring additional overheads such as cache pollution, memory bus contention, or inefficient CPU stalling. A naive, opportunistic use of hardware acceleration can thus be detrimental, rather than beneficial, in certain scenarios. In addition, when multiple system parameters interact, the relationship between configuration choices and their performance impact often becomes non-linear. Under such conditions, heuristic approaches or simple lookup tables become difficult to apply reliably, as they fail to capture the complex interactions among parameters.

Compounding these challenges is the growing heterogeneity within modern applications themselves. Workflows increasingly involve a mix of small metadata messages and large, unstructured payloads within the same communication stream [30, 31, 32]. The diversity in data size, format, and reuse patterns demands IPC strategies that are aware of content characteristics, system conditions, and hardware capabilities simultaneously. Static or one-size-fits-all IPC models are no longer sufficient.

Addressing these challenges requires a new generation of IPC systems that can dynam-

ically adapt to workload behaviors, intelligently leverage emerging hardware features, and manage system resources in a coordinated and responsive manner. This dissertation investigates how integrating lightweight resource management, selective hardware offloading, and runtime workload awareness into IPC can significantly improve system performance and scalability in modern data-intensive environments.

## 1.1 Statement of Problem

- **Conventional IPC prevents memory-efficient sharing of heavyweight runtimes.**

Modern applications increasingly rely on heavyweight runtime frameworks such as CUDA, TensorFlow, and PyTorch. These runtimes are embedded into each application instance, substantially inflating memory footprint and creating a scaling bottleneck in resource-fixed environments.

Decoupling lightweight application logic from shared runtime services is a promising way to mitigate this memory pressure. By amortizing the cost of heavyweight runtimes across instances, such deployments can significantly reduce per-instance memory usage. However, realizing this design requires frequent communication between tenant instances and shared services.

Conventional IPC mechanisms are ill-suited for this setting. They impose high latency and serialization overheads when crossing isolation boundaries and operate as passive data conduits, independent of runtime-induced resource demand or backend resource availability. As a result, communication overheads dominate, undermining the benefits of runtime sharing and pushing systems back toward memory-inefficient monolithic deployments.

- **Conventional IPC fails to expose diverse data-movement mechanisms through a unified interface.**

As applications become increasingly data intensive, efficient data transfer has be-

come a first-order concern. Modern systems already provide multiple data-movement mechanisms, including CPU-driven copies, kernel-assisted paths, and programmable copy engines. Each mechanism exhibits distinct performance characteristics and configuration parameters, and no single choice is optimal across workloads.

Despite this diversity, conventional IPC presents data transfer through a monolithic abstraction. It assumes a single fixed copy path, typically CPU-based, and hides alternative data movers behind the system boundary. As a result, applications are unable to express transfer-specific intent, such as latency sensitivity, throughput orientation, or locality constraints, even when suitable mechanisms exist in the underlying system.

This thesis addresses this limitation by exposing multiple data-movement mechanisms through an abstract IPC interface. Rather than hardwiring a single transfer path, the interface makes data-movement choices explicit and provides a common set of configuration knobs that allow applications or higher-level runtimes to select appropriate mechanisms. By elevating data movers to first-class IPC components, the system enables informed trade-offs without entangling applications with low-level hardware details.

- **Existing IPC models lack automated adaptation to growing configuration complexity.**

Modern data-intensive applications exhibit substantial variability in data size, structure, and access patterns. At the same time, systems increasingly offer a diverse set of data-movement mechanisms, each with its own performance trade-offs and configuration parameters. As workload diversity and data-mover diversity grow in tandem, the space of possible configurations expands rapidly.

This expansion has turned configuration into a significant burden. The performance impact of data-movement choices is often non-linear, and small changes in workload

characteristics can shift the optimal configuration across data movers or parameter settings. Manual tuning and static policies are therefore brittle and difficult to maintain, particularly as workloads evolve over time.

Existing IPC models provide no support for managing this complexity. They neither observe workload behavior nor automate the selection and tuning of data-movement mechanisms. This thesis argues that IPC must incorporate machine-learning-based automation to reduce configuration burden and adapt transfer behavior dynamically at runtime. Such automation is essential for sustaining performance in the presence of heterogeneous workloads and increasingly diverse data-movement options.

## **1.2 Thesis Statement**

This thesis argues that Inter-Process Communication must evolve from a passive data conduit into an intelligent runtime layer that enables scalable performance in modern data-intensive systems. Conventional IPC falls short because it is fragmented and cannot adapt to dynamic workloads. This thesis introduces a runtime-aware resource manager (Pocket), a unified data mover (Rocket), and an adaptive controller (SkyRocket) that together provide an active IPC runtime with capabilities beyond static mechanisms.

## **1.3 Contributions**

This dissertation advances IPC efficiency by developing runtime-adaptive, multi-backend designs for data-intensive systems. It introduces Pocket, which embeds resource hints into messages for dynamic memory management, and Rocket, a unified IPC runtime that orchestrates multiple data movement backends, including legacy memory copies and software- or hardware-assisted transfer engines. Building on these systems, SkyRocket pursues a more general form of runtime adaptation that selects and composes heterogeneous backends according to workload demands. In support of the thesis statement, this dissertation makes the following contributions:

- **IPC-Driven Resource Amplification for Efficient Split-Architectures.** (Chapter 3)

To make split-architecture deployments viable at the edge, this thesis proposes Pocket, a high-performance IPC system that eliminates the boundary overhead between decoupled components. Pocket introduces the concept of inline resource amplification, where the IPC mechanism itself acts as a conduit for dynamic resource provisioning. By embedding resource hints directly into the message passing protocol, Pocket allows lightweight frontends to instantaneously "amplify" the backend's capabilities only for the duration of a request. This transforms IPC from a passive data pipe into an active enabler of high-density serving, achieving the isolation benefits of containers with the performance of monolithic function calls.

- **Memory Offloading for IPC Acceleration with Abstract API for Diverse Data Movers** (Chapter 4)

Recognizing that memory copy operations increasingly dominate IPC costs in data-intensive applications, this thesis develops Rocket, a high-performance IPC runtime that selectively offloads data movement to backends such as hardware accelerators or kernel services like Intel's DSA or Copier [24]. Rocket exposes configurable execution modes, cache-aware transfer policies, and hybrid polling strategies to adapt offloading decisions to workload characteristics and system conditions. The novelty of Rocket lies in its systematic characterization of the critical factors, such as transfer size, data reuse patterns, cache pressure, and synchronization overheads, that govern performance trade-offs in memory offloading. Beyond mere analysis, Rocket leverages these insights to provide an adaptive framework that dynamically balances efficiency and throughput, avoiding the pitfalls of naive offloading. Rocket achieves significant improvements in throughput, CPU cycle reduction, and system efficiency compared to both CPU-only IPC and static offloading approaches.

- **Self-Configuring IPC Middleware for Heterogeneous Workloads.** (Chapter 5)

This dissertation presents a self-configuring IPC middleware that manages transfer-configuration complexity in the presence of heterogeneous and multimodal workloads. Rather than exposing a growing set of backend-specific parameters to applications, the IPC runtime incorporates a lightweight decision component that selects effective transfer configurations based on observed execution context.

The system relies on a learned performance model to guide configuration choices, enabling IPC to adapt to workload variation without manual tuning or static policies. Evaluation across diverse workloads shows that runtime-managed configuration consistently improves transfer efficiency compared to fixed configurations that fail to generalize under changing conditions. This contribution elevates configuration management to a core responsibility of the IPC runtime, aligning adaptive behavior with the goals of unified middleware design.

## **CHAPTER 2**

### **BACKGROUND**

Modern data-intensive systems are undergoing a fundamental architectural shift. As applications increasingly rely on large-scale data movement, performance bottlenecks have migrated from computation to communication [33, 34]. Systems must now adapt to constraints stemming from memory pressure, resource contention, and inefficient data transfer mechanisms that were not designed for today’s data volumes.

In response to these application trends, a new class of data movers is emerging. These mechanisms now span a broad stack, from user-level software to kernel-level components, and extend across hardware domains, including on-chip processors and memory controllers. As a result, the data movement landscape is rapidly evolving. New capabilities, such as user-space programmable memory engines, are becoming increasingly available on commodity data center platforms.

This chapter provides essential context for understanding this shift. It first highlights the growing complexity of inter-process communication in data-centric workloads and quantifies the overheads imposed by traditional IPC designs. It then presents the architectural opportunities enabled by hardware-assisted memory operations, which are becoming critical to addressing these inefficiencies.

#### **2.1 Cost of Data Movement in Datacenter Systems**

Modern datacenter workloads are increasingly constrained by the cost of data movement rather than computation. Large-scale production studies show that a significant fraction of execution time and energy is spent moving data across memory hierarchies, not performing arithmetic operations. As a result, memory access and internal data transfers now contribute substantially to overall system overhead.

Prior measurements from Google datacenters indicate that memory-related activities, including intra-process and inter-process data movement, account for a large share of resource utilization [34]. Figure 2.1 illustrates this breakdown, highlighting that the cost of moving data through the memory subsystem already rivals, and in some cases exceeds, the cost of computation. These observations suggest that improvements in compute throughput alone are insufficient to deliver proportional end-to-end performance gains.

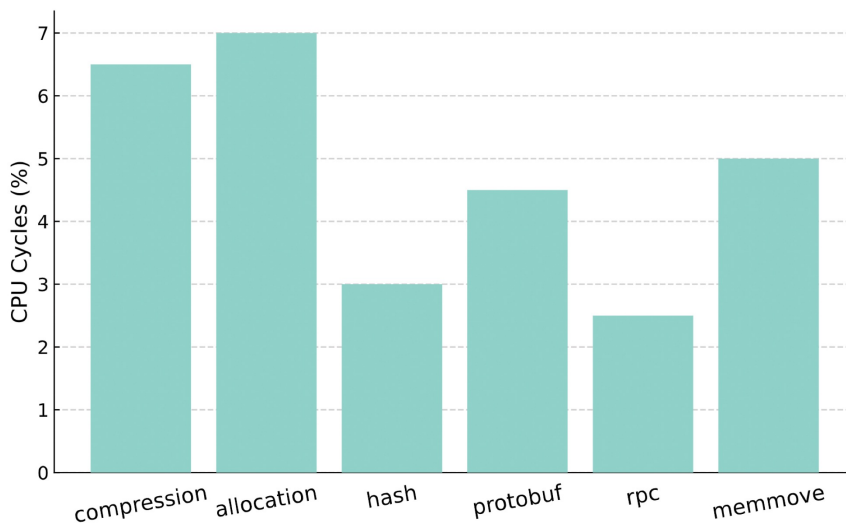


Figure 2.1: Breakdown of Datacenter Tax (Reconstructed from [34])  
Breakdown of datacenter resource utilization, showing the growing contribution of memory access and data movement.

This imbalance reflects a broader architectural trend. Compute capability continues to scale through additional cores and specialized accelerators, while memory bandwidth and capacity improve at a much slower rate [35]. Consequently, internal data movement has emerged as a first-order bottleneck in warehouse-scale systems, with direct implications for efficiency, cost, and scalability.

These pressures are not limited to a narrow class of workloads. Emerging applications increasingly rely on frequent transfers of large intermediate data objects as part of their normal execution [36, 7]. As data volumes and pipeline complexity continue to grow, the cost of data movement is expected to increase further, intensifying the strain on existing system abstractions. [36, 15]

## 2.2 Modern Applications and the Data Movement Bottleneck

This section identifies the key performance bottlenecks that arise when current IPC systems are used in modern data-intensive pipelines. Large unstructured data transfers consume memory bandwidth inefficiently and require substantial CPU cycles. These inefficiencies intensify memory pressure in multi-tenant and edge environments, where heavy-weight frameworks increase resource contention and reduce system stability. The discussion shows that existing IPC mechanisms do not account for data size or resource constraints in their core behavior. As a result, there is an urgent need for IPC designs that incorporate data volume and resource management directly into communication operations.

### 2.2.1 Data-Intensive Applications

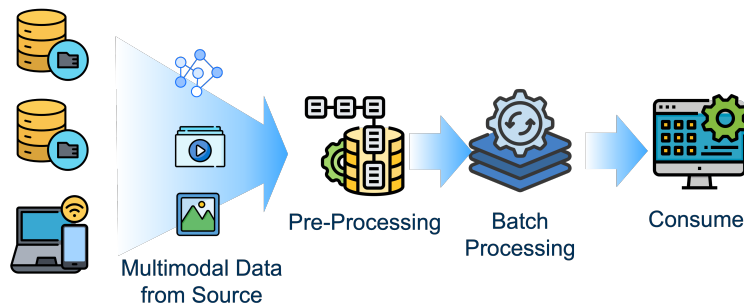


Figure 2.2: Modern Application Structure

Modern applications are structured as pipeline-style workflows that process multimodal data, resulting in frequent movement of intermediate data between successive stages.

Modern data-intensive applications, ranging from machine learning inference pipelines to large-scale graph analytics and video processing, are increasingly organized as multi-stage workflows (Table 2.1, Figure 2.2). Unlike traditional web services that exchange small text-based messages, these pipelines routinely exchange large intermediate results between processing stages [36], with typical data units reaching tens or hundreds of megabytes per transfer (Table 4.1). In many cases, payloads consist of large, unstructured data objects such as high-resolution images, embeddings, or feature vectors, often accompanied by lightweight structured metadata [13, 14].

Table 2.1: Representative Data Modality in Modern Applications

Representative memory sizes and batching configurations across common multimodal inputs, illustrating potential data movement overheads.

<b>Modality</b>	<b>Per-Input Size (Uncompressed in Memory)</b>	<b>Batching</b>	<b>Approx. Single Request</b>
Image [37, 38, 39] (low-res, 224×224)	224×224×4 = 200 KB (RGB, fp32)	32–256	Batch of 256: 51 MB
Image [40] (high-res, 4K)	3840×2160×4 = 33 MB (RGB, fp32)	32	Batch of 32: GB scale
Medical Image (DICOM)	A few MBs [23]	10	100+MB or GB scale w/ batching
Video	Frame size × FPS × duration	5–10	GB-scale possible
Multimodal (text + image)	Text embeddings + image tensors	1–N	Possibly 100+MB

This shift toward large, opaque data transfers fundamentally alters the performance characteristics of inter-process communication. Because the runtime treats intermediate payloads as uninterpreted binary objects, communication costs are driven primarily by data movement rather than message processing. As a result, efficient transfer mechanisms are critical to overall system performance.

This evolution has dramatically increased the volume and criticality of intra-node data movement. However, existing IPC mechanisms [9, 10, 11] were not designed with such demands in mind. Instead, they continue to rely on CPU-managed data copies through conventional memory hierarchies, resulting in redundant copying, high CPU cycle consumption, and inefficient use of system memory bandwidth.

Current IPC primitives are not optimized for large-scale data transfers and therefore incur significant overhead when directly applied to data-intensive pipelines (further illustrated in Section 2.2.2). In such settings, data movement, not computation, often dominates the overall cost, exposing performance bottlenecks. To address this, IPC systems must

evolve beyond basic message-passing interfaces to allow for optimizations that consider factors such as data size, transfer cost, and system resource constraints.

### 2.2.2 Memory Pressure and Resource Contention in Fixed-Resource Systems

As data volumes continue to increase across modern applications, memory bandwidth and capacity have emerged as critical constraints [35]. While computational throughput has improved through additional cores and specialized accelerators, the ability to move and stage large amounts of data within the system remains limited by relatively static memory infrastructure.

This constraint becomes particularly acute in fixed-resource environments such as edge computing platforms with multi-tenant deployments, where available memory resources are limited and shared across multiple workloads [41]. These challenges are further compounded by heavy-weight runtime frameworks such as TensorFlow [3] and PyTorch [4], whose static memory reservations and layered abstractions often inflate per-application memory footprints. When co-located on shared infrastructure, these frameworks exacerbate contention, making it difficult to reason about and control actual memory usage.

In these settings, bursts of memory demand can exceed the available capacity, leading to contention, latency spikes, or outright allocation failures.

Despite the growing importance of memory resource management, traditional IPC mechanisms provide no means to regulate memory usage or adapt communication patterns in response to changing system conditions. They are designed primarily to deliver data reliably, without mechanisms to coordinate resource consumption or mitigate the effects of transient load surges.

This lack of resource-aware communication primitives leaves systems vulnerable to performance degradation under high memory load, and highlights the need for IPC designs that are capable of reacting to resource pressure, not through external orchestration, but as an integral part of communication behavior itself.

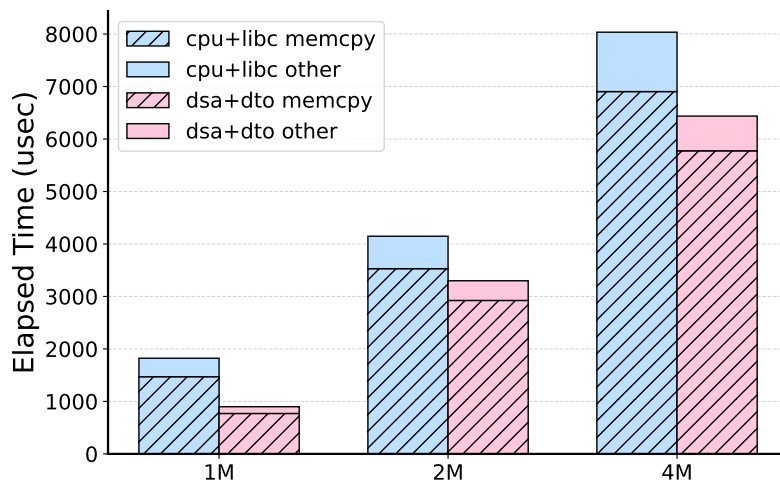


Figure 2.3: Execution time breakdown of a shared memory IPC microbenchmark. As the data payload increases from 1MB to 4MB, the memory copy operation (hatched area) consumes the vast majority of CPU cycles.

## 2.3 Closing the Memory Copy Bottleneck: Leveraging Emerging User-Space Memory Engines in IPC

This section quantifies the performance bottlenecks in shared memory IPC and shows that raw memory copy operations dominate end-to-end latency. It examines the constraints of current hardware offloading techniques such as Intel’s DSA, where software overheads frequently reduce the expected throughput benefits. The analysis also identifies a widening gap between emerging user-programmable memory engines and existing software stacks, which struggle to exploit these capabilities effectively. Taken together, these findings underscore the need for flexible and adaptive IPC designs that can manage hardware and software trade-offs in a more intelligent and responsive manner.

### 2.3.1 Quantifying the Bottleneck: The Dominance of Memory Copy

To understand where time is actually spent in shared memory IPC, we profiled a simple echo microbenchmark and decomposed the end-to-end latency into data movement and non-data-movement components (Figure 2.3). Despite the advantages of shared memory,

the cost of explicitly moving payloads into and out of shared buffers remains substantial.

Figure 2.3 shows the latency breakdown for payloads ranging from 1MB to 4MB. Across all sizes, the memory copy portion (hatched area) dominates the total request latency, while the remaining operations contribute only a small fraction of the total cost.

For the baseline CPU implementation (`cpu+libc`), the elapsed time increases almost entirely due to the scaling of the copy itself. `memcpy` accounts for the overwhelming majority of latency, making clear that data movement, rather than control logic, is the limiting factor in shared memory IPC.

A similar pattern appears when the copy is offloaded to a DMA engine such as Intel’s Data Streaming Accelerator (DSA) (`dsa+dto`). Although the accelerator can move data without occupying the CPU, the end-to-end improvement remains limited. The offload path introduces its own software overheads, including descriptor preparation, queue submission, and completion handling. These steps are not negligible, and in current systems they interact with the accelerator in ways that prevent it from reaching its effective peak throughput. As a result, the benefits of hardware-assisted data movement are partly offset by the inefficiencies of the surrounding software stack, and the total latency continues to be dominated by the cost of transferring multi-megabyte payloads.

### 2.3.2 Rethinking IPC with Novel Data Movers

As data center architectures demand tighter coupling between compute and data movement, the gap between IPC software assumptions and hardware capabilities has become more pronounced. The availability of user-space programmable engines marks a turning point where rethinking IPC integration is not just possible, but necessary for scalable performance.

Traditional mechanisms for accelerating data movement have historically relied on hardware-driven approaches such as DMA. While effective for fixed I/O paths, conventional DMA interfaces are typically constrained by kernel mediation (a few hundreds of

CPU cycles), static buffer registration, and limited programmability. These design choices make them ill-suited for the dynamic memory allocation patterns and diverse data paths common in modern applications.

Recent systems have also introduced software-level data movers [24] that are implemented across different layers of the system stack, leveraging optimized memory copies, shared memory abstractions, or runtime-managed buffering. Together, these developments reflect a broader trend toward exposing data movement as a first-class system capability, spanning both hardware accelerators and software-managed mechanisms.

On the hardware side, recent architectures have begun to expose more flexible data movement primitives, ranging from memory-controller-assisted mechanisms [25] to programmable memory engines [26, 27]. For example, Intel’s Data Streaming Accelerator (DSA) introduces a class of programmable engines that allow user-space submission of copy operations, support virtual memory addressing, and provide optional cache management features.

These capabilities open the door for IPC systems to offload large data movements without heavy system-level coordination, offering the potential to free CPU cycles, reduce memory bus contention, and mitigate the data center tax associated with internal data movement.

However, this growing diversity of data movers has also increased the complexity of the IPC design space. Existing IPC abstractions were developed under the assumption of CPU-managed data copies and provide limited support for reasoning about alternative transfer mechanisms. As a result, integrating heterogeneous data movers—ranging from programmable hardware engines to software-based transfer paths—often requires ad hoc decisions that are external to the IPC layer.

Without systematic support for selecting and coordinating among these mechanisms, systems struggle to balance performance, resource usage, and portability. This fragmentation motivates IPC designs that can accommodate multiple classes of data movers and

adapt their behavior to workload characteristics and system conditions.

Moreover, the characteristics of emerging data-intensive applications, such as frequent large-payload transfers, multimodal data, and irregular access patterns, present unique challenges. While these traits align with the intended benefits of programmable memory engines, empirical studies on how real workloads interact with these features remain limited. Understanding this interaction is critical to building IPC systems that can adapt to both hardware and application behavior.

## 2.4 Summary

Modern data-intensive applications increasingly stress system resources through large-scale intra-node data movement. Traditional IPC mechanisms, originally designed for small control messages and CPU-managed memory transfers, are ill-suited for the bandwidth and memory demands of contemporary workloads. As a result, IPC overheads emerge as significant performance bottlenecks, particularly in multi-stage pipelines processing large, heterogeneous data payloads.

At the same time, advances in hardware accelerators such as programmable memory engines provide new opportunities to offload and optimize data movement. Memory acceleration techniques such as Intel’s DSA or data transfer as a kernel service, enable user-space, flexible memory operations with features such as virtual memory support and cache management. However, integrating these capabilities into IPC systems requires new mechanisms for dynamic transfer selection and resource-aware coordination, which are absent in traditional designs.

Existing research has explored both software-based IPC optimizations and hardware-assisted memory operations. Yet, prior efforts largely focus on regular, homogeneous workloads and do not address the dynamic, multimodal, and resource-variable conditions found in real-world applications. Therefore, this thesis contributes the design and evaluation of adaptive, hardware-aware IPC systems that leverage emerging memory engines to improve

performance under diverse and evolving workload behaviors.

## **CHAPTER 3**

### **IPC RUNTIME FOR DYNAMIC RESOURCE ALLOCATION IN RESOURCE-FIXED ENVIRONMENT**

The increasing demand for machine learning (ML) serving has driven a move toward multi-tenancy, where shared infrastructure runs inference on behalf of multiple diverse tenants. In these environments, maximizing resource efficiency and supporting higher tenant density are critical goals. While these challenges are pervasive across modern infrastructure, they are particularly acute in settings with fixed hardware capacity, where the manner in which each application consumes resources determines overall system scalability. In such environments, how each application consumes resources is a key determinant of overall system scalability.

Conventional cloud-native deployment models assume monolithic application instances that embed their own ML frameworks and runtimes. Even with containers or micro-VMs, identical libraries and runtime states are repeatedly loaded for each instance. While this redundancy is less problematic in resource-rich data centers, it directly limits the number of instances that can be supported on edge servers with fixed memory and compute capacity.

In this context, this chapter focuses on removing unnecessary redundancy across the multi-tenant ML serving stack to improve resource efficiency. This approach is relevant for improving ML serving efficiency in multi-tenant settings and for scaling up the number of clients that can be served with a fixed set of resources, whether in the cloud or at the edge. Reducing per-instance resource allocation alone risks performance degradation and increased latency. The core challenge is therefore to maintain performance within a limited resource budget while enabling the sharing and reuse of duplicated ML runtimes and framework overheads.

### 3.1 The Need for Split-Architecture for ML Serving and Design Challenges

Machine learning (ML) applications are transforming not only how businesses operate but also everyday activities. In order to serve end users with lower latency and to reduce the cost of data movement from end users to the cloud, ML applications are expanding across the entire computing spectrum, from the cloud to the mobile edge [42, 43, 44]. As a result, the mechanisms and systems which support these inherently resource-intensive applications need to be adapted to operate efficiently in shared environments with fixed resource capacities.

A similar goal of improving resource efficiency and scalability has driven innovation toward lightweight sandbox abstractions for datacenter clouds. Concrete examples include cloud-native stacks based on Docker containers [45], Kubernetes [46], and micro virtual machines (uVMs) [47]. These offer robust infrastructure, near-native performance, and isolation, and are increasingly shaping the edge computing landscape [48, 49, 50, 51]. However, we observe that these technologies are not sufficient to address the constraints at the edge when considering resource-intensive ML applications.

The primary bottleneck arises from the **monolithicity** of these cloud-native technologies. Underlying frameworks such as TensorFlow [3] and PyTorch [4], along with specialized hardware packages [52, 53, 54, 55], lead to excessive resource usage. For instance, an application based on the YOLO model [56, 57] consumes over 1,000 MB of memory (1,096 MB in FireCracker and 1,074 MB in Docker). In a mobile edge computing (MEC) node equipped with 32 GB of memory, this limits the capacity to only 31 instances. While horizontal scale-out is common in datacenters, it is often unfeasible at the edge due to power and space constraints. Consequently, as the number of instances increases, total resource usage grows linearly (Figure 3.1a), leading to severe **linear bloat**. Each instance redundantly loads its own dependencies and runtime state, preventing the node from scaling to the 100s or 1000s of users required for practical MEC deployment [58].

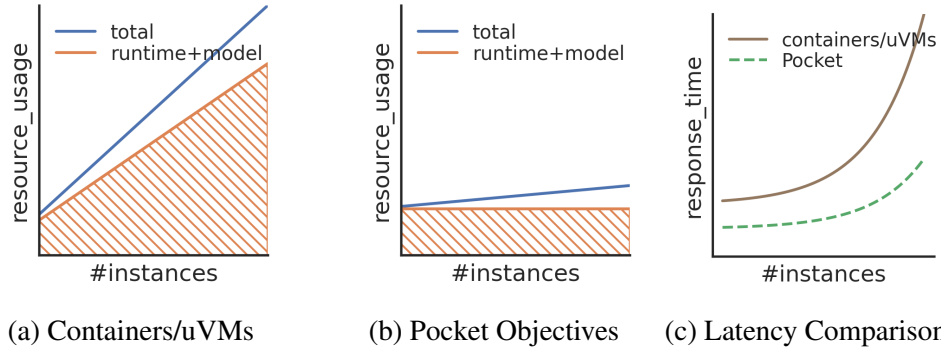


Figure 3.1: Existing solutions based on monolithic approaches scale poorly due to resource requirements of ML runtimes. End-to-end request latency in (c) (y-axis: milliseconds, x-axis: the number of concurrent instances). Pocket aims to serve more instances with better performance by sharing and efficiently handling the runtimes’ resource demand.

To mitigate this, we propose a **split architecture** that separates the application into two components: a lightweight application-specific frontend and a shared backend runtime component (Figure 3.1b). Transforming the deployment from the current monolithic model (Figure 3.2a) to a *Runtime-as-a-Service* model (Figure 3.2b) allows for better resource efficiency by amortizing the ML runtime demand across multiple clients. Under monolithic designs, latency increases sharply beyond a concurrency threshold due to memory contention and runtime duplication, whereas Pocket sustains lower latency by amortizing shared runtime costs. Pocket aims to achieve strong resource efficiency while also delivering performance, such as response time, that is superior to or competitive with existing solutions (Figure 3.1c).

However, realizing this split architecture introduces significant design challenges. First, naively separating these components shifts the performance bottleneck to Inter-Process Communication (IPC). Traditional mechanisms like gRPC introduce per-client memory demand and runtime overheads that are particularly significant given the large parameter sizes of ML APIs, often failing to satisfy strict latency requirements. Second, sharing a backend raises critical concerns regarding the isolation of tenant-specific data. Finally, managing diverse frontends alongside a shared backend complicates resource allocation on limited edge hardware.

This thesis presents **Pocket**, a system designed to address these challenges by providing a high-performance, resource-efficient, and isolated split-architecture for ML serving at the edge. Pocket introduces a novel interface for shared heavyweight runtimes, enabling efficient resource amortization while minimizing the deployment and initialization overheads of bulky ML frameworks. Our design integrates (1) lightweight IPC for high performance, (2) nested namespaces for robust multi-tenant isolation, and (3) a dynamic resource amplification mechanism that transiently redistributes server resources during runtime service calls. As a result, Pocket improves performance by up to 80% and supports  $1.3\text{--}9.9\times$  more instances on CPUs and  $2\text{--}20\times$  more on GPUs compared to traditional approaches.

- We design Pocket, a split-architecture system for edge ML serving that replaces monolithic deployments with a shared runtime and lightweight application units.
- We introduce a suite of mechanisms to overcome the overheads of backend sharing, including high-performance IPC, cross-client isolation, and transient resource management via inlined resource amplification.
- We evaluate Pocket on both CPU and GPU platforms, demonstrating its ability to improve application performance and enable edge servers to scale by up to an order of magnitude more instances compared to monolithic or statically partitioned strategies.

### 3.1.1 Split Architecture Components

This study assumes a decomposition of ML applications into two roles to improve resource efficiency.

- **Frontend (FE):** A lightweight container responsible for the user interface, preprocessing and postprocessing of input data, and business logic. Each user runs an isolated FE instance.

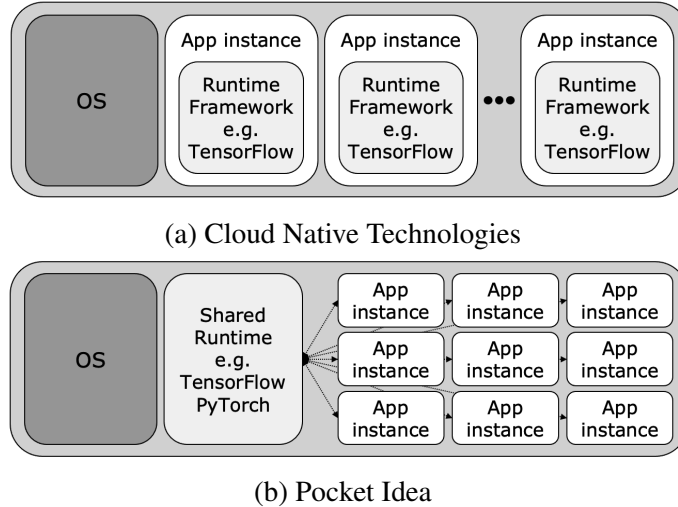


Figure 3.2: ML applications and their resource-demanding ML runtimes lead to resource bloat when used with existing cloud-native technologies, which limits scaling. By introducing a shared backend, the ML runtime resource demand is expected to be amortized.

- **Shared Backend (BE):** A shared service container that hosts heavy ML runtimes such as TensorFlow or PyTorch and loads the models. It processes inference requests from multiple FEs.

### 3.1.2 Problem Statement: Bottlenecks from Physical Separation and IPC

Despite reducing memory redundancy, the physical separation of FE and BE introduces a critical performance bottleneck in **IPC**. This issue is especially urgent in edge environments that require strict latency guarantees.

Traditional IPC mechanisms such as gRPC merely relay data without addressing runtime-level contention. As a result, they fail to meet these tight latency constraints and do not resolve the competition that emerges when multiple FEs share the same runtime. Consequently, **IPC becomes a limiting factor that threatens the viability of the split architecture itself.**

## 3.2 Obstacles to Scaling ML Serving at the Edge

### 3.2.1 Inefficiency of Conventional Cloud-Native Stacks

A common strategy for deploying ML-centric applications [59, 60, 61, 44, 62] in edge environments involves migrating established cloud-based software infrastructures to edge-located servers [63, 64, 65]. However, our empirical analysis suggests that these cloud-native architectures are ill-suited for the edge. While these technologies effectively address specific cloud-level issues, they offer diminishing returns when subjected to the unique constraints of edge-based ML workloads.

Specifically, we tested modern lightweight virtualization and containerization, including Docker and Firecracker microVMs [47], which are frequently used to power serverless platforms. While these tools successfully minimize isolation overhead, they fail to address the fundamental problem: the massive memory footprint of ML frameworks. Regardless of how optimized the underlying sandbox is, each instance must still load extensive ML and hardware-specific libraries (e.g., CUDA, TensorFlow), leading to significant resource consumption. This issue is particularly acute in microVMs, which require additional system-level components like bootloaders and file systems for every instance.

Although Firecracker excels at rapid application startup compared to other microVMs, it does not alleviate the resource saturation caused by heavy ML dependencies. As demonstrated in Table 3.1, a Firecracker VM configured to support TensorFlow requires substantially larger per-instance memory and system resources than the minimal configurations evaluated in prior work due to the inclusion of full ML frameworks and supporting system components. As additional instances are deployed, this fixed per-instance overhead accumulates, resulting in a linear increase in total resource usage, mirroring the trend shown in Figure 3.1a (detailed metrics are provided in Section 3.4). These traditional methods do not facilitate scaling to a high density of clients on the fixed resource pools typical of edge nodes.

Table 3.1: Comparison of Firecracker configurations: a configuration for a simple function from [47] and a configuration for running TensorFlow-based YoloV3 applications. In the NSDI evaluation, the image size of Firecracker is zero because the boot time was measured as the loading time of a bare minimum kernel, not including any supporting services and not having underlying file systems.

	<b>NSDI ‘20 Evaluations [47]</b>	<b>Settings for YoloV3-tf2</b>
<b>Memory</b>	256MB	1152MB
<b>Linux Distro</b>	vanilla kernel	Ubuntu 18.04
<b>Image Size</b>	0	2GB
<b>Intended Use Case</b>	Serverless applications with basic functionality	Complex applications relying on heavy frameworks

### 3.2.2 Architectural Hurdles in Decoupled Runtimes

To overcome the bloat inherent in monolithic ML deployments, a logical alternative is to decouple the runtime stack from the application logic and share it across multiple tenants. In this ”split” model, the backend serves as a centralized entity hosting shared frameworks (e.g., TensorFlow) and pre-loaded models. The frontend remains lightweight, containing only application-specific logic, input handling, and preprocessing. While this approach directly targets the root cause of resource inefficiency, it introduces significant design hurdles.

**The Communication Bottleneck** The physical separation of components necessitates a high-performance communication layer. We evaluated gRPC, the industry standard for microservices [66], but found it unsuitable for edge ML. gRPC introduces significant latency, typically hundreds of microseconds, which is exacerbated in interpreted or JIT-compiled environments. Furthermore, transmitting large data blobs (e.g., high-resolution video frames) through gRPC incurs heavy costs from serialization and kernel-level network stacks. Even more concerning is the memory overhead; gRPC consumes roughly 460 MB of memory for every new connection. Given that a standalone YOLO instance consumes

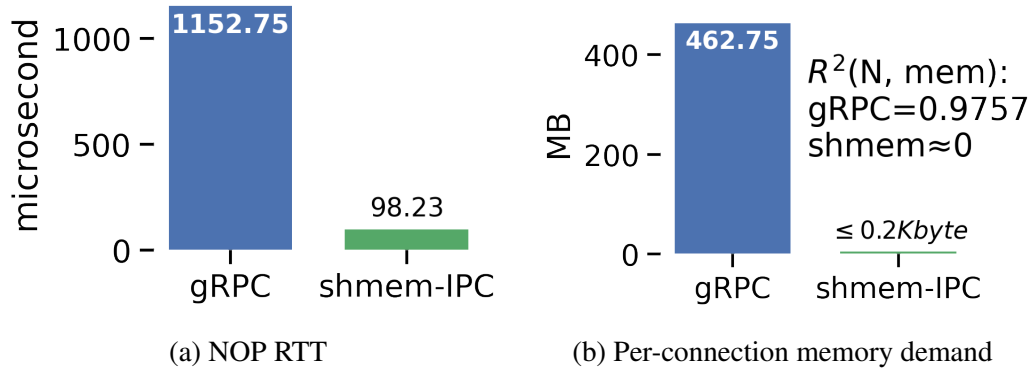


Figure 3.3: Comparison of (a) round-trip time (unit:  $\mu sec$ ) and (b) incremental memory footprint per connection for gRPC and shmem-IPC. The regression fit ( $R^2$ ) represents the correlation between the number of concurrent tenants and the resulting server memory demand.

about 1074 MB, the memory cost of gRPC connections would negate nearly half of the savings gained from sharing the backend. To meet the latency and memory-efficiency requirements, we target a shared-memory-based IPC mechanism that avoids serialization, kernel crossings, and per-connection memory duplication (Figure 3.3).

**Multitenant Isolation Risks** Shifting to a shared backend architecture compromises the inherent security boundaries provided by per-client containers. In a multitenant edge environment where users do not necessarily trust one another, shared channels and backend objects become potential vectors for unauthorized data access or compromise. Existing fast-IPC solutions [67] often prioritize speed over security, leaving these vulnerabilities unaddressed. To ensure Pocket is viable for public edge deployments, we must implement specialized isolation mechanisms that restrict access to communication channels to only verified stakeholders.

**Dynamic Resource Management** Decomposing applications raises complex resource management questions. In monolithic cloud models (Figure 3.2a), resources are typically partitioned equally. However, in a shared-backend model, static allocation becomes inefficient. While a backend requires significant resources to process inference requests,

frontends may occasionally need bursts of power for heavy pre-processing (as seen in our SSDResNet50 tests). A rigid allocation strategy either starves the frontend or prevents the backend from utilizing available capacity during peak demand. Ideally, edge server resources should be dynamically and transiently redistributable between the frontend and backend based on real-time service demands, a capability currently missing in standard service infrastructures.

### 3.3 System Design for Efficient Resource Sharing and Isolation

Pocket re-architects the edge server stack by adopting a "Runtime-as-a-Service" model, as illustrated in Figure 3.2. Under this paradigm, ML applications are *pocketized* into lightweight frontend containers. These frontends manage application-specific tasks such as device interaction and data pre/post-processing, while delegating heavy ML execution to a shared backend service container. This decoupling allows infrastructure operators to deploy multiple ML framework versions (e.g., specific TensorFlow or PyTorch backends) as centralized services accessible via standard APIs. Figure 3.4 provides an architectural overview of these components and their interactions.

To maximize multi-tenant scalability without compromising performance, the design of Pocket rests on three technical pillars:(i) Lightweight IPC, which minimizes the overhead of cross-container delegation;(ii) Hardened Isolation, ensuring that shared resources do not weaken the security semantics of standard containers; and(iii) Resource Amplification, a just-in-time mechanism that dynamically shifts hardware resources to the backend during active service calls.

#### 3.3.1 Lightweight Pocket IPC

The communication backbone of Pocket must resolve three primary inefficiencies: redundant data copying, control-plane latency, and memory bloat. Standard RPC stacks [66] often force data to traverse the entire container and host networking stacks, incurring sig-

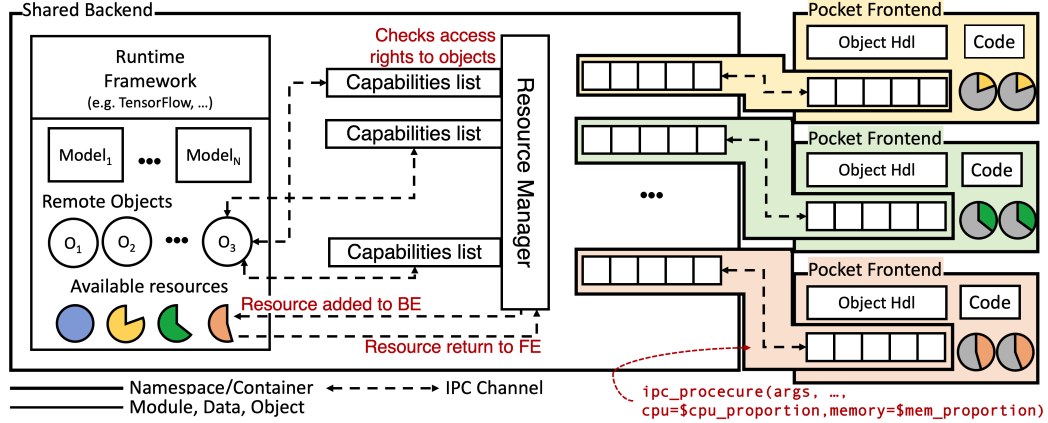


Figure 3.4: Design overview of Pocket. Pocket frontends connect to one of the shared backends via lightweight IPC and use it as an ML server. Every request is checked via capabilities list to prevent illegal access, and an additional layer of namespace and private queues for each channel hardens the isolation. A Resource Manager takes care of dynamic resource reallocation.

nificant marshaling overhead and memory pressure.

To address these, Pocket implements a custom IPC mechanism built on System V shared memory and message queues. Unlike state-of-the-art solutions like Nightcore [67], which may not fit the specific memory constraints of edge ML, our IPC is designed from the ground up to minimize data movement. By utilizing shared memory, Pocket frontends and backends can exchange large tensors without intermediate copies.

As shown in Figure 3.3, this design allows Pocket to maintain a memory footprint that is nearly independent of the number of active tenants, in contrast to the linear scaling observed in gRPC. While gRPC incurs a substantial per-connection state, averaging 462.75 MB per additional tenant, Pocket’s demand remains  $\leq 0.2\text{KB}$ . This discrepancy highlights the architectural difference in connection management: gRPC’s footprint is driven by the cumulative overhead of maintaining per-tenant HTTP/2 flow control buffers [68, 69] and framework-specific session contexts within TensorFlow [70]. In contrast, Pocket reduces the communication state drastically, utilizing a passive shared memory region where each additional connection requires only negligible metadata. This efficiency enables the high-density tenant scaling essential for edge environments.

### 3.3.2 Multi-Tenant Isolation

Sharing a backend inherently challenges the isolation boundaries established by Linux namespaces [71]. In a shared-memory model, a malicious tenant might attempt to "peek" into the communication channels of others or access unauthorized remote objects. To restore container-level security, Pocket introduces three layers of protection:

**Nested IPC Namespaces** Pocket employs an additional layer of nested namespaces to secure the communication channels. While the shared backend resides in a parent namespace, allowing it to manage all active queues, each frontend is restricted to a child namespace containing only its specific channel. This hierarchy is established transparently during initialization. Creating these namespaces is highly efficient, taking only  $54\mu\text{s}$ , and remains outside the critical request-processing path.

**Capability-Based Access Control** To prevent unauthorized object access within the backend's memory space, Pocket maintains individual capability lists for every frontend. When a frontend requests an operation on a specific object handle, the backend validates the handle against the frontend's capabilities. Access is granted only if the ownership is verified, effectively preventing cross-tenant data leakage.

**Private Per-Frontend Queues** Each channel is supported by a dedicated private queue visible only to the specific frontend-backend pair. This prevents starvation and allows for future integration of custom QoS policies. While the current implementation focuses on best-effort delivery, this per-queue architecture provides the necessary infrastructure for priority-based or weighted scheduling.

### 3.3.3 Resource Amplification

Standard monolithic applications rely on static resource limits, which are difficult to "right-size" for the unpredictable loads of edge ML. Static allocation is particularly inefficient at

the edge, where reserving peak resources for every idle instance leads to rapid exhaustion.

Pocket solves this through Resource Amplification, which couples IPC service calls with a temporary reallocation of hardware limits. When a frontend initiates a request, it "amplifies" the resources available to the backend for the duration of that task. This ensures the shared service has sufficient capacity exactly when needed, proportional to the concurrent load.

**Inlined Execution** Initial attempts to implement this via a centralized daemon were unsuccessful, as the 180ms overhead of daemon-mediated migration negated any performance gains. Consequently, Pocket utilizes inlined resource amplification via direct cgroup manipulation. By modifying cgroup resource limits directly during the IPC call, we reduce the reallocation overhead to just  $800\mu\text{s}$ —a  $220\times$  improvement over the daemon-based approach.

**Context-Aware Allocation** Resource parameters are configurable per frontend to accommodate diverse application needs. For instance, an application performing heavy image preprocessing requires its frontend to retain more memory than a simple text-based client. Pocket provides an API for developers to specify amplification factors at launch time, ensuring the system dynamically balances the fixed edge resources between frontends and the shared backend as workloads shift.

### 3.4 Evaluation: IPC Performance and Resource Efficiency

We conduct an extensive empirical study to validate Pocket's efficacy across four key dimensions: (i) achieving competitive or superior end-to-end latency compared to monolithic architectures; (ii) optimizing resource utilization to increase tenant density on both CPU and GPU platforms; (iii) identifying specific deployment scenarios where Pocket offers the most significant advantages; and (iv) ensuring that the system's operational overhead remains negligible within multi-tenant edge environments.

Table 3.2: Testbed Setup.

Items	Specification
<b>Processors</b>	Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz; 2 Processors; 24 cores; 48 threads
<b>GPU</b>	NVIDIA K80 (only for GPU experiments)
<b>Memory</b>	128GiB
<b>Operating System</b>	Ubuntu 18.04.3 LTS GNU/Linux 4.15.0-76-generic x86_64
<b>Software</b>	TensorFlow:2.1.0; GCC 7.4.0 python 3.6.9

### 3.4.1 Experimental Environment and Workload Characterization

Our evaluation was performed using the Chameleon Cloud platform [72], with hardware specifications detailed in Table 3.2. We utilized a benchmark suite consisting of six diverse ML applications, as listed in Table 3.3. These applications span object detection, image classification, and natural language processing (NLP), representing a broad range of model sizes and computational complexities.

Unless otherwise noted, all experiments use a batch size of one to reflect latency-sensitive edge serving rather than throughput-oriented offline inference. Workloads follow a closed-loop model, in which each client issues a new request only after receiving the previous response, ensuring that measurements capture steady-state per-request performance under concurrency rather than open-loop queueing effects.

For CPU-only evaluations, we compare Pocket against three baselines: monolithic Docker containers (*mono*), Firecracker microVMs (*FC*), and a *static* shared-backend configuration that employs lightweight IPC but lacks dynamic resource reallocation. For GPU evaluations, we focus on Pocket’s ability to facilitate accelerator sharing, a task typically requiring complex GPU virtualization [73]. Here, we compare Pocket against a *static* baseline and a *none* baseline, where resources are partitioned uniformly without dynamic adjustment. For NLP workloads, we use fixed-length inputs consisting of approximately

Table 3.3: Applications used in the evaluation. We include applications that represent different functionality and exhibit different resource requirements.

	Application	Serialized CPU Time	Memory Consumption	Size
Image Classification	MobileNetV2	7.267sec	287.24MB	“Little”
	ResNet50	12.361sec	528.59MB	“Large”
Object Detection	SSDMobileNetV2	23.185sec	946.54MB	“Little”
	SSDResNet50	72.905sec	2134.89MB	“Large”
Language Processing	SmallBERT	19.268sec	889.98MB	“Little”
	TalkingHeads	53.841sec	1331.22MB	“Large”

300-word movie review texts, and apply the same sequence length across all experiments to ensure comparability.

### 3.4.2 Temporal Efficiency and Scaling Results

**CPU Performance and Latency** We first examine end-to-end performance on CPU platforms. As shown in Figure 3.5, a naive shared-backend approach (*static*) significantly degrades performance due to insufficient resource allocation. In contrast, Pocket’s resource amplification mechanism dynamically boosts the backend’s capacity during IPC calls by adjusting `cfs_quota_us` values. This allows the backend to utilize more cores than a monolithic instance, resulting in visible acceleration for latency-sensitive models like ResNet50 and SmallBERT. For heavy-compute models like SSDResNet50, Pocket remains competitive until high tenant density leads to backend congestion, suggesting that additional backend instances may be required as the node reaches saturation.

**Density and Memory Efficiency** Pocket demonstrates superior resource efficiency, with memory consumption growing much more slowly than in monolithic deployments (Figure 3.5, middle). For example, supporting seven SSDResNet50 instances requires 9.1 GB in Docker but only 5.5 GB in Pocket, a 40% reduction. Consequently, Pocket increases

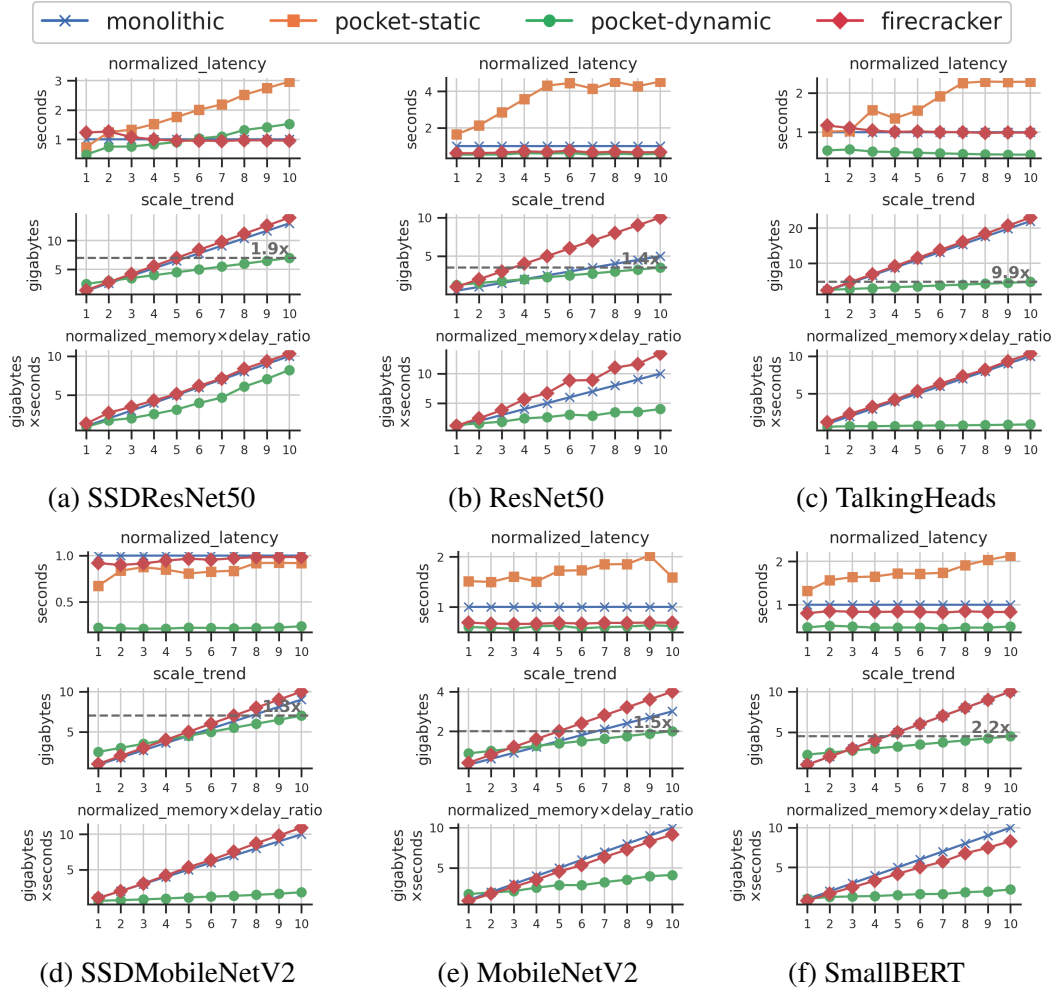


Figure 3.5: Performance and resource efficiency metrics for each application. Overall, with the suggested resource budget configurations, Pocket achieves better performance with on average  $5\times$  better resource efficiency, which points to its ability to serve more instances at once with the same amount of resources.

client density by 1.3–9.9 $\times$  compared to Docker and even more relative to Firecracker. To quantify this, we use the memory-delay product (similar to the energy-delay product); Pocket reduces this metric by 1.15 to 10 $\times$ , proving that it delivers higher throughput per unit of memory even when response times are slightly longer.

**GPU Performance Breakdown** On GPU-enabled nodes, Pocket provides up to a 3 $\times$  speedup over the *static* policy by resolving frontend bottlenecks. In models requiring significant image preprocessing (e.g., SSDResNet50), static allocation starves the frontend,

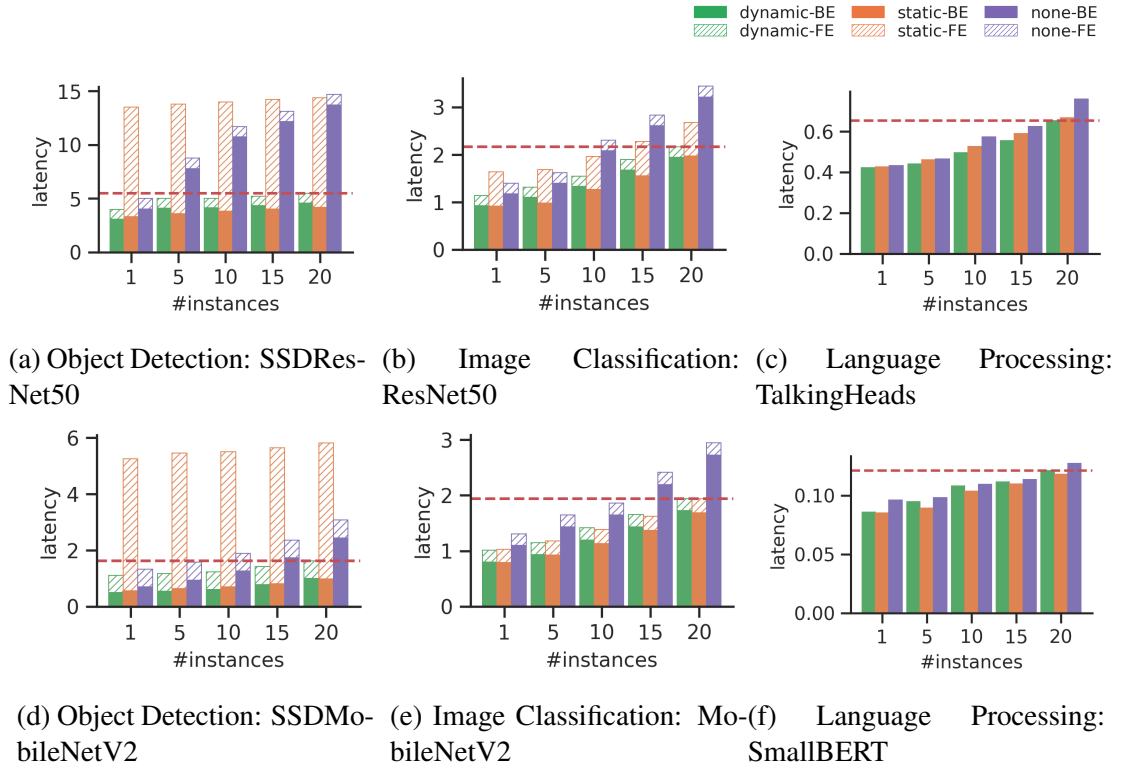


Figure 3.6: Execution time breakdown of GPU execution. Even for GPU execution, pocket-dynamic is superior to the other two policies, static(ally reassigning the resource) and none (no resource reassignment). When input preprocessing is the bottleneck for an application, pocket-dynamic has an advantage over static by allocating the resources required for input preprocessing to the frontend. The red dotted line represents the latency when running 20 instances as pocket-dynamic. This reveals how many instances each policy can run concurrently with comparable performance.

creating a massive end-to-end delay. Pocket’s dynamic reallocation ensures the frontend has the memory required for preprocessing while the backend possesses the power for inference. As shown in Figure 3.6, Pocket allows for a 1.16–20× increase in the number of concurrent instances that can be served while maintaining a target latency, compared to the *none* baseline.

### 3.4.3 Efficacy of System Mechanisms

**Benefits of Dynamic Amplification** Our experiments confirm that resource amplification is vital for balancing frontend and backend requirements. Figure 3.7 illustrates that

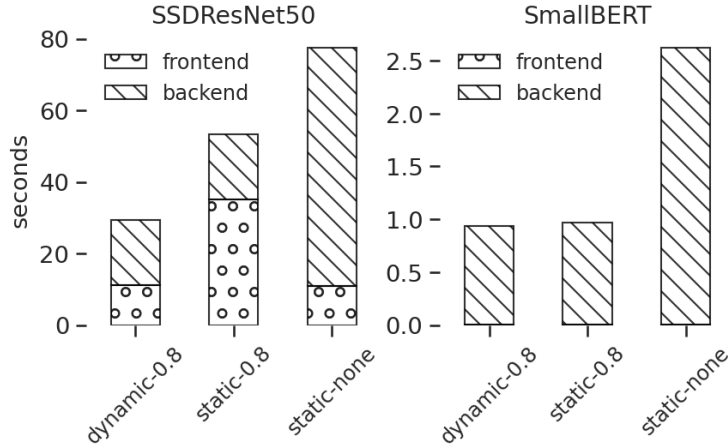


Figure 3.7: Pocket execution time breakdown. SSDResNet50 is an application that requires input preprocessing from frontend and SmallBERT language processing application requires little preprocessing. Allocating most resources statically to the backend may result in worse latency than dynamic allocation, depending on application.

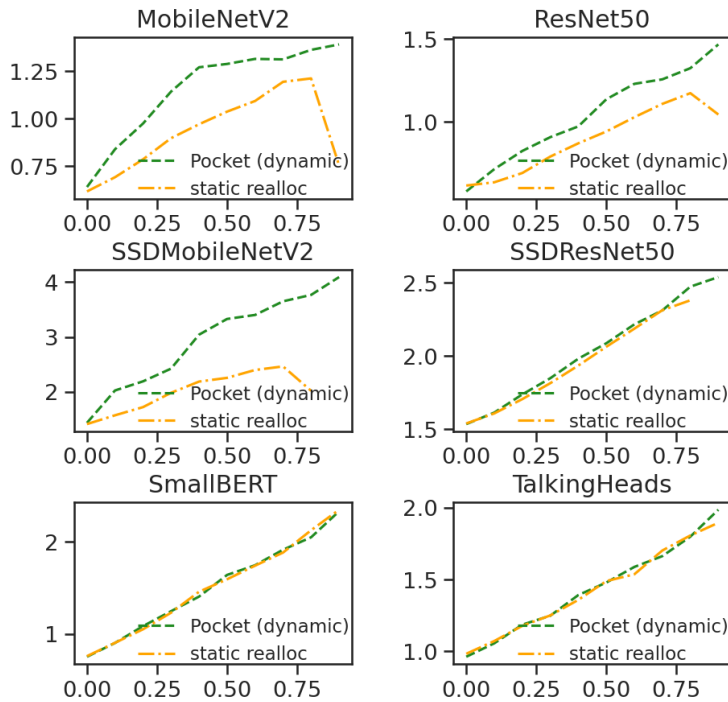


Figure 3.8: Speed up trend with varying resource transfer proportions. X axis represents a proportion to transfer to backend. Y axis is  $\frac{\text{latency}(\text{monolithic})}{\text{latency}(\text{static})}$  or  $\frac{\text{latency}(\text{monolithic})}{\text{latency}(\text{pocket})}$ , which represents the amount of speed up (higher the better). Larger resource transfer leads to more speed up for Pocket.

for SSDResNet50, Pocket matches the low frontend latency of unconstrained containers and the low backend latency of centralized services. Furthermore, Figure 3.8 shows that

increasing the resource transfer ratio directly correlates with performance gains, up to an 80% amplification factor, beyond which frontends may face memory exhaustion.

Table 3.4: Whether it is a backend with dynamic resource transfer or without resource transfer, no meaningful differences in latency have been observed.

	before-inference-operations		after-inference-operations		
(unit: sec)	(1) pocket-realloc	(2) static-realloc	(3) pocket-realloc	(4) static-realloc	$\frac{(3)}{(4)}$
null_ipc	1.360773	1.360144	1.383121	1.386921	0.997
mobilenetv2	0.014561	0.015054	0.016904	0.018561	0.911
resnet50	0.014467	0.014359	0.017145	0.01681	1.020
ssdmobilenetv2	0.058052	0.0577544	0.054542	0.053890	1.012
ssdresnet50	0.053913	0.055380	0.054690	0.051903	1.054

**Operational Overhead** Dynamic reallocation via cgroups adds only 840  $\mu$ s per transfer. Table 3.4 demonstrates that this mechanism introduces no measurable degradation in latency, as cgroups manage resource quotas without triggering expensive page-in/page-out operations.



Figure 3.9: To evaluate how Pocket works when serving multiple applications, we create 8 workload mixes. Each one of them is a subset of the 6 models that we use in the earlier experiments. The y axis shows the normalized value of Pocket compared to a monolithic Docker deployment for application latency and total memory usage (total\_mem). Lower is better. Overall, Pocket leads to shorter latency results and shows reduced memory usage, compared to the monolithic approach.

**Mixed Workload Performance** In multi-model scenarios (Figure 3.9), Pocket generally provides lower latency and footprint across eight distinct workload mixes. We did observe that "Little" models can experience queuing delays when mixed with "Large" models due to our current round-robin scheduling. This suggests that future iterations could further improve performance by implementing priority-aware scheduling policies.

### 3.5 Summary

This chapter presented Pocket, which is a resource-aware IPC system designed to enable high-density machine learning serving in resource-constrained edge environments. Traditional monolithic deployment models suffer from significant memory inefficiency because they redundantly load heavy frameworks for every application instance. To resolve this problem, Pocket introduces a split-architecture that decouples lightweight application logic into frontends and consolidates heavyweight runtimes into a shared backend service.

Pocket overcomes the performance and isolation challenges of this design through three primary concepts. First, it utilizes a shared-memory IPC mechanism to minimize data movement and reduce memory overhead compared to standard networking protocols. Second, the system ensures security in the shared environment through nested isolation and capability-based access control. Third, Pocket introduces resource amplification, which is a mechanism that carries resource expectations within messages to temporarily reallocate hardware limits during active service calls. This approach ensures that the shared service has sufficient capacity only when it is needed.

The evaluation demonstrates that Pocket significantly improves scalability by supporting many more concurrent instances on both CPU and GPU platforms compared to monolithic deployments. Furthermore, Pocket achieves an average of five times better resource efficiency while maintaining competitive end-to-end latency.

Although Pocket is evaluated under the stringent constraints of edge environments, the mechanisms proposed in this work, including resource-aware IPC and dynamic amplifi-

cation, target fundamental inefficiencies in multi-tenant machine learning serving. As a result, their applicability extends beyond the edge to a wide range of shared computing infrastructures.

### 3.5.1 Relationship to the Thesis Statement

The findings in this chapter provide the initial evidence for the central thesis that IPC must evolve from a passive data conduit into an active and intelligent runtime layer. Pocket establishes that when an IPC mechanism is aware of execution context and resource demands, it can dynamically manage hardware resources to bridge the gap between monolithic efficiency and microservice flexibility.

This resource-aware foundation is essential for the subsequent stages of this dissertation. While Pocket addresses resource management at the communication boundary, the following chapters expand this intelligent IPC framework to include hardware-aware data movement in Rocket and workload-driven adaptation in SkyRocket. Together, these systems demonstrate a comprehensive path toward efficient data movement in the era of hardware-accelerated and data-centric computing.

## **CHAPTER 4**

### **UNIFIED IPC RUNTIME FOR NOVEL DATA MOVERS ACROSS HARDWARE AND SOFTWARE STACK**

The transition toward modular and split-architecture systems, while enabling unprecedented flexibility for data-intensive services, has fundamentally altered the performance profile of modern computing. As specialized accelerators and decoupled runtimes become the norm, the system-level bottleneck has migrated from core computation to the efficiency of data exchange across process boundaries. In the era of multimodal AI and high-throughput data pipelines, this "boundary tax", defined by redundant memory copies and the resulting CPU saturation, increasingly dictates the actual end-to-end performance and energy efficiency of complex software stacks.

Conventional inter-process communication (IPC) mechanisms, however, treat data movement as a passive, CPU-bound library task, failing to leverage the emerging landscape of heterogeneous data movement engines and hardware-level memory services. While hardware-assisted offloading offers a promising path to reclaim CPU cycles, its naive integration into user-space IPC introduces complex trade-offs involving synchronization overhead, cache hierarchy interference, and execution parallelism. This chapter addresses these challenges by introducing a unified and hardware-aware transport layer, moving beyond simple offloading toward an orchestration-driven approach that reconciles the gap between raw hardware capabilities and application-level communication requirements.

#### **4.1 Introduction: The Escalating Cost of Data Movement in Modern IPC**

Building upon the resource-aware communication strategies established in the previous chapter, this part of the research addresses a critical shift in the performance landscape of modern systems. While computational acceleration has seen massive progress, contem-

Table 4.1: Metrics characterizing data transfer and memory behavior for representative multimodal workloads in application-pipeline IPC scenarios.

Aspect	MobileNetV2	XGBoost	PageRank	MilvusDB
<b>Bytes (req/resp)</b>	120MB/ 800KB	25MB/ 800KB	76MB/ 320KB	1MB/ 320MB
<b>Memcpy time in IPC (ms)</b>	203	25	46	323
<b>Memory behavior</b>	Large input, low reuse	Low reuse, compute-bound	High reuse, memory-bound	Bulky response
<b>Config</b>	Batch size 200 [77]	200K rows, 30 features	Graph w/ 10M edges [78]	Batched search

porary AI and data analytics pipelines are increasingly bottlenecked by inter-process data movement rather than computation. Emerging multimodal workloads such as encompassing high-resolution video, image-text pairs, and large-scale tabular data, routinely exchange hundreds of megabytes per request [74, 75, 76]. For instance, batching 256 RGB images for an offline inference task involves moving over 100 MB, a volume that scales even further in visual analytics pipelines where a single 4K image exceeds 30MB [23, 19, 20, 21, 22].

#### 4.1.1 The Data Movement Bottleneck and the "Boundary Tax"

Table 4.1 characterizes this trend across several representative multimodal workloads. As inter-process transfers grow to the range of 50–500 MB, the time spent on memory copies becomes the dominant factor in both latency and energy consumption. This phenomenon effectively imposes a "boundary tax" on modular architectures, where the cost of moving data across process boundaries outweighs the benefits of specialized execution units.

In iterative AI training and inference cycles, data exchange often saturates memory bandwidth and CPU cycles before the core processing logic can even begin [79]. This inefficiency persists even in intra-node pipelines that move data between caches [80] and inference backends [5, 81]. In serverless workflows, data movement within a node can account for up to 92% of end-to-end latency [82], while scientific pipelines scale these

transfers up to 1GB chunks [83, 84]. Furthermore, traditional datacenter workloads already spend over 5% of total CPU cycles on `memcpy` operations, leading to significant cache pollution and energy overhead [34, 85].

#### 4.1.2 Limitations of CPU-Driven IPC Stacks

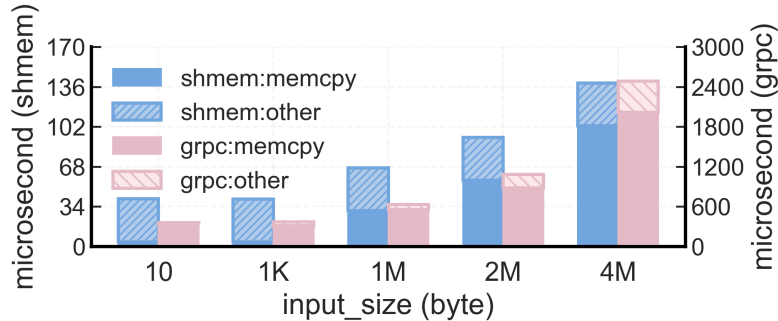


Figure 4.1: Breakdown of end-to-end latency for intra-node echo RPCs implemented using shared memory (shmem) and gRPC. The figure quantifies the portion of total latency attributed to `memcpy` as a function of message size.

Figure 4.1 demonstrates that `memcpy` time dominates both optimized shared-memory stacks (e.g., Nightcore [67]) and general-purpose frameworks like gRPC. As transfer sizes increase, the conventional reliance on CPU-driven copies becomes unsustainable, saturating both core execution resources and cache hierarchies. This growing gap between compute and communication efficiency necessitates a new class of runtimes that treat data movement as a first-class, offloadable system service.

While hardware engines like Intel’s DSA [86] and software-based offload mechanisms at the OS level [24] offer potential relief, existing IPC stacks lack a cohesive model to coordinate these heterogeneous backends effectively. Most prior work has treated memory offloading in isolation, focusing on intra-process tasks or inter-node network acceleration [87, 28, 88]. Integrating such engines into an inter-process, intra-node setting introduces complex trade-offs involving cache interference and synchronization overheads that remain largely underexplored in modern client-server interactions [15].

### 4.1.3 Rocket: A Unified Runtime for Heterogeneous Data Movers

Naïve integration of data-movement accelerators often results in broken cache locality or amplified page fault overhead, which can negate the performance gains of offloading. Addressing these challenges requires an orchestration layer that is aware of the underlying hardware and software data movers.

Building on the architectural insights of this dissertation, this chapter introduces Rocket, a unified IPC runtime suite designed to bridge the gap between diverse data movement engines and shared-memory communication. Rocket moves beyond the view of accelerators as standalone devices. Instead, it serves as a comprehensive orchestration layer that integrates multiple backends including hardware accelerators, kernel-resident services, and memory-controller-assisted mechanisms into a seamless communication framework. By managing synchronization, cache visibility, and concurrency across these heterogeneous engines, Rocket transforms data movement from a peripheral task into a managed system capability.

The contributions of this research phase are as follows:

**(1) Systematic Bottleneck Analysis.** We identify the critical system-level factors affecting both hardware and software-based memory offloading in user-space IPC. This analysis reveals the fundamental trade-offs between CPU conservation, cache behavior, and synchronization costs that determine the effectiveness of a unified runtime approach. (§ 4.4)

**(2) Unified Protocol Design and Implementation.** We design and implement Rocket, an IPC suite that abstracts diverse data movement backends into a single shared-memory pipeline. Rocket optimizes end-to-end efficiency through asynchronous pipelining, adaptive parallelism, and targeted cache injection, while leveraging advanced architectural features such as power-saving coordination instructions on x86. (§ 4.5)

**(3) Empirical Evaluation of Multi-Backend IPC.** We evaluate Rocket on real-world workloads using systems equipped with Intel DSA, demonstrating up to a 22% reduction in instruction counts and a 15% improvement in throughput compared to CPU-only baselines.

Our findings show that a coordinated, multi-backend IPC runtime provides benefits that extend far beyond simple memory copy speedups. (§ 4.7)

Ultimately, this work provides a practical IPC architecture that lays the groundwork for future memory-accelerated systems, ensuring that communication efficiency keeps pace with the demands of data-intensive, modular software environments.

## 4.2 Background: Evolution of Decoupled Data Movement

Building upon the necessity for resource-aware communication discussed in previous chapters, modern system architectures increasingly strive to decouple data movement from core CPU execution. Traditional `memcpy`-based transfers, while ubiquitous, have become a significant source of inefficiency in bandwidth-heavy workloads. This is primarily because CPU-driven copies not only consume valuable execution cycles but also frequently pollute cache hierarchies with transient data.

To address these bottlenecks, a paradigm shift is underway where memory offloading is treated as a programmable, first-class system service [89, 24, 90]. This shift allows bulk data transfers to proceed outside the CPU’s critical execution path. Recent developments, ranging from kernel-managed asynchronous engines to integrated hardware accelerators like Intel’s DSA, reflect this trend. This background section contextualizes **Rocket** within this evolving landscape, motivating the need for a unified runtime that bridges the gap between raw offloading capabilities and high-level IPC requirements.

### 4.2.1 The Multi-Layered Landscape of Memory Offloading

In recent years, memory offloading has transitioned from a niche hardware feature to a prominent design principle [27, 89]. Historically treated as simple, synchronous library calls, memory operations are now recognized as primary scalability barriers in data-intensive systems due to cache interference and synchronization delays. Modern approaches to mitigating these issues span multiple layers of the system stack.

At the hardware layer, emerging research proposes performing copies directly at the memory controller or within cross-accelerator DMA engines [25, 90]. These mechanisms reduce CPU stalls and allow data movement to overlap with computation. At the system and kernel level, frameworks such as Copier [24] reinterpret copy operations as coordinated services delegated to specialized background threads. These systems track data dependencies to overlap copy-and-use phases, aiming to transform data movement into a schedulable operation.

Table 4.2: Representative memory-offload mechanisms across system layers.

	<b>(MC)<sup>2</sup> [25]</b>	<b>DMX [90]</b>	<b>Copier [24]</b>
<b>Level</b>	Mem. controller	Cross-accelerator	OS kernel
<b>Cache pollution</b>	Lazy flush	Fence DMA	Not handled
<b>Applicability</b>	Hardware-tied	Specialized setup	OS-integrated
<b>Overhead</b>	Very low	Moderate	Moderate-high

However, as shown in Table 4.2, a significant gap persists in the applicability of existing memory-offload mechanisms to user-space IPC. While current efforts focus on the mechanisms of *enabling* offload at specific system layers such as memory controllers, accelerators, or the OS kernel, they remain either tightly bound to specialized hardware or exposed only through low-level interfaces, limiting their direct usability in application-level dataflows. This resulting disconnect between hardware/kernel potential and end-to-end system exploitation defines the design space for Rocket. Our objective is to provide the missing architectural layer: a unified runtime that orchestrates these heterogeneous offloading backends specifically for user-space IPC.

### 4.3 Case Study with Intel DSA

Modern memory offloading engines are becoming core components in high-performance systems. However, their real value depends on how effectively software can use them in practice. To make this discussion concrete, we focus on Intel’s DSA as a representative

case study. This section uses DSA to highlight two aspects. First, it shows the capabilities that modern offloading hardware can provide. Second, it exposes the software integration challenges that limit their practical impact. Together, these observations motivate the need for a unified runtime abstraction.

#### 4.3.1 DSA Primer

Intel’s DSA, incorporated into Sapphire Rapids processors, exemplifies contemporary memory offloading engines. DSA augments conventional DMA by providing user-level programmability, support for virtual memory, and fine-grained cache control. Together, these capabilities are designed to mitigate cache pollution, release CPU resources, facilitate overlap between computation and memory operations, and enhance effective bandwidth utilization.

**Native Virtual Memory and PASID.** Unlike legacy DMA engine [27] that required complex physical address mapping, DSA supports virtual memory, which drastically simplifies its integration into user-space applications [89]. By leveraging Process Address Space IDs (PASID), DSA can manage multiple address spaces without requiring explicit memory pinning. This capability is essential for the multitenant and virtualized environments where modern IPC runtimes frequently operate.

**High-Efficiency User-Space Submission.** A critical advantage of DSA is its low-latency programming model. Rather than relying on expensive system calls (e.g., `ioctl`), DSA allows applications to submit work descriptors directly from user space using specialized instructions like `ENQCMD`. This reduces the overhead of initiating a transfer from tens of microseconds [91] to approximately 200ns [89]. Furthermore, the atomic nature of `ENQCMD` facilitates lock-less work submission in highly concurrent multithreaded environments [28].

### 4.3.2 Current Challenges in Software Integration

Despite these hardware advancements, existing software stacks for memory offloading—such as low-level drivers or transparent interception libraries present a difficult trade-off between control and ease of use.

**The Programmability-Control Gap.** Low-level interfaces provide fine-grained control over accelerator operation. This control allows applications to explicitly manage descriptor preparation, work submission, and completion handling. However, it also introduces significant programmability overhead. Developers must manage descriptor lifecycles manually. They must also enforce correct dependency ordering across submitted work. Completion must be monitored explicitly, most commonly through active polling or instructions such as `UMWAIT`.

While the CPU is technically freed during the data movement itself, the surrounding control logic remains on the critical path. The application must coordinate progress between the CPU and the accelerator. To avoid idle stalls, it must restructure execution into asynchronous dataflow stages (Figure 4.2). This restructuring increases code complexity and raises the risk of subtle synchronization errors. It also reduces portability across execution environments. In practice, the effort required to manage this asynchrony often offsets the benefits of fine-grained control. This mismatch sustains a persistent gap between hardware capability and practical programmability.

**Limitations of Static Offloading.** Static offloading frameworks simplify accelerator adoption, but they inherently limit performance and generality. Systems such as Intel’s DSA Transparent Offload (DTO) [92] achieve ease of use by transparently intercepting standard `memcpy()` calls. This design removes the need for explicit accelerator management. However, it also strips the system of critical execution context.

As a result, DTO makes offloading decisions without awareness of transfer size, call-site semantics, or surrounding computation. It frequently offloads small or latency-sensitive copies. In these cases, descriptor setup and submission costs exceed any potential benefit.

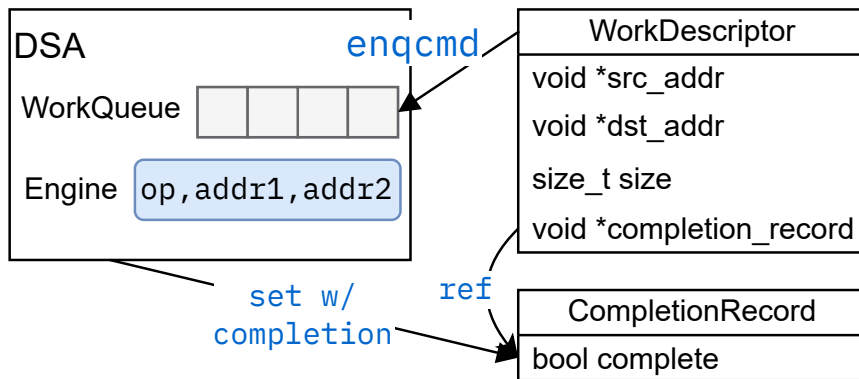


Figure 4.2: DSA programming model. The CPU prepares the task descriptor and submits it to DSA. DSA executes the task and sets the completion flag. The CPU then checks the completion flag to determine if the task is complete.

This inefficiency is not an implementation artifact. It follows directly from the static nature of interception-based offloading.

More critically, DTO enforces a synchronous execution model. Each offloaded copy blocks the calling thread until completion. This design prevents meaningful overlap between data movement and useful CPU computation. It also serializes control and data paths that could otherwise proceed independently. Together, these limitations prevent static offloading frameworks from serving as general-purpose IPC mechanisms.

These limitations point to a structural gap that existing approaches cannot resolve. A Unified IPC Runtime is therefore necessary. Such a runtime must coordinate IPC execution holistically rather than delegating decisions to isolated mechanisms.

Rocket is designed to address this need. It introduces an orchestration layer that reasons about IPC execution at runtime. This layer understands the performance trade-offs of different backends. It considers factors such as transfer size, synchronization cost, and overlap potential. Based on this context, Rocket dynamically selects the execution path that best matches the communication pattern. By doing so, it avoids the rigidity of static offloading and the burden of low-level control, enabling IPC to adapt to application behavior rather than constrain it.

## 4.4 Motivation: Integration Challenges in Offloaded IPC

Incorporating memory offloading into inter-process communication introduces subtle trade-offs that can negate its benefits when applied naively [34, 28]. Offload engines relieve the CPU of data movement but shift synchronization, visibility, and caching responsibilities to the software layer [89]. This interplay exposes three recurring tensions: (i) synchronization granularity between submission and completion, (ii) address visibility and page-fault handling, and (iii) cache injection and data reuse control [87, 24]. Understanding these tensions is essential for integrating offload mechanisms into runtime systems effectively and motivates the design principles for Rocket discussed in § 4.5. In this section, we illustrate and quantify these trade-offs using Intel DSA as a primary case study.

### 4.4.1 Hardware-level Trade-offs

Offloading memory operations frees CPU cycles and reduces cache pollution, potentially improving overall system performance. For example, offloading a 1MB transfer saves approximately  $33\mu s$  with DSA, which translates to roughly 130,000 CPU cycles that can be repurposed for application-level tasks. However, these benefits are highly context-dependent and may incur significant system-level overhead if not carefully managed [93, 91].

In IPC workloads, integration of a unified runtime presents several hardware-level trade-offs. Key factors include synchronization overhead from busy-waiting, loss of CPU cache locality due to cache bypass, cache pollution from injected data, latency spikes from page faults, and potential bus contention [94, 95]. The following sections examine each factor and its impact on overall system behavior.

**Overheads from Completion Check.** Regardless of whether a backend is used synchronously or asynchronously, offload completion must be detected by reading a completion flag, often located in an uncacheable memory-mapped I/O region. Figure 4.3 com-

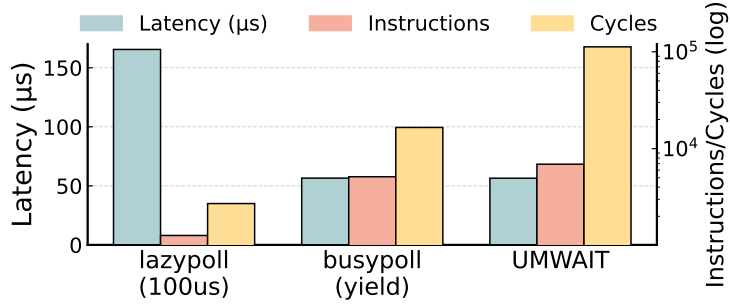


Figure 4.3: Comparison of polling strategies on latency and CPU usage (1MB transfer. `lazypoll`: polling every  $100\mu\text{s}$ ; `busypoll`: polling with `yield` but no sleep; `UMWAIT`: polling with usermode interrupt.)

compares three common polling strategies. While busy-waiting provides the lowest latency, it consumes excessive CPU cycles. Conversely, lazy-waiting proves inefficient for latency-sensitive IPC. The `UMWAIT` instruction offers latency comparable to busy-waiting without true asynchronous behavior—it places the CPU in a shallow wait state, effectively polling at  $25\mu\text{s}$  intervals [93]. Although `UMWAIT` may appear to execute a large number of instructions and consume substantial CPU cycles, this behavior reflects its use of low-power execution states rather than active computation. In single-threaded settings without competing work, `UMWAIT` primarily reduces power consumption by lowering core frequency and gating execution resources, but does not improve responsiveness or reduce polling overhead compared to busy-waiting.

Furthermore, polling introduces nontrivial system-level costs. In many accelerator-driven designs, including our DSA-based configuration, completion flags are placed in uncached or weakly cacheable memory regions to avoid cache pollution and excessive coherence traffic under frequent polling [89, 92]. Each read to an uncacheable flag bypasses the CPU cache and traverses the memory bus, increasing contention [94]. Accesses to memory-mapped regions enforce strict ordering, hindering out-of-order execution and introducing pipeline stalls [95]. These effects highlight the need for the low-overhead, responsive synchronization mechanisms implemented in Rocket.

**Impact of Page Faults.** While modern engines support virtual memory, page faults still

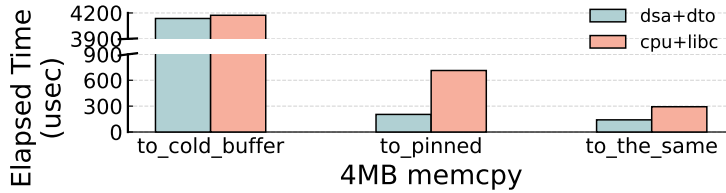


Figure 4.4: Performance comparison of DSA and CPU `memcpy` under different memory conditions. Copying to a pinned buffer reduces latency by 95%, and reusing the same buffer achieves a 97% reduction, both relative to cold-buffer access.

introduce significant latency. As shown in Figure 4.4, when faults occur, hardware offloading provides no clear advantage over CPU-based `memcpy`. In contrast, with pinned memory, the accelerator significantly outperforms the CPU [88]. These results emphasize the importance of utilizing pre-mapped or pinned memory regions, a core design choice in Rocket’s shared-memory management.

#### 4.4.2 Cache Interference and Path Divergence: CPU vs. Accelerator

Memory offloading alters fundamental assumptions about temporal locality. Unlike CPU operations that benefit from automatic cache retention, offload engines typically bypass caches, potentially causing cold-cache effects upon data reuse [34]. While this reduces cache pollution for ephemeral data, it increases latency for near-term accesses. To mitigate this, Rocket leverages explicit cache injection to route selected data into the LLC during transfer. As shown in Figure 4.5, cache injection improves LLC hit rates in single-threaded workloads but may degrade performance under heavy multi-threaded contention [93].

**Execution Implications by Access Direction.** We summarize how memory access patterns affect offloading decisions:

- **Read-In (DRAM → Cache):** CPU loads reused data into the cache, leveraging locality. Memory accelerators may bypass the cache, risking cold-start penalties (Figure 4.6a).
- **Write-Out (Cache → DRAM):** Memory accelerators may avoid polluting the cache

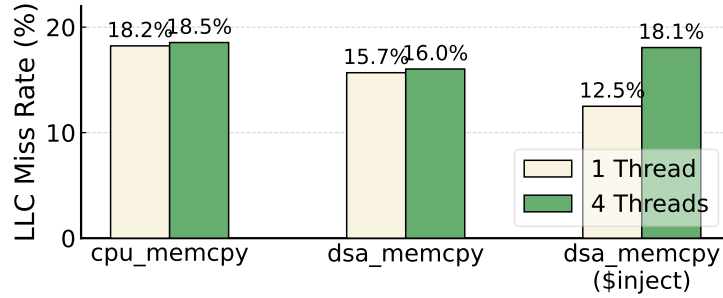


Figure 4.5: LLC miss rates under single and four-threaded execution, comparing `cpu_memcpy`, `dsa_memcpy`, `dsa_memcpy ($inject)` (Microbenchmark: summation over all elements in the destination buffer after memory copy).

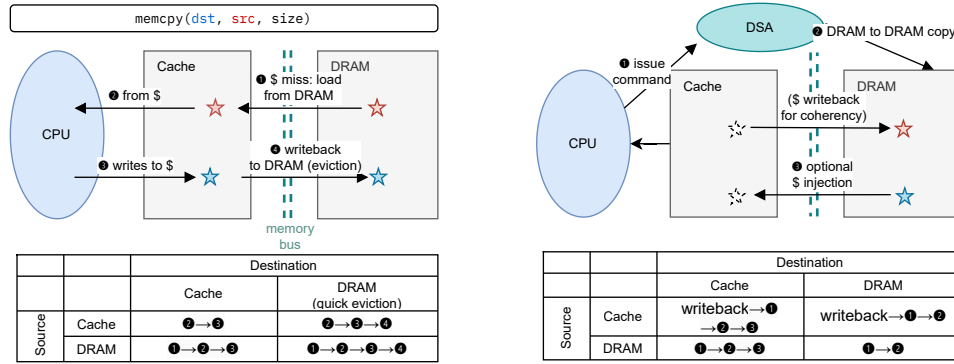


Figure 4.6: Comparison of memory copy execution paths between CPU and DSA. CPU-based memcpy naturally integrates with the cache hierarchy, while DSA-based memcpy bypasses the cache, accessing DRAM directly.

with write-out data, which is beneficial for ephemeral or one-way transfers. However, reuse-sensitive data may suffer from bypass-induced misses (Figure 4.6b).

#### 4.4.3 Summary of the Tradeoffs

In summary, our analysis shows that hardware offloading alone does not guarantee improved IPC performance. The benefit of hardware engines depends strongly on system-level factors, including synchronization overhead, cache state, and virtual memory management costs. If these factors are not carefully managed, they can outweigh the gains from higher raw throughput. Table 4.3 condenses these findings into concrete design principles

Table 4.3: Trade-offs of DSA offloading and implications for system design. Each factor highlights a key limitation and how Rocket addresses it through configurable or default design decisions.

Factor	Observed Trade-off	Microbenchmark Insight	Design Implications
<b>Data Size</b>	Offloading not always beneficial for small transfers	Offloading 1MB saves 33 $\mu$ s; breakeven 4KB raw, higher by setup [28]	Use CPU-based memcopy for small transfers; apply size-based threshold
<b>Page Faults</b>	DSA supports virtual memory but performance drops with page faults	page faults eliminate DSA speedup; pinned memory yields best performance (Figure 4.4)	Reuse shared memory to avoid PFs; enforce pre-mapping or pinning
<b>Cache Injection</b>	May improve or harm performance depending on reuse timing	Boosts hit rate in single-threaded case; degrades multi-threaded performance due to pollution (Figure 4.5)	Enable cache injection only under low contention. (e.g. single-threaded sync/async modes)
<b>Completion Check</b>	Frequent polling causes bus contention and stalls	UMWAIT reduces active polling cost but limited to 25 $\mu$ s (Figure 4.3, [93])	Use hybrid polling (UMWAIT + timeout); defer checks in pipelined mode
<b>Parallelism</b>	Untapped unless explicitly orchestrated in software	No parallel execution in idxd; DTO blocks until completion	Enforce structured async/pipelined execution to leverage concurrency
<b>Software Support</b>	Existing tools lack intelligent decision logic	idxd offers low-level control; DTO disables parallelism	Provide high-level API with tunable execution modes and cache options

and emphasizes the need for a dynamic orchestration layer. By replacing static offloading with hardware-aware decision logic, Rocket overcomes the core limitations of existing approaches. These principles define the key requirements of the Rocket architecture in § 4.5 and ensure robustness across the diverse bottlenecks of modern data-intensive pipelines.

#### 4.5 Design of the Rocket Unified IPC Runtime

To address the integration hurdles and performance tensions identified in Section 4.4, this research phase introduces Rocket: a comprehensive runtime suite that orchestrates both hardware-assisted and software-defined memory offloading for high-performance intra-node communication. Unlike conventional IPC stacks that treat data movement as an incidental CPU task, Rocket establishes a unified execution model over shared memory. It

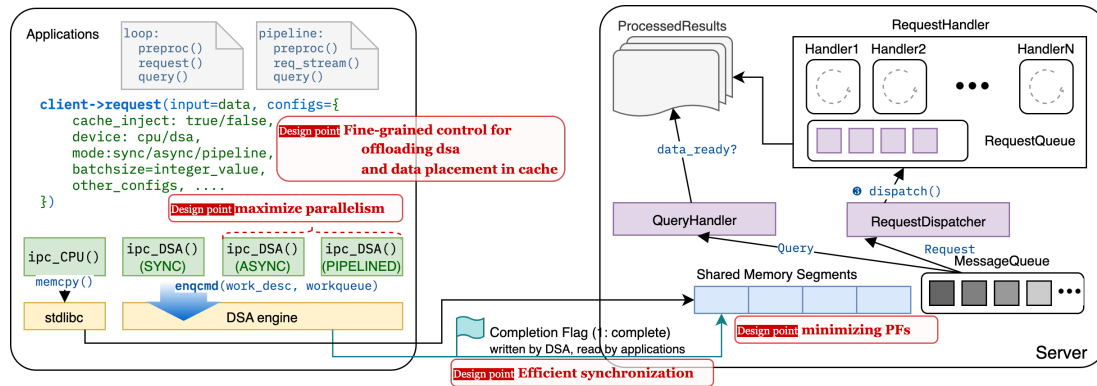


Figure 4.7: Overview of Rocket architecture. Its components reflect key design principles such as page fault avoidance, maximizing parallelism, configurable selective offloading with or without cache injection enabled, and efficient synchronization.

moves away from static offloading policies in favor of a configurable framework where execution, synchronization, and cache-visibility behaviors are dynamically aligned with the underlying hardware capabilities and workload characteristics [34, 28].

#### 4.5.1 Architectural Overview and Core Principles

The architecture of Rocket, as illustrated in Figure 4.7, is designed as a modular orchestration layer capable of managing multi-client connections while selectively delegating data movement to heterogeneous backends. The runtime environment—comprising a centralized message queue, a request dispatcher, and specialized query handlers—facilitates efficient CPU-accelerator overlap through asynchronous batching and deterministic coordination.

The design of Rocket is governed by five hardware-aware principles derived from the systemic bottlenecks of modern memory subsystems:

**(1) Persistent Shared Memory Management.** To mitigate the “boundary tax” associated with memory remapping and page-fault latency, Rocket reuses long-lived shared-memory segments that remain mapped for the duration of a communication session, across the entire communication lifecycle. By maintaining buffer continuity through a pre-allocated memory pool and persistent message queues, the runtime ensures that data movement backends

interact with resident pages, thereby maximizing the efficiency of hardware engines like Intel DSA [88, 89].

**(2) Adaptive Backend Orchestration.** Recognizing that offloading is not universally beneficial, Rocket provides an abstraction layer for selective offloading. Unlike rigid frameworks such as Intel DTO, our runtime enables fine-grained, size-aware thresholds to determine whether a transfer should be handled by the CPU or a dedicated hardware engine, balancing setup overhead against raw throughput gains [34].

**(3) Native Support for Execution Overlap.** Rocket natively integrates the synchronization primitives necessary for true asynchronous transfers. By abstracting the complexities of work-descriptor management typically required in low-level APIs, the runtime enables concurrent CPU processing while data movement is in-flight, which is a critical capability for maximizing system-wide parallelism [24, 84].

**(4) Locality-Aware Cache Control.** The runtime exposes cache injection as a tunable software knob. Within multi-client environments, the server maintains the necessary execution context to allow clients to enable injection selectively, ensuring that data is placed in the LLC only when high temporal reuse is predicted and cache contention is low [93, 80].

**(5) Deterministic Synchronization and Polling.** To minimize the bus contention caused by frequent flag-polling, Rocket implements a time-prediction-based synchronization model. By deferring completion checks based on deterministic latency estimates and utilizing power-efficient instructions like `UMWAIT`, the system achieves low-latency coordination without the overhead of continuous busy-waiting [94, 95].

#### 4.5.2 Unified IPC API and Execution Modes

The Rocket API is architected to provide a flexible interface that supports diverse data-intensive workloads while ensuring performance portability. This interface decouples the functional request from the underlying execution strategy, allowing developers to optimize for throughput or latency as needed.

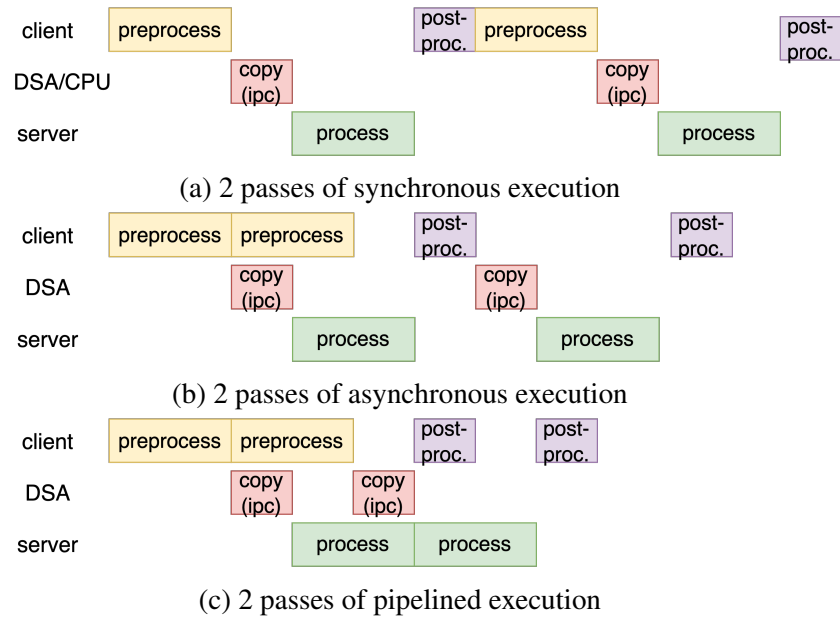


Figure 4.8: Execution mode structure in Rocket. Each mode differs in synchronization and overlap strategy.

**Execution Paradigms.** Rocket supports three distinct execution modes (`sync`, `async`, and `pipelined` (Figure 4.8)), and each optimized for specific points in the communication trade-off space:

- **Synchronous Mode (`sync`):** Operates in a blocking manner where the issuing thread waits for completion. This mode is optimized for latency-critical, sequential tasks where immediate data visibility is paramount. It typically utilizes cache injection to ensure data is "warm" for immediate CPU processing [89].
- **Asynchronous Mode (`async`):** Decouples request submission from completion tracking. This mode returns control to the application immediately, allowing the CPU to execute independent logic while the data transfer is processed by the backend. This is the preferred mode for hiding the "boundary tax" in parallel pipelines [24].
- **Pipelined Mode (`pipelined`):** Processes memory operations in batched stages to maximize aggregate throughput. By deferring individual completion checks and reusing buffers across multiple stages, this mode amortizes the coordination over-

head, making it ideal for massive data transfers where throughput is the primary constraint.

**Configurable Parameters.** To support workload-specific tuning, the API exposes parameters for **Offload Control** (CPU vs. Hardware), **Cache Injection Policies**, and **Execution Paradigm Selection**. These controls allow Rocket to adapt its behavior to the specific reuse patterns and contention levels of the host system.

### 4.5.3 Runtime Internals and Implementation

**Shared Memory Region Reuse.** Rocket relies on persistent shared memory regions to reduce page fault overhead. During connection initialization, the server allocates a fixed-size memory pool and assigns each client a dedicated queue pair. Each pair consists of transmit (client-to-server) and receive (server-to-client) buffers that are mapped once and reused for the lifetime of the session. This approach removes repeated remapping costs and provides stable, low-latency memory access, which is critical for efficient DSA transfers. While inspired by RDMA queue pairs, the design is adapted to the copy-based semantics of DSA.

**Asynchronous DSA Engine.** To expose parallelism and mask memory latency, Rocket incorporates a lightweight asynchronous engine for DSA command management. The engine encapsulates low-level DSA operations and presents a simple interface to the IPC driver. Requests are directed to mode-specific execution paths, where the engine manages command submission, completion tracking, and batching in `pipelined` mode. A hybrid polling scheme achieves low CPU overhead while maintaining timely completion detection.

**Backend Integration and DMA Semantics.** Rocket manages a lightweight asynchronous engine that abstracts low-level hardware-specific primitives into a clean IPC interface. Drawing inspiration from RDMA queue-pair semantics, the runtime assigns each client a dedicated transmit/receive pair to eliminate remapping costs and provide stable, low-

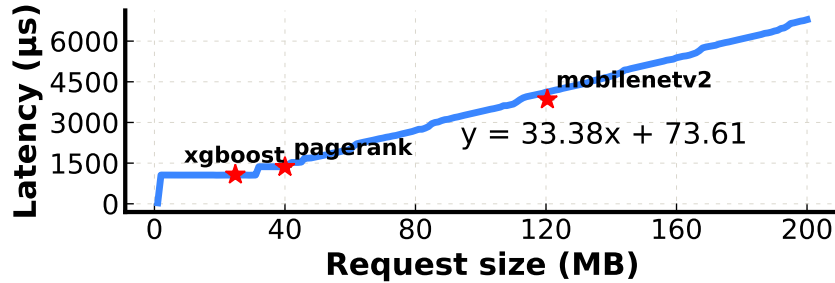


Figure 4.9: The latency of `memcpy` increases linearly with the target buffer size, at approximately  $33.4 \mu\text{s}$  per 1MB on our hardware.

latency access paths for the data movement engines.

**Hybrid Polling Strategy and Latency Prediction.** A core innovation within Rocket is its hybrid polling strategy designed to balance responsiveness with CPU efficiency. Since instruction-level waits like `UMWAIT` are limited to approximately  $25 \mu\text{s}$  [93], they are insufficient for the large transfers common in modern AI pipelines (Figure 4.9).

Rocket addresses this by employing a size-aware deferral mechanism. Let  $L$  be the predicted completion latency:  $L = L_{\text{fixed}} + \alpha \cdot \text{size\_in\_MB}$ . In our reference implementation, these constants ( $L_{\text{fixed}} = 73.6 \mu\text{s}$ ,  $\alpha = 33.4 \mu\text{s}/\text{MB}$ ) are calibrated via an automated profiling script during deployment to account for machine-specific bus speeds. The runtime sleeps for  $0.95 \cdot L$  to yield the CPU before initiating passive `UMWAIT`-based polling, thereby significantly reducing unnecessary memory bus traffic [94, 95]. The profiling methodology and sensitivity of these parameters are analyzed in more detail in our prior work [96].

**Orchestration Stack Internals.** The execution stack separates submission, handler logic, and query management. The *RequestDispatcher* routes incoming IPC messages to thread-local *RequestHandlers*. In `pipelined` mode, these handlers batch commands to minimize the number of work-submission instructions (e.g., `ENQCMD`). Simultaneously, the *QueryHandler* manages the completion lifecycle, allowing for decoupled, high-concurrency result collection across multiple outstanding jobs.

**Request Batching Support.** To sustain high throughput, particularly in `pipeline` mode, Rocket supports application-level batching. Incoming messages are categorized as execu-

tion requests or result queries. Requests are forwarded to workload-specific handlers, such as MobileNetV2 or graph processing, which are registered through a unified interface. Handlers execute asynchronously and write outputs directly to shared memory. By separating request submission from result retrieval, Rocket enables deferred collection and batch execution. This design improves buffer reuse, lowers synchronization costs, and allows multiple outstanding requests to be processed together with minimal coordination.

## 4.6 Implementation Details

The Rocket runtime is implemented in C++17, totaling approximately 12,000 lines of code. It interfaces with the Intel IDX driver to provide high-level abstractions over hardware-specific primitives while leveraging advanced ISA features such as UMONITOR, UMWAIT, and atomic work-submission (ENQCMD).

The implementation integrates seamlessly with standard data-intensive libraries, including ONNX for AI inference [97], XGBoost for analytics [98], and MilvusDB for vector search [99]. This ensures that Rocket can serve as a drop-in replacement for traditional IPC mechanisms in modern, modular software stacks.

Listing 4.1: Rocket API for Multi-Mode Execution

```
// Synchronous: Traditional blocking request
client->request(mode="sync", op="mobilenetv2", data);

// Asynchronous: Non-blocking future-based request
auto future = client->request(mode="async", op="mobilenetv2", data);
// Perform concurrent computation here...
future.get();

// Pipelined: Batched throughput-oriented request
for (int i = 0; i < batch_size; i++) {
    job_ids[i] = client->request(mode="pipeline", op="mobilenetv2", data[i]);
}
```

```

}
// Collect batched results after processing
for (int i = 0; i < batch_size; i++) {
    results[i] = client->query(job_ids[i]);
}

```

**Deployment and Configuration Effort.** Rocket exposes three primary configuration axes: the device (CPU or Hardware), `cache_injection` (On/Off), and the mode (`sync`, `async`, `pipeline`). While **Rocket** provides heuristic defaults, such as enabling cache injection for synchronous transfers and disabling it for pipelined batches, it allows for explicit user overrides. This allows the runtime to navigate the complex trade-offs between transfer size, CPU-accelerator bus contention, and temporal reuse patterns that define the efficiency of modern inter-process data movement.

## 4.7 Evaluation of the Rocket Unified Runtime

In this section, we experimentally demonstrate that Rocket significantly improves end-to-end throughput and latency across a suite of representative application benchmarks. Our results validate that the hardware-aware design choices within Rocket effectively mitigate the systemic inefficiencies associated with inter-process data movement.

### 4.7.1 Experimental Methodology

**Testbed Configuration.** The experimental platform is summarized in Table 4.4. We utilize a high-performance system equipped with an Intel Sapphire Rapids processor (Xeon Gold 6438Y+), featuring 32 physical cores and an integrated Intel DSA device. The system is further augmented with an NVIDIA A100 GPU to evaluate the impact of Rocket in environments where compute acceleration shifts the bottleneck toward the communication layer.

**Workload Selection.** Our evaluation encompasses five application benchmarks that cap-

Table 4.4: Experimental Setup

Component	Specification
CPU	Intel(R) Xeon(R) Gold 6438Y+ 4.0GHz (Sapphire Rapids), 32 cores
GPU	NVIDIA A100 (PCIe, 40GB HBM2 memory, 6912 CUDA cores, 312 TFLOPS FP16)
Cache	60MiB LLC
DSA	1 Intel DSA device with 1 workqueue
RAM	704GB DDR4 4800MT/s
OS	Ubuntu 22.04.5 LTS, Kernel 6.5.0-41
Compiler	GCC 11.4.0
Libraries	glibc 2.35, accel-config, numactl, DTO, PyTorch 2.1.2, ONNX (1.19.1), OpenCV, XGBoost, BoostGL, OpenVINO

ture diverse dimensions of data-movement stress in data-intensive systems: MobileNetV2 [38] (dense tensor streaming), XGBoost [98] (fine-grained feature batching), PageRank [100] (irregular, reuse-heavy graph traversal), MilvusDB [99] (batched vector search), and Vision Transformer (ViT) [101] (GPU-intensive deep learning). These workloads cover dense, sparse, and batched access patterns, i.e. the primary drivers of the "boundary tax" in modular architectures.

Table 4.5: Pipeline stages per benchmark workload.

Benchmark	Pre-processing	Processing	Post-processing
MobileNetV2	Image decoding, resizing, normalization	CNN inference using ONNX Runtime	Parsing, formatting output
XGBoost	Building feature vector	Inference using pre-trained boosted trees	Parsing prediction output
PageRank	Building adjacency list	Iterative PageRank computation	Extracting top-10 vertices
MilvusDB w/ image embeddings	Image decoding, resizing, embedding extraction	Approximate NN, top-3 most similar images	Parsing prediction output
Vision Transformer (ViT)	Image decoding, resizing, normalization	Deep learning transformer-based image inference	Parsing, formatting output

**Performance Characteristics.** Our evaluation spans both CPU-only and GPU-accelerated settings. In CPU-only scenarios, computation typically dominates the total runtime, with IPC accounting for roughly 1% of latency. However, as compute phases accelerate (e.g., via GPU or AMX), data movement becomes a disproportionately larger bottleneck [5, 102]. Rocket is designed to scale its benefits in these high-throughput environments by reducing memory bus contention and improving the overlap between compute and communication.

#### 4.7.2 Impact on Application Performance

**Measurement Methodology** To ensure a rigorous evaluation, we define and measure latency and throughput as independent metrics. This distinction is essential in asynchronous and pipelined systems, where concurrent execution decouples the completion of individual operations from overall system progress.

- **Latency.** We measure end-to-end latency at the granularity of individual IPC operations. For each request, we record a timestamp immediately before the client thread initiates the call and another immediately after the final response is received. The reported average latency is computed as the mean of these per-operation measurements across all client threads.
- **Throughput.** We measure throughput by recording the total wall-clock time required to complete the entire benchmark. Throughput is then computed by dividing the total number of operations by this duration.

In a strictly serial, single-threaded setting ( $n = 1$ ), latency and throughput exhibit a direct inverse relationship. This coupling breaks down in multi-threaded and oversubscribed configurations ( $n > 1$ ), where resource contention and overlapping execution cause the two metrics to diverge. By decoupling latency and throughput, we can accurately quantify how Rocket improves system efficiency through coordinated parallelism rather than through simple per-operation speedups.

Figure 4.10 illustrates the impact of Rocket’s execution modes and cache policies on application-level performance across varying system loads. A key observation is that as the number of parallel clients increases, the gains in throughput and the reductions in latency follow distinct trajectories. Rocket’s asynchronous and pipelined modes consistently outperform the CPU-based baseline by allowing the CPU to prepare subsequent requests while the DSA engine handles data movement in the background.

PageRank shows a distinct performance pattern in Figure 4.10, especially across Rocket’s execution modes. Unlike request-driven ML inference, PageRank follows an iterative, bulk-synchronous model that repeatedly exchanges large intermediate states. Because each iteration is bounded by global synchronization, asynchronous execution alone provides limited benefit, leading to similar performance for the *sync* and *async* modes. In contrast, the *pipeline* mode overlaps computation and data transfer across iteration boundaries, amortizing data movement over larger transfers and sustaining DSA utilization. This difference explains the substantially larger gains observed for PageRank under pipelined execution. We present a more detailed per-workload and per-mode analysis of these effects in [96].

To isolate the specific sources of performance improvement, Figure 4.11 provides an end-to-end latency decomposition for the **MobileNetV2** workload across different backends and execution modes. This breakdown allows us to distinguish between the gains achieved through raw hardware acceleration and those resulting from the architectural orchestration within Rocket.

The transition from `sync_cpu` to `sync_dsa` illustrates the baseline impact. While the dedicated engine reduces data movement time, the CPU remains stalled during the transfer. In contrast, the shift to `async_dsa` highlights the primary strength of our design. As shown in the figure, the significant reduction in end-to-end latency in the asynchronous path stems both from this orchestrated parallelism and the hardware copy speed. This confirms that while accelerators provide the necessary bandwidth, a unified, offload-aware runtime is essential to truly mitigate the “boundary tax” by enabling effective compute-

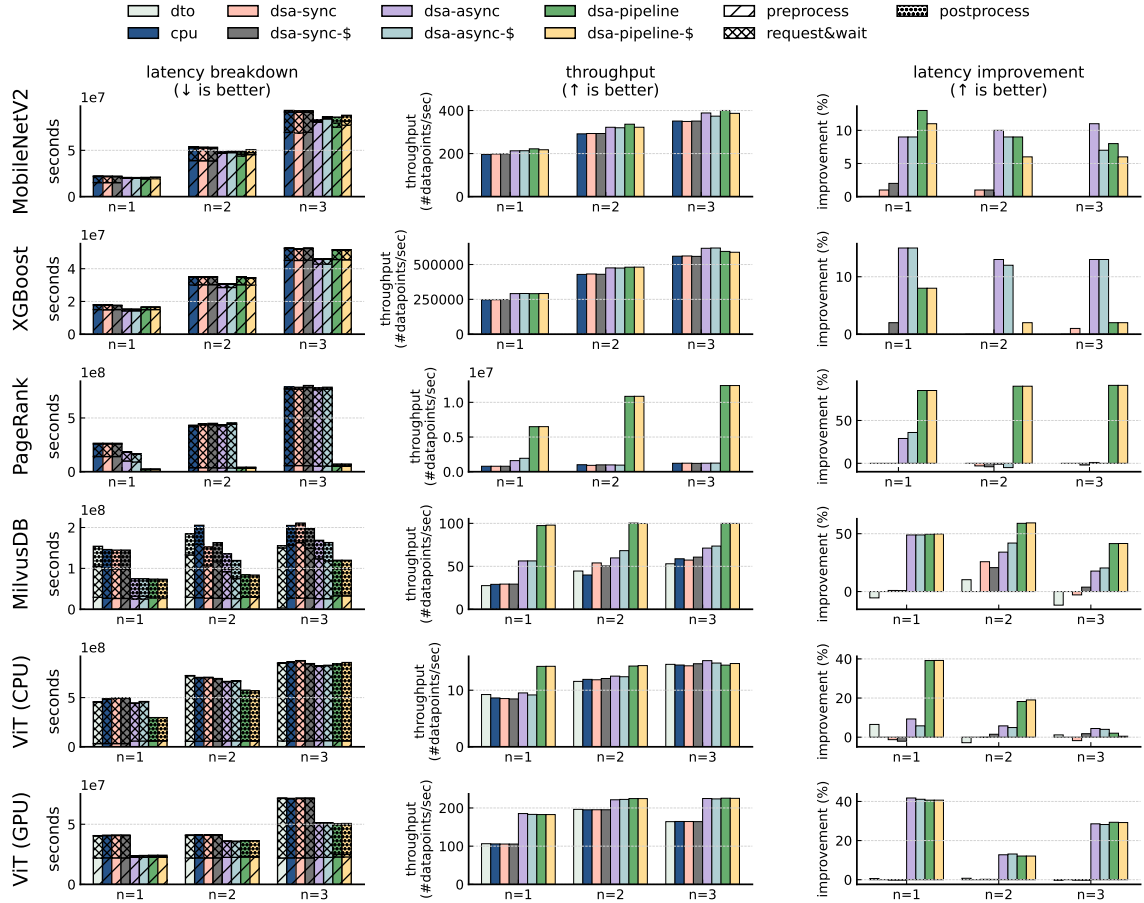


Figure 4.10: Impact on execution time breakdown (left), throughput in images/sec (middle), and end-to-end latency improvement over CPU-based baseline (right) across for different Rocket IPC implementations for MobileNetV2, XGBoost, PageRank, MilvusDB, blue ViT (CPU), ViT (GPU) under varying system load: (a) undersubscribed ( $n=1$ ), (b) matched ( $n=2$ ), and (c) oversubscribed ( $n=3$ ).

communication overlap. A more detailed breakdown of the per-stage latency contributions and overlap behavior is presented in our prior work [96].

**Comparison with Industry Baselines.** We evaluate Rocket against standard CPU `mmap` and vendor-supported frameworks like Intel DTO. Our results show that DTO consistently underperforms, often falling behind even the CPU baseline. This is due to its indiscriminate offloading of all transfers, which introduces queuing delays for small data chunks where CPU-based movement is more efficient. In contrast, Rocket uses its adaptive orchestration layer to selectively offload only when hardware acceleration provides a tangible net gain.

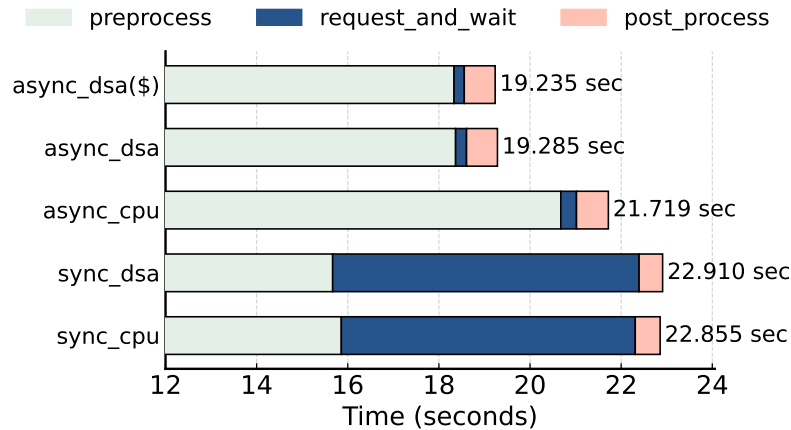


Figure 4.11: End-to-end latency decomposition across devices and execution modes (MobileNetV2).

**Effectiveness of Execution Paradigms.** Asynchronous modes (`async` and `pipelined`) consistently outperform synchronous baselines.

- **Asynchronous Mode:** Provides the best balance for individual request latency by decoupling submission from completion, allowing the CPU to hide transfer costs during preprocessing.
- **Pipelined Mode:** Maximizes throughput by batching commands and deferring synchronization. This is particularly effective in high-load scenarios like MilvusDB and ViT, where aggregate data volume is high.

**Role of Cache Injection.** Selective cache injection proves beneficial in low-contention, single-threaded scenarios (e.g., MobileNetV2 in Figure 4.10), where placing data directly in the LLC reduces immediate cold-start penalties. However, Rocket correctly identifies when to disable this feature under multi-threaded oversubscription to prevent cache thrashing and reference churn.

### 4.7.3 Architectural Sources of Improvement

**Instruction and Bus Cycle Reduction.** By offloading the data movement logic, Rocket reduces the CPU’s instruction count and cycle consumption by up to 23% compared to syn-

chronous baselines (Figure 4.12). The pipelined mode achieves the highest efficiency by amortizing the cost of work submission across multiple requests.

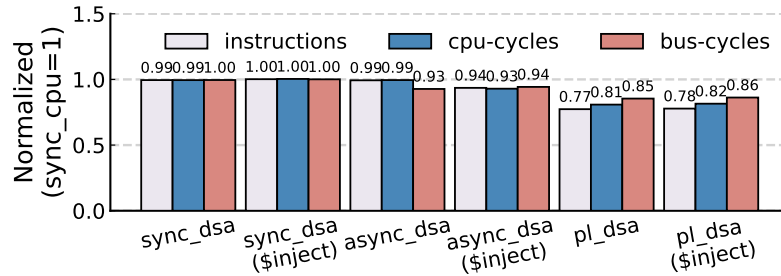


Figure 4.12: Normalized instruction counts, CPU cycles, and bus cycles with DSA-based offloading, relative to the synchronous CPU baseline from MobileNetV2 benchmark.

**Cache and TLB Efficiency.** Rocket mitigates CPU stalls by reducing the volume of cache accesses required for bulk data movement. While asynchronous offloading slightly increases dTLB misses (as the IOMMU handles translation), the overall reduction in CPU-resident stall rates (down to 0.89 in optimal configurations) confirms that the trade-off favors improved parallelism.

**Acceleration in GPU Environments.** As shown in the last 2 rows in Figure 4.10, the most dramatic gains are observed in the ViT workload when using GPU acceleration. While the CPU compute-bound phases shrink, the IPC overhead expands by nearly 10×. In this regime, Rocket’s pipelined execution yields up to a 40% throughput improvement over optimized CPU baselines. This confirms that as data-intensive systems transition to faster compute engines, the “boundary tax” becomes the defining performance constraint, making Rocket’s unified orchestration essential.

In summary, Rocket avoids the pitfalls of rigid offloading by providing a flexible, hardware-aware runtime. By intelligently managing synchronization, cache visibility, and backend selection, it delivers stable performance gains across varying workloads, data sizes, and system-level contention scenarios.

## 4.8 Summary

This chapter presented **Rocket**, a unified IPC runtime suite designed to mitigate the "boundary tax" in modern data-intensive and AI-driven pipelines. As systems transition toward modular and split-architectures, the primary bottleneck has shifted from computation to the efficiency of inter-process data movement. Traditional IPC stacks, which rely on synchronous, CPU-driven `memcpy`, fail to exploit the emerging landscape of hardware and software-based data movers. To resolve this, **Rocket** introduces an orchestration layer that decouples data movement from core execution and treats it as a managed system service.

**Rocket** overcomes the integration challenges and performance trade-offs of modern accelerators through three primary architectural pillars. First, it provides a unified orchestration of heterogeneous backends, integrating dedicated hardware engines like Intel DSA, kernel-resident services, etc., into a single shared-memory framework. Second, it implements asynchronous and pipelined execution models that enable true overlap between compute and communication, transforming serialized data transfers into parallel system-wide dataflows. Third, **Rocket** introduces hardware-aware synchronization and cache management, utilizing hybrid polling via `UMWAIT` and selective cache injection to balance latency, CPU efficiency, and cache hierarchy pressure.

The evaluation demonstrates that **Rocket** significantly enhances system-wide efficiency across a range of real-world workloads, including AI inference, graph analytics, and vector databases. By intelligently delegating transfers to the most suitable backend, **Rocket** achieves up to a  $2.1\times$  improvement in throughput and a 72% reduction in end-to-end latency. Furthermore, it reduces CPU instruction counts by up to 22%, proving that coordinated IPC orchestration is essential for reclaiming cycles in oversubscribed or GPU-accelerated environments where data movement becomes the dominant bottleneck.

#### 4.8.1 Relationship to the Thesis Statement

The findings in this chapter reinforce the central thesis that IPC must evolve from a passive data conduit into an active, intelligent runtime layer. While the previous chapter on **Pocket** established the necessity of resource-awareness at the communication boundary, **Rocket** demonstrates that this intelligence must extend to the physical data movement path itself. It proves that when an IPC mechanism is aware of the underlying hardware characteristics and temporal reuse patterns, it can effectively bridge the gap between raw hardware capabilities and high-level application requirements.

This hardware-aware foundation serves as a critical bridge in this dissertation. By providing a flexible and high-performance transport layer, **Rocket** sets the stage for the final evolution of this framework. While this chapter focuses on optimizing the physical movement through unified orchestration, the following chapter introduces **SkyRocket**, which leverages these capabilities to provide workload-driven, adaptive IPC policies. Together, these systems establish a comprehensive architecture for efficient, intelligent data movement in modern data-centric computing environments.

## CHAPTER 5

### ADAPTIVE IPC RUNTIME CONFIGURATION WITH ML-BASED TUNING MODEL

SkyRocket brings workload-driven adaptation to IPC systems by supporting heterogeneous data-movement backends, including CPU-based copiers and hardware accelerators. While Rocket provides an efficient IPC software suite that supports flexible execution modes and cache management, it currently requires applications to manually select configurations. Skyrocket seeks to extend this foundation by introducing runtime workload-driven adaptation, optimizing IPC behavior based on the workload characteristics based on runtime execution information such as modality, size, lifespan, the number competing threads, and etc.

#### **5.1 From Fixed to Fluid: The Need for Dynamic Policy Orchestration in Unified IPC Runtimes**

Modern data-centric systems increasingly rely on heterogeneous mechanisms for moving data across processes. Hardware accelerators such as Intel DSA, software engines like high-performance copy libraries, and new industry standards (e.g., OCP efforts on unified data-movement APIs [103]) collectively expand the range of available execution paths for IPC. As these options grow, the cost of a data transfer no longer depends solely on the underlying hardware, but also on how the system selects among diverse backends and configuration modes.

Selecting the right execution policy is non-trivial. The performance of a transfer changes significantly with properties such as data size, reuse locality, and memory-hierarchy pressure. Small latency-critical payloads often benefit from cache residency and minimal indirection, while large streaming transfers achieve higher throughput when bypassing caches

or using offload engines. These characteristics fluctuate across workloads and are rarely known with certainty at compile time, making it necessary to adjust execution policies dynamically at runtime.

Systems like Rocket illustrate this expanding design space. Rocket decouples IPC from a single backend and exposes multiple execution paths, including CPU copies and hardware DMA engines, together with configuration knobs such as cache-injection modes and pipelined transfers. This flexibility enables high performance across heterogeneous scenarios, but it also shifts the responsibility of selecting an appropriate policy to the application. When a workload changes or when system conditions evolve (e.g., LLC pressure or multi-client contention), a previously effective policy may become suboptimal. The runtime provides no mechanism to adapt, leading to wasted cache capacity, serialized transfers, and reduced throughput.

As the option space grows richer, manual tuning becomes brittle and difficult to maintain. Future IPC systems will need principled ways to evaluate available execution modes and adjust behavior as workloads evolve. This motivates an adaptive layer that can reason about data-movement trade-offs and select appropriate configurations without developer intervention.

SkyRocket uses lightweight runtime signals and a learned performance model to determine dynamic policy choices, while simultaneously providing mode-level predictions that serve as compile-time hints for configurations that cannot be changed at runtime. It infers whether a transfer should remain in cache, bypass cache, or be offloaded to an accelerator, and applies the corresponding configuration at request time. By making IPC policy selection data-driven and adaptive, SkyRocket ensures that Rocket’s flexible backend architecture is used effectively under diverse and changing conditions.

## 5.2 Design Overview

SkyRocket is designed as a standalone adaptive control layer that operates above a generic shared-memory IPC substrate. Modern IPC frameworks often expose rich but low-level configuration surfaces—multiple data-movement backends ([26, 24]), optional cache behaviors, and diverse synchronization APIs—yet even when these abstractions are fully available, the space of possible configurations and their performance trade-offs remains difficult for users to evaluate. Selecting a backend or cache policy typically requires detailed awareness of transfer size, locality, concurrency, and memory-hierarchy pressure—conditions that fluctuate at runtime and are rarely observable with precision at the application level. As a result, developers must either commit to static choices or rely on coarse rules of thumb that can be fragile under dynamic workloads.

SkyRocket addresses this gap by turning IPC configuration into a per-request, fine-grained, and runtime-adaptive decision problem. Rather than expecting users to manually reason about subtle interactions between request characteristics and backend behaviors, SkyRocket provides a systematic mechanism for profiling workload signals, learning their performance implications offline, and applying optimal configurations online on a per-request basis. Figure 5.1 summarizes this architecture.

### 5.2.1 System-Level Workflow Overview

At a high level, SkyRocket follows a workflow pattern common to adaptive systems:

1. **Offline policy construction**
2. **Online policy application**
3. **Feedback for future revision**

**Offline Policy Construction.** The system collects profiling traces under a workload mix that captures a range of data-movement patterns. In practice, this can be instantiated either

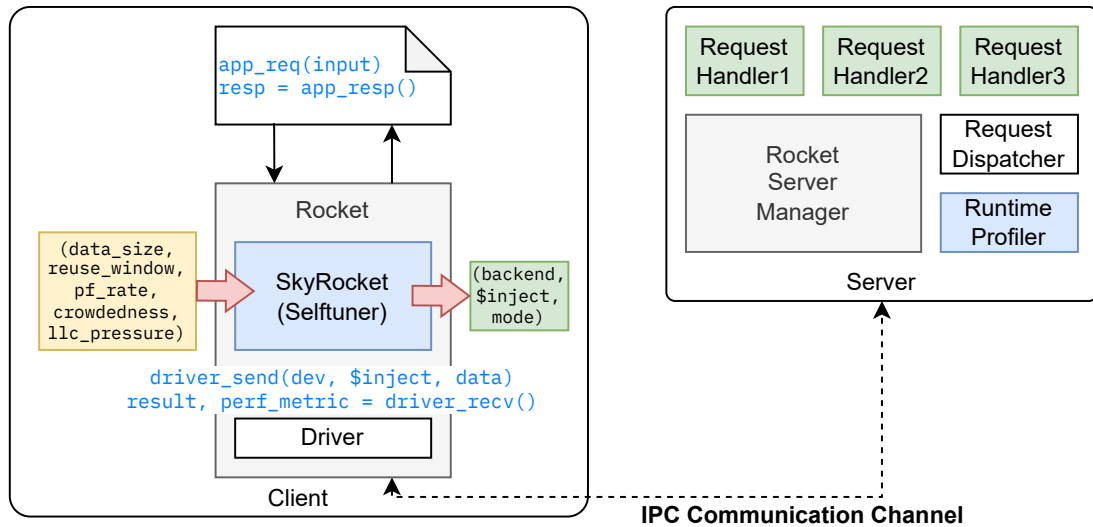


Figure 5.1: Design blueprint of Skyrocket.

as a hardware-level profiling suite that is reused across applications, or as application-specific traces when tighter tuning is desired. These traces capture broad categories of observable signals: *workload-driven*, *resource-pressure*, and *concurrency-related* indicators. A predictive model is trained using an 80/20 train–test split (43,200 samples total), and the model outputs a configuration tuple representing the preferred policy under each feature combination.

**Online Policy Application.** At runtime, the client performs inexpensive profiling of each outgoing request. The embedded predictor evaluates the current feature vector and issues a configuration update. SkyRocket’s control plane is designed to operate across arbitrary adaptation timescales, allowing decision frequency to align with workload dynamics rather than a fixed interval. In our evaluation, we instantiate this mechanism with request-level updates to illustrate the model’s responsiveness.

**Feedback for Future Revision.** The system logs predicted optimal structural modes, enabling refinement during later profiling phases. Lightweight statistics gathered online support stable normalization and feature scaling.

This workflow defines an explicit *control plane* for IPC behavior, where policy decisions are grounded in data rather than fixed heuristics.

### 5.2.2 Configuration Space Model

SkyRocket structures the IPC configuration space according to when decisions can be safely applied and how they interact with execution semantics.

**Structural (Compile-time) Parameters** Structural parameters influence control flow or the programming model, making them unsuitable for runtime mutation. SkyRocket evaluates these parameters during offline policy construction and stores the best-performing mode as a static deployment hint.

**Dynamic (Runtime) Parameters** Dynamic parameters influence backend selection and memory-hierarchy effects but do not affect program logic. SkyRocket treats these parameters as *fine-grained policy actions*, adaptable at runtime on a per-request basis.

This partitioning exposes a clear decision boundary: structural choices remain stable, while behavioral choices adapt at high temporal resolution.

### 5.2.3 Policy Construction Pipeline

SkyRocket's offline stage collects profiling traces and trains a regression-based model over observable features. The features are grouped into broad categories applicable across diverse IPC backends:

- **Workload characteristics:** payload magnitude, short-term locality
- **Resource-pressure signals:** cache load, memory interference
- **Concurrency indicators:** number of active clients, interleaving effects

The dataset (43,200 samples) is divided into 80% training (34,560 samples) and 20% testing (8,640 samples). The predictor outputs values for the configuration parameters exposed by the underlying IPC backend. For Rocket, this corresponds to the tuple:

$$(device, cache\_injection, mode_{hint})$$

Only the first two fields are applied online, but all three are emitted to support deployment time structural decisions.

The trained model is serialized into a compact artifact embedded in the client runtime. This pipeline converts a high-dimensional configuration problem into a predictable, data-driven policy function.

### 5.3 Prediction Model

SkyRocket introduces a predictive configuration layer that treats backend selection as a runtime policy decision problem. The goal of this section is to describe how we identify useful runtime signals, motivate our feature hypothesis, and develop the model that predicts transfer cost.

#### 5.3.1 Quantitative Characterization of IPC Backends

To understand which runtime signals meaningfully influence IPC performance, we conduct a characterization study on the backend options provided by our prototype IPC substrate. The purpose of this study is not to derive backend-specific heuristics, but to identify general, repeatable tendencies that can be exploited for policy inference.

These factors correspond to three broad categories that influence IPC behavior: (1) workload size and locality, (2) memory- and cache-pressure indicators, and (3) concurrency-related signals. Together, these categories establish the necessary context for the two fundamental optimization axes central to IPC design: the choice of transfer mechanism (e.g.,

Table 5.1: Runtime context factors used by SkyRocket for execution policy inference.

<b>Factor</b>	<b>Runtime meaning</b>
data_size_mb	amount of data transferred per call
reuse_window	reuse locality from the receiver
pf_rate	degree of page fault activity
llc_pressure	cache contention from co-located workloads
num_clients	multi-tenant contention proxy

maximizing throughput via DMA or latency via CPU copies) and the optimal use of the memory hierarchy (e.g., cache injection). Their relevance is grounded in architectural considerations: data size affects whether AVX512-based CPU copies or DMA engines are advantageous; reuse-window, page-fault rate, and LLC pressure capture conditions where cache residency is beneficial or harmful, and the number of active clients acts as a proxy for queueing and interference.

### 5.3.2 Runtime Features and Design Hypothesis

Empirically, our prototype system exhibits consistent behavior across a wide range of experiments: certain combinations of these signals correlate with the relative advantage of CPU copy, cache-injected paths, or hardware offload. These observations motivate the following design hypothesis:

**Hypothesis:** *A compact set of workload-, pressure-, and concurrency-related runtime signals is sufficient to infer which IPC execution policy minimizes transfer cost.*

This hypothesis is *not* a claim of universal sufficiency. Rather, it reflects trends repeatedly observed in our prototype, and serves as the foundation for our predictive model. Whether the learned model generalizes and achieves high accuracy is evaluated empirically in § 5.5.

These patterns motivate using a learning-based model rather than hand-tuned rules, enabling SkyRocket to infer backend behavior directly from the observed context.

Table 5.2: Context–policy relationships commonly observed in our prototype system.

<b>Runtime observable</b>	<b>Implication for policy selection</b>
Large <code>data_size_mb</code>	Stream/offload large transfers to avoid LLC pollution
High <code>pf_rate</code>	Avoid CPU stall amplification; prefer DMA/streaming
High <code>llc_pressure</code>	Disable cache injection to preserve shared capacity
Short <code>reuse_window</code>	Favor cache residency for short-term reuse

### 5.3.3 Learning-Based Configuration Engine

Backend performance exhibits nonlinear interactions across multiple runtime signals. Threshold effects, workload-dependent locality, and pressure-induced behavior shifts make simple heuristics insufficient. SkyRocket therefore employs a regression model based on Gradient Boosting Decision Trees (GBDT).

SkyRocket’s configuration engine operates over a small, discrete policy space and makes decisions based on instantaneous system conditions, such as request size, concurrency, and cache pressure. Although the configuration itself is relatively static, the interactions among these signals exhibit nonlinear threshold effects that make fixed rules or lookup tables brittle without extensive manual tuning. Models that emphasize temporal correlation are less effective in this setting, as decisions are driven by momentary system states rather than long-term trends, while neural networks introduce unnecessary runtime complexity for a low-dimensional control path. Gradient Boosting Decision Trees provide a lightweight mechanism to capture nonlinear decision boundaries with predictable inference cost and minimal overhead, making them a natural fit for SkyRocket’s configuration setting.

GBDT is well-suited for this setting for three reasons:

- **Threshold semantics.** Tree splits naturally express threshold boundaries (e.g., data-size cutoffs) that frequently appear in cache-aware and hardware-assisted transfer paths.
- **Feature interactions.** Many policies are defined not by a single signal but by combi-

nations (e.g., cache injection is beneficial only when data is small *and* cache pressure is low).

- **Cost-based predictions.** The regressor outputs predicted service time rather than a categorical backend label, enabling SkyRocket to select the policy with the lowest estimated cost.

SkyRocket trains this model using profiling traces collected offline from representative workloads. These traces capture diverse combinations of the runtime factors listed in Table 5.1 and provide the supervision signal for predicting transfer cost under each configuration option.

#### 5.3.4 Performance Prediction Model

SkyRocket casts configuration selection as a cost-minimization problem. It employs a regression model to estimate the service time across available execution paths, treating the predicted latencies as relative cost metrics for policy ranking. Let the training dataset be

$$\mathcal{D} = \{(\mathbf{x}_i, t_i)\}_{i=1}^N, \quad t_i > 0,$$

where  $\mathbf{x}_i$  denotes the workload–configuration feature vector and  $t_i$  is the measured service time in microseconds. To stabilize variance and reduce the skew inherent to latency measurements, we apply a logarithmic transformation:

$$y_i = \log(1 + t_i). \tag{1}$$

SkyRocket adopts LightGBM as its prediction model. The model represents the regression function as an additive ensemble of decision trees:

$$f_M(\mathbf{x}) = \sum_{m=1}^M \eta h_m(\mathbf{x}), \tag{2}$$

where  $h_m$  is the  $m$ -th regression tree,  $\eta$  is the learning rate, and  $M$  is the number of boosting iterations. Parameters are obtained by minimizing the standard  $\ell_2$  objective:

$$\min_f \mathcal{L}(f) = \sum_{i=1}^N (y_i - f(\mathbf{x}_i))^2 + \Omega(f), \quad (3)$$

where  $\Omega(f)$  regularizes model complexity (e.g., number of trees, number of leaves). LightGBM optimizes Eq. (3) using second-order gradient boosting, efficiently incorporating both numerical and categorical workload features.

During inference, predicted log-latencies are mapped back to the original microsecond scale by inverting the log transformation:

$$\hat{t}(\mathbf{x}) = \exp(f_M(\mathbf{x})) - 1. \quad (4)$$

At request time, the runtime evaluates the model for each candidate configuration and selects the policy  $\mathbf{x}^*$  that minimizes the predicted service time:

$$\mathbf{x}^* = \arg \min_{\mathbf{x}} \hat{t}(\mathbf{x})$$

By focusing on the relative ordering of these costs, SkyRocket identifies the optimal execution knob even in the presence of minor variance in absolute latency estimation.

### 5.3.5 Transparent Runtime Integration

SkyRocket deploys the trained regressor as a lightweight prediction module within the IPC client library. Applications continue to issue IPC calls using their existing API; the system monitors runtime context, evaluates the embedded model, and applies fine-grained configuration choices per request. This integration preserves API semantics while enabling dynamic, context-sensitive policy selection without developer intervention.

## 5.4 Runtime Adaptation

This section details the design and implementation of SkyRocket’s adaptive execution loop, which leverages an embedded predictor to dynamically tune IPC parameters on a per-request basis. It explains how the system captures fine-grained workload signals to optimize device selection and cache injection with minimal runtime overhead. Furthermore, the text distinguishes between these online-tunable settings and static structural configurations, such as execution modes, which are fixed at deployment time. Finally, it outlines the workflow for training and integrating workload-specific models into the runtime stack.

### 5.4.1 Adaptive Execution Loop

During execution, SkyRocket forms a feature vector for each request using lightweight runtime observations and submits it to the embedded predictor. The predictor outputs updates for the parameters that are online-tunable in the underlying IPC backend. In Rocket, this corresponds to:

*(device, cache\_injection)*

However, the mechanism generalizes to any backend that exposes a similar runtime configuration surface. SkyRocket is designed to support a configurable adaptation interval, allowing policy updates to be applied at any granularity from per-request triggers to batch-level adjustments. In our evaluation, we apply configuration updates at a resolution that matches the fluctuation of workload signals. Metrics such as compute crowdedness and LLC pressure can fluctuate at the granularity of individual requests (or small batches). These fluctuations directly affect whether cache injection is beneficial and whether CPU or accelerator execution paths are preferable. Because these signals largely determine transfer behavior, adapting configuration at the same fine-grained resolution allows the runtime to capture short-lived optimization opportunities that coarser-grained policies would miss.

In Rocket’s design, applying such updates is both safe and low-cost. The IPC runtime

already handles ordering, buffer ownership, and completion tracking. As a result, SkyRocket does not require additional correctness mechanisms, and configuration adjustments do not perturb the data plane. When workload characteristics change more slowly, the same mechanism may operate at a coarser cadence. The choice of cadence is left to the control plane rather than fixed by the system design.

This adaptive loop allows SkyRocket to respond closely to variations in locality, interference, and system pressure. While the prototype is implemented on Rocket, the underlying principle (adapting online-tunable parameters at the timescale of workload variation) extends naturally to IPC backends that expose similar runtime configuration surfaces.

#### 5.4.2 Implementation Notes

SkyRocket is implemented as a modular policy component inside the runtime stack. The predictor introduces a minimal overhead of approximately  $75 \mu s$  on the critical path, executing on a per-request basis. This investment is highly cost-effective, as the resulting optimal policy decisions reduce the overall end-to-end latency by up to several seconds. Stable prediction accuracy is maintained through periodic refresh of normalization statistics.

The core policy actions (i.e., applying the derived configuration) occur at the control-plane level, ensuring the data plane remains fully unchanged. This design principle means that SkyRocket’s decisions decouple the adaptive behavior from direct modifications to the data plane components, allowing adaptation without entangling application logic with backend heterogeneity, even though the low-latency prediction itself is invoked per-request.

#### 5.4.3 Deployment-time Configuration and Policy Integration

While SkyRocket provides fine-grained, dynamic policy decisions at runtime (e.g., `device` and `cache_injection` selection), its operation relies on **structural configuration choices** set at deployment time. This section clarifies the role of these coarse-grained knobs and

details the process by which SkyRocket’s learned model is integrated into the runtime stack.

**Structural Knobs and Static Hints.** The IPC configuration space is partitioned into dynamic parameters that can be safely adapted per-request, and **structural parameters** that govern the control flow, such as the **synchronous, asynchronous, or pipelined execution modes**. These structural choices must be fixed at compile or deployment time as they fundamentally influence the application’s programming model. SkyRocket addresses this through **static deployment hints**: the offline policy pipeline produces a `mode_hint` recommendation alongside the runtime parameters. This recommendation guides the user or the build system toward the most efficient structural mode for their specific workload.

**Model Integration and Automation Scope.** Integrating SkyRocket involves the **initial configuration and deployment of its prediction artifact**. Instead of traditional manual code or system parameter changes, the primary deployment-time effort involves:

1. **Workload Profiling:** Running the provided data collection scripts to gather traces under representative workload mixtures.
2. **Model Training:** Executing the learning scripts to train a **workload-specific GBDT model**.
3. **Artifact Integration:** Serializing the trained model into a compact artifact and embedding it into the client runtime for online inference.

Alternatively, users can opt for a **default pre-trained model** to bypass this initial profiling and training. The coarse-grained knobs are thus configured indirectly by selecting the most appropriate prediction model and the corresponding Rocket DSA structural mode. The future scope for automation lies in developing a mechanism to automatically select and integrate the best **structural mode** based on the generated `mode_hint`, reducing the entire deployment process to simple integration of the final model artifact.

## 5.5 Evaluation

This section presents the empirical evaluation of SkyRocket, validating both the accuracy of its learning components and their runtime efficiency. It begins by quantifying the predictive performance of the regression and classification models, demonstrating high fidelity in latency estimation and configuration selection. The analysis then examines the statistical significance of the input features and evaluates the computational overhead of the system, confirming that the cost of decision making remains negligible. Finally, a timeline-based case study illustrates SkyRocket’s ability to dynamically adapt execution policies such as cache injection and device offloading in response to fluctuating workload characteristics.

### 5.5.1 Regression Model Performance

Table 5.3: Evaluation summary of the regression model.

Category	Metric	Value
<b>Regression Performance (log-scale)</b>		
	Train RMSE	0.1523
	Test RMSE	0.1608
<b>Regression Performance (original scale, <math>\mu</math>s)</b>		
	Test MAE	6,635.06
	Test RMSE	16,237.06
	$R^2$	0.9778
<b>Optimal Configuration Classification</b>		
	Accuracy	74.3%
	Macro F1	0.75
	Workloads (Test)	1161
	Deviation from Optimal (MAE)	5,913.76 $\mu$ s
<b>Per-Class Metrics</b>		
	async_cpu.\$bypass	Precision 0.78, Recall 0.72, F1 0.75, Support 191
	async_dsa.\$bypass	Precision 0.76, Recall 0.70, F1 0.73, Support 193
	async_dsa.\$inject	Precision 0.72, Recall 0.84, F1 0.77, Support 216
	sync_cpu	Precision 0.70, Recall 0.73, F1 0.72, Support 170
	sync_dsa.\$bypass	Precision 0.81, Recall 0.68, F1 0.74, Support 186
	sync_dsa.\$inject	Precision 0.71, Recall 0.82, F1 0.76, Support 205

Table 5.3 summarizes the predictive accuracy of SkyRocket’s regression and classifica-

tion components. The regression model achieves an RMSE of 0.1523 on the training data and 0.1608 on the held-out test set in log-scale space, indicating stable generalization with no evidence of overfitting. When errors are projected back to the original latency domain, the model attains a MAE of 6.6ms and an RMSE of 16.2ms, corresponding to an  $R^2$  of 0.9778. These results indicate that the model captures the dominant latency structure of IPC operations with high fidelity and preserves the explained variance across scales.

The policy selection stage likewise shows strong performance. By selecting the configuration with the minimum predicted latency, SkyRocket achieves 74.3% accuracy across 1,161 distinct workloads, with a macro-F1 score of 0.75. Per-class metrics in Table 5.3 show balanced behavior across CPU/DSA and `bypass/inject` combinations, with F1-scores ranging from 0.72 to 0.77. Notably, classes involving cache injection (`async_dsa_inject` and `sync_dsa_inject`) exhibit high recall (0.84 and 0.82), indicating that the model reliably detects scenarios where injection policies are important. When the selected configuration differs from the oracle-optimal choice, the mean performance gap is only 5.9 us. This negligible penalty confirms that even "misclassified" instances result in near-optimal performance, as the model effectively captures the underlying cost structure.

Overall, the measurements in Table 5.3 demonstrate that SkyRocket's learning components provide accurate latency prediction and robust configuration recommendations, enabling adaptive policy selection without compromising end-to-end performance.

### 5.5.2 End-to-End System Validation

SkyRocket's runtime adaptation preserves optimal end-to-end performance at the application level. We evaluate a full MobileNetV2 inference pipeline to assess whether dynamic IPC decisions affect overall latency.

Figure 5.2 compares SkyRocket with a static oracle-optimal configuration. The oracle reflects the best fixed policy selected with full knowledge of each transfer. SkyRocket matches this performance without prior information. It dynamically selects between CPU

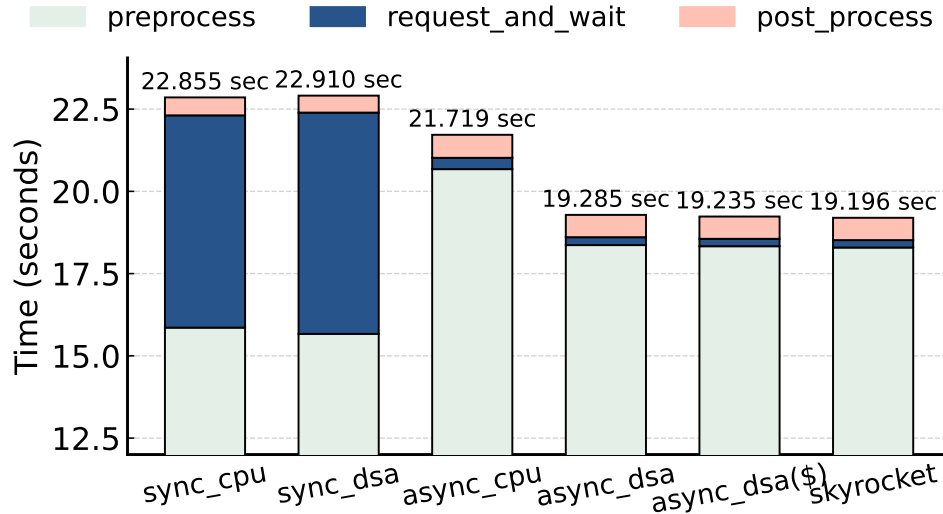


Figure 5.2: Comparison of end-to-end results demonstrating that the async-\$ configuration yields performance equivalent to the optimal setting.

copies and hardware acceleration to follow the oracle’s effective configuration at runtime. These results show that SkyRocket achieves optimal end-to-end behavior without manual tuning or workload-specific policies. The end-to-end results confirm that our method achieves performance equivalent to the optimal configuration and follows the same configuration trajectory.

### 5.5.3 Decision Analysis of Input Features

Table 5.4 summarizes how each input feature contributes to SkyRocket’s two runtime decisions: whether to enable cache injection and whether to use the DSA device. The results follow a consistent pattern that aligns with the model behavior previously observed in Table 5.3.

For cache-injection decisions, two features—`data_size_mb` and `llc_pressure`—exhibit clear statistical significance ( $p \approx 0.012$  and  $p \approx 0.013$ , respectively). This agrees with the classifier’s strong recall for cache-injection classes, indicating that the model reliably identifies workloads where enabling cache injection yields performance benefits.

Other features show little or no statistical influence. For instance, `pf_rate` exhibits

Table 5.4: Significance test results for decision-related input features.

<b>Decision</b>	<b>Input Feature</b>	<b>P-Value / Significance</b>
<b>cache_inject</b>		
	data_size_mb	0.012001 (Yes)
	llc_pressure	0.013044 (Yes)
	pf_rate	0.137650 (Somewhat)
	num_clients	0.507112 (No)
	reuse_window	0.902850 (No)
<b>device</b>		
	pf_rate	0.928966 (No)
	num_clients	0.989061 (No)
	llc_pressure	0.183641 (Somewhat)
	data_size_mb	0.058129 (Yes)
	reuse_window	0.829394 (No)

only a weak effect, while `num_clients` and `reuse_window` show no meaningful contribution. This mirrors the classifier’s behavior, where these features do not substantially shift decision boundaries.

Device-selection outcomes follow a simpler pattern. Among all features, only `data_size_mb` reaches marginal significance ( $p = 0.058$ ), whereas the remaining features exhibit no detectable effect. This observation is consistent with the balanced F1 scores across CPU and DSA classes in Table 5.3. In practice, data size dominates the decision between CPU and DSA, while other characteristics primarily influence the absolute service time without altering the preferred device.

Overall, the significance tests indicate that SkyRocket’s decisions are driven by meaningful runtime signals. Cache-injection choices depend mainly on data size and LLC pressure, whereas device selection is governed almost entirely by data size. These findings are statistically consistent with the model’s observed behavior and aligned with expected characteristics of IPC workloads.

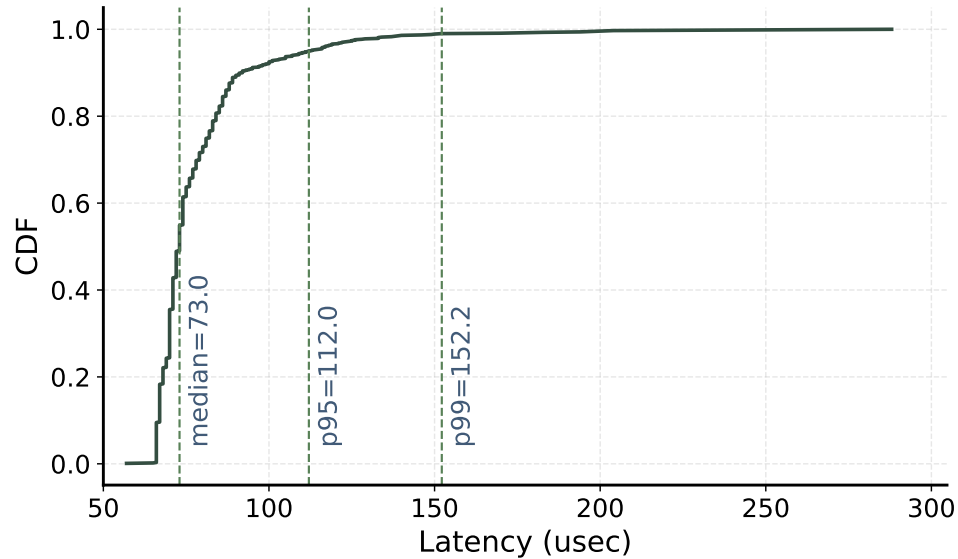


Figure 5.3: CDF of end-to-end decision overhead introduced by the SkyRocket configuration engine. The distribution shows that the vast majority of predictions complete within tens of microseconds, indicating negligible impact on overall IPC latency.

#### 5.5.4 Overhead Analysis

Figure 5.3 shows that SkyRocket’s prediction overhead is statistically insignificant relative to the overall IPC execution time. The end-to-end inference cost averages 75  $\mu$ s (stdev 13  $\mu$ s), with more than 95% of predictions completing below 112  $\mu$ s. This narrow distribution demonstrates that the prediction path is both stable and low-variance.

When placed in context, the magnitude of this overhead becomes negligible. For example, a baseline configuration of the same benchmark (mobilenetv2, CPU) exhibits 22.285 s end-to-end latency, and selecting the optimal configuration yields a 1.705 s improvement. The prediction cost is orders of magnitude smaller than both the total runtime and the performance gains SkyRocket enables. Importantly, the tail percentiles (p95 and p99) remain well below thresholds that could perturb IPC progress, indicating that prediction does not introduce queueing, serialization, or contention with the data plane.

Taken together, the empirical evidence supports a simple conclusion: SkyRocket’s decision engine operates safely within the noise floor of application-level runtime. Its overhead

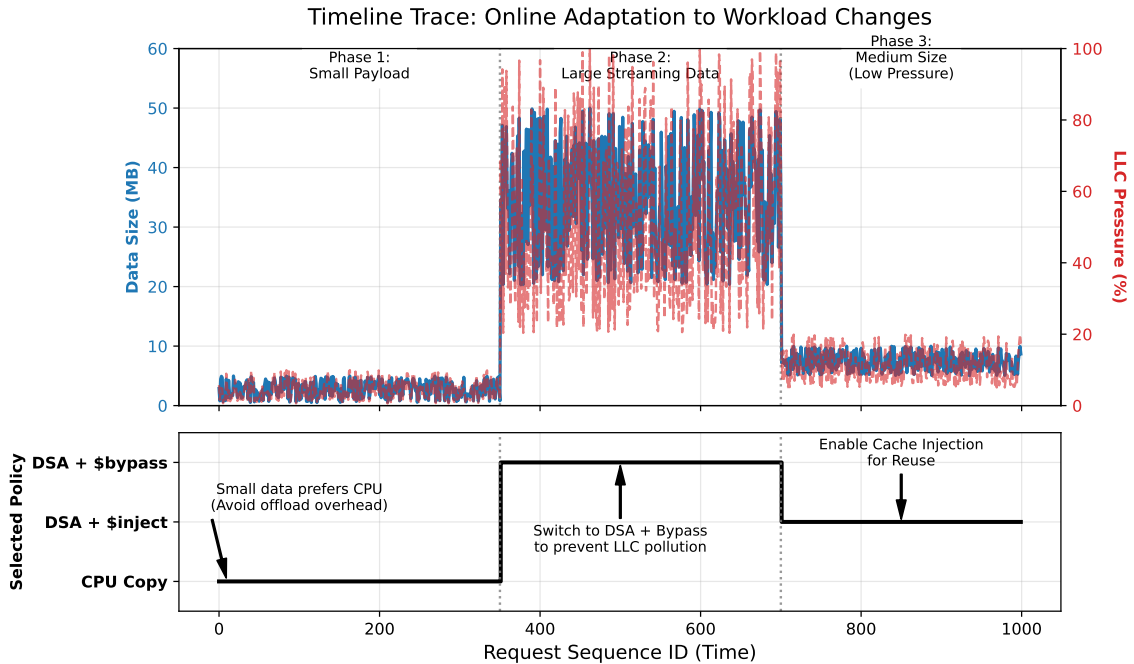


Figure 5.4: Timeline trace showing SkyRocket’s online adaptation to dynamic workload changes. The system switches between CPU Copy, DSA + \$bypass, and DSA + \$inject based on real-time monitoring of data size ( $< 5$  MB vs.  $> 10$  MB) and LLC pressure, effectively preventing cache pollution and optimizing transfer efficiency.

is sufficiently low to permit continuous, per-request adaptation without compromising system throughput or altering workload behavior.

### 5.5.5 Case Study: Online Adaptation to Workload Changes

To demonstrate the runtime adaptability of SkyRocket, we evaluate the trained model using a synthetic benchmark designed to simulate dynamic shifts in workload intensity. We intentionally vary the request data sizes across three distinct regimes—small ( $< 5$  MB), medium (5–10 MB), and large ( $> 10$  MB)—to stress-test the model’s decision logic against prescribed parameter transitions. Figure 5.4 illustrates the variation in input features—specifically data size and LLC pressure—across a sequence of 100 requests, alongside the corresponding execution policies selected by SkyRocket. The experiment reveals three distinct phases of adaptation:

- **Phase 1 (Small Payload, Requests 0–35):** In the initial phase, the transfer size remains small ( $< 5$  MB), and system LLC pressure is low. SkyRocket determines that the initialization overhead of the hardware accelerator outweighs the transfer benefit for such small payloads. Consequently, it consistently selects the `CPU Copy` mode to minimize latency.
- **Phase 2 (Large Streaming Data, Requests 36–70):** The workload shifts to a streaming pattern characterized by large data spikes and elevated LLC pressure. SkyRocket detects these signals and immediately switches to the `DSA + $bypass` (cache bypass) mode. This decision offloads the massive transfer to the accelerator while bypassing the cache, effectively preventing LLC pollution and preserving shared cache capacity for other processes.
- **Phase 3 (Medium Size & Low Pressure, Requests 71–100):** In the final phase, data sizes stabilize at a medium range (5–10 MB), and the LLC pressure subsides. Recognizing the availability of cache capacity and the potential for data reuse, SkyRocket switches to the `DSA + $inject` (cache injection) mode. This strategy warms up the cache for the consumer, thereby reducing subsequent access latency.

This trace demonstrates that SkyRocket successfully transitions beyond static policies. By continuously monitoring runtime signals (as detailed in Table 5.1), the system dynamically selects the optimal data movement path—`CPU`, `DSA-Bypass`, or `DSA-Inject`—for each specific context.

## 5.6 Summary

This chapter presented **SkyRocket**, a workload-driven adaptive IPC control plane designed to automate policy selection in increasingly complex and heterogeneous data-movement environments. While **Rocket** provided the necessary physical backends and execution modes, it left the burden of choosing the optimal configuration to the developer. **SkyRocket**

resolves this by introducing an intelligent adaptation layer that transforms IPC from a set of static configuration knobs into a self-optimizing system service that responds dynamically to runtime environmental shifts.

**SkyRocket** overcomes the challenges of manual tuning and policy brittleness through three primary architectural pillars. First, it implements a **data-driven performance model** using GBDT to predict the service time of diverse IPC configurations based on lightweight runtime signals. Second, it introduces a **fine-grained adaptive execution loop** that monitors workload characteristics such as transfer size, LLC pressure, and page-fault rates, to make per-request decisions on backend selection and cache management. Third, it establishes a **hybrid configuration model** that distinguishes between structural parameters (static deployment hints) and behavioral parameters (runtime policy actions), ensuring that adaptation occurs without violating the application’s execution semantics.

The evaluation demonstrates that **SkyRocket** achieves high-fidelity performance inference, with an score of 0.9778 in latency prediction and a 74.3% accuracy in optimal configuration selection. By enabling per-request adaptation with a negligible overhead of only 75 s, **SkyRocket** consistently identifies the most efficient data-movement path—whether it be `CPU Copy`, `DSA + bypass`, or `DSA + inject`. This automation allows the system to match the performance of hand-tuned optimal configurations across fluctuating workloads, proving that machine-learning-based orchestration can effectively navigate the non-linear trade-offs of modern hardware accelerators.

### 5.6.1 Relationship to the Thesis Statement

The findings in this chapter represent the final evolutionary step of the central thesis: that IPC must evolve into an active, intelligent runtime layer. While **Pocket** established the need for resource-awareness and **Rocket** provided the data-mover-aware substrate, **SkyRocket** provides the **dynamic intelligence** required to orchestrate these capabilities. It proves that a “one-size-fits-all” approach is no longer viable for modern data-centric computing and

that the IPC layer itself must possess the reasoning capacity to bridge the gap between high-level workload patterns and low-level hardware characteristics.

As the concluding technical contribution of this dissertation, **SkyRocket** synthesizes the principles of resource-awareness, hardware-integration, and workload-adaptation into a single unified framework. Together with the preceding chapters, it completes the transition of Inter-Process Communication from a passive data conduit into a proactive, intelligent manager of system-wide dataflows, setting a new standard for the design of high-performance, modular system architectures.

## **CHAPTER 6**

### **RELATED WORK**

Prior work on optimizing IPC for data-intensive workloads has largely followed three directions: software-level runtime optimization, hardware-assisted data movement, and heuristic-based control. These approaches have reduced communication overhead, but they typically address resource management, hardware offloading, and runtime adaptation in isolation.

This chapter reviews these three bodies of work to position our contributions. We compare Pocket with existing container isolation mechanisms, distinguish Rocket from static hardware offloading abstractions, and place SkyRocket within the literature on ML-for-systems as a learned control plane that enables coordinated adaptation across the system stack.

#### **6.1 Sharing, Amplification, and Transient Use of Resources**

This section situates Pocket within prior work on improving the efficiency of containerized and virtualized workloads. We review existing approaches to sharing execution state, amplifying resources on demand, and managing transient resource use, and clarify how Pocket departs from these designs to address the demands of ML serving in resource-constrained environments.

##### 6.1.1 Efficient Container Runtimes

A substantial body of recent work focuses on improving the resource footprint, performance, and responsiveness of containerized workloads through reduced startup latency, improved memory sharing, runtime specialization, or workload partitioning.

**Sharing warm read-only state.** Several systems reduce container startup latency by reusing pre-initialized execution state. SOCK [104] enables fast container launch by restoring containers from snapshots of pre-initialized images. While this approach significantly shortens startup time, each container instance still maintains its own runtime, and runtime memory overhead during steady-state execution remains largely unchanged.

Related approaches, including AWS Layers [105] and Catalyzer [106], promote sharing of read-only state through memory or image file overlays [107]. These techniques, particularly when combined with warm restart, further reduce launch time compared to monolithic container images. However, they require explicit separation of shared and custom state and additional program restructuring. More fundamentally, regardless of the extent of memory sharing, these systems retain a distinct runtime instance for each application instance.

Pocket differs in a fundamental way. Rather than focusing solely on startup latency, Pocket enables sharing of the runtime itself, offering warm shared runtime. As a result, Pocket targets persistent execution-time resource overheads, while remaining compatible with complementary warm-start mechanisms.

**Leveraging language runtimes.** Another line of work exploits language-level mechanisms in managed runtimes to improve efficiency across multiple instances. Catalyzer [106] and related systems [108] rely on runtime-specific techniques to enable sharing and optimization. While effective within their scope, such approaches are inherently limited to specific programming languages.

Other work leverages compile-time optimization to reuse cached results of static functions across executions [109]. These techniques enable a limited form of shared execution but rely on statically compiled binaries and do not support concurrent sharing across independent invocations.

Pocket addresses the same efficiency challenge through a more dynamic approach. It is not tied to a specific programming language and does not rely solely on static compilation.

This design choice allows Pocket to support heterogeneous, multi-language ML workloads that are common in containerized serving environments.

**Partitioning workloads.** Cntr [110] proposes splitting a container into a slim image that contains the application and a fat image that includes auxiliary tools for debugging, auditing, and maintenance. This approach reduces container image size while preserving development flexibility.

Although Pocket also adopts a partitioned design, it operates in the opposite direction. Pocket attaches minimal frontend containers to a shared backend runtime service. This backend is placed directly on the execution critical path and is central to resource management and performance. In contrast, Cntr’s fat containers remain off the critical path, and their resource usage and performance are not primary concerns of the system design.

**Reducing system overheads.** Some systems target isolation overheads at the system layer. Firecracker reduces virtualization overhead to enable lightweight microVMs and appears promising for resource-constrained deployments.

However, experimental evidence shows that for complex ML runtimes such as TensorFlow, Firecracker does not significantly reduce overall resource usage compared to traditional containers. In these workloads, the dominant resource consumption occurs at the application layer rather than the system layer. This observation motivates Pocket’s focus on application- and runtime-level sharing rather than isolation mechanisms alone.

### 6.1.2 Resource Amplification: Academic Roots

A central contribution of Pocket is its use of resource amplification. While the mechanism itself is new in this context, it draws inspiration from earlier systems research that explored amplification under different assumptions and trade-offs.

**Amplification.** Prior work employed amplification to improve efficiency or to balance performance and security. The concept of temporarily borrowing and releasing resources to enhance system flexibility was pioneered by Exokernel [111], which introduced revocable resource allocation to provide applications with direct hardware control. This lineage of dynamic allocation was further developed for multi-tenant environments in systems such as Merlin [112], Cellular Disco [113], and Barrelfish [114]. Other forms of amplification include Active Messages [115], which streamlined communication by inlining service invocations into messages, and Hydra [116], which introduced capability amplification to temporarily elevate privileges during remote procedure calls to trade off security guarantees for performance.

Pocket reinterprets these ideas for modern ML serving. It introduces resource amplification as a mechanism that temporarily increases available resources at invocation time. To the best of our knowledge, Pocket is the first system to implement resource amplification in this form for ML workloads, enabling efficiency gains without permanently increasing resource reservations.

## **6.2 Data Movement Across Hardware, OS, and IPC Layers**

This section reviews prior work on hardware-assisted data movement and copy offloading, with a focus on mechanisms that reduce CPU involvement in memory transfers. While modern systems already provide a range of data movers at the hardware and kernel levels, existing abstractions fall short of integrating these capabilities into latency-sensitive, intra-node inter-process communication (IPC). Rocket is motivated by this gap and targets the missing software layer between offload-capable engines and application-level dataflows.

### 6.2.1 A Landscape of Data Movers Across the Stack

Recent systems expose multiple data-movement mechanisms at different layers of the stack. Although they differ in design and scope, they share the common goal of offloading

memory copies from the CPU.

**On-chip memory copy acceleration.** Intel DSA [26] represents a modern on-chip accelerator for general-purpose memory operations. It enables bulk data movement without consuming CPU cycles on the copy path and exposes low-overhead submission mechanisms to software. As an on-chip accelerator, DSA offers high throughput but requires careful coordination to maintain cache coherence and memory ordering.

**Memory-controller-level copy deferral.**  $MC^2$  [25] introduces a lazy `memcpy` mechanism that defers copying until the destination is accessed. By performing copy operations within the memory controller,  $MC^2$  reduces CPU stalls transparently. However, this transparency limits fine-grained software control and can introduce latency variability for large transfers when deferred copies are triggered.

**Hardware engines for heterogeneous data motion.** DMX [90] accelerates data restructuring and chaining across accelerators, reducing CPU orchestration overhead in accelerator pipelines. While effective in its target domain, DMX follows a hardware-centric design and does not address how software runtimes should dynamically coordinate offload decisions for IPC.

**Traditional DMA and network offload.** Intel I/O Acceleration Technology (I/OAT) [27] explored early DMA-based copy offloading, including IPC-oriented designs such as ADCE [29]. RDMA-capable NICs [117] have since become the standard solution for inter-node communication offload. These approaches demonstrate that offloading is well established for specific transfer domains, yet they also highlight that intra-node IPC lacks a comparable, flexible integration layer for modern on-chip movers.

**Kernel-level copy as a coordinated service.** Copier [24] elevates data movement to a first-class kernel service by providing asynchronous copy execution and system-wide coor-

dination. XPC [118] offers kernel-level IPC primitives. While complementary to hardware movers, kernel mediation introduces overheads that can conflict with the responsiveness requirements of fine-grained IPC.

## 6.2.2 Empirical Studies and System Adoption of Intel DSA

A growing body of work has empirically characterized Intel DSA and explored its adoption in system-level mechanisms that depend on efficient memory movement.

**Microarchitectural characterization of DSA.** A foundational study [28] provides a detailed analysis of DSA’s execution model, work queues, and descriptor batching. The study reports speedups of 1.7–3.0x for intra-socket transfers and 3.5–4.5x for inter-socket transfers compared to single-core CPU copies. It further shows that two DSAs suffice in most scenarios unless both source and destination reside in high-bandwidth memory. Importantly, DSA transfers do not interfere with compute-intensive or scalar workloads, while interference arises for data-intensive memory operations.

**DSA from the perspective of in-memory data processing.** The work by Berthold et al. [87] builds on prior characterization results and evaluates DSA as a practical accelerator for in-memory workloads. The study demonstrates that DSA can both improve performance and free CPU cores for other computation. It further argues that DSA is broadly applicable to software systems that frequently copy medium- to large-sized memory regions, including inter-process communication when zero-copy mechanisms cannot be applied.

**System mechanisms adopting DSA.** DSA has been incorporated into concrete system designs. Para-ksm [119] accelerates memory deduplication by parallelizing copy-intensive operations using DSA. DSA-2LM [120] leverages DSA to manage data movement in tiered-memory systems with minimal CPU involvement. Together with prior empirical

studies [28, 87], these systems establish DSA as a deployable building block for real systems rather than a purely experimental accelerator.

Rocket is informed by this body of work but addresses a different integration boundary. Rather than embedding DSA into a specific system service, Rocket targets intra-node IPC, where data movement must be integrated into application-level communication paths under strict responsiveness constraints.

### 6.2.3 Limitations of Existing Offloading Abstractions

Despite the availability of capable data movers, existing software abstractions do not adequately support their use in IPC.

**Static adoption through transparent interception.** DTO [92] enables DSA offloading by transparently intercepting `glibc memcpy` calls. This design represents a static adoption of DSA as a drop-in replacement for CPU copies. However, DTO remains largely synchronous and does not provide a programmable interface for selectively choosing when offload is appropriate.

**Deployment constraints of legacy DMA-based IPC.** ADCE [29] demonstrates the feasibility of DMA-based copy offloading but depends on kernel-mode drivers [27] and lacks support for modern virtualization features such as IOMMU and VT-d. These constraints limit applicability in cloud and multi-tenant environments and motivate the use of modern on-chip accelerators such as DSA.

**CPU-centric IPC designs.** High-performance IPC systems such as Nightcore [67] remain fundamentally CPU-centric on the data movement path. They optimize CPU usage but treat copying as an unavoidable cost rather than a selectable operation. Rocket departs from this assumption by treating data movement as a controllable component of IPC.

#### 6.2.4 Workloads Motivating Efficient Intra-Node Data Movement

**Evidence from data-intensive pipelines.** Plumber [75] identifies data handling and IPC as recurring bottlenecks in machine learning pipelines, showing that copy overheads can dominate end-to-end performance once pipelines scale. These findings motivate Rocket’s focus on reducing CPU copy pressure and integrating offload decisions directly into IPC paths.

#### 6.2.5 Synthesis

Prior work shows that multiple data movers exist across hardware and the operating system, and that DSA has been empirically characterized and increasingly adopted in real systems [87]. However, these efforts adopt offload either statically or at coarse granularity, without integrating it into IPC semantics.

**Positioning of Rocket.** Rocket addresses the missing software layer between offload-capable engines and application-level IPC. Rather than proposing another data mover, Rocket focuses on integration, coordination, and responsiveness, enabling principled use of hardware-assisted data movement within intra-node IPC.

### **6.3 Runtime Adaptation for Heterogeneous Data Movement**

An intra-node IPC system operates under rapidly changing hardware conditions and exposes a wide range of data-movement choices. CPU copies, on-chip accelerators such as DSA, legacy DMA engines, and shared-memory mechanisms each exhibit distinct performance characteristics. Selecting among these options at runtime is no longer a matter of fixed thresholds or simple heuristics. This section reviews prior work that motivates SkyRocket’s design, moving from the limits of manual configuration to the need for learned control planes, and finally to the challenge of making learning viable on the IPC critical path.

### 6.3.1 The Intractability of Manual IPC Configuration

**Explosion of data-movement choices.** Modern IPC is no longer a single-path operation. Systems must choose among CPU-based copying, accelerator-assisted transfers such as Intel DSA [89], legacy DMA engines including I/OAT [27], and cache-sensitive shared-memory paths. Each option trades off latency, CPU utilization, cache pollution, and memory bandwidth pressure. Prior characterization work on DSA demonstrates that these trade-offs vary significantly with transfer size, socket locality, and concurrent memory activity [28]. As a result, no single mechanism dominates across all conditions.

**Non-linearity and cross-resource interference.** The performance landscape of IPC is highly non-linear. Small changes in input size or access pattern can trigger cascading effects on last-level cache occupancy, memory bandwidth contention, and accelerator queuing delays [121, 122]. Empirical studies of accelerator-based data movement show that performance can shift abruptly depending on whether workloads are compute-intensive or data-intensive, and whether they interfere at shared resources [28, 123, 124, 125]. These interactions make it difficult to predict performance outcomes using simple rules.

**Limits of static heuristics.** Despite this complexity, many existing systems rely on static heuristics [126]. Standard libraries such as `glibc select memcopy` implementations using fixed size thresholds that ignore runtime hardware state. Similar threshold-based decisions appear throughout systems software [127]. While such heuristics are easy to deploy, they implicitly assume that performance relationships are stable. In practice, IPC workloads violate this assumption at per-request granularity, especially under noisy-neighbor effects and fluctuating cache pressure. This makes manual tuning brittle and fundamentally intractable.

### 6.3.2 The Case for Learned Control Planes

**Learning as a systems abstraction.** Systems research has increasingly adopted learned models to replace rigid heuristics in complex decision-making paths. Learned index structures [128] demonstrate that data access decisions can be learned rather than hard-coded. Resource management systems such as Paragon [129] and Quasar [130] use machine learning to reason about interference and placement in large-scale clusters. Configuration auto-tuning frameworks such as OpenTuner [131] further show that learning-based search can navigate large, irregular configuration spaces more effectively than human-designed rules. These successes suggest a clear direction for IPC. Selecting a data-movement path is fundamentally a prediction problem over a complex, non-linear space. A learned control plane can adapt to shifting conditions and automatically capture interactions that are difficult to encode manually. However, IPC imposes tighter latency constraints that distinguish it from prior ML-for-systems work. Decisions must be made at microsecond granularity, often on the critical path of application execution. This raises a key challenge: learning must be fast enough to not negate its own benefits.

### 6.3.3 The Latency Requirement in IPC

**Decision latency and Workloads.** Many ML-driven systems make decisions at coarse time scales. Cluster schedulers and interference-aware placement systems operate on intervals of seconds or minutes [129, 130]. Even when inference overhead is amortized, such timescales are incompatible with IPC, where decisions must be made per request and directly affect tail latency.

**Inference overhead on the critical path.** Deep learning models exacerbate this problem. Neural-network inference introduces overheads of a millisecond to second scale that are non-negligible at microsecond scales [132]. In IPC, where the cost of a copy operation itself may be only a few microseconds, inserting a heavy inference step can result in a net

loss. This creates a “control-path bottleneck,” where the mechanism intended to optimize performance instead dominates it.

**Mismatch between model complexity and IPC needs.** The mismatch is not conceptual but practical. IPC requires models that can reason about non-linear interactions while executing with minimal instruction overhead and predictable latency. Existing ML-based systems were not designed with such tight constraints in mind, leaving a gap between the promise of learned control and the realities of IPC execution.

#### 6.3.4 SkyRocket: Lightweight Gradient Boosting on the Critical Path

**Why gradient-boosted decision trees.** SkyRocket addresses the latency barrier by adopting gradient-boosted decision trees, implemented using LightGBM (LGBM) [133]. Decision trees naturally capture non-linear interactions among features, making them well suited to the complex performance landscape of IPC. At the same time, their structure allows inference to be compiled into a sequence of simple conditional branches, avoiding the overhead of matrix operations or iterative optimization. Because tree-based models reduce to a small number of comparisons [134], they can execute efficiently on the CPU pipeline and fit directly on the IPC critical path. This property places LGBM in a sweet spot between expressiveness and speed. SkyRocket leverages this characteristic to make per-request decisions without introducing measurable control overhead.

**Positioning within ML-for-systems.** Prior work has explored decision trees and boosted models in latency-sensitive domains such as packet classification and scheduling [135]. SkyRocket extends this line of thinking to intra-node IPC, demonstrating that learned control is feasible even under microsecond-level constraints. By combining lightweight models with runtime data-movement adaptation, SkyRocket shows that learning can replace brittle heuristics without sacrificing responsiveness.

## CHAPTER 7

### CONCLUSION

#### 7.1 Dissertation Summary: Reimagining IPC for the Data-Centric Era

This dissertation establishes that modern computing now hinges on data movement rather than computation, and it demonstrates why this shift demands a fundamental rethinking of inter-process communication. Data-intensive workloads—most notably end-to-end machine learning pipelines—expose the limits of traditional IPC designs. These designs rely on small, CPU-mediated message passing and static resource management, and they no longer sustain the scale or complexity of contemporary data-centric systems.

Revisiting this problem foregrounds the central insight of this work: IPC must evolve from a passive conduit into a resource-aware, hardware-aware, and workload-adaptive runtime substrate. This dissertation demonstrates this evolution through the Pocket, Rocket, and SkyRocket systems. Each system addresses an escalating bottleneck in the data path, and together they form a coherent trajectory toward efficient data-centric computing. Their progression shows that overcoming IPC bottlenecks is not a sequence of isolated optimizations but a unified effort to redesign runtime communication around the realities of modern hardware and emerging workloads.

#### 7.2 Synthesis of Contributions

This dissertation demonstrates that Pocket, Rocket, and SkyRocket collectively mark a staged evolution toward intelligent and efficient IPC. Their contributions are not isolated results. They form a coherent progression in which IPC gradually expands its role from a passive message conduit, to an active resource manager, to a hardware-adaptive substrate capable of orchestrating large-scale data movement. This synthesis highlights how each

system resolves a specific bottleneck while enabling the next stage of capability.

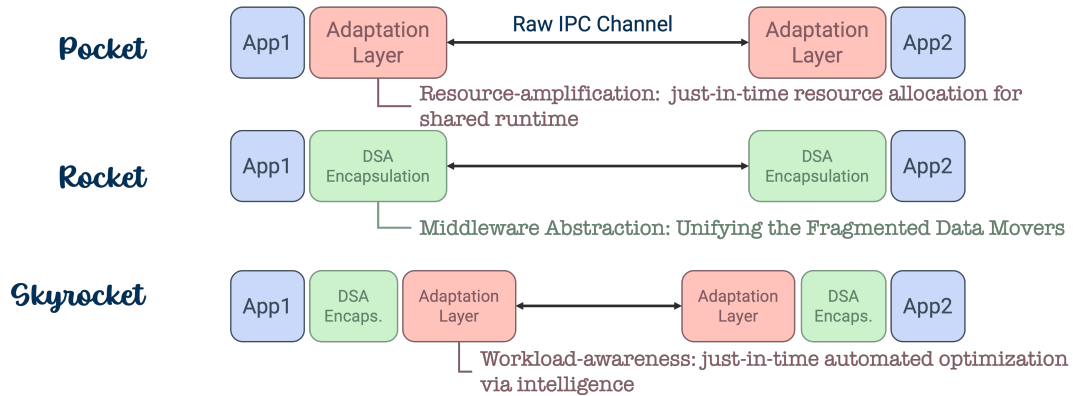


Figure 7.1: Design blueprint of Skyrocket.

### 7.2.1 IPC with Resource Management: Pocket

This dissertation establishes that Pocket resolves the resource overhead and isolation challenges inherent to split-architecture edge ML environments. Pocket introduces inline resource amplification, a mechanism that embeds execution-context authority directly into IPC messages. By attaching resource control semantics to the data path, Pocket demonstrates that IPC can dynamically reassign system resources rather than merely transport payloads. This shift reframes IPC as an active resource manager and sets the conceptual foundation for Rocket and SkyRocket, which build on this capability to address increasingly demanding data-centric workloads.

### 7.2.2 IPC with Unified Coordination for Heterogeneous Data Movement: Rocket

Rocket advances the evolution of IPC by establishing it as a unified coordinator for heterogeneous data-movement backends. This dissertation demonstrates that alleviating data-movement bottlenecks requires integrating not only hardware accelerators such as Intel DSA, but also kernel-level memory services like Copier and optimized software routines. Rocket abstracts these diverse backends behind a single IPC interface, allowing the runtime to select the most effective mechanism without exposing their complexity to applications.

This unification ensures that IPC remains the central control point for orchestrating data movement across heterogeneous engines.

Rocket goes beyond simple offloading. It introduces a coordinated IPC runtime that consistently manages backend-specific characteristics such as asynchronous execution behavior, cache-bypass modes, and kernel–user transition costs. By controlling these properties within the software stack, Rocket provides the mechanism needed for applications to exploit a wide range of copy engines without being tied to any particular implementation. This contribution establishes the physical foundation on which SkyRocket’s adaptive policies operate and extends the capabilities enabled by Pocket’s resource-aware design.

### 7.2.3 IPC with Adaptive Intelligence: SkyRocket

SkyRocket completes the evolutionary arc of this dissertation by establishing IPC as an adaptive decision-making system. Building on Rocket’s hardware-coordinated mechanisms, SkyRocket resolves the core challenge of selecting the optimal execution path among many alternatives—CPU vs. DSA, synchronous vs. asynchronous execution, and cache-injection vs. non-injection modes. These choices depend on dynamic runtime conditions such as data size, contention, and pipeline configuration, and the complexity of navigating them exceeds what static heuristics can provide.

This dissertation demonstrates that a lightweight learned model can serve as an effective adaptive control plane for IPC. SkyRocket employs this model to revise communication strategies in real time, ensuring that each data-movement operation aligns with current system conditions. By doing so, SkyRocket adds the “intelligence layer” that Pocket and Rocket intentionally set the stage for: Pocket enabled resource-aware IPC; Rocket enabled hardware-aware IPC; and SkyRocket unifies these capabilities into a workload-adaptive runtime. Together, they produce an end-to-end system that embodies the central thesis of this work—IPC must evolve into an intelligent, adaptive substrate for efficient data-centric computing.

### 7.3 Key Insights and Broader Implications

This dissertation establishes several design principles that extend beyond the individual systems developed in this work. These insights clarify how future communication substrates must evolve to support data-centric computing at scale.

**The Breakdown of the Black-Box Abstraction.** This dissertation demonstrates that the long-standing practice of treating IPC as a black box (reduced to `send()` and `recv()` semantics) is fundamentally misaligned with data-centric workloads. Efficiency now depends on exposing hardware characteristics and data properties such as modality and size to the IPC layer. When IPC is aware of these attributes, the runtime can select communication paths that better exploit hardware capabilities and reduce unnecessary data movement. The Pocket–Rocket–SkyRocket progression shows that once the abstraction boundary becomes permeable, IPC can act with precision rather than generality, yielding substantial performance and efficiency gains.

**Co-design of Communication and Resource Management.** We establish that communication and resource management cannot remain decoupled components of the system stack. Treating them as independent responsibilities leads to inflated overheads and unpredictable latency in dense and heterogeneous workloads. Pocket demonstrates that unifying these concerns enables IPC to dynamically allocate resources along the data path, producing low-latency and high-throughput operation even under load. This co-design principle underpins Rocket’s hardware coordination and SkyRocket’s adaptive policies, showing that integrated control is a prerequisite for efficient data-centric systems.

**Adaptivity Is Essential, Not Optional.** This dissertation demonstrates that fixed heuristics and static policies break down in environments marked by rapidly changing system states and heterogeneous hardware. As accelerators proliferate and data pipelines grow in scale, the system must continuously revise its communication strategy to reflect current conditions. SkyRocket’s adaptive control plane embodies this requirement by learning

from runtime signals and selecting the most efficient execution path on demand. Adaptivity therefore emerges not as an optimization but as a foundational requirement for next-generation IPC systems and, more broadly, for efficient data-centric computing.

## 7.4 Future Directions

This dissertation establishes the foundations of intelligent, resource-aware, and hardware-coordinated IPC. At the same time, it acknowledges several limitations that open meaningful avenues for future research. These directions extend the ideas developed in Pocket, Rocket, and SkyRocket to broader and more complex system environments.

**Scope of Heterogeneity.** This work concentrates on intra-node scenarios and evaluates heterogeneity primarily through a single accelerator class, such as Intel DSA. Future systems, however, will rely on far more diverse hardware substrates. Extending the proposed IPC mechanisms to CXL-based memory pooling, GPU-direct communication, and other disaggregated architectures will be essential for validating their generality. These environments introduce unique latency characteristics, shared-memory topologies, and failure modes that an intelligent IPC runtime must learn to navigate.

**Distributed Systems.** Pocket’s resource propagation and SkyRocket’s adaptive control demonstrate clear benefits within a single machine. Extending these mechanisms to inter-node settings raises significant challenges. Network latency variability, distributed state synchronization, and multi-node resource arbitration complicate the semantics of resource-aware and adaptive IPC. A future research direction is to develop communication substrates that preserve the core guarantees established in this dissertation even when data moves across network fabrics. Achieving this goal would unify intra-node and distributed communication under a single adaptive IPC framework.

**Complexity of Learning Models.** SkyRocket shows that lightweight learned models can effectively guide IPC decisions in dynamic environments. Yet more intricate workload patterns and long-horizon dependencies may exceed the representational capacity of such

models. Reinforcement-learning-based IPC controllers represent a compelling extension. These controllers could acquire policies through interaction, adapt autonomously to evolving system conditions, and coordinate multiple heterogeneous data-movement engines simultaneously. Advancing this line of work would deepen the role of intelligence within IPC and extend the adaptive capabilities established in this dissertation.

## **7.5 Concluding Remarks**

This dissertation demonstrates that modern data-centric applications demand communication substrates capable of managing data movement with far greater speed, autonomy, and intelligence than traditional IPC can deliver. In response to this need, the dissertation presents concrete architectures and methodologies that transform IPC from a passive conduit into an active, intelligent manager of data movement. Pocket, Rocket, and Sky-Rocket together offer a practical blueprint for the co-evolution of hardware and software in next-generation systems. Their progression establishes that intelligent, resource-aware, and hardware-coordinated IPC is not an incremental enhancement but a foundational requirement for efficient data-centric computing.

## REFERENCES

- [1] NVIDIA Corporation, *CUDA c++ programming guide*, <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>, Version 12.x, Accessed: 2025-12-21, 2025.
- [2] Advanced Micro Devices, Inc. (AMD), *ROCm documentation*, <https://rocm.docs.amd.com/>, Version 6.x, Accessed: 2025-12-21, 2025.
- [3] Google Brain Team, *Tensorflow: An end-to-end open source machine learning platform*, Accessed: 2025-05-16, 2025.
- [4] PyTorch Team, *Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration*, Accessed: 2025-05-16, 2025.
- [5] M. Park, K. Bhardwaj, and A. Gavrilovska, “Pocket: MI serving from the edge,” in *Proceedings of the Eighteenth European Conference on Computer Systems*, ser. EuroSys ’23, Rome, Italy: Association for Computing Machinery, 2023, pp. 46–62, ISBN: 9781450394871.
- [6] S. Karandikar, C. Leary, C. Kennelly, J. Zhao, D. Parimi, B. Nikolic, K. Asanovic, and P. Ranganathan, “A hardware accelerator for protocol buffers,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’21, Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 462–478, ISBN: 9781450385572.
- [7] S. Nayak, V. Rangwani, K. Dubey, R. Mondal, T. Gupta, and R. Shah, “Poster: Reducing data movement tax for serialization in microservices,” in *Proceedings of the 20th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT ’24, Los Angeles, CA, USA: Association for Computing Machinery, 2024, pp. 17–18, ISBN: 9798400711084.
- [8] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, “Firecracker: Lightweight virtualization for serverless applications,” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, USENIX Association, 2020, pp. 419–434.
- [9] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, 2010, Comprehensive guide to POSIX IPC (`shmopen`, `mqopen`, `semopen`).
- [10] W. R. Stevens and S. A. Rago, *Advanced Programming in the UNIX Environment*, 3rd. Addison-Wesley Professional, 2013, Covers System V IPC mechanisms: shared memory, semaphores, and message queues.

- [11] ZeroMQ Development Team, *Zeromq: An open-source universal messaging library*, <https://zeromq.org/>, Accessed: 2025-12-21, 2025.
- [12] J. Jackson, *Return of the monolith: Amazon dumps microservices for video monitoring*, <https://thenewstack.io/return-of-the-monolith-amazon-dumps-microservices-for-video-monitoring/>, The New Stack, Accessed: 2025-12-22, 2023.
- [13] K. Bayouhd, R. Knani, F. Hamdaoui, and A. Mtibaa, “A survey on deep multimodal learning for computer vision: Advances, trends, applications, and datasets,” *The Visual Computer*, vol. 38, no. 8, pp. 2939–2970, Aug. 2022.
- [14] Z. Liu, S. Cheng, G. Tan, Y. You, and D. Tao, *Elasticmm: Efficient multimodal llms serving with elastic multimodal parallelism*, 2025. arXiv: 2507.10069 [cs.DC].
- [15] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinisky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, “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*, ser. ASPLOS ’19, Providence, RI, USA: Association for Computing Machinery, 2019, pp. 3–18, ISBN: 9781450362405.
- [16] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Foo, Z. Haque, T. Hayakal, M. Ispir, V. Jain, L. Koc, et al., “TFX: A TensorFlow-based production-scale machine learning platform,” in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 1387–1395.
- [17] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, “Flashattention: Fast and memory-efficient exact attention with io-awareness,” in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, 2022, pp. 16 344–16 359.
- [18] T. Dao, “Flashattention-2: Faster attention with better parallelism and work partitioning,” in *International Conference on Learning Representations (ICLR)*, 2024.
- [19] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, *Accurate, large minibatch sgd: Training imagenet in 1 hour*, 2018. arXiv: 1706.02677 [cs.CV].
- [20] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [21] UHD Alliance, *Understanding 4k: Technical overview of 4k ultra hd*, <https://uhdalliance.org/>, Accessed: 2024-04-10, 2020.

- [22] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The hadoop distributed file system,” in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, IEEE, 2010, pp. 1–10.
- [23] W. R. Lionheart, *An mri dicom data set of the head of a normal male human aged 52*, Zenodo, 2015.
- [24] J. He, Y. Dong, D. Du, M. Zou, Z. Yu, Y. Ren, N. Jia, Y. Xia, and H. Chen, “How to copy memory? coordinated asynchronous copy as a first-class os service,” in *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, ser. SOSP ’25, Lotte Hotel World, Seoul, Republic of Korea: Association for Computing Machinery, 2025, pp. 1062–1081, ISBN: 9798400718700.
- [25] A. K. Kamath and S. Peter, “(mc)2: Lazy memcopy at the memory controller,” in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 1112–1128.
- [26] Intel Corporation, *Intel® data streaming accelerator (intel® dsa)*, Accessed: 2025-05-16, 2025.
- [27] J. Corbet, “Memory copies in hardware,” *LWN.net*, 2005, Accessed: 2024-02-23.
- [28] R. Kuper, I. Jeong, Y. Yuan, R. Wang, N. Ranganathan, N. Rao, J. Hu, S. Kumar, P. Lantz, and N. S. Kim, “A quantitative analysis and guidelines of data streaming accelerator in modern intel xeon scalable processors,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS ’24, La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 37–54, ISBN: 9798400703850.
- [29] K. Vaidyanathan, W. Huang, L. Chai, and D. K. Panda, “Designing efficient asynchronous memory operations using hardware copy engine: A case study with i/oat,” in *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007, pp. 1–8.
- [30] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, and I. Stoica, “Ray: A distributed framework for emerging AI applications,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 561–577.
- [31] A. Kalia, M. Kaminsky, and D. G. Andersen, “Datacenter RPCs can be Thousand Times Faster,” in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019, pp. 199–212.
- [32] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Richter, M. Gallina, et al., “An open-source benchmark suite for microservices and

their hardware-software implications,” in *24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 441–457.

- [33] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafae, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, “Clearing the clouds: A study of emerging scale-out workloads on modern hardware,” in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII, London, England, UK: Association for Computing Machinery, 2012, pp. 37–48, ISBN: 9781450307598.
- [34] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, “Profiling a warehouse-scale computer,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA ’15, Portland, Oregon: Association for Computing Machinery, 2015, pp. 158–169, ISBN: 9781450334020.
- [35] W. A. Wulf and S. A. McKee, “Hitting the memory wall: Implications of the obvious,” *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.
- [36] S. D. Fu, H. Zhang, R. Teoh, T. Priadka, and S. Ratnasamy, “Toward data-centric service composition,” in *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*, ser. HotNets ’24, Irvine, CA, USA: Association for Computing Machinery, 2024, pp. 360–367, ISBN: 9798400712722.
- [37] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” 2016, pp. 770–778.
- [38] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” *arXiv preprint arXiv:1704.04861*, 2017, <https://arxiv.org/abs/1704.04861>.
- [39] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.
- [40] Y. Tang, Y. He, V. Nath, P. Guo, R. Deng, T. Yao, Q. Liu, C. Cui, M. Yin, Z. Xu, H. Roth, D. Xu, H. Yang, and Y. Huo, “Holohisto: End-to-end gigapixel wsi segmentation with 4k resolution sequential tokenization,” *CoRR*, vol. abs/2407.03307, 2024.
- [41] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, “Edge computing: Vision and challenges,” *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

- [42] H. Liu, F. Eldarrat, H. Alqahtani, A. Reznik, X. de Foy, and Y. Zhang, “Mobile Edge Cloud System: Architectures, Challenges, and Approaches,” *IEEE Systems Journal*, vol. 12, no. 3, pp. 2495–2508, Sep. 2018.
- [43] C. Nguyen, A. Mehta, C. Klein, and E. Elmroth, “Why cloud applications are not ready for the edge (yet),” in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, ser. SEC ’19, Arlington, Virginia: Association for Computing Machinery, Nov. 2019, pp. 250–263, ISBN: 978-1-4503-6733-2.
- [44] K.-J. Hsu, K. Bhardwaj, and A. Gavrilovska, “Couper: DNN Model Slicing for Visual Analytics Containers at the Edge,” in *Proceedings of the 4th International ACM/IEEE Symposium on Edge Computing (SEC’19)*, Washington DC., 2019.
- [45] *Docker*, <https://www.docker.com/>.
- [46] *Kubernetes*, <https://kubernetes.io>.
- [47] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, “Firecracker: Lightweight virtualization for serverless applications,” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434, ISBN: 978-1-939133-13-7.
- [48] *Akraino Edge Stack*, <https://www.lfedge.org/projects/akraino/>, 2018.
- [49] *LF EDGE: Building an Open Source Framework for the Edge*, <https://www.lfedge.org>.
- [50] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo, “Adaptive resource efficient microservice deployment in cloud-edge continuum,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, pp. 1825–1840, 2022.
- [51] K. Bhardwaj, M.-W. Shih, P. Agarwal, A. Gavrilovska, T. Kim, and K. Schwan, “Fast, Scalable and Secure Onloading of Edge Functions Using AirBox,” in *1st IEEE/ACM Symposium on Edge Computing (SEC’16)*, Washington DC, 2016.
- [52] A. Albanese, P. Crosta, C. Meani, and P. Paglierani, “GPU-accelerated Video Transcoding Unit for Multi-access Edge Computing Scenarios,” in *The Sixteenth International Conference on Networks (ICN’17)*, Apr. 2017.
- [53] S. Biokaghazadeh, M. Zhao, and F. Ren, “Are FPGAs Suitable for Edge Computing?” In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, Boston, MA: USENIX Association, Jul. 2018.

- [54] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, “Real-Time Video Analytics: The Killer App for Edge Computing,” *Computer*, vol. 50, no. 10, pp. 58–67, 2017.
- [55] *The NVIDIA EGX Platform for Edge Computing*, <https://www.nvidia.com/en-us/data-center/products/egx-edge-computing/>.
- [56] Z. Zhang, *Zzh8829/yolov3-tf2*, <https://github.com/zzh8829/yolov3-tf2>, original-date: 2019-04-03T17:57:49Z, Feb. 2020.
- [57] J. Redmon and A. Farhadi, “Yolov3: An incremental improvement,” *arXiv preprint arXiv:1804.02767*, 2018.
- [58] *5G vs 4G LTE: what are the differences?* <https://blog.antenova.com/5g-vs-4g-lte-what-are-the-differences>.
- [59] S. George, J. Wang, M. Bala, T. Eiszler, P. Pillai, and M. Satyanarayanan, “Towards drone-sourced live video analytics for the construction industry,” in *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '19, Santa Cruz, CA, USA: ACM, 2019, pp. 3–8, ISBN: 978-1-4503-6273-3.
- [60] P. Patel, M. Intizar Ali, and A. Sheth, “On using the intelligent edge for iot analytics,” *IEEE Intelligent Systems*, vol. 32, no. 5, pp. 64–69, 2017.
- [61] M. Huzaifa, R. Desai, S. Grayson, X. Jiang, Y. Jing, J. Lee, F. Lu, Y. Pang, J. Ravichandran, F. Sinclair, B. Tian, H. Yuan, J. Zhang, and S. V. Adve, *Exploring extended reality with illixr: A new playground for architecture research*, 2021. arXiv: 2004.04643 [cs.DC].
- [62] J. Heo, C. Phillips, and A. Gavrilovska, “FLiCR: A Fast and Lightweight LiDAR Point Cloud Compression Based on Lossy RI,” in *ACM Symposium on Edge Computing (SEC'22)*, Seattle, WA, 2022.
- [63] *Chick-fil-A*, <https://www.redhat.com/architect/cloud-to-edge>.
- [64] *VaporIO*, <https://www.vapor.io/>.
- [65] *Types and Locations of Edge Data Centers*: <https://www.missioncriticalmagazine.com/ext/resources/whitepapers/white-papers-2/TIA-White-Paper-Types-and-Locations-of-Edge-Data-Centers.pdf>.
- [66] A. Perdanaputra and A. I. Kistijantoro, “Transparent tracing system on grpc based microservice applications running on kubernetes,” in *2020 7th International Con-*

*ference on Advance Informatics: Concepts, Theory and Applications (ICAICTA)*, 2020, pp. 1–5.

- [67] Z. Jia and E. 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)*, 2021, pp. 152–166.
- [68] gRPC Authors, *Grpc on http/2 — engineering a robust, high-performance protocol*, <https://grpc.io/blog/grpc-on-http2>, Explains how gRPC builds on HTTP/2’s long-lived connections and multiplexing, 2018.
- [69] Microsoft Learn Documentation, *Performance best practices with grpc*, <https://learn.microsoft.com/en-us/aspnet/core/grpc/performance?view=aspnetcore-10.0>, Official guidance on gRPC channel reuse and HTTP/2 connection management, 2025.
- [70] TensorFlow Authors, *Module: Tf.config.threading*, [https://www.tensorflow.org/api\\_docs/python/tf/config/threading](https://www.tensorflow.org/api_docs/python/tf/config/threading), TensorFlow API Documentation (v2.x). Accessed: 2026-01-09, n.d.
- [71] *Namespaces(7) linux user’s manual*, Oct. 2021.
- [72] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, “Lessons Learned from the Chameleon Testbed,” in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC ’20)*, USENIX Association, Jul. 2020.
- [73] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, “GVIM: GPU-Accelerated Virtual Machines,” in *Proceedings of the 3rd ACM Workshop on System-Level Virtualization for High Performance Computing*, ser. HPCVirt ’09, Nuremberg, Germany: Association for Computing Machinery, 2009, pp. 17–24, ISBN: 9781605584652.
- [74] R. Bachkaniwala, H. Lanka, K. Rong, and A. Gavrilovska, “Lotus: Characterization of machine learning preprocessing pipelines via framework and hardware profiling,” in *2024 IEEE International Symposium on Workload Characterization (IISWC)*, 2024, pp. 30–43.
- [75] M. Kuchnik, A. Klimovic, J. Simsa, V. Smith, and G. Amvrosiadis, “Plumber: Diagnosing and removing performance bottlenecks in machine learning data pipelines,” in *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu, Eds., vol. 4, 2022, pp. 33–51.

- [76] A. Audibert, Y. Chen, D. Graur, A. Klimovic, J. Šimša, and C. A. Thekkath, “Tf.data service: A case for disaggregating ml input data processing,” in *Proceedings of the 2023 ACM Symposium on Cloud Computing*, ser. SoCC ’23, Santa Cruz, CA, USA: Association for Computing Machinery, 2023, pp. 358–375, ISBN: 9798400703874.
- [77] S. Yun and Y. Ro, “Shvit: Single-head vision transformer with memory efficient macro design,” in *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024, pp. 5756–5767.
- [78] J. Leskovec and A. Krevl, *SNAP Datasets: Stanford large network dataset collection*, <https://snap.stanford.edu/data>, Jun. 2014.
- [79] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, “Data movement is all you need: A case study on optimizing transformers,” in *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica, Eds., vol. 3, 2021, pp. 711–732.
- [80] D. Graur, D. Aymon, D. Kluser, T. Albrici, C. A. Thekkath, and A. Klimovic, “Cachew: Machine learning input data processing as a service,” in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA: USENIX Association, Jul. 2022, pp. 689–706, ISBN: 978-1-939133-29-43.
- [81] F. Strati, X. Ma, and A. Klimovic, “Orion: Interference-aware, fine-grained gpu sharing for ml applications,” in *Proceedings of the Nineteenth European Conference on Computer Systems*, ser. EuroSys ’24, Athens, Greece: Association for Computing Machinery, 2024, pp. 1075–1092, ISBN: 9798400704376.
- [82] H. Wu, J. Deng, M. Yu, Y. Yu, Y. Liu, H. Fan, S. Wu, and W. Wang, *Faastube: Optimizing gpu-oriented data transfer for serverless computing*, 2024. arXiv: 2411.01830 [cs.DC].
- [83] T. Hobson, O. Yildiz, B. Nicolae, J. Huang, and T. Peterka, “Shared-memory communication for containerized workflows,” in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2021, pp. 123–132.
- [84] J. Tarraga-Moreno, J. Escudero-Sahuquillo, P. J. Garcia, and F. J. Quiles, *Understanding intra-node communication in hpc systems and datacenters*, 2025. arXiv: 2502.20965 [cs.AR].
- [85] D. Gosnell, “3 considerations for adding real-time ml to applications,” *RTInsights*, Aug. 2022.

- [86] T. Benz, M. Rogenmoser, P. Scheffler, S. Riedel, A. Ottaviano, A. Kurth, T. Hoefler, and L. Benini, “A high-performance, energy-efficient modular dma engine architecture,” *IEEE Transactions on Computers*, vol. 73, no. 1, pp. 263–277, 2024.
- [87] A. Berthold, C. Fürst, A. Obersteiner, L. Schmidt, D. Habich, W. Lehner, and H. Schirmeier, “Demystifying intel data streaming accelerator for in-memory data processing,” in *Proceedings of the 2nd Workshop on Disruptive Memory Systems*, ser. DIMES ’24, Austin, TX, USA: Association for Computing Machinery, 2024, pp. 9–16, ISBN: 9798400713033.
- [88] A. Baumstark, L. Martins, and K.-U. Sattler, “Uncore your queries: Towards cpu-less query processing,” in *Proceedings of the 21st International Workshop on Data Management on New Hardware*, ser. DaMoN ’25, Association for Computing Machinery, 2025, ISBN: 9798400719400.
- [89] I. Corporation, *Intel® data streaming accelerator user guide*, <https://cdrdrv2-public.intel.com/759709/353216-data-streaming-accelerator-user-guide-002.pdf>, Accessed: 2025-05-16, 2024.
- [90] S.-T. Wang, H. Xu, A. Mamandipoor, R. Mahapatra, B. H. Ahn, S. Ghodrati, K. Kailas, M. Alian, and H. Esmaeilzadeh, “Data motion acceleration: Chaining cross-domain multi accelerators,” in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 1043–1062.
- [91] P. Fent, A. v. Renen, A. Kipf, V. Leis, T. Neumann, and A. Kemper, “Low-latency communication for fast dbms using rdma and shared memory,” in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1477–1488.
- [92] I. Corporation, *Dsa transparent offload (dto)*, <https://github.com/intel/DTO>, Accessed: 2025-05-16, 2024.
- [93] Intel Corporation, *Power management: User wait instructions - power saving for dpdk pmd polling workloads (technology guide)*, Accessed: 2025-03-30, 2024.
- [94] A. Zuepke, A. Bastoni, W. Chen, M. Caccamo, and R. Mancuso, “Mempol: Policing core memory bandwidth from outside of the cores,” in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2023, pp. 235–248.
- [95] V. Saravanan, K. D. Pralhaddas, D. P. Kothari, and I. Woungang, “An optimizing pipeline stall reduction algorithm for power and performance on multi-core cpus,” *Human-centric Computing and Information Sciences*, vol. 5, pp. 1–13, 2015.

- [96] M. Park, R. Dubey, Y. Yuan, N. S. Kim, and A. Gavrilovska, *Rethinking inter-process communication with memory operation offloading*, 2026. arXiv: 2601.06331 [cs.OS].
- [97] ONNX Community, *Open neural network exchange (onnx)*, <https://onnx.ai>, Accessed: 2025-04-10, 2019.
- [98] T. Chen and C. Guestrin, “Xgboost: A scalable tree boosting system,” *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 785–794, 2016.
- [99] Zilliz, *Milvus: An open-source vector database for scalable similarity search*, <https://github.com/milvus-io/milvus>, Accessed: 2025-06-21, 2020.
- [100] L. Page, S. Brin, R. Motwani, and T. Winograd, “The pagerank citation ranking: Bringing order to the web.,” Stanford InfoLab, Technical Report 1999-66, 1999.
- [101] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, “An image is worth 16x16 words: Transformers for image recognition at scale,” in *International Conference on Learning Representations (ICLR)*, 2021.
- [102] *Mlperf inference: Datacenter benchmark suite*, MLCommons benchmark suite, Accessed on 2025-07-30, 2025.
- [103] Open Compute Project, *Ocp accelerator module (oam) specification*, version 1.1, Open Compute Project Foundation, 2021.
- [104] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, “{SOCK}: Rapid Task Provisioning with Serverless-Optimized Containers,” in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 57–70.
- [105] Amazon Web Services, *Aws lambda layers*, <https://docs.aws.amazon.com/lambda/latest/dg/chapter-layers.html>, Accessed: 2025-12-24, 2025.
- [106] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, “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, pages = 467–481, series = ASPLOS '20*, 2020.
- [107] The Linux Kernel Organization, *Overlay filesystem documentation*, <https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt>, Accessed: 2025-12-24, 2014.

- [108] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, “Putting the ”Micro” Back in Microservice,” in *USENIX Annual Technical Conference (ATC’18)*, 2018, pp. 645–650, ISBN: 978-1-939133-01-4.
- [109] J. Carreira, S. Kohli, R. Bruno, and P. Fonseca, “From warm to hot starts: Leveraging runtimes for the serverless era,” in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ACM, Jun. 2021, pp. 58–64, ISBN: 9781450384384.
- [110] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci, “Cntr: Lightweight OS Containers,” in *USENIX Annual Technical Conference (ATC’18)*, 2018, pp. 199–212, ISBN: 978-1-939133-01-4.
- [111] 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*, ser. SOSP ’95, Copper Mountain, Colorado, USA: Association for Computing Machinery, 1995, pp. 251–266, ISBN: 0897917154.
- [112] P. Tembey, A. Gavrilovska, and K. Schwan, “Merlin: Application- and Platform-aware Resource Allocation in Consolidated Server Systems,” in *ACM Symposium on Cloud Computing (SOCC’14)*, Seattle, WA, 2014.
- [113] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum, “Cellular disco: Resource management using virtual clusters on shared-memory multiprocessors,” in *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, ser. SOSP ’99, New York, NY, USA: ACM, 1999, pp. 154–169.
- [114] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhanian, “The barrellfish operating system,” in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, ser. SOSP ’09, New York, NY, USA: ACM, 2009, pp. 29–44.
- [115] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer, “Active messages: A mechanism for integrated communication and computation,” in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ser. ISCA ’92, Queensland, Australia: Association for Computing Machinery, 1992, pp. 256–266, ISBN: 0897915097.
- [116] E. Cohen and D. Jefferson, “Protection in the hydra operating system,” in *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, ser. SOSP ’75, Austin, Texas, USA: Association for Computing Machinery, 1975, pp. 141–160, ISBN: 9781450378635.
- [117] A. Kalia, M. Kaminsky, and D. G. Andersen, “Design guidelines for high performance rdma systems,” in *Proceedings of the 2016 USENIX Conference on Usenix*

*Annual Technical Conference*, ser. USENIX ATC '16, Denver, CO, USA: USENIX Association, 2016, pp. 437–450, ISBN: 9781931971300.

- [118] Y. Xia, D. Du, Z. Hua, B. Zang, H. Chen, and H. Guan, “Boosting inter-process communication with architectural support,” *ACM Trans. Comput. Syst.*, vol. 39, no. 1–4, Jul. 2022.
- [119] H. Ji, M. Kim, S. Oh, D. Kim, and N. S. Kim, “Para-ksm: Parallelized Memory Deduplication with Data Streaming Accelerator,” in *Proceedings of the 2025 USENIX Annual Technical Conference (USENIX ATC)*, To appear, Boston, MA, USA: USENIX Association, 2025.
- [120] R. Liu, T. Ma, M. Zhang, J. Huang, Y. Shan, Z. Liu, L. Xiang, Z. Lin, H. Lu, J. Rao, K. Chen, and Y. Wu, “DSA-2LM: A CPU-Free Tiered Memory Architecture with Intel DSA,” in *Proceedings of the 2025 USENIX Annual Technical Conference (USENIX ATC)*, To appear, Boston, MA, USA: USENIX Association, 2025.
- [121] Q. Chen, H. Yang, J. Mars, and L. Tang, “Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16, Atlanta, Georgia, USA: Association for Computing Machinery, 2016, pp. 681–696, ISBN: 9781450340915.
- [122] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang, “Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers,” in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 406–418.
- [123] P. B. G. Cox, J. Vesely, and A. Basu, “Suv: Static analysis guided unified virtual memory,” in *Proceedings of the 2024 57th IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '24, Austin, TX, USA: IEEE Press, 2024, pp. 293–308.
- [124] P. Gera, H. Kim, P. Sao, H. Kim, and D. Bader, “Traversing large graphs on gpus with unified memory,” *Proc. VLDB Endow.*, vol. 13, no. 7, pp. 1119–1133, Mar. 2020.
- [125] G. Costa. “I/o access methods: Scylla’s path to user-space i/o.” Accessed: 2025-12-24, ScyllaDB.
- [126] The GNU C Library Development Team, *The gnu c library (glibc) - implementation of x86\_64 memcpy with static thresholds*, version 2.38, Accessed: 2025-10-08. See `sysdeps/x86/dl-cacheinfo.h` for cache-based thresholding logic., 2023.

- [127] W. de Bruijn, *MSG\_ZEROCOPY*, Linux Kernel Documentation, Accessed: 2025-12-24, 2017.
- [128] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, “The case for learned index structures,” in *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, ACM, 2018, pp. 489–504.
- [129] C. Delimitrou and C. Kozyrakis, “Paragon: Qos-aware scheduling for heterogeneous datacenters,” in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, New York, NY, USA: ACM, 2013, pp. 77–88.
- [130] C. Delimitrou and C. Kozyrakis, “Quasar: Resource-efficient and qos-aware cluster management,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, New York, NY, USA: ACM, 2014, pp. 127–144.
- [131] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O’Reilly, and S. Amarasinghe, “OpenTuner: An extensible framework for program autotuning,” in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Edmonton, Canada, Aug. 2014.
- [132] A. Ignatov, R. Timofte, A. Kulik, S. Yang, K. Wang, F. Baum, M. Wu, L. Xu, and L. V. Gool, *Ai benchmark: All about deep learning on smartphones in 2019*, 2019. arXiv: 1910.06663 [cs.PF].
- [133] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, “Lightgbm: A highly efficient gradient boosting decision tree,” in *Advances in Neural Information Processing Systems (NIPS/NeurIPS)*, vol. 30, Curran Associates, Inc., 2017, pp. 3146–3154.
- [134] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Second. Springer Science & Business Media, 2009.
- [135] P. Gupta and N. McKeown, “Classifying packets with hierarchical intelligent cuttings,” *IEEE Micro*, vol. 20, no. 1, pp. 34–41, Jan. 2000.