UNIVERSITY OF CALIFORNIA,
IRVINE

Self-aware Memory Management for Emerging Architectures

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Biswadip Maity

2023

DEDICATION

To my beloved parents, Papiya and Santanu Kumar Maity, whose sacrifices have shaped my journey and made me who I am today. And to Aheli Ghosh, for fostering my love for research and being my constant source of inspiration.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

None of the work presented here would have been possible without my advisor, Nikil Dutt. Nik's unwavering guidance and support throughout my doctoral journey gave me the freedom to explore my research interests I am grateful for his patience and constant encouragement during the ups and downs of my graduate life. I would also like to thank my committee members, Fadi Kurdahi and Nalini Venkatasubramanian, for their valuable feedback, suggestions and the opportunity to collaborate with their research groups.

Parth Malani for his continued guidance and for industry perspectives that shaped the direction and impact of my research. David Carrillo-Cisneros for his mentorship and extending the scope of my research from academia to an large-scale industrial setting. My colleagues and friends at Dutt Research Group: Amir, Donny, Tiago, Kasra, Majid, Hamid for guiding me through the initial years of my PhD. Alex, Caio, Hans for being an incredible source of friendship and making my time at UCI memorable. Aditi and Gopi who have made my time at UCI feel like a home away from home.

My work was generously supported by funding and resources from the Donald Bren School of Information and Computer Sciences, National Science Foundation (NSF) grant CCF-1704859, Deutscher Akademischer Austauschdienst (DAAD) grant 57440917, and a research gift from Meta Platforms, Inc.

Finally, thanks to IEEE, ACM, and Springer for granting permission to reuse previously published work in my thesis. Portion of Chapter 2 of this dissertation is a reprint of the material as it appears in [74], used with permission from IEEE. Portion of Chapter 3 of this dissertation is a reprint of the material as it appears in [79], used with permission from IEEE. Portion of Chapter 3 of this dissertation is a reprint of the material as it appears in [75], used with permission from ACM. Portion of Chapter 4 of this dissertation is a reprint of the material as it appears in [76], used with permission from Springer. Portion of Chapter 4 of this dissertation is a reprint of the material as it appears in [99], used with permission from ACM. Portion of Chapter 5 of this dissertation is a reprint of the material as it appears in [80], used with permission from ACM.

VITA

Biswadip Maity

EDUCATION

Doctor of Philosophy in Computer Science	**2023**
University of California, Irvine	*Irvine, CA*
Master of Science in Computer Science	**2019**
University of California, Irvine	*Irvine, CA*
Bachelor of Engineering in Computer Science and Engineering	**2011**
Jadavpur University	*Kolkata, India*

RESEARCH EXPERIENCE

Graduate Research Assistant	**2017–2023**
University of California, Irvine	*Irvine, California*

TEACHING EXPERIENCE

Associate Instructor	**2022**
University of California, Irvine	*Irvine, California*
Teaching Assistant	**2017–2021**
University of California, Irvine	*Irvine, California*

INDUSTRY EXPERIENCE

Research Intern	**2022**
Meta Platforms Inc.	*Menlo Park, California*
Research Intern	**2020**
Meta Platforms Inc.	*Menlo Park, California*
Cloud Infrastructure Intern	**2019**
Tinder	*West Hollywood, California*
Software Engineer	**2015–2017**
Microsoft India R&D	*Hyderabad, India*

REFEREED JOURNAL PUBLICATIONS

The Self-Aware Information Processing Factory Paradigm for Mixed-Critical Multiprocessing **2022**
IEEE Transactions on Emerging Topics in Computing

SEAMS: Self-Optimizing Runtime Manager for Approximate Memory Hierarchies **2021**
ACM Transactions on Embedded Computing Systems

Chauffeur: Benchmark Suite for Design and End-to-End Analysis of Self-Driving Vehicles on Embedded Systems **2021**
ACM Transactions on Embedded Computing Systems

Self-Adaptive Memory Approximation: A Formal Control Theory Approach **2020**
IEEE Embedded Systems Letters

HESSLE-FREE: Heterogeneous Systems Leveraging Fuzzy Control for Runtime Resource Management **2019**
ACM Transactions on Embedded Computing Systems

REFEREED CONFERENCE PUBLICATIONS

Expanding Datacenter Capacity with DVFS Boosting: A safe and scalable deployment experience **2024**
Architectural Support for Programming Languages and Operating Systems

Locate: Low-Power Viterbi Decoder Exploration Using Approximate Adders **2023**
Great Lakes Symposium on VLSI

Information Processing Factory 2.0 - Self-awareness for Autonomous Collaborative Systems **2023**
Design Automation & Test in Europe

ProSwap: Period-aware Proactive Swapping to Maximize Embedded Application Performance **2022**
International Conference on Networking, Architecture, and Storage

Self-aware Memory Management for Emerging Energy-efficient Architectures **2020**
International Green Computing Conference

Workload characterization for memory management in emerging embedded platforms **2019**
International Embedded Systems Symposium

The information processing factory: a paradigm for life cycle management of dependable systems **2019**
CODES/ISSS

SOFTWARE

Chauffeur `https://github.com/duttresearchgroup/Chauffeur/`
Open source benchmark suite for self-driving cars.

MARS `https://github.com/duttresearchgroup/mars/`
Middleware for Adaptive Reflective Computer Systems

ABSTRACT OF THE DISSERTATION

Self-aware Memory Management for Emerging Architectures

By

Biswadip Maity

Doctor of Philosophy in Computer Science

University of California, Irvine, 2023

Distinguished Professor Nikil Dutt, Chair

The ever-increasing demands of data-intensive applications and the rapid evolution of computer architectures have posed significant challenges in memory performance and energy efficiency. Efficient memory management is crucial to meet the requirements of these applications while optimizing the utilization of memory resources. Traditional approaches that rely on workload-specific optimizations and static memory configurations are no longer sufficient to address the dynamic nature of modern computing systems.

To overcome these challenges, the concept of computational self-awareness (CSA) has emerged as a promising approach. Computational self-awareness draws inspiration from psychology and neuroscience and aims to develop intelligent systems that can learn from past experiences, reason about their current state, and make informed decisions at runtime.

In this thesis, I explore the application of computational self-awareness in the context of memory management. I investigate the different degrees of self-awareness applied across the memory subsystem and examine their benefits on memory performance and energy consumption. The results highlight the potential of computational self-awareness in addressing the challenges posed by data-intensive applications and evolving computer architectures, paving the way for improved performance, energy efficiency, and bandwidth utilization in memory systems.

Chapter 1

Introduction

The ever-increasing demands of data-intensive applications and the rapid evolution of computer architectures have posed significant challenges in memory performance and energy efficiency. Efficient memory management is crucial to meet the requirements of these applications while optimizing the utilization of memory resources. Traditional approaches that rely on workload-specific optimizations and static memory configurations are no longer sufficient to address the dynamic nature of modern computing systems.

To overcome these challenges, the concept of computational self-awareness (CSA) has emerged as a promising approach. Computational self-awareness draws inspiration from psychology and neuroscience and aims to develop intelligent systems that can learn from past experiences, reason about their current state, and make informed decisions at runtime. By applying self-awareness principles to memory management, it becomes possible to create an energy-efficient memory subsystem that adapts to changing workloads and optimizes memory usage across different architectures and technologies.

The objective of this thesis is to explore the application of computational self-awareness in the context of memory management. Figure 1.1 illustrates the abstraction layers in

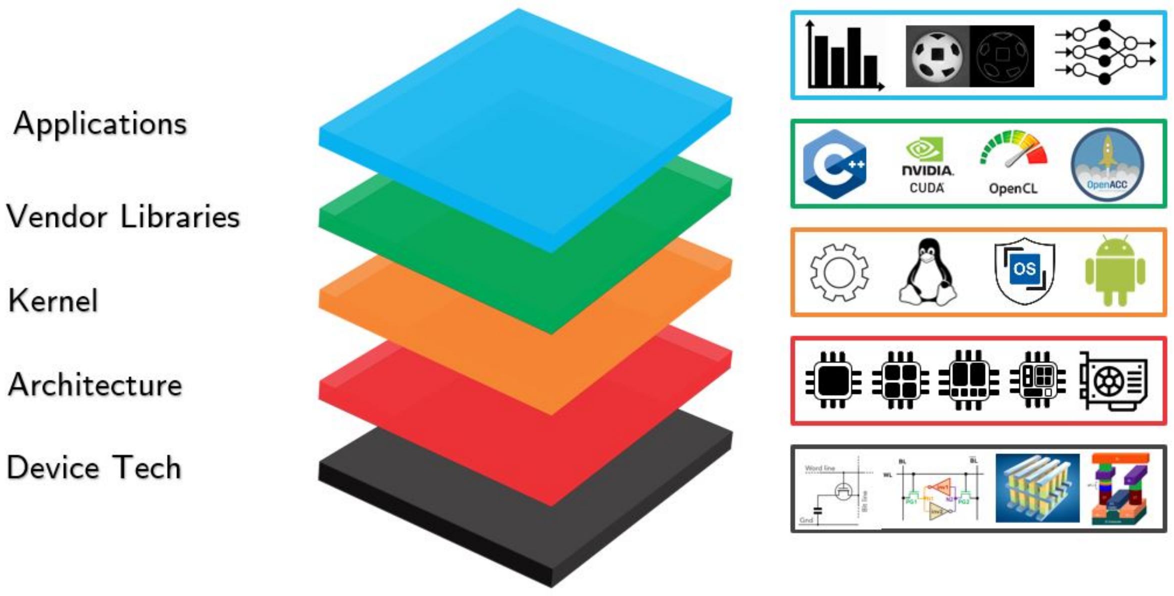


Figure 1.1: Abstraction layers in the application layer of the memory subsystem.

the application layer of the memory subsystem, providing a visual representation of the components and interactions involved. By integrating self-awareness capabilities into the memory subsystem, we aim to design an intelligent system that can dynamically adjust its memory configuration, allocation, and access policies to optimize performance and energy efficiency. This thesis investigates the different degrees of self-awareness that can be achieved in the memory subsystem and examines their impact on memory performance and energy consumption.

In this thesis, we will delve into the principles and techniques of self-aware memory management, focusing on the hardware layer, operating system layer, and application layer. We will analyze the challenges faced in memory performance and energy efficiency and propose innovative solutions that leverage computational self-awareness to address these challenges. Through extensive experimental evaluation, we will validate the effectiveness of our self-aware memory management techniques and demonstrate their benefits in terms of performance improvement and energy savings.

1.1 Background and Related Work

Efficient memory management plays a critical role in meeting the demands of data-intensive applications and optimizing the utilization of memory resources. However, traditional approaches and static configurations are no longer sufficient to address the dynamic nature of modern computing systems. In this section, we provide a background on the systems and machine learning landscape, followed by an exploration of the concept of computational self-awareness (CSA) and its application in memory management.

1.1.1 Systems and Machine Learning Landscape

Current efforts related to ”intelligent systems” typically deploy machine learning (ML) techniques to solve a wide range of problems, both at the application and system levels. We therefore begin with a brief overview of the research landscape addressing energy efficiency at the intersection of systems and ML.

Systems for Machine Learning Machine learning applications have been widely adopted in domains ranging from low-power Internet-of-Things (IoT) devices, edge networks, autonomous vehicles, to large-scale data centers. Application researchers have looked into various facets of machine learning: model accuracy, interpretability, security, bias, privacy, model scalability, and opportunities for acceleration. Heterogeneous many-core systems are continuously evolving to support the data-centric nature of ML applications [104]. We refer to these systems as ***Systems for ML***. Some of these applications (e.g., deep learning algorithms such as convolutional neural networks) consist of a large number of floating-point multiplications and additions which are well supported by graphics processing units (GPUs). GPUs have evolved into highly parallel many-core processing elements allowing efficient manipulation of large blocks of data. GPUs with dedicated main-memory (server GPUs) can perform extremely fast floating-point arithmetic compared to general-purpose processing units (CPUs). However, the energy consumption of server GPUs often limits its applicability in embedded domains. Alternatively, embedded GPUs in systems-on-chip (SoCs) share main-memory with the general-purpose CPUs while offering more energy-efficiency [127] than the server GPUs, an essential requirement for battery-driven mobile devices. Embedded GPUs offer embedded designers an opportunity to use the streaming multiprocessors for general-purpose parallel processing [159]. Machine learning software frameworks like Tensorflow and Pytorch provide libraries for ML application researchers to efficiently utilize heterogeneous resources without specialized knowledge about the underlying hardware. However, due to the limited number of registers in the small cores, GPU kernels require many memory accesses to the shared main-memory. Access to the main-memory remains the significant performance and energy bottleneck in embedded systems [76]. To honor the low-power constraints while increasing performance (i.e., accuracy, throughput, and scalability), accelerators have gained traction for machine learning applications. Literature in different domains ranging from healthcare applications [139] to deep learning [131] demonstrates that application-specific accelerators can achieve higher performance throughput with better energy-efficiency. Accelerating common building blocks with specialized hardware still requires general-purpose processors to launch the kernels with the initial data and fetch the results at the end of execution to continue the rest of the application. Sriraman *et al.* [120] show that the orchestration spent around core-application logic, which includes copying, allocating, and freeing memory, can consume up to 37% of cycles for datacenter workloads. In emerging systems for ML, data movement remains a critical bottleneck for performance and energy-efficiency.

Machine Learning for Systems We now focus our attention on energy-efficiency challenges faced by designers during the design as well as runtime execution of embedded systems [103]. While embedded systems (e.g., a battery-powered mobile phone) are purpose-built, they are also expected to run various applications throughout their lifetime. Some of these applications are data-centric (e.g., rendering a game), while others are less resource-intensive (e.g., browsing emails). In some cases, users expect applications to deliver a minimum performance (e.g., 30 frames-per-second (FPS) refresh rate in games), which we define as the quality-of-service (QoS). It is the embedded system designer's responsibility to configure the system parameters before deployment and further deploy runtime policies that deliver the required performance while still being energy-efficient at runtime. We refer to the intelligent strategies used for design and management of systems as ***ML for Systems*** and review some related efforts.

The plethora of on-chip and off-chip resources (e.g., compute, memory, network) available in a system presents a challenging task for an embedded system designer: configuring the system to meet the application's QoS requirements while minimizing the energy consumption. The operating parameters for CPUs, GPUs, memory, and interconnect creates a large design space. Together with runtime decisions (e.g., scheduling, mapping), parameter configuration puts the burden on system designers to identify operating points that meet the performance requirements while being energy-efficient. In the face of dynamic workloads, performing workload-specific optimizations for runtime resource allocation and dynamic power management for energy-efficiency is infeasible at design-time. Recent efforts have leveraged machine-learning-based techniques to guide the design of specialized hardware, as well as improve the computational efficiency of hardware design optimization [138]. Online learning techniques (e.g., reinforcement learning) can also be leveraged to automatically learn policies specific to workloads, reducing the burden on system designers [31].

While machine learning techniques have been instrumental in enhancing system efficiency, they alone may not be sufficient to address the complexities and dynamic nature of modern computing systems. To fully realize the potential of intelligent and adaptive systems, the concept of computational self-awareness (CSA) becomes crucial. By incorporating self-awareness principles inspired by human cognitive processes, CSA complements machine learning techniques and enables systems to observe, reason, and adapt their behavior based on their internal and external states.

1.1.2 Computational Self-Awareness (CSA)

Computational self-awareness (CSA) is an emerging research area that aims to develop intelligent systems capable of self-learning, self-reasoning, and self-adaptation. Inspired by human cognitive processes, CSA combines techniques from psychology, neuroscience, autonomic computing, machine learning, and artificial intelligence to create systems that can observe, reflect upon, and adapt their behavior based on their internal and external states.

CSA enables systems to go beyond traditional rule-based approaches and exploit their knowledge and understanding of their own behavior to optimize their performance, energy efficiency, reliability, and resource utilization. By building models of themselves and their environments, self-aware systems can make informed decisions, predict future states, and take proactive actions to achieve desired goals.

In the context of memory management, computational self-awareness offers promising opportunities to overcome the limitations of traditional approaches. By incorporating self-awareness capabilities into the memory subsystem, it becomes possible to dynamically adapt memory configurations, access patterns, and allocation policies to optimize performance and energy efficiency. This can lead to significant improvements in overall system performance and responsiveness.

Several research efforts have explored the application of computational self-awareness in memory management. These studies have focused on various aspects, such as runtime optimizations, approximation techniques, goal-oriented adaptation, and self-healing mechanisms. By leveraging machine learning algorithms, control theory, and system identification techniques, these approaches have demonstrated the potential to enhance memory performance, reduce energy consumption, and improve system reliability.

In this thesis, we aim to contribute to the field of self-aware memory management by investigating novel techniques and algorithms that leverage computational self-awareness principles. We will explore the integration of self-awareness capabilities at different layers of the memory subsystem, including the hardware layer, operating system layer, and application layer. By developing intelligent memory management strategies, we strive to achieve optimal memory performance and energy efficiency in dynamic computing environments.

1.2 Challenges and Contributions

The field of self-aware memory management poses several challenges that need to be addressed to harness the full potential of computational self-awareness. These challenges include the design of efficient self-awareness mechanisms, the development of accurate models for memory behavior, the integration of self-awareness at different layers of the memory subsystem, and the coordination of self-awareness across multiple layers and components of the system. Additionally, there is a need to balance the trade-offs between performance, energy efficiency, and reliability in self-aware memory management.

In this thesis, we aim to tackle these challenges and make the following contributions:

- Chapter 2: Self-Aware Memory Management

  In Chapter 2, we provide a comprehensive overview of self-aware memory management techniques. We discuss the motivations behind self-aware memory management, the properties of computational self-awareness relevant to memory management, and the implementation of self-awareness through the observe-decide-act (ODA) loop.

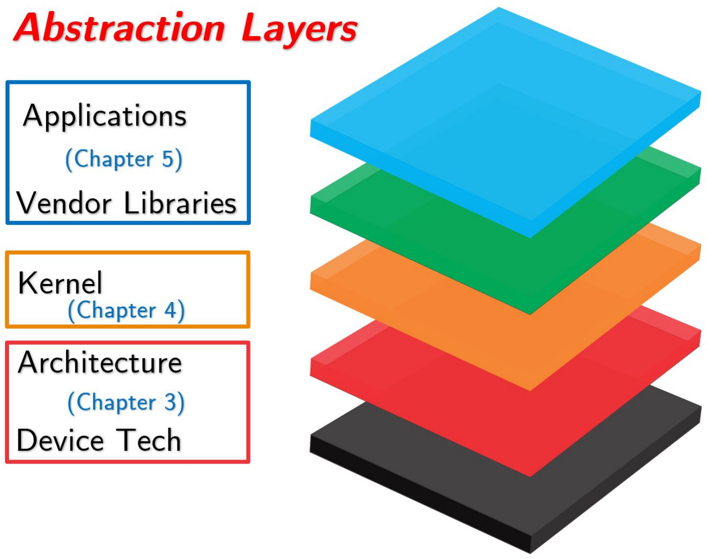


Figure 1.2: Representation of abstraction layers divided into chapters.

- Chapter 3: Hardware Layer

  In Chapter 3 (red box in Fig. 1.2), we focus on the hardware layer of the memory subsystem through a use case on approximate memories as an exemplar of self-aware memory management. We investigate techniques for incorporating self-awareness into memory controllers, cache hierarchies, and emerging memory technologies. We explore how self-aware algorithms can be used to optimize memory access patterns, dynamically adjust memory configurations, and adapt to changing workloads. We also discuss the challenges and trade-offs involved in implementing self-awareness at the hardware layer.

- Chapter 4: Operating System Layer

  In Chapter 4 (orange box in Fig. 1.2), we shift our focus to the operating system layer of the memory subsystem. We examine how self-awareness can be integrated into memory allocation policies, working set size management, and algorithms to manage the main memory controller. We explore the use of self-awareness techniques to predict memory demands, optimize resource allocation, and improve overall system performance.

- Chapter 5: Application Layer

  Chapter 5 (blue box in Fig. 1.2) explores the application layer of the memory subsystem, with a focus on emerging data-centric applications and end-to-end software pipelines. We discuss the motivation for benchmarking and optimizing data-centric applications on embedded systems, emphasizing the significance of efficient memory management in this context.

  In the first case study, we present Chauffeur, a novel open-source end-to-end benchmark suite for self-driving vehicles. Chauffeur provides a comprehensive set of self-driving application categories, enabling the evaluation and benchmarking of different instantiations of self-driving pipeline. We highlight the usefulness of Chauffeur in facilitating the assessment of different self-driving scenarios and identify opportunities for future system designer.

  In the second case study, we focus on datacenter autoscalers that automatically allocate machines to sustain the increasing demands of Machine Learning (ML) workloads. Performance requirements and capacity constraints become challenging to resolve as datacenter capacity gets crunched and workloads become increasingly dynamic. We propose a new approach: Self-Optimizing Autoscaler for MLaaS Inference Serving Systems OASys using multi-agent reinforcement learning to dynamically alter the batch-sizes as well as turbo configuration in realtime, responding to workload fluctuations and power budgets in datacenter racks.

Overall, this thesis aims to contribute to the field of self-aware memory management by providing a comprehensive understanding of computational self-awareness principles, exploring their application in different layers of the memory subsystem, and developing intelligent techniques and algorithms to optimize memory performance and energy efficiency. By addressing the challenges and leveraging the benefits of self-awareness, we strive to pave the way for more efficient and adaptive memory management in modern computing systems.

Chapter 2

Self-aware Memory Management using CSA

Memory management is a critical aspect of computing systems, playing a vital role in achieving optimal performance, energy-efficiency, and reliability. Traditional memory management techniques have been designed based on static configurations and assumptions about workload behavior. However, as computing systems become more complex and diverse, these traditional approaches often fall short in meeting the dynamic requirements of modern applications.

To overcome the limitations of traditional memory management, researchers have turned to self-awareness principles and techniques. Self-aware computing systems have the ability to learn, adapt, and make decisions based on their own observations and reasoning. By incorporating self-awareness into memory management, systems can dynamically adjust their memory configurations and operations to optimize resource utilization and meet application goals.

This chapter focuses on the application of self-awareness principles to memory management

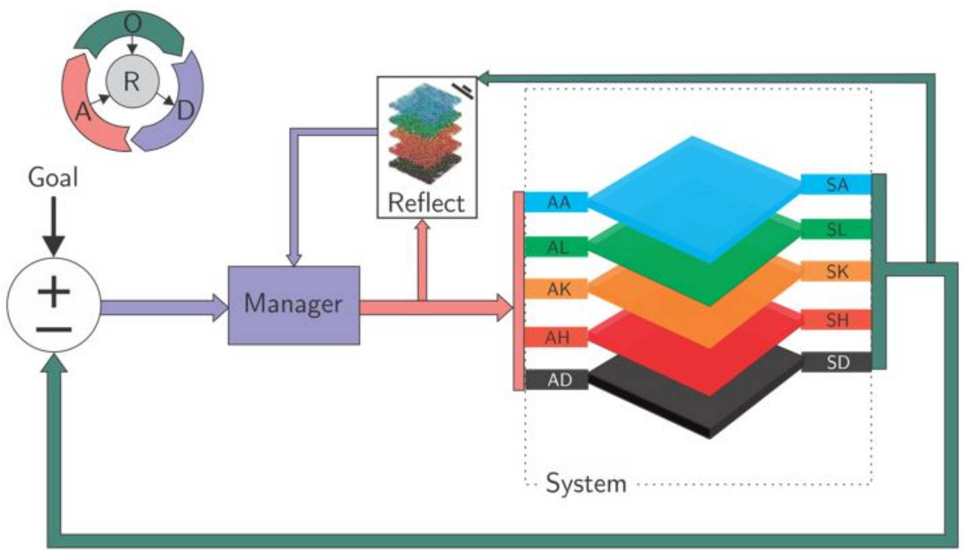


Figure 2.1: Observe-Decide-Act (ODA) loop in an embedded system with reflection (R) for self-aware management [129]. The layers shown are application (blue), vendor libraries (green), kernel (orange), hardware (red), and device (black). Each layer has sense (S) and act (A) capabilities.

using the concept of Computational Self-Awareness (CSA). CSA combines the principles of introspection, approximation, goal orientation, adaptation, and self-healing to enable intelligent memory management. These self-* properties empower memory management systems to observe their own behavior, make informed decisions, and adapt to changing workload requirements and system conditions.

The chapter begins by discussing the self-* properties and how they apply to memory management. We explore the role of introspection in observing the behavior of the memory subsystem at runtime, the use of approximation techniques to optimize resource utilization, the goal-oriented nature of memory management to align with user or application objectives, the adaptive mechanisms for dynamically adjusting memory configurations, and the self-healing capabilities to ensure reliability in the face of errors.

2.1 Self-* properties and how it applies to Memory Management

Computational self-awareness involves the application of self-* properties to memory management. Self-* properties refer to the characteristic properties of self-aware systems, including introspection, approximation, goal orientation, adaptation, and self-healing. In the context of memory management, these properties play a crucial role in optimizing system performance, energy-efficiency, and reliability.

Kounev *et al.* [140] defines self-aware computing systems as systems with the following properties: (1) **Modelling**: the ability to *learn models* by capturing knowledge on an ongoing basis about the system as well as the environment in which the system is running, and (2) **Reflection**: ability to make decisions by *reasoning* using the models, and perform actions based on the decision. The model here is a generic abstraction of the system and environment. Examples are: (a) descriptive model that captures system performance-related parameters, (b) prescriptive model that defines actions based on different system states, and (c) predictive model to perform 'what-if' queries. The learning can include static information gathered during design-time, along with dynamic information gathered during runtime. Figure 2.1 shows an instance of such a system with the different abstraction layers and corresponding sensors. Examples of sensors in the memory subsystem: cache miss rate at various levels, main-memory bandwidth, main-memory latency, working set of processes, numbers of errors in data transmissions, or memory access, CUPTI [92] sensors for CUDA GPU kernels, and application-level QoS (e.g., FPS).

The properties associated with a self-aware system (which we refer to as self-* properties) are domain-specific and different, for example, in a collective system [135] versus in robotics [142]. Agarwal *et al.* [126] examine the fundamental properties that pertain to self-aware computation: introspection, approximation, goal orientation, adaptation, and self-healing.

Bellman *et al.* [136] review the challenges of applying self-awareness principles in resource-constrained cyber-physical systems. Following the road-map laid out in prior literature, we discuss some of the self-* properties and define them in the context of memory-management:

- **Introspection:** Introspection allows the memory management system to observe the behavior of the memory subsystem at runtime. This observation is facilitated by telemetry information collected from multiple abstraction layers, such as cache miss rates, main-memory bandwidth, latency, working set size, and application-level quality of service (QoS) metrics. Introspection enables the system to gain insights into the memory behavior and make informed decisions based on the observed information.

- **Approximation:** Memory approximation involves selecting the level of precision required for memory operations to achieve system or application goals. By leveraging the concept of approximation, the memory management system can optimize resource utilization and improve energy-efficiency. The challenge lies in determining the optimal trade-off between precision and performance, considering factors like data-centric application requirements and constrained resources.

- **Goal Orientation:** Goal orientation focuses on meeting user or application goals while optimizing memory management under given constraints. The goals can vary based on specific requirements, such as maintaining a certain QoS level, maximizing energy-efficiency, or minimizing power consumption per unit of work executed. The memory management system aims to align its operations with the defined goals, considering dynamic changes in workload and system states.

- **Adaptation:** Adaptation refers to the ability of the memory management system to dynamically change its operating configuration based on observed information and current goals. Through adaptive mechanisms, the system can adjust various parameters and knobs, such as memory-controller schedule, bandwidth reservation, voltage and frequency scaling, and load/store accuracy. Adaptation ensures that the memory management system continuously optimizes its performance and resource utilization.

- **Self-healing:** Self-healing mechanisms in memory management systems ensure correct operation in the face of unexpected errors or emergent behavior. The system detects errors at runtime, identifies their root causes, and takes appropriate actions to mitigate the errors. Self-healing is particularly crucial for safety-critical applications running on memory subsystems, as it helps maintain system reliability and prevents potential failures.

Table 2.1 provides examples of self-awareness properties for realizing a runtime memory-approximation manager. These properties showcase different degrees of self-awareness and their corresponding characteristics.

Reflection uses observed knowledge to aid decision-making by *reasoning*, and performs actuations based on these decisions. Continuous cross-layer observations together with *reflection* allow the system to *introspect*, which is one of the key properties of computational self-awareness.

Through *reflection*, intelligent systems can consider past observations as well as predictions made from past observations [128] during the decision making process. Reflection and predictions involve 'what-if' queries to two types of models: models for the subsystem(s) under control (e.g., memory subsystem, GPU subsystem), and models for other decision-making policies. Some of these models can be obtained at design-time (e.g., through system identification), while others can be generated at runtime (e.g., through linear regression, binning). In Figure 2.1, a self-model of the system is being used to performed the *reflection* (shown in white box).

The runtime manager (violet box in Figure 2.1) is responsible for closing the loop by making decisions about the system under control. Runtime resource management through decision

Table 2.1: Examples of self-awareness properties for realizing a runtime memory-approximation manager.

Property	**Degree of self-awareness**		
	Degree 1: (low)	***Degree 2: (medium)***	***Degree 3: (high)***
Introspection	Reactive: A closed-loop system that reacts to observed behavior by tuning approximation knobs (e.g., if observed quality drops below the threshold, increase the precision).	Reflective: Use predictive models of approximation knobs (e.g., model the bit-error-rate (BER) relationship to voltage and temperature for SRAM).	Meta self-aware: The system is aware that it is self-aware.
Approximation	Target a single layer in the memory hierarchy as candidates of approximation (e.g., L1 cache, DRAM).	Policies can automatically tune different layers of memory (e.g., on-chip cache and off-chip main-memory).	Polices can determine knobs for multi-layer memory hierarchies, as well as device variations.
Goal-orientation	Single-objective goal (e.g., maximize energy-efficiency).	Multi-objective goals (e.g., maintain QoS while minimizing energy), which are dynamic.	Goals specified in different abstraction layers that may conflict with each other.
Adaptation	Model-based closed loop control.	Self-optimizing model-free control.	Robust and self-optimizing model-free control.
Self-healing	Detect failures and terminate gracefully.	Detect failures and take corrective actions to continue execution.	Find the root cause of failure and take action to mitigate the error.

making is a well-researched area. Several efforts have been undertaken for energy-efficient management of the memory subsystem and can broadly be categorized into: (1) heuristic-based [83, 134], (2) control-theory-based [85, 137], and (3) machine-learning-based [132, 133]. Decisions enable *adaptivity* in systems by specifying a mechanism to update the system state based on the difference between observed information and the current goals.

2.2 Overview of Memory Approximation and its Challenges

Memory approximation plays a significant role in self-aware memory management. Modern data-centric applications often tolerate imprecision in certain data sections, enabling more efficient utilization of resources. Approximation techniques aim to utilize the minimum required precision for memory operations while achieving system or application goals.

However, memory approximation presents several challenges. One of the key challenges is determining the optimal level of approximation for different layers of the memory hierarchy. This decision requires considering factors such as the trade-off between precision and performance, the impact on application behavior, and the characteristics of emerging memory technologies.

Another challenge is the dynamic nature of workload requirements and system conditions. Memory approximation techniques should be able to adapt to changing workloads, varying resource constraints, and evolving system states. The ability to dynamically adjust the approximation level based on real-time observations and goals is crucial for achieving efficient memory management.

Additionally, memory approximation techniques need to ensure the correctness and reliability of memory operations. They should be able to handle potential errors introduced by approximation methods and mitigate their impact on system functionality and application behavior. Self-aware memory management systems incorporate self-healing mechanisms to address potential errors and ensure reliable operation.

By addressing these challenges, memory approximation techniques within the framework of self-aware memory management can effectively optimize system performance, energy-efficiency, and resource utilization.

2.3 Conclusion

In this chapter, we explore self-awareness principles, particularly in the context of memory management. By incorporating self-awareness into memory management, systems can dynamically adjust their configurations and operations to optimize energy utilization. The different degrees of self-awareness enable the system to achieve varying levels of efficiency and effectiveness in managing energy consumption.

While machine learning-based black-box methods are commonly used today, they lack interpretable reasoning and cannot fully leverage the available system resources. In contrast, self-awareness principles, particularly through reflection, empower system managers to reason using models when making decisions about system configuration. This reasoning capability allows for more informed and effective energy-saving strategies.

Adopting computational self-awareness principles in memory management represents an exciting direction for research and development. It holds great potential for improving the performance, energy-efficiency, reliability, and fast adoption of emerging computer architectures and newer memory substrates, as we see in more details in the next chapters.

Chapter 3

Hardware Abstraction Layer

3.1 Overview of memory approximation and its challenges

Memory approximation aims to reduce the bottleneck by trading off quality for performance or energy gains. This technique applies to applications that can tolerate degradation in the quality of service (QoS). However, in order to maintain the required QoS at runtime, additional mechanisms are needed to monitor the QoS and tune the knobs to meet the requirements. In this report, we describe a simulation infrastructure for runtime control of such a system equipped with quality-configurable approximate memory.

As applications become increasingly resource-intensive, intelligent trade-offs between performance and energy in battery-powered systems have become essential. Memory-accesses have emerged as one of the most significant performance and energy bottlenecks [20]. In light of this, the literature in approximate computing has been growing, showing that we can achieve higher performance and energy efficiency by trading off an acceptable loss in precision. Approximate memory, which relaxes the need for high-precision storage for some data structures, is an effective way to alleviate the energy bottleneck in memory for applications that can tolerate output errors.

Current approximation techniques depend on design-time modeling of workloads to determine the optimal knob configurations for a specific system configuration. This approach makes the techniques application- and platform-specific, and introduces significant overhead to utilize approximate memory for each new memory technology. Prior work on memory approximation required programmers to statically set approximation knobs to appropriate values through trial and error in order to reach the desired quality of service (QoS) [115, 69, 86].

In response to these challenges, we first introduce a new control-theory based approach where developers only specify a target QoS metric and the system uses a formal control-theoretic approach to tune the memory reliability knobs of a quality-configurable memory to guarantee the desired QoS. We outline how to develop a system model using System Identification theory for the memory components and how they react to different approximation settings using a statistical black-box modelling technique. Using the developed system model, we design a controller that observes the application's behavior at fixed epochs and tunes knobs automatically to deliver the desired QoS despite changing workloads and system variations. Our proposed methodology is able to maintain the QoS when operating on parts of the memory subsystem and can reach the required behavior much faster than a manual calibration scheme without any tuning by the programmer. This is the first work to introduce the idea of using a formal control-theory based approach for memory approximation.

Building upon this control-theory based approach, we develop SEAMS, a model-free method to tune memory knobs without prior observation of the system. SEAMS eases the design of systems with approximate memory by enabling deployment without requiring design-time exploration of configuration knobs. Once deployed, SEAMS can learn optimal knob configurations for unknown applications, resulting in self-optimizing systems. Furthermore, SEAMS enables coordination between multiple memory system knobs without explicit communication. We believe SEAMS will enable quick adoption of approximation using a variety of memory nodes.

The rest of the chapter, is organized as follow:

1. Introduction to Formal Control Theory, Self-adaptivity, Self-optimizing properties: This section provides an essential background in formal control theory and its relevance to self-adaptive and self-optimizing systems. This foundational knowledge sets the stage for the subsequent discussions and case studies.

2. Addressing the Challenge of Designing Memory Systems: A discussion of the challenge at hand: designing memory systems that not only ease the burden on programmers but also guarantee a desired quality of service (QoS).

3. Case Study 1 - Self-Adaptive Memory Approximation: This case study offers a comprehensive exploration of a formal control-theoretic approach for tuning memory approximation knobs. The subsequent sections delve into the system identification technique for capturing memory approximation effects and a discussion of the results when compared with a manual calibration scheme.

4. Case Study 2 - Self-Optimizing Runtime Manager: This case study delves into the limitations of existing approximation techniques for full memory hierarchies. It also presents the proposed self-optimizing runtime manager, SEAMS, including a detailed explanation of its methodology, coordination of runtime decisions for interdependent knobs and subsystems, and a demonstration of its effectiveness in reducing energy usage and QoS violations.

5. Summary: The chapter concludes with a summary, tying together the various discussions and demonstrating how they contribute to the broader thesis. This summary also

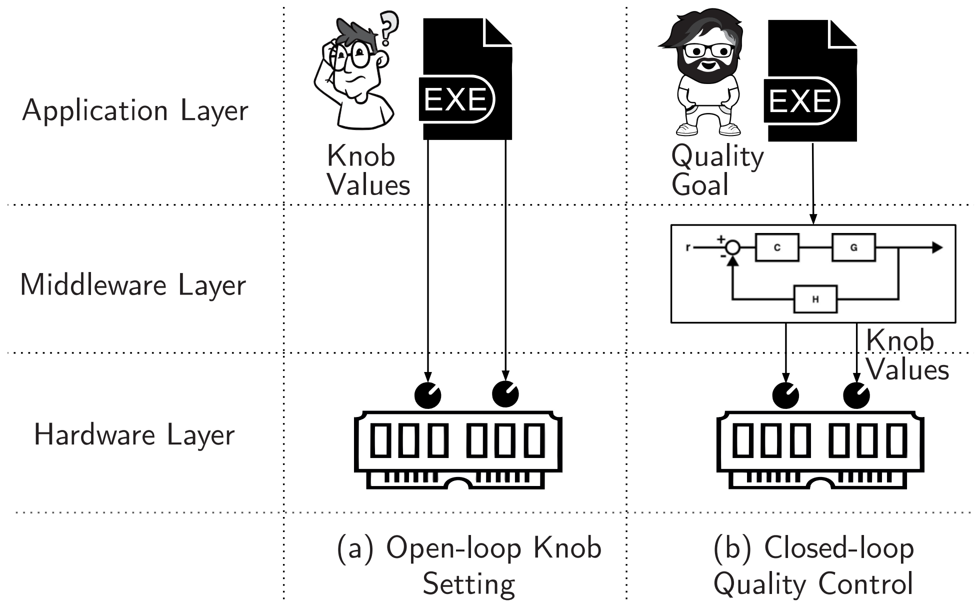


Figure 3.1: Open-loop knob settings vs. closed-loop quality control.

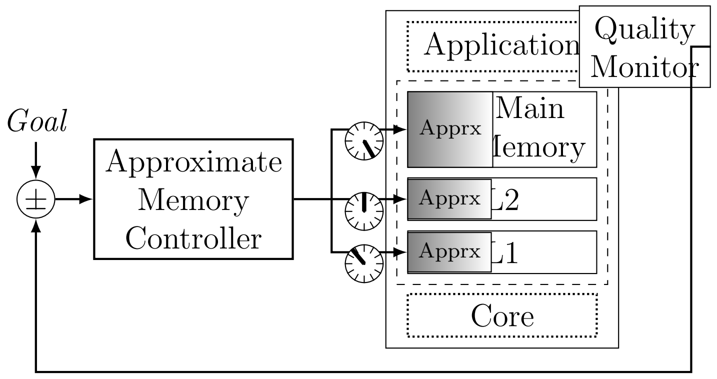


Figure 3.2: Runtime management of approximation knobs using output quality monitoring.

highlights the implications and potential future directions for the research presented in the chapter.

3.2 Brief intro to Formal control theory, Self-adaptivity, Self-optimizing properties

Control theory is a multifaceted discipline that finds its origins in mechanical and electrical engineering but has since permeated various domains including computing systems. At its core, control theory is concerned with influencing the behavior of systems to achieve a specific outcome or maintain a particular state, often in the presence of disturbances. It involves the use of feedback from the system state to adjust the control inputs dynamically.

Formal control theory, in particular, deals with rigorous mathematical techniques for the analysis and design of control systems. It provides a structured framework to understand the dynamics of complex systems, predict their behavior, and determine the appropriate control mechanisms. This approach allows for the effective control of system behavior while ensuring system stability, robustness, and performance.

On the other hand, self-adaptivity refers to the capability of a system to adjust its behavior dynamically in response to changes in the system state or environment. This is often achieved through a feedback loop, similar to control systems, where the system monitors its performance or other critical parameters and modifies its behavior to maintain optimal or desired operation. Self-adaptive systems are especially relevant in contexts where the system operates in unpredictable or changing environments, or where manual tuning of system parameters is infeasible or inefficient.

Self-optimizing properties take self-adaptivity a step further. A self-optimizing system not only adapts to changes but also seeks to optimize its performance or other objectives. This might involve balancing trade-offs between conflicting goals, such as maximizing performance while minimizing energy usage.

In the context of computing systems, these principles can be applied to manage and optimize various system resources. For instance, memory systems can be controlled and adapted dynamically to meet the desired Quality of Service (QoS) while minimizing resource usage. This approach can alleviate the burden on programmers by automating the tuning of system parameters and enable the system to handle dynamic workloads or operating conditions more effectively.

3.2.1 Related work - Bit Error Rate (BER) models

Previous works in the field of memory approximation have looked into the application of approximation in different memory technologies. Depending on the type of memory technology, researchers have come up with one or more knobs that can tune the degree of approximation in the memory subsystem to save energy by trading-off accuracy. The degree of approximation is represented by the probability of bit flips when accessing the memory and is called bit error rate (BER). Although BER metric is common across technologies, the mapping of BER to a physical technology knob depends on the type of memory used. In this section, we summarize the models of BER for various memory technologies:

1. **Static RAM (SRAM)**: In SRAM cells, the minimum operating voltage (Vmin) is the control knob to trade accuracy for energy gains. Wang and Calhoun developed models of cell failure probability for read, write, and hold operations in [145]. Ansari *et. al.* modeled the failure rate of an SRAM cell based on the Vmin in a 90nm technology [4]. The model is shown in Figure 3.3.

2. **Dynamic RAM (DRAM)**: Previous works have shown that BER in a DRAM can be modelled as a function of temperature and refresh-rate [13, 69]. One such model from [69] which relates the error rates under different refresh cycles (at $40°C$) is presented in Table 3.1.

3. **Spin-Transfer-Torque Magnetic RAM (STT-MRAM)**: Oboril *et. al.* proposed

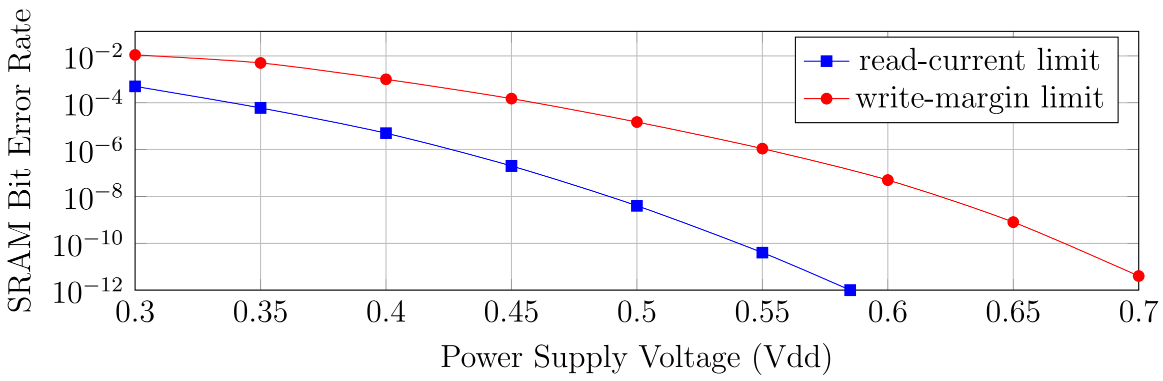


Figure 3.3: Bit error rate for an SRAM cell with varying Vdd values in 90nm. Figure from [4].

Refresh Cycle [s]	Error Rate per DRAM cell	Bit Flips per Byte
1	4.0×10^{-8}	3.2×10^{-7}
2	2.6×10^{-7}	2.1×10^{-6}
5	3.8×10^{-6}	3.0×10^{-5}
10	2.0×10^{-5}	1.6×10^{-4}
20	1.3×10^{-4}	1.0×10^{-3}

Table 3.1: DRAM error rate under different refresh cycle (at $40°C$). Table from [69] based on results from [13].

relaxing thermal stability factor of STT-MRAM to enable fast and energy-efficient cache memories[95]. Ranjan *et. al.* developed a model for error probabilities as the read/write energy is varied for Spintronic Memories [105] and is shown in Figure 3.4.

In the experiments concerned with the simulation of quality-configurable memory, the underlying technology is abstracted by choosing BER as the control knob. Since there are models for each technology which maps the technology-dependent control knob to BER, the models can be incorporated in the infrastructure by adding another lookup-table or an equation which translates between BER and the technology-dependent control knob. As an example, the equation for STT-MRAM writes, as obtained from the model in Figure 3.4 (b), is:

$$WriteEnergy(pJ) = \min\Big\{-4.836 \times log_{10}(BER) - 1.022, 36\Big\} \tag{3.1}$$

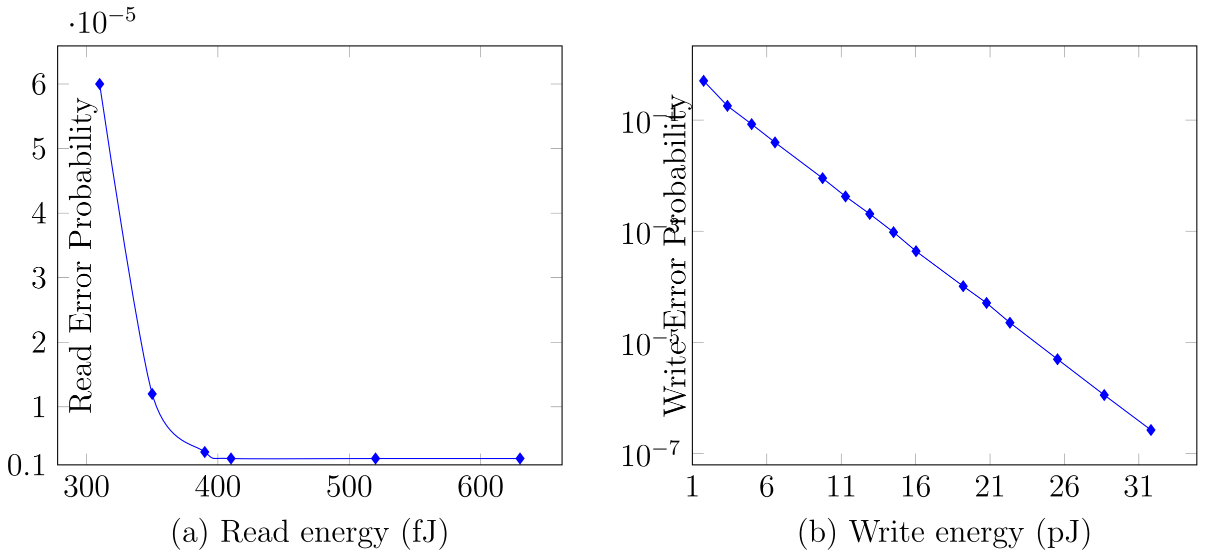


Figure 3.4: Energy vs. error probability trade-off for an STT-MRAM bit-cell from [105].

The minimum function is used to limit the energy to the value when write operation is performed without any approximation (i.e., BER = 0).

3.2.2 Simulation infrastructure

To simulate the behavior of a system with approximate memory, we developed a Sniper-based [19] memory fault injector (FI). The simulation infrastructure is described in this section.

Host system

To carry out the experiments a host system with an eight-core Intel Xeon(R) CPU E3-1230 V2 processor with 16 GB of RAM running *Ubuntu 16.04* Operating System is used. Sniper 6.1 is compiled with *gcc-4.7.4* and uses *PIN* front end.

Simulated system

The simulated system in Sniper has a single-core processor with *x86* instruction set architecture (ISA) based Gainestown micro-architecture and two levels of on-chip cache (L1 and L2). A fault injector (FI) is added which can inject faults into read / write operations of the memory hierarchy (e.g., cache, TLB, DDR) [1]. We make the following changes to implement the FI:

1. The application source code is first analyzed to detect the non-critical parts of the data. Although future work could explore automatically detecting the non-critical data elements, currently programmer expertise is needed to detect these addresses. In our target applications, we used a combination of both static analysis and Valgrind to select the non-critical data elements. Typically, large data structures which hold signal buffers (e.g., images, video) are good candidates for non-critical data.

2. To inject faults only into the non-critical data objects of the program, the source code of the program is annotated with `add_approx()` and `remove_approx()` methods to declare the address of the non-critical data objects in the program. The annotations use SimAPI commands in Sniper which are magic-instructions to communicate from the user application to the simulator at runtime. The annotated user-program is then compiled with Sniper API libraries.

   When developing a real platform, we expect to have a quality configurable memory subsystem with separate instances of: (1) Regions which do not have any quality configuration, and do not allow any errors; (2) Regions with configurable knobs (e.g., voltage in SRAM, refresh rate in DRAM, read/write current amplitude in NVM). The critical data elements can only be mapped to the non-configurable regions, whereas the non-critical can be mapped to either region.

[1] Code repository at https://github.com/duttresearchgroup /memapprox-control.

3. Whenever `add_approx()` or `remove_approx()` methods are invoked, the methods are captured by the FI. FI records these addresses into a table (dedicated memory) in the simulator. At runtime, all the memory accesses are instrumented. Sniper is used with Pin as frontend and runs as a Pintool which is used for dynamic instrumentation. For each memory access, if the virtual address of the access falls into the any of the given address boundaries in the table, FI attempts to inject a fault into the part of data referenced by that memory access. The probability of a bit-flip is determined by the value of BER in the simulation framework.

4. A separate controller, which is implemented in the middleware, is capable of receiving the results from quality monitors at runtime from user-applications. The controller continually monitors the application's output quality. The controller can communicate with the simulator and dynamically change the degree of approximation through SimAPI commands. The controller changes the degree of approximation by setting the read/write BER knobs in the simulation framework.

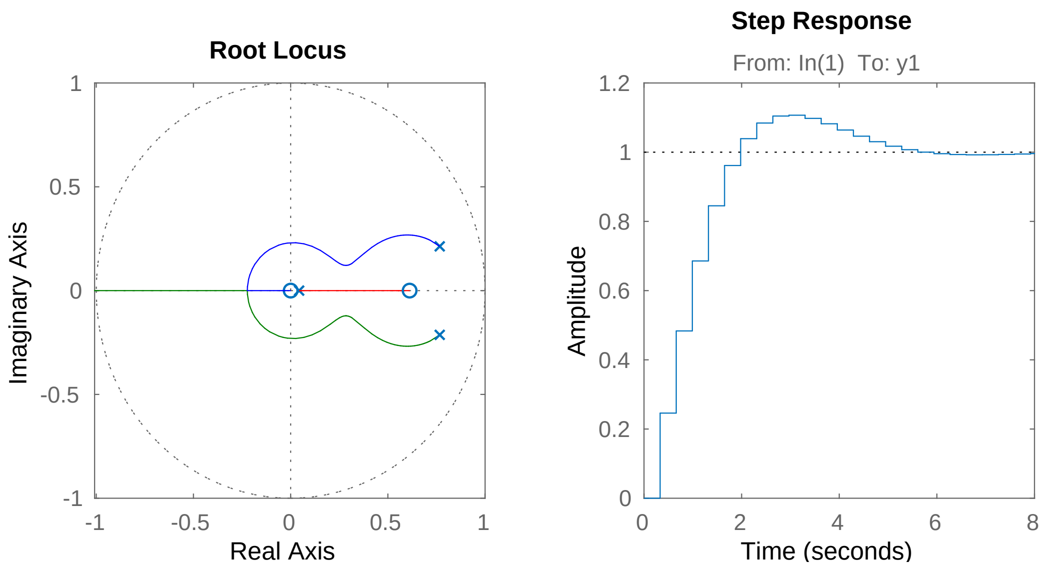


Figure 3.5: Root locus and step response for system identification of L1 data Writes.

3.2.3 System Model

The goal of the System Identification process is to identify a model of the system using black-box techniques. We use the System Identification toolbox in Matlab to model the system and understand the system dynamics of the generated model. The modeling and analysis are performed for each memory component for reads and writes. This section outlines the steps in detail.

Generating the data

In the simulation framework described in Section 3.2.2, the target application is executed with different bit error rates (BER) to observe the loss in accuracy. The execution starts with BER set to 0, and then it is varied in steps of $2.5e^{-7}$ after every 10 frames are processed. This creates a waveform with step-like input function. The data collection is terminated after 120 steps ($BER = 3e^{-5}$). This captures the degradation in output quality with variation of BER knobs. Depending on the application's tolerance to errors, the number of steps can be changed to capture most of expected quality values during runtime.

Discrete-time identified transfer function

A system can be modelled using Z-transforms which describe systems using transfer functions. This *transfer function* of a system describes how an input $U(z)$ is transformed into the output $Y(z)$ and is defined as $G(z) = \frac{Y(z)}{U(z)}$. We use the System Identification toolbox in Matlab to model the system using data generated in previous step.

Stability

A closed-loop system is considered stable, if all the closed-loop poles have a magnitude of less than 1. To determine the magnitude of the closed-loop poles, once the transfer function is obtained, the root locus of the system is plotted in Matlab. The root locus is the locations of all possible roots of the transfer function. The root locus and the step response obtained after modeling L1 write errors are presented in Fig 3.5. It can be seen that all the poles are within the unit circle, hence the system can be considered stable.

Maximum overshoot

The *maximum overshoot* is the absolute value of the largest difference between the output signal and its steady-state value, divided by steady state value and is denoted by: $M_P = \dfrac{|y_{max} - y_{ss}|}{|y_{ss}|}$.

Summary of system dynamics

We present the summary of the system dynamics for memory writes in the table below:

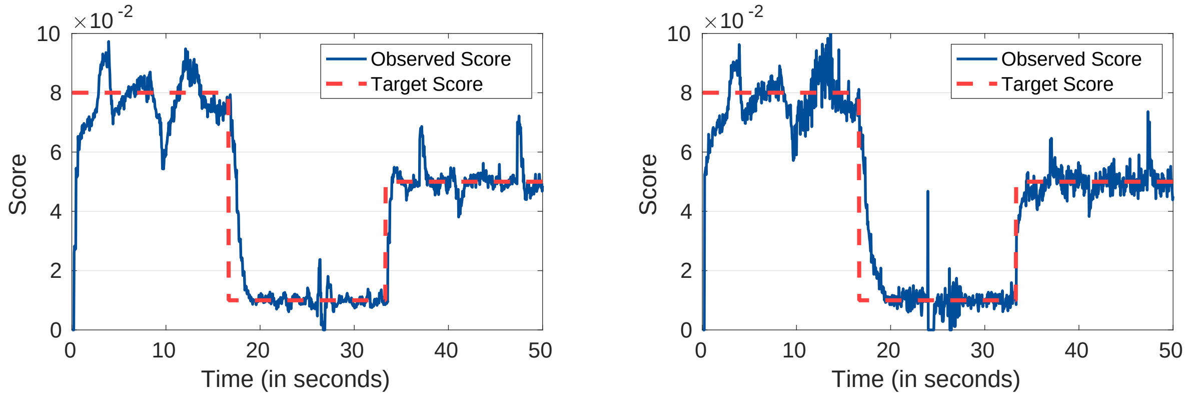
Component	Transfer function	Stable	Max-overshoot	Mode
L1D	–	-	137%	Static BER=$1e^{-3}$
L1D	$\dfrac{379.8z^{-1}}{1 - 0.823z^{-1} + 0.0276z^{-2}}$	Yes	21.6%	PI
L1D	–	-	25%	Manual
L2D	$\dfrac{39.89z^{-1}}{1 - 1.817z^{-1} + 0.8378z^{-2}}$	Yes	26%	PI
L2D	-	Yes	7.5%	Manual
DRAM	$\dfrac{1.796z^{-1}}{1 - 0.3254z^{-1} - 0.1103z^{-2}}$	Yes	38.3%	PI

Algorithm 2 Pseudo-code for manual recalibration algorithm used in tuning of memory approximation knob.

```
1: $upper\_bound \leftarrow 1.1$                                   ▷ Upper bound for settling
2: $lower\_bound \leftarrow 0.9$                                   ▷ Lower bound for settling
3: $resolution \leftarrow 3E-6$                                    ▷ Minimum change of manual knob
4: procedure MANUAL_CALIBRATIONS($current\_knob, current\_error, target\_error$)   ▷ The manual recalibration scheme which returns the next knob.
5:     $next\_knob \leftarrow knob$
6:     if $current\_error > upper\_bound \times target\_error$ then
7:         if $target\_error > 0$ then
8:             $multiplier \leftarrow log(current\_error/target\_error) + 1$
9:         else
10:            $multiplier \leftarrow 1$
11:        end if
12:        $step \leftarrow resolution \times multiplier$
13:        $next\_knob \leftarrow next\_knob - step$
14:    else if $current\_error < upper\_bound \times target\_error$ then
15:        if $current\_error > 0$ then
16:            $multiplier \leftarrow log(target\_error/current\_error) + 1$
17:        else
18:            $multiplier \leftarrow 1$
19:        end if
20:        $step \leftarrow resolution \times multiplier$
21:        $next\_knob \leftarrow next\_knob + step$
22:    end if
23:    return $next\_knob$                                         ▷ The new value is next_knob
24: end procedure
```

3.2.4 Runtime control algorithms

After the system identification process, the next step is to configure the memory subsystem knobs correctly. If the BER is too low, then the relaxed-accuracy is under exploited, whereas if the BER is too high, then the QoS requirement is not met. This calls for a runtime management of the BER knobs to meet the dynamic QoS requirement. A good algorithm should be able to adapt to changes in QoS quickly and maintain the target without overshoots and undershoots.



(a) PI Controller (b) PID Controller

Figure 3.6: Quality tracking of a streaming video using different controllers. The L1 data cache writes are exposed to errors based on the value of the BER knob.

Manual control

A manual algorithm to set the knob by observing the deviation from the QoS target is presented in Algorithm 1. The minimum change of the BER knob defines the granularity of the control. The algorithm proceeds in steps towards the target QoS with a scaling factor to respond quickly to large changes. The results of tracking using the manual algorithm are presented in [77]. Although the algorithm can reach the target quality, the settling time is very large. Moreover, the manual algorithm does not take into account the history of errors which occurred before.

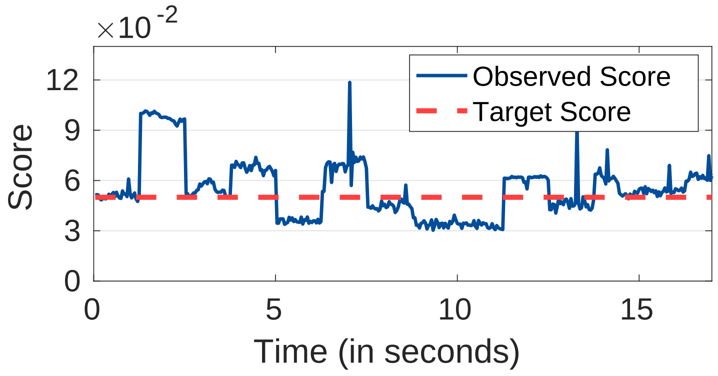
PI and PID control

Manual control cannot look into the history of errors and does not have any understanding of system dynamics. Moreover, it lacks any formal analysis to guarantees of stability. We have developed a formal control theory based approach in [77]. As a related work, we discuss the performance of a PI and PID controller here. Figure 3.6 (a) shows the quality tracking using a PI controller and 3.6 (b) shows the quality tracking for a PID controller.

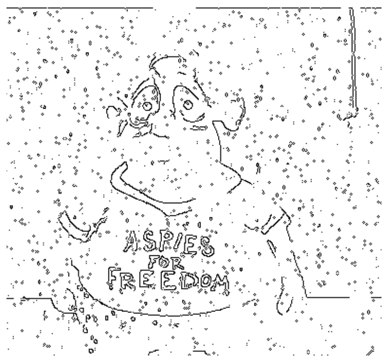
In order to assess the reaction to stochastic inputs, we first calculate a trailing moving average of the score at each frame based on the last six frames. Then, we evaluate the Error Sum of Squares (SSE), which is the sum of the squared differences between each observation and the moving mean corresponding to the frame. This metric gives us insight into the reaction of the controller to stochastic variations in the system's input. A low value of this metric would imply that the controller is stable, and is better capable of tracking the score. For the experiment shown in Figure 3.6, with a PI controller this value is 0.0141, and with a PID controller this value is 0.0187. Thus, we can see that PI is more effective in reacting to the stochastic nature of the changing inputs. Moreover, from the figure, we can see that PID tracking has a lot more ups and downs compared to the PI tracking, which affirms our analysis.

3.3 Challenge: Design memory systems that eases the programmer's burden and guarantees a desired quality of service (QoS)

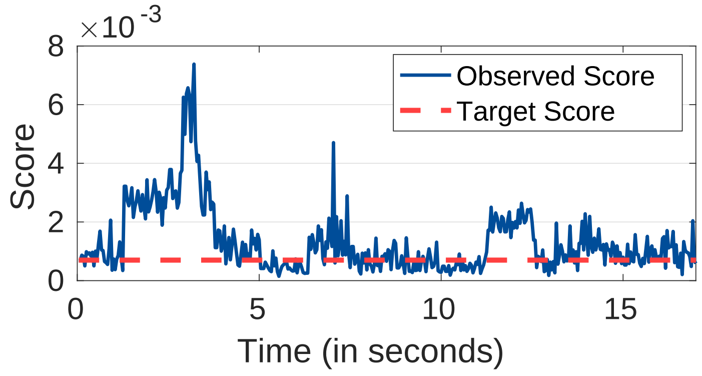
The QoS delivered via approximations is affected by multiple parameters including configuration of the memory hierarchy, the application input and temporal relationship between



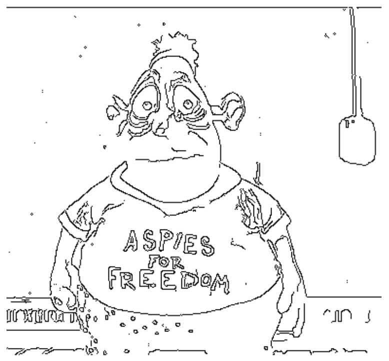
(a) BER = 1E-3, Expected Score = 0.05



(b) Score = 0.05



(c) BER = 1E-5, Expected Score = 0.001



(d) Score = 0.001

Figure 3.7: (a) and (c) shows variation of the quality of the edge detection in a video scene when the BER is constant. (b) and (d) shows a frame in the video with the expected score for the given setting of BER.

inputs, making manual tuning extremely challenging.

As an example we illustrate the challenge raised by varying application load that affects the delivered QoS. Traditionally, developers profile an application and determine the best possible knob for a given target and expect the same QoS for a given setting of the knob throughout the application's lifetime. However, fixed knob settings result in varying QoS in the face of changing workloads, as shown in Figure 3.7. It shows the quality of edge detection in a video composed of multiple scenes when the write bit error rate is kept constant. Figures 3.7 (a) and (c) show significant variations in quality across different inputs, demonstrating the drawback of an open-loop set-once-and-execute approach. This traditional open-loop approach suffers several drawbacks:

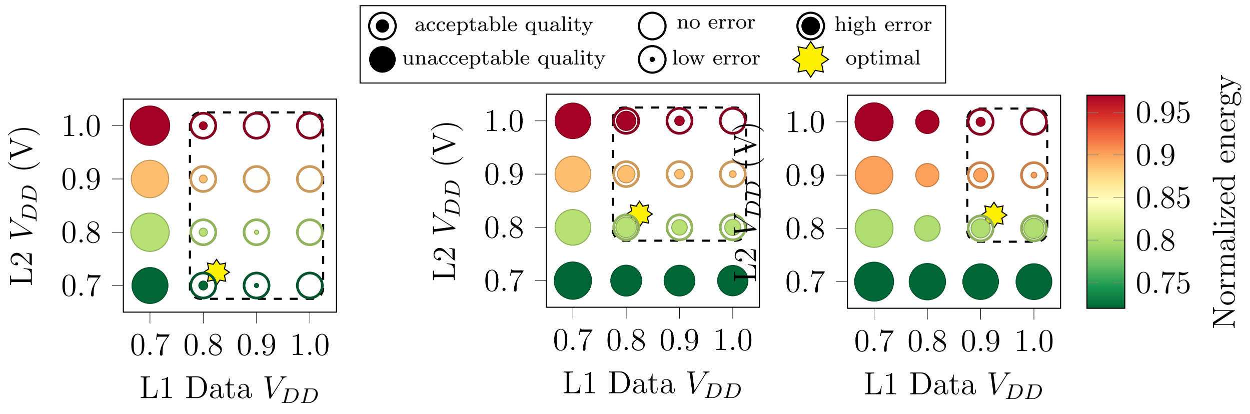
1. It is difficult to model an under-designed memory in order to measure the output accuracy at different settings. Temporal faults that are variability-induced, temperature-induced, etc. cannot be modeled easily. Unlike software approximation strategies that are easier to evaluate, hardware approximation requires rigorous runtime tuning. Application profiling to generate fixed parameter knobs cannot yield a consistent quality metric.

2. They make approximation decisions based on average or worst-case input behavior. These techniques rely on training with inputs that attempt to represent real-world inputs, which are difficult to achieve in practice. Laurenzano *et. al.* [63] have shown that accuracy of approximate programs depends heavily on program input.

3. Different components in the memory subsystem react differently to each workload due to differences in memory access patterns. Although techniques like memory profiling can help estimate the knobs for a given system, once the application is ported to a different system, the programmer needs to manually recalibrate the knobs in order to achieve desired quality.

The QoS achieved by a configuration of approximation knobs varies widely based on the application and current input. Even for a fixed workload (application and input), the configuration space grows exponentially with each additional knob (e.g., one knob = 4 states, two knobs = 16 states, three knobs = 64 states). Knobs are at least partially interdependent in a memory hierarchy: changing one knob affects multiple subsystems in ways that are complex to predict. (e.g., changing L1 V_{DD} introduces errors in L1, that propagate to L2.) This makes the configuration problem extremely challenging.

Consider the system equipped with an approximate memory subsystem in Figure 3.12. The application's source code is annotated with a quality monitor and is running on a system that supports reconfigurable approximate memory. The approximate memory subsystem consists



(a) Application A

(b) Application B, input B1

(c) Application B, input B2

Figure 3.8: Effect of configuration knobs on cache layers (L1 data cache and L2 shared cache) for two different applications (A and B), and different inputs within an application (B1 and B2 within B). The dot diameter indicates the number of errors (smaller is better: no dot means no errors), and the color indicates normalized total energy usage (normalized to 1V:1V case). The outer circle represents the quality-constraint which the system must meet. For knob configurations where there is no outer circle, the system fails to meet the quality constraint. Feasible operating regions that can achieve the target QoS are outlined in dashed rectangles, and the optimal setting is indicated by a star.

of three layers of hierarchy: SRAM L1 cache; SRAM L2 cache; and DRAM main memory. These memories have an 'exact' and 'approximate' region in which application data can be mapped. Approximation can be controlled at each layer of the memory hierarchy, and the approximation knob varies based on the memory technology: in the context of this work, we use the voltage level for SRAM cache and refresh rate for DRAM main memory. Each knob setting impacts the application QoS measured by the quality monitor. `malloc` calls from the application to the Linux kernel are modified by the developer to indicate which data can be mapped to approximate regions.

Figure 3.8 shows QoS observed for the system in Figure 3.12 for different applications and configurations of L1 and L2 approximation knobs.[2] The dots' size represents the QoS (i.e., number of errors, smaller is better). We observe the effect of configuration knobs on two applications:

- **Application A**: A memory write-read kernel that writes 512 64-bit numbers in the main memory and then reads the numbers from main memory. The QoS metric for this kernel is defined as the total number of bit flips that occur during the write-read cycle. The QoS and average energy for each knob configuration is shown in Figure 3.8a.
- **Application B**: The Canny-edge detection application as described in Section 3.5.4. The QoS metric for this application is the `rmse` (Root Mean Square Error) between the pixels of the approximate runs and the exact runs of the application. The QoS and average energy for knob configurations corresponding to two different inputs (i.e., scenes), B1 and B2, is shown in Figures 3.8b and 3.8c respectively.

We make three key observations: First, we observe that configurations achieving a target QoS vary both *within* and *between* applications. In Figure 3.8, we define a feasible region (dashed

[2]The DRAM knob is fixed for the sake of simplicity. The experimental setup for generating the illustrative example is the same setup as described in Section 3.5.4.

rectangle) by identifying the set of configurations that achieve acceptable QoS. Depending on the workload, the feasible regions of operation are different. The difference is seen in the varying bounding boxes of Figure 3.8a (Application A), and Figure 3.8b (Application B). Even within the same application, the acceptable regions of operation vary based on the dynamic inputs to the application at runtime as seen in Figure 3.8b (input B1) and Figure 3.8c (input B2).

Second, we observe that within the feasible regions, the achieved QoS varies across applications and inputs. In some cases, the outer circle and inner circle are well separated, implying that there is still room for approximation. However, in some cases, the inner circle is very close to the outer circle, implying that the QoS is reaching its threshold.

Third, we observe that the same configuration of knobs (e.g., L1:0.7 V, L2:0.9 V) yield different energy consumption with respect to different applications, and different inputs within an application. This results in varying optimal design points (marked with a star).

This simple example demonstrates that even for the same memory technology, it is hard to predict QoS and energy consumption when knobs are changed in only two layers of the memory hierarchy; i.e., the dynamics between the system and application vary both within and between applications. We expect that finding the optimal configuration for additional layers of a memory hierarchy or new memory technologies will only exacerbate these challenges, with current state-of-the-art techniques insufficient for determining the complex interactions of knob configurations for multi-level approximate memories.

3.3.1 Approximation: State-of-the-art

Open-loop Control

A common approximation strategy is to use design-time techniques to find optimal knob configurations [84, 59, 121, 88]. Based on the application profile, approximation knobs are determined before deployment and are expected to meet the QoS requirements throughout the application's lifetime. Designers must account for the worst-case scenario in an open-loop system, and therefore are unable to exploit the full potential of the approximation knobs at runtime. Additionally, application programmers are burdened with the task of setting memory approximation knobs through intensive profiling of the target workloads at design time [59, 156, 125]. Thus, open-loop control techniques are application-specific and not portable to new systems.

Model-based Closed-loop Control

State-of-the-art alternatives reconfigure approximation knobs using closed-loop controllers in order to address the lack of reconfiguration in open-loop systems [161, 109, 40, 85]. The controllers are generated based on a system model identified at design-time. Closed-loop control aims to alleviate the programmer's burden at design-time by using feedback at runtime. Design-time models consider the difficulty of specifying an under-designed memory's parameters by measuring the output accuracy in different settings. However, with the number of system parameters on the rise, system identification is becoming impractical for capturing the effects of one knob on another. Coordination in control theory requires a formal Multiple-input-multiple-output (MIMO) method, but designing a MIMO controller requires nontrivial design-time effort. Additionally, such models are rigid: models must be generated for each memory technology, with an underlying assumption that the system is available for observation ahead of deployment. Thus, closed-loop control techniques are also application-specific and suffer from significant design-time overhead.

3.3.2 Benefits of Model-independence

A static model identified during development does not take into account complex system dynamics (e.g., variability between applications). Self-learning intelligent agents without apriori knowledge are attractive candidates to find optimal solutions through runtime observation. Reinforcement learning [123] is prevalent in the field of self-learning agents, demonstrating success in decision-making for services such as recommendation engines and games. In this work, we utilize a model-free reinforcement learning approach to develop an approximate memory controller that can learn the behavior of knobs through runtime experience. Model-free control techniques can provide a general-purpose solution, independent of the application and system dynamics.

3.4 Case study 1: Self-Adaptive Memory Approximation

3.4.1 Detailed explanation of the formal control-theoretic approach for tuning memory approximation knobs

In this section, we show that despite the randomness of errors introduced into the execution of programs due to memory approximation, a formal control-theoretic technique can capture the system dynamics effectively. The effectiveness is demonstrated by quality control of the program even in the presence of stochastic (non-deterministic) behaviors.

In our case, the target *system* is composed of both hardware (with quality-configurable memory) and software (the application running on the hardware) as shown in Figure 3.9. Depending on the memory technology, there can be one or more knobs that can tune the degree of approximation in the memory subsystem. Some examples of control knobs are (1) Voltage in SRAM memory, (2) Refresh rate in DRAM, (3) Read/Write current amplitude in non-volatile memories like STT-MRAM. To make our experiments technology-independent, we use Bit Error Rate (BER) as our knob which is the probability with which each bit flips during a memory read/write operation. With a pre-determined frequency, the quality monitor routine measures the current quality of the output and compares it against the quality goal. A positive difference means there is still room to relax the reliability requirements of the memory and the controller accordingly sets a more aggressive knob setting. A negative difference indicates that the quality has degraded more than what was intended. The controller accordingly sets a more conservative knob setting.

System Identification Technique for Memory Approximation

The use of statistical or black-box methods to construct models of a system is known as *system identification*. A common practice to design a feedback system using a controller

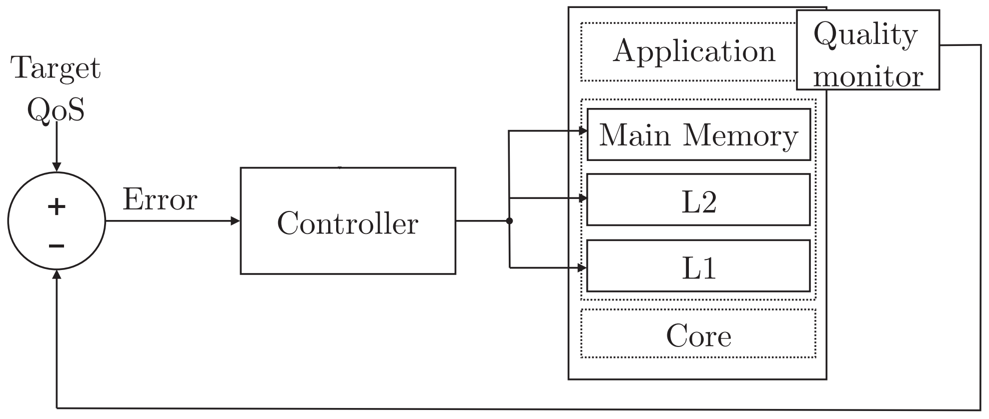


Figure 3.9: Closed loop approach for tuning memory approximation knob(s).

is to extract the dynamic model of complex systems through System Identification Theory [71, 64, 47]. By varying BER experimentally, data is collected to see the effects of BER on the measured output (score) for each memory component. A waveform with a step-pattern is applied at the inputs, and the output is continuously monitored. The monitoring process is repeated for several video streams and average output quality at each BER knob to used to model the relationship.

The relationship between control inputs and measured outputs can be specified using linear difference equations. A simple first order linear difference equation can be approximated as: $y(k+1) = ay(k)+bu(k)$, where a and b are parameters that can be identified using parameter estimation methods such as *least square regression*. This equation represents a first-order model where the next output depends only on the inputs and outputs from one time-unit in the past. The control inputs and measured outputs are used to estimate a linear-parametric model through a transfer function. If we are given a system with transfer function $G(z)$ and input $U(z)$, then the output of the system can be described as $Y(z) = G(z) \times U(z)$. Transfer functions have properties such as stability, steady-state gain, settling time, and maximum overshoot. We use the System Identification toolbox in MATLAB to estimate the transfer functions when BER control knob is applied on L1 data cache, L2 cache, and DRAM. The estimated models can be found in [78]. For all the components, we used a second order model with 2 poles ($n_p = 1$) and 1 zero ($n_z = 1$).

Application and Quality Metric

In this work, we target streaming applications which have temporal dependencies between consecutive inputs. The application is given a sequence of inputs, and the results from processing previous inputs can be used to adjust the knobs for successive inputs. The adjustment is possible due to the correlation of the inputs as well as the temporal behavior of the memory errors. We use canny video edge detection [18] as our case study. Edge detection is the process of identifying sharp changes in image brightness. For video processing, edge detection is often conducted on a frame-by-frame basis independently. Adjacent frames of a scene in a video have temporal similarity, and it allows the controller to adjust the quality based on the history of the system.

The quality measurement method is application dependent, and normally the programmer provides a software routine for measuring it at runtime. In many cases, this quality measurement would require computing the precise and approximate versions of the output for comparison [10, 9]. We use miss-classification error (ME) as our QoS metric, that is the ratio of the total number of pixels mistakenly classified as edge/non-edge to the total number of pixels in the frame. To evaluate the performance of the method, the settling time is computed. Settling time is the time for the output to reach the target value after a change in one of the inputs. We consider the output to have settled when it is within 2% of its target value.

Controller Design

Our system is a simple single-input-single-output (SISO) control system with bit error rate (BER) as a control input and edge detection miss-classification rate as measured output. We use a proportional-integral (PI) controller to control this system. The proportional term refers to the fact that the controller output is proportional to the amplitude of error signal, while *Integral* indicates that the controller output is proportional to the integral of all past errors [47]. The PI control law has the form:

$$u(k) = u(k-1) + (K_P + K_I)e(k) - K_P e(k-1) \tag{3.2}$$

Where K_P and K_I denote the coefficients for the proportional and integral terms, respectively. *Controller design* is a mature field which utilizes many tools that provide off-the-shelf controllers. We use MATLAB PID tuner toolbox to design our controllers. It is important to note that although derivative control law is helpful to add predictability to the controller, stochastic variations in the system output may cause inaccuracy in the controller. This issue becomes more severe in computer systems as they commonly have a significant stochastic component. Therefore, for computer systems PI controllers are preferred over proportional-integral-derivative (PID) controller [47]. PI control benefits from both integral control (zero steady-state error) and proportional control (fast transient response).

3.4.2 Comparison of self-adaptive with a manual calibration scheme

We evaluate the performance of the self-adaptive system by comparing it with a manual recalibration scheme. In the self-adaptive system, we define a target QoS accuracy while the middleware controls the knobs automatically. We expect (1) the system to adapt to changes in application input automatically, and (2) the settling time to be significantly less than the manual recalibration scheme. More details about the experimental setup and system dynamics can be found in [78].

To simulate the behavior of a system with approximate memory, we developed a Sniper-based [19] memory fault injector (FI)[3]. This FI can inject faults into read / write operations of the memory hierarchy (e.g., cache, TLB, DDR). To inject faults only into the non-critical data objects of the program, the source code of the program is annotated with `add_approx()` and `remove_approx()` methods to declare the address of those data objects in the program. These methods are called in the program at appropriate places and are captured by the FI. The FI records these addresses into a table. During the execution, it instruments all the memory accesses. If the virtual address of the access falls into the any of the given address boundaries, it attempts to inject a fault into the part of data referenced by that memory access. The controller, which is implemented in the middleware, is capable of receiving the

[3]Code repository at https://github.com/duttresearchgroup/memapprox-control.

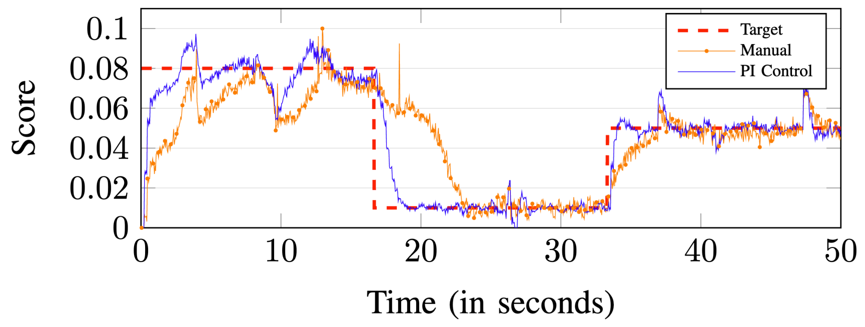


Figure 3.10: Runtime quality tracking for L1 Data cache write errors. Self-evaluation done every 5 frames.

results from quality monitors at runtime and set the read/write BER knobs in the simulation framework.

Control Experiment - Manual Recalibration

The manual scheme measures the difference between desired quality and current quality. If this difference is within -+10% it does not change the knobs. Otherwise, it changes the knob in one direction with fine-grained steps until the quality returns to the acceptable quality region. To recalibrate dynamically, it multiplies the steps with the *logarithm* of difference in quality.

Candidate Experiment - Self-adaptive System

Quality tracking is simulated using different control mechanisms in Figures 3.10 and 3.11 and the system's performance is evaluated. The figures show how the feedback loop operates in practice for video inputs. The red dashed curve shows the quality goal (or expected behavior). The blue curve shows achieved quality (or observed behavior) for PI control. The orange curve shows the achieved quality for manual control.

The blue curve in Figure 3.10 shows the performance of a self-aware system equipped with

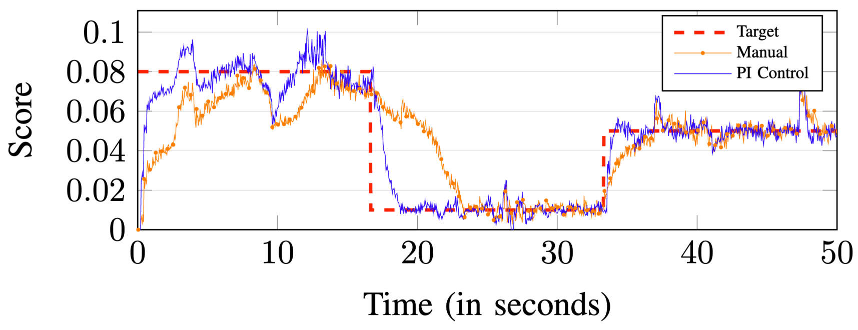


Figure 3.11: Runtime quality tracking for L2 cache write errors.
Average settling time for PI control = 2.04s, and average settling time for Manual control = 6.25s.

a PI controller in tracking target quality for L1 data (L1D) cache write errors. The tracking can be performed with an average settling time of 1.96*secs*. The orange curve shows the performance of a system with a manual recalibration scheme. In the manual scheme, the average settling time is 6.13 seconds, making the settling time 3× faster for the L1D writes when using the self-adaptive system. Figure 3.11 show a similar evaluation where we approximate the memory writes of L2 cache instead of L1D. For L2 write errors, the average speedup of settling time is also about 3×. In the best case, both L1D and L2 write errors have a speedup in settling time of 5×.

Our controller is capable of effectively tracking the quality when we approximate on-chip memory. However, there remain challenges when performing this approximation on DRAM. Our initial investigation suggests that DRAM is more tolerant to errors because, for the same score, the expected BER knob value of DRAM is almost 5× more than on-chip BER knob values. We note that our attempt to track the quality using our self-adaptive approach was unsuccessful; we hypothesize that our controller cannot take advantage of the temporal similarity across frames since DRAM access patterns are non-uniform, therefore, the quality monitor, being the only feedback is insufficient in controlling the target effectively. This highlights the need to investigate further opportunities for using proposed method in the context of off-chip memory accesses.

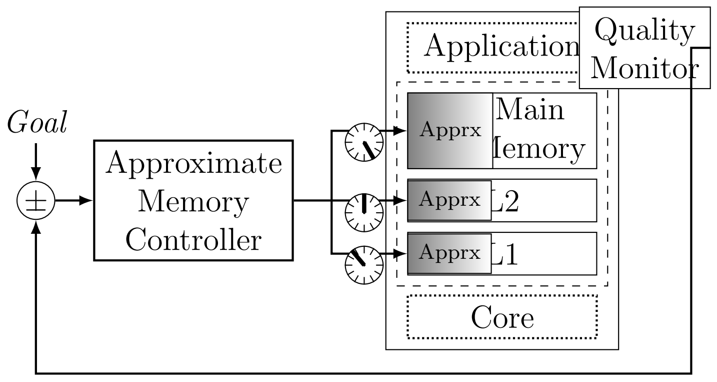


Figure 3.12: Runtime management of approximation knobs using output quality monitoring.

3.5 Case study 2: Self-Optimizing Runtime Manager

3.5.1 Overview of limitations in existing approximation techniques for full memory hierarchies

Consider the system shown in Figure 3.12. The memory hierarchy (L1 cache, L2 cache, and main memory) exposes tunable knobs (e.g., operating voltage for L1 and L2, data refresh period for main memory) that control the degree of approximation. Each knob introduces a new degree of freedom and increases the configuration space exponentially. Runtime optimization of multiple objectives is required to determine the desired system configuration, and is non-trivial. Researchers have proposed frameworks for exploring the configuration space at design-time and determining static optimal knob settings for an approximate memory hierarchy before deployment [156, 51]. More flexible solutions have been proposed to provide dynamic configuration of knobs at runtime, but require identifying workload-specific system dynamics at design-time [125, 77]. Requiring apriori knowledge assumes that the system and workload are observable ahead of deployment, and inherently limits the runtime adaptability. Determining the optimal knob configuration for unknown applications and new inputs at runtime is an extremely challenging decision process.

Table 3.2: Examples of approximate memory technology knobs. (*) used to evaluate SEAMS' performance in Section 3.5.5.

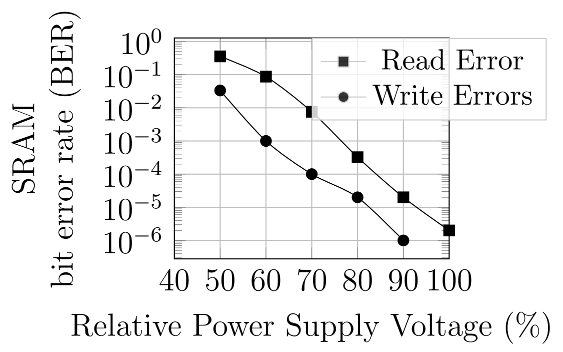
	Technology	**Memory Type**	**Technology Knobs**	**Knob objective**	**Reference**
Cache	SRAM	Volatile	(*) Operating Voltage (V_{DD})	Energy savings	ASPLOS'12[34], ESL'15[116]
	STT-RAM	Non-Volatile	Read Voltage (V_{read}) Write Pulse Duration (t_{write})	Energy savings	HPCA'11[117], CASES'15[110] ISLPED'17[87]
Main Memory	DRAM	Volatile	(*) Data Refresh Period (t_{REF}) Operating Voltage (V_{DD}) Row Activation Delay (t_{RCD})	Energy savings Reduce Latency	ASPLOS'11[70], ISLPED'14[24] MICRO'19[59]
	PCM	Non-Volatile	Data Comparison Write (Th)	Energy savings Increase lifetime	MICRO'09[101], MICRO'13[111]

Approximate Memory Technologies

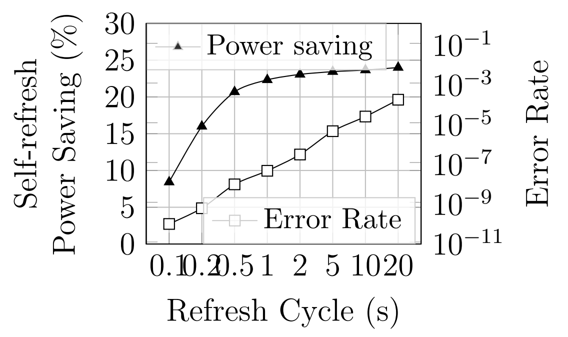
Table 3.2 summarizes some standard memory technologies and specifies associated approximation knobs and objectives. Each memory type exposes different technology knobs and allows room for optimization based on the technology knob. SEAMS methodology is technology-agnostic and can potentially leverage all of the technology knobs described in Table 1. However, for evaluation purposes, we focus on the most commonly used memory types: SRAM cache and DRAM main memory, marked with (*) in Table 3.2.

SRAM Voltage-scaling and power-gating lead to leakage energy savings in SRAM. When scaling the supply voltage (V_{DD}) in SRAM cells, read and write errors are dominant; hence hold failures are not considered. For experimentation in Section 3.5.5, we require a model that relates SRAM cell supply voltage (V_{DD}) to read/write error probability. Several power models exist in literature for different SRAM nodes (e.g., 90 nm [3], 70 nm [25], 28 nm [162, 7]). We use a model for a 6T SRAM at 65 nm node from [156] for comparison with related memory approximation work. Figure 3.13a shows the bit error rate for reads/writes corresponding to relative power supply voltage for the model under consideration.

DRAM Off-chip main memory in commodity systems uses DRAMs to store large amounts of data at high density, but comes with power and performance overhead compared to SRAM. Unlike SRAMs, DRAM cells need periodic refreshing (one row at a time) to retain stored data. During standard DRAM operation, the external memory controller is responsible



(a) Bit error rate for a 6T SRAM cell with varying V_{DD} values in 65nm. Data from [156].



(b) Error rate and power savings in self-refresh mode for different refresh cycles in DRAM array under 48°C. High refresh part is 1/4 of DRAM array. Data from Flikker [70].

Figure 3.13: Error and Power models used for (a) L1/L2 cache and (b) DRAM

for issuing refresh commands regularly to the DRAM. The frequent refresh operations (e.g., sending an AUTO-REFRESH command every 7.81 µs for refreshing the entire array in 64 ms) leads to power and performance overheads. Researchers have explored several knobs (e.g., operating voltage V_{DD}, row activation delay t_{RCD}) with different objectives (e.g., energy, latency) to reduce DRAM overhead. In this work we focus on optimizing energy, and use refresh-rate as the knob for experimentation.

Well-known work [102, 70] demonstrates significant power savings through memory approximation by using a feature available in mobile DRAMs known as self-refresh mode. In self-refresh mode, the DRAM array is periodically refreshed without any commands from the memory controller, even if the processor is in sleep mode. Low-power DRAM modules have a feature for further enhancing the self-refresh mode: Partial Array Self Refresh (PASR). PASR partitions the DRAM array in two separate regions, refresh and no-refresh. The partitioning allows refresh to be limited to the portion of the memory that is being used to store data. DRAM cells that are not refreshed will lose data in PASR. Liu *et al.* in Flikker [70] propose an approximation method by extending PASR to support multiple DRAM regions with different refresh rates, providing reduced reliability of data in part of memory instead of complete data loss. We use the Flikker DRAM model as our approximate DRAM memory technology with configurable refresh rate for the non-critical section of the memory array. Flikker is also used by a state-of-the-art memory approximation manager [156], allowing for comparison. Although the exact refresh rate required for reliable operation at runtime is effected by temperature, device capacity, and cell layout [53], these variations are not considered in this work.

Configuring Approximate Memory

Several methods have been proposed to tune memory approximation knobs. Table 3.3 identifies the most recent and relevant research, and positions SEAMS with respect to these prior works. We define self-adaptivity as the ability to adapt to user-specified application goals or system constraints (e.g., increased target QoS). We define self-optimization as the ability to find desirable system configurations given a fixed goal in the face of external disturbances (e.g., a scene change). Manual methods rely on designer expertise to optimize approximation knobs (e.g., t_{REF} in DRAM). In EDEN [59], Koppula *et al.* show the effectiveness of manual tuning for neural networks, which have an intrinsic capacity of tolerating errors in memory accesses. EDEN uses approximate DRAM to reduce energy consumption and increase the performance of DNN inference. EDEN is limited to machine learning workloads and does not apply to a multi-level memory hierarchy. The absence of a runtime quality monitor in EDEN prevents dynamic reconfiguration of the approximation knobs (e.g., row activation delay t_{RCD}, operating voltage V_{DD}).

Maity *et al.* [77] propose a solution to maintain a quality target at runtime using classical control theory. Quality configuration tracking is modelled as a formal quality-control problem, and black-box modelling is used to capture memory approximation effects with variations in application input and system architecture. However, this scheme assumes only one level of the memory hierarchy is tuned at runtime and fails to address the problem of coordination between multiple knobs.

In AdAM [125], Teimoori *et al.* investigate memory approximation by managing approximation knobs across the memory hierarchy. AdAM solves a design-time ILP optimization problem and uses a runtime algorithm to adapt to new tasks by re-estimating the execution time. Although optimization techniques are a natural choice for simple architectural tuning, the lack of a feedback mechanism makes them too rigid for any sort of adaptivity (e.g., unknown inputs, disturbance from other applications). Their use-case only addresses a two-layer memory hierarchy, with an on-chip STT-RAM and an off-chip PCM Main Memory, and the design-time algorithm is technology-dependent.

In DART [156], Yarmand *et al.* propose a framework for a three-layer memory hierarchy (SRAM L1, SRAM L2, and an off-chip DRAM) without any technology-specific assumptions. DART uses a branch and bound algorithm to consider all possibilities at design time, and creates a search tree to perform error probability analysis. Although DART considers the full memory hierarchy, it requires the programmer to: (1) analyze the program during design time, (2) generate a memory profile for each application that would run on the system, and (3) estimate the worst-case probability of errors that would occur due to under-designed memory. Therefore DART requires apriori knowledge of the application and assumes that the system is available for full observation before deployment.

In the broader scope of runtime resource management, machine learning approaches have been explored. Researchers have investigated the feasibility of machine learning methods for quality configuration in the approximation domain [81, 82]. However, conventional machine learning methods require extensive training to learn the correlation between the system's inputs and outputs. Static models that are defined ahead of deployment fail to handle new situations outside of expected behavior. Online learning methods aim to address this issue and have shown promising results for resource management [31, 74].

Table 3.3: Memory approximation approaches and the key challenges addressed ($*$ = uniquely addressed by SEAMS).

Features	**EDEN** [59]	**Control Theory** [77]	**AdAM** [125]	**DART** [156]	**SEAMS**
Technology Independent		✓		✓	✓
Memory Hierarchy			✓	✓	✓
Application Agnostic					$*$
Coordination					$*$
Self-Adaptivity		✓			✓
Self-Optimization					$*$
Model-Independence	✓			✓	✓
Real System Evaluation	✓				✓

SEAMS incorporates the features highlighted in Table 3.3 using online learning methods. The SEAMS approach improves upon prior work by eliminating design-time modeling, being memory technology-agnostic, and coordinating multiple knobs at runtime to exploit approximation for multi-level memory hierarchies; enabling quick adoption of approximation for diverse platforms.

3.5.2 Explanation of the proposed self-optimizing runtime manager, SEAMS

Figure 3.14 from [75] presents the SEAMS realization of the logical architecture described in Figure 3.12, consisting of the following components: ① a hardware platform with a processing unit, cache subsystem, and main memory. This hardware controls the degree of approximation at each memory layer by configuring the specific technology knobs available on the platform. Examples of technology knobs are in Table 3.2. The processor core contains special registers to set knobs (e.g., L1 V_{DD} updated from 0.7 V to 0.8 V) through special instructions. In our current RISC-V realization of the processor, we deploy unused control and status registers (CSRs) for this purpose. ② Custom ISA extensions. Custom instructions

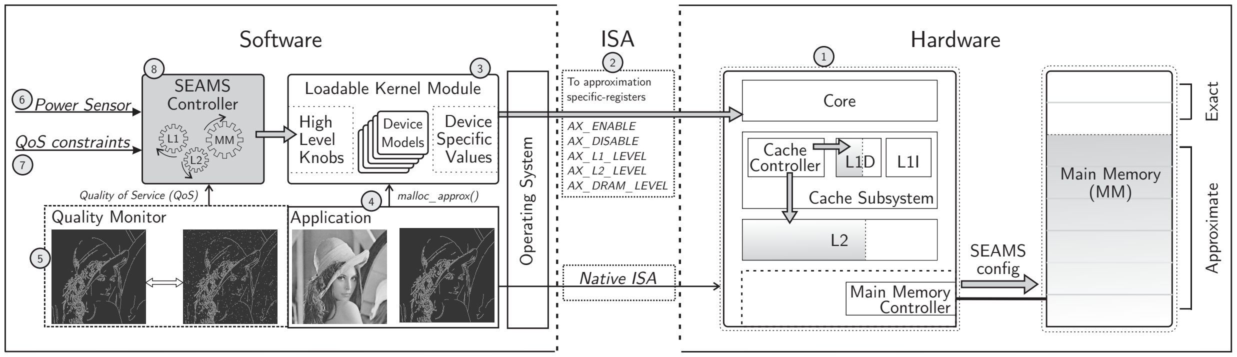


Figure 3.14: Overview of SEAMS system architecture.

write to CSRs to form an extension of the processor's ISA (RISC-V in our implementation), and are used to manage approximate elements at runtime. Truffle [34] is another example of a microarchitecture design that efficiently supports ISA extensions for disciplined approximate programming. Instructions supported through our CSRs include `AX_ENABLE` to enable approximation, `AX_DISABLE` to disable approximation, `AX_L1_LEVEL` to set the technology-specific knob for Level 1 cache, `AX_L2_LEVEL` to set the knob for Level 2 cache, `AX_DRAM_LEVEL` to set the technology-specific knob for DRAM. ③ A loadable kernel module. The module maps the high-level knob values (e.g., low approximation) to technology-specific knob values (e.g., V_{DD}). For new technologies, the module must be updated to reflect the available actuation knobs (e.g., available write pulse duration (t_{write}) for STT-RAM). The kernel module also allows applications to indicate which parts of the application's virtual memory can be placed physically in the approximate regions (explained further in Section 3.5.4). ④ The user application, specifying the non-critical sections of the data using a `malloc_approx()` call. ⑤ A quality monitor that computes the QoS periodically at runtime and reports it to SEAMS. ⑥ Power sensors that measure the current system power. ⑦ Expected QoS specified by the user. Expected QoS can be updated at runtime to adapt to different system objectives (e.g., a strict quality constraint provides more accurate execution, whereas a relaxed quality constraint provides energy savings). ⑧ The SEAMS controller. The controller is the final component of the architecture and is responsible for runtime control of the memory approximation knobs.

The SEAMS controller is a model-free decision-making mechanism for tuning configurable approximation knobs throughout the memory hierarchy. SEAMS follows the observe-decide-act (ODA) paradigm: the environment is observed through sensors during normal execution, and the controller is periodically invoked in order to (re)configure the system using knobs. We design our decision-making logic by first defining our problem as a Markov Decision Process [152]: (S, A, P_a, R_a), where S refers to state space, A refers to action space, P_a refers to the transition probabilities from $S \rightarrow S'$ given action A, and R_a refers to the expected reward for selecting action a in state s. As is common in real systems, we assume system dynamics are unknown and can change continuously. To address this well-known problem, we apply an appropriate established reinforcement learning solution: temporal difference (TD) learning [122].

Design Methodology

Our goal is to design a controller that coordinates each layer of a unified 3-layer memory hierarchy to achieve acceptable application QoS while minimizing energy consumption. First, we must define the structure of our environment.

State Space (S) The state is a representation of the current system under control. In SEAMS, we define a state vector that consists of high-level approximation settings (e.g., no/low/medium/high approximation) of each memory layer, as well as the current QoS error. The quantitative definitions of the different approximation settings are in Table 3.4. The state vector can be summarized as:

1. L1D: current level 1 data cache configuration
2. L2: current level 2 shared cache configuration
3. Main memory: current main memory configuration

4. Discretized QoS error ($Q_{threshold} - Q$), where Q is the measured QoS, and $Q_{threshold}$ is the provided constraint

The state includes the current knob settings, as well as how effectively they are achieving the QoS requirement set by the application. This allows us to translate the dynamics between application behavior and hardware configuration. The QoS error is normalized to the worst-case QoS value (max_Q) to make SEAMS portable across applications, and high-level knobs allow SEAMS to be independent of memory technologies.

Action Space (A) The action space contains all possible operations the controller may take to configure the system each time the controller is invoked. The SEAMS action vector consists of the relative changes to the high-level knobs for layers in the memory hierarchy:

1. L1D: Increase/Decrease/No change
2. L2: Increase/Decrease/No change
3. Main memory: Increase/Decrease/No change

Initially, the SEAMS controller policy does not have any information regarding what actions are desirable, and must discover which actions yield the maximum reward in each state via exploration (e.g., when there is no QoS constraint, actions which decrease power yield the maximum reward).

Reward (R) The reward provides immediate feedback to the controller on how the previous state-action pair helped achieve the system goal. In our case, the goal is to find the optimal configuration corresponding to minimum energy with acceptable QoS. Power is used as a direct proxy for the higher-level optimization goal (energy), and instantaneous power is integrated over time to compute the total energy. We define the reward in an unconstrained system as

$$\begin{aligned} reward_P &= 1 - \frac{Power}{max_{Power}} \\ reward_P &\in \{x \mid 0 \leqslant x \leqslant 1\} \end{aligned} \tag{3.3}$$

where $Power$ is measured power, max_{Power} is the power consumed when the approximation is disabled at all layers of the memory hierarchy, and $reward_P$ is the reward obtained in terms of optimizing power consumption. This function represents a power optimization objective with a target power of zero. In an unconstrained system, operating at the highest power yields no reward, while operating at zero power yields the maximum reward. However, we must constrain the total reward in order to account for the quality threshold.

The controller should take actions that minimize the number of violations of the quality constraint specified by the application developer. The reward with respect to quality is calculated as

$$\begin{aligned} reward_Q &= -\frac{Q - Q_{threshold}}{max_Q} \\ reward_Q &\in \{x \mid -1 \leqslant x \leqslant 1\} \end{aligned} \tag{3.4}$$

where $reward_Q$ is the reward obtained by staying within the quality constraint. In case of violations, $reward_Q$ is negative, indicating an action that led to a QoS violation.

Finally, the reward R is calculated from the $reward_P$ and $reward_Q$ and measured by the controller as

$$R = \begin{cases} reward_P, & \text{if } Q \leqslant Q_{threshold} \\ reward_Q, & \text{otherwise} \end{cases} \tag{3.5}$$

This reward function effectively minimizes power (by maximizing $reward_P$) while the QoS requirement is being achieved ($Q \leqslant Q_{threshold}$), otherwise it minimizes error (by maximizing

$reward_Q$). Note that quality metric in case of approximation is quantified by the number of errors (lower Q is better).

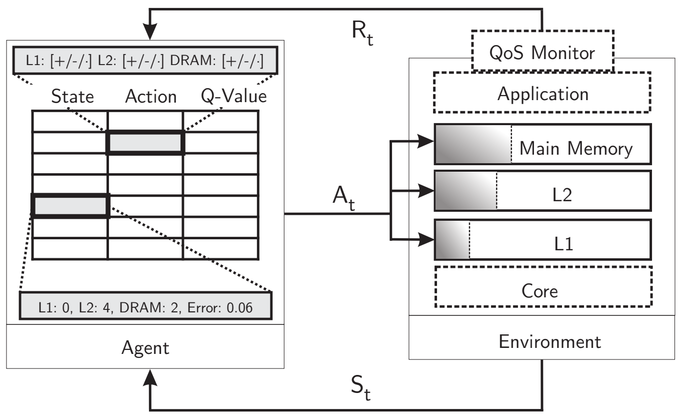


Figure 3.15: SEAMS taking actions against the environment, and the environment returns observations (updated state) and reward.

Approx. degree	L1 (SRAM)	L2 (SRAM)	Main memory (DRAM)
No	1V	1V	0.1s
Low	0.9V	0.9V	1.0s
Medium	0.8V	0.8V	5.0s
High	0.7V	0.7V	20s

Table 3.4: Example instance of memory hierarchy knob used to evaluate SEAMS

SEAMS Controller: Model-free Control

Given the definition of the environment and goals, we simply need to implement a decision-making mechanism (SEAMS) to find the optimal policy. The policy is defined as a state-action value function, and captured in a table that stores all the state variables and possible actions. Initially, the SEAMS controller does not have any information regarding the environment and explores the state-space by taking purely arbitrary decisions (actions). SEAMS uses *temporal-difference* (TD) learning [123] to learn directly from experience without a model of the environment's dynamics. Figure 3.15 shows the logical structure of the SEAMS controller and its relation to the environment, i.e., system under control. The controller interacts with the environment through actions, and the environment provides rewards and updated state information to the controller. Actions that lead the system to optimize power without violating quality constraints are rewarded well.

Q-learning [150] is a popular TD control algorithm. Q-learning aims to learn a state-action value function, Q, which directly approximates $q*$, the optimal state-action value function. A variation of Q-learning combines eligibility traces to obtain a more general method that may learn more efficiently. Eligibility traces look backward to recently visited states and act as short-term memory. This algorithm, where Q-learning is combined with a backward short-term memory using eligibility traces, is known as TD(λ) [122]. SEAMS uses the TD(λ) algorithm to update and optimize the approximation management policy continuously throughout runtime. A Q-table is populated with the Q-value of each state-action pair (Figure 3.15). The controller is invoked periodically and performs the following steps during each invocation:

1. Measure the power and QoS to evaluate the reward R
2. Update the table (Q values) based on reward R
3. Sense the current approximation levels and QoS to determine the current state S
4. Given the current state S and updated Q values, select next action A

The detailed algorithm is outlined in Algorithm 3.

line 1: The dilemma presented during any controller design is determining control parameters, whether the implementation uses classical control theory or reinforcement learning. In the TD(λ) algorithm, learning parameters are interpretable, and can be selected several ways, e.g., using designer intuition or empirical observation. In our case we determine learning parameters (α=0.6, γ=0.1, and λ=0.95) empirically by simulating our control logic on system traces for `canny`. No matter the controller deployed, these parameters must be determined. However, we define our control logic in such a way that the values apply to the *type* of control (i.e., memory approximation knobs), as opposed to the *application* under control (i.e., edge detection). Thus, the parameter values remain relevant across new applications.

Algorithm 3 TD(λ) algorithm [123] for determining SEAMS policy.

```
1: Algorithm parameters: step size, discount factor, trace decay $\alpha, \gamma, \lambda \in (0, 1]$  ▷ Empirically determined parameters
2: Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$ ▷ Initialize action-value function (Q)
3: for each episode do
4:     $E(s, a) = 0, \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$    ▷ Eligibility trace (E) is initialized to 0
5:     Initialize $S, A$
6:     for each step of episode do
7:         Take action $A$, observe $R$, $S'$    ▷ Take selected action, observe reward ($R$) and next state ($S'$)
8:         Choose $A'$ from $S'$ using policy derived from $Q$    ▷ Choose next action using the policy
9:         $\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$    ▷ TD error = Expected value - Current value
10:        $E(S, A) \leftarrow E(S, A) + 1$    ▷ Increase eligibility trace by 1 for visited state-action pair
11:        for each $s \in \mathcal{S}, a \in \mathcal{A}(s)$ do
12:            $Q(s, a) \leftarrow Q(s, a) + \alpha\delta E(s, a)$    ▷ Update Q in the direction of TD target
13:            $E(s, a) \leftarrow \gamma\lambda E(s, a)$    ▷ Decay the eligibility of previously visited state-action pairs
14:        end for
15:        $S \leftarrow S'$; $A \leftarrow A'$    ▷ Update the state ($S$) and action ($A$)
16:    end for
17: end for
```

line 2: The value function $Q(s, a)$ considers transitions from state–action pair to state–action pair, and indicates how desirable it is to perform a given action in a given state. The value function is initialized arbitrarily as the state-space is yet to be explored.

line 3: During the policy initialization phase, the same sequence of inputs are repeatedly used to explore the state-action pairs and populate the value function $Q(s, a)$. This set of inputs form an *episode* which group the controller-environment interactions into smaller sequences. After the policy is initialized and SEAMS is exposed to new inputs, then inputs are not grouped anymore, and this outer for loop will be skipped. During the policy initialization phase, the processing of 100 steps are grouped into an episode, where each step corresponds to the actuation frequency. If the actuation frequency is 1, then each input is considered as a step. A detailed analysis of the best size of each step is presented in Section 3.5.5. At the end of each episode, some information is updated, which is explained in lines [4-5].

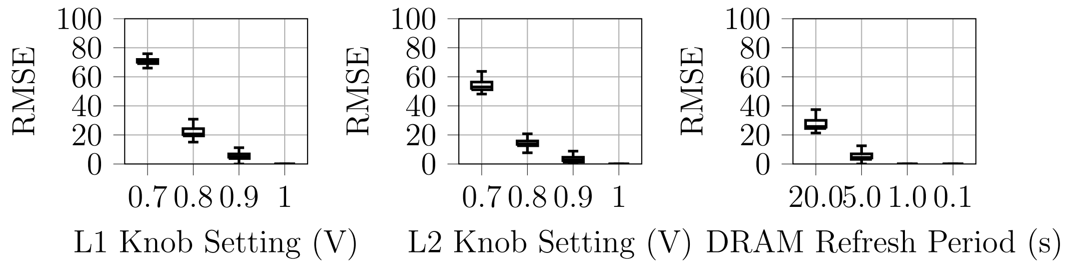
line 4: For all the state-action pairs, the eligibility values are initialized to zero. The eligibility matrix estimates the degree that previously-visited state-action pairs contribute to the latest reward. Staler state-action pairs have a low eligibility value, and recent state-action pairs have a high eligibility value.

line 5: Initially we start with the state of no approximation and the best action for the current state is selected based on the value function $Q(s, a)$.

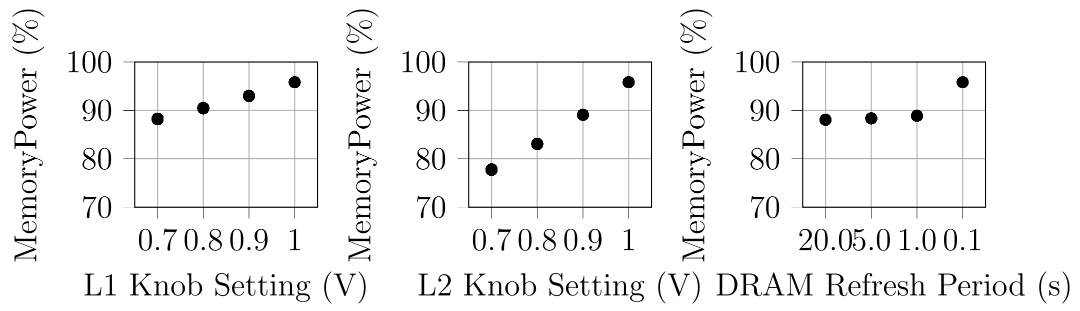
line 6: This loop is executed for each step corresponding to a new input.

line 7: Set the appropriate knob values based on the selected action A. The controller yields to normal execution for a set period of time, and is invoked again after the invocation period expires. Upon invocation, the new state S' and the reward R are determined based on measured behavior.

line 8: The next action A' is selected based on the new state S' and value function $Q(S', A*)$.



(a) RMSE Sensitivity analysis.



(b) Memory Power Sensitivity analysis.

Figure 3.16: Sensitivity analysis of memory configuration knobs on QoS (RMSE, top) and memory power (normalized to L1:1 V, L2:1 V, DRAM:0.064 s) for `canny`.

line 9: The TD error is calculated to determine how well the value function predicted the actual reward.

line 10: The eligibility value of the visited state-action pair is updated.

line 11: This loop visits the entire state-action space to update the value function Q.

line 12: The Q-value for all state-action pairs is updated based on the latest reward, TD error, and eligibility value.

line 13: All eligibility values are decayed.

line 15: The current state and action S, A are updated based on the observed state S' and selected action A'.

3.5.3 Discussion of SEAMS' key features

Figure 3.14 outlines the SEAMS system architecture, with the SEAMS controller (⑧) responsible for runtime control of the hardware memory approximation knobs. Our implemented environment consists of a unicore RISC-V processor with a three-layer memory hierarchy: L1 SRAM data cache, L2 SRAM shared cache, and DRAM main memory. SEAMS is implemented in software and runs in userspace. A loadable kernel module translates the device-specific translations from high-level configurations (e.g., low approximation) to technology-specific values (e.g., 0.9 V for SRAM caches).

The state vector S is made up of discrete integer values that represent (1) high-level configurations corresponding to memory layers, and (2) the QoS error. The L1 and L2 voltage levels (V_{DD}) are between 0.7-1.0 V in increments of 0.1 V [156]. The main memory refresh periods are 0.1 s, 1 s, 5 s, 20 s [70]. The QoS error is normalized and discretized into 16 buckets of step size $log_2 16$. The inclusion of the QoS error in the state differentiates desirable actions for the same voltage level, depending on the QoS error explicitly.

The action vector A contains a field for adjusting each of the L1, L2, and main memory knobs. The knob values are voltage levels for L1 and L2 cache, and refresh periods for DRAM main memory. Knob actions only consist of increment, decrement, or remain the same. Figure 3.16 shows the sensitivity analysis of the approximation knobs on the QoS and power outputs. The analysis is performed using the experimental setup described in Section 3.5.4 and the Canny edge detection application (described in Section 3.5.4). We make three key observations. (1) As we move up the memory hierarchy (i.e., from L1 to L2 to main memory), the quality is less affected by higher degrees of approximation. (2) The contribution to power from individual levels of the hierarchy varies. Although main memory techniques can save around 23% DRAM power, when the full memory hierarchy is considered, DRAM power savings saturate at 12%. (3) Four knob values capture the range of power/quality tradeoff while keeping the state-space manageable. We conclude that four knob values for each level provides sufficient control to achieve our goal.

Reward R is calculated based on Equation 3.5. SEAMS uses software sensors to determine the application's output quality and calculate the reward. Although the quality metric is domain-specific and generated by a quality monitor, normalizing the value keeps SEAMS domain-agnostic. We update Q values using the reward as specified in Algorithm 3.

We deploy a hardware platform that emulates the effects of approximation in order to demonstrate the efficacy of SEAMS to coordinate approximation knobs in the memory hierarchy. The effect of approximation knobs in each layer in the memory hierarchy is determined using known models [156, 70].

3.5.4 Coordinate runtime decisions for interdependent knobs and subsystems

Figure 3.17 shows an overview of our evaluation platform. Prior work on configurable memory approximation uses simulation to evaluate methods [78]. Our desire is to practically evaluate a full system that captures dynamic workload and operating system behavior, which leads us to a hybrid setup with hardware in the loop. We implement SEAMS in software running on Openpiton [11], an open-source framework designed to enable scalable architecture research prototypes. We use Openpiton with a single Ariane [160] core, a 64-Bit RISC-V core capable of running Linux. Ariane implements the 39-bit page-based virtual memory scheme SV39 and boots Linux on a single core on an FPGA. The platform is synthesized on a DIGILENT NexysVideo board with a Xilinx Artix-7 FPGA(XC7A200T-1SBG484C). The parameters used to synthesize the system are summarized in Table 3.5.

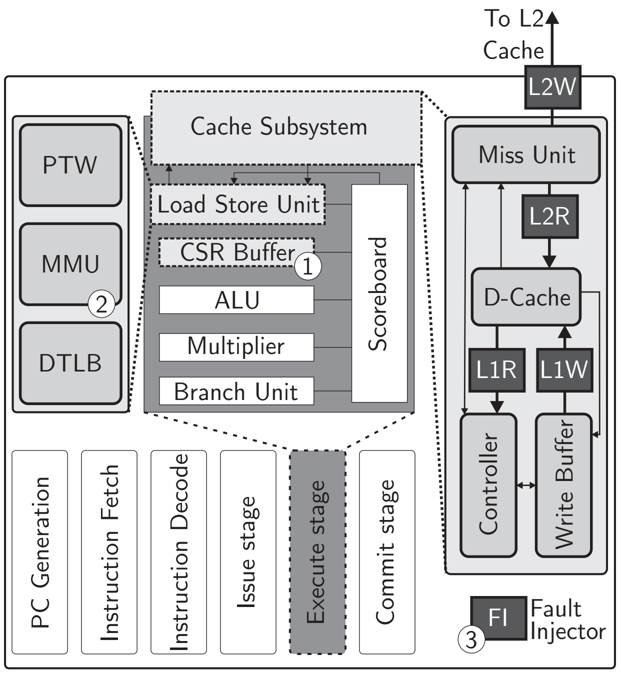


Figure 3.17: Modification of Ariane RISC-V core to emulate on-chip approximate memory. ① Addition of new CSRs to communicate with SEAMS kernel module. ② Modification of address translation logic in Memory Management Unit (MMU) to generate `approx` signal. ③ Fault injectors that introduce errors in the memory bus.

Modifications to RISC-V Core

We further modify the Ariane core to support fault injection throughout the memory hierarchy. The synthesized core, running on the NexysVideo board, does not support real knobs for approximation. Instead, we rely on existing works that map device-specific approximation knobs to observable bit error rates [156, 70], and introduce bit errors through fault injection to emulate the effect of approximation knobs.

SRAM Fault Injection The RISC-V specification defines separate addresses for Control and Status Registers (CSRs) associated with each hardware thread [149]. Unused CSRs are utilized by the kernel to communicate information required for the configuration of the approximation knobs. Additional CSRs are denoted with ① in Figure 3.17. In particular, the following information is stored in CSRs:

1. L1 data cache Read and Write bit error rate
2. L2 shared cache Read and Write bit error rate
3. Starting and ending physical address of the non-critical memory segment

The bit error rates correspond to specific memory nodes and are translated from technology-

Table 3.5: System configuration used for SEAMS evaluation.

Component	Configuration
Cores	1
TLBs	Number of entries (16)
L1 D-Cache	Number of sets, ways (**16kB, 4-way**)
L2 Cache	Number of sets, ways (**64kB, 4-way**)
Floating-Point Unit	Present
Main Memory	Onboard (512MB DDR3, 800 Mbps data rate)
Clock frequency	30 MHz
Cache Replacement Policy	**L1**: Pseudo Random (using LFSR) **L2/TLB**: Pseudo Least Recently Used (LRU)

specific values described in Section 3.5.1. The knob settings from the CSRs are propagated to the ② Memory Management Unit (MMU), where address translation takes place. The MMU uses this information to generate an additional `approx` bit along with the `index` and `tag` bits to indicate if an address is in the valid range of approximation. The `approx` bit generation is repeated whenever a virtual address is converted to a physical address. The cache controller uitilizes the `approx` bit in conjunction with the CSRs to determine bit error rate, control the degree of approximation, and contain it to the non-critical parts of the application data. A Fault Injector (`FI`) module emulates the effects of approximation by introducing bit flips on the memory bus. Four `FI` modules are instantiated in the cache subsystem as shown in Figure 3.17 ③. The `FI` modules generate a bit-flip mask for each memory access using a Linear-Feedback Shift register (LFSR) that introduces randomness in the injected errors.

The `FI`s are located in (1) Data Cache Memory emulating the bit flips corresponding to L1 data reads and L2 reads, (2) Write Buffer emulating the bit flips corresponding to L1 data writes, and (3) Miss Unit emulating the bit flips corresponding to L2 writes.

DRAM Fault Injection DRAM cells store data in capacitors that lose charge over time. In order to keep the data consistent, the DRAM cells are periodically refreshed. DRAM cell strength is non-uniform due to manufacturing variability, i.e., some DRAM cells lose charge faster than others. The number of bit-flips in DRAM increases as the refresh period increases due to a higher number of DRAM cells losing charge before they are refreshed. The number of bit-flips is also affected by DRAM data reads and writes. Therefore, implementing a `FI` module for DRAM requires tracking faulty DRAM cells for each refresh-rate knob and the hold times of the data in each DRAM cell. Given the DRAM size, maintaining this information requires a lookup table of impractical size on an FPGA. The lookup table would also introduce considerable latency to DRAM accesses. There are multiple ways to emulate a variable refresh knob in DRAM hardware, e.g., using precharge-powerdown and ACT-

Table 3.6: Applications used for SEAMS' evaluation along with their inputs and QoS.

Application	Domain	Input Size	Quality Metric
canny [18]	Image-Processing	352x288 (Grayscale)	Image Diff (RMSE)
k-means [157]	Machine Learning	426x240 (RGB)	Image Diff (RMSE)
blackscholes [157]	Finance	4K entries	Avg. Relative Error

PRE for short wakeups, or row-granular refresh. However, we believe software emulation is sufficient to demonstrate the efficacy of SEAMS. To emulate DRAM errors, we implement a software-based `FI` for DRAM. Initially, a map `DRAM_MAP` of faulty DRAM cells[4] for the maximum refresh period (20 s) knob is generated randomly using a uniform distribution. The faulty DRAM cells for higher refresh rates are a subset of `DRAM_MAP`. The data being loaded into the DRAM is modified using the `DRAM_MAP` and the current refresh rate knob. The exact read and write accesses to the DRAM are not impacted.

User application and QoS metric

The SEAMS methodology is well suited for a large class of memory-intensive workloads (e.g., video processing, machine learning). Table 3.6 summarizes the applications used for SEAMS' case study. (1) `Canny` edge detection [18] algorithm operates on a video stream and marks the edges in each frame. (2) `k-means` is a machine-learning application [157] that partitions 3-dimensional input points (RGB pixels) into six different clusters. (3) `blackscholes` [157] is a financial analysis application that solves partial differential equations to perform price estimations.

Each application's source code is modified to indicate non-critical data elements. Several techniques have been have been proposed [144, 39] to systematically analyze and report how different parts of applications are affected by errors. Depending on the application, there are one or more candidate data segments (e.g., image data, video data, signal data) for

[4]The bit error rates are sourced from [70] for temperatures under 48°C.

accuracy/energy tradeoffs. We identify these segments in the source code, and replace their `malloc()` calls to the kernel with `malloc_approx()` calls. For `canny`, the image buffer is marked as non-critical. For `k-means`, the data structure for the image buffer is modified to separate the non-critical pixel data, and raw pixel information is converted from float representation to unsigned char representation since each pixel value lies between 0 and 255. For `blackscholes`, the buffer data structure is left unmodified: the non-critical approximate memory consists of a buffer of floats. Errors can impact the bits differently, and in case of extreme approximation may produce relative errors of 100 %. The `malloc_approx()` calls are intercepted by a custom Linux kernel module, described in next section.

In addition to specifying the non-critical data elements, a quality-monitor specific to the application domain is required. The quality monitor is a lightweight software routine invoked to evaluate the application QoS and calculate the reward. The QoS metric indicates the quality degradation caused by the configuration of approximation knobs. Typically, application developers generate a software routine that is capable of measuring the quality at runtime. In `canny`, the QoS is determined by evaluating the *Root Mean Square Error* (RMSE), which is the mean of pixel differences squared between an exact result and an approximate result. For `k-means` and `blackscholes` the quality monitors are RMSE and *Average Relative Error*, used directly from AxBench [157]. The quality monitor software routine is invoked at runtime, and the result of an exact run of the application is compared with an approximate version [10, 33]. The quality evaluation is not repeated for every input so that the benefits of approximation can justify the overhead. Depending on the status of learning, the frequency of quality evaluation should be adapted. A detailed overhead analysis is presented in Section 3.5.5. If additional cores are available, ground truth comparison can be performed in parallel. In our unicore setup, ground truth comparison incurs unavoidable overhead during the initial exploration phase.

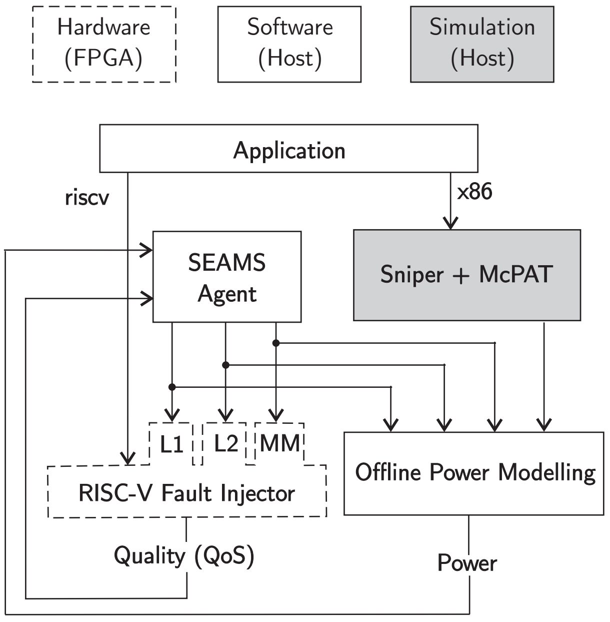


Figure 3.18: Tools flow for experimental setup. The RISC-V Fault injector (shown in dashed lines) allows us to emulate faults using different models while considering a full system running an operating system. The Sniper+McPat simulations determine power consumption without approximation, which is scaled based on technology-specific models for approximation.

Kernel Support for Approximation Knobs

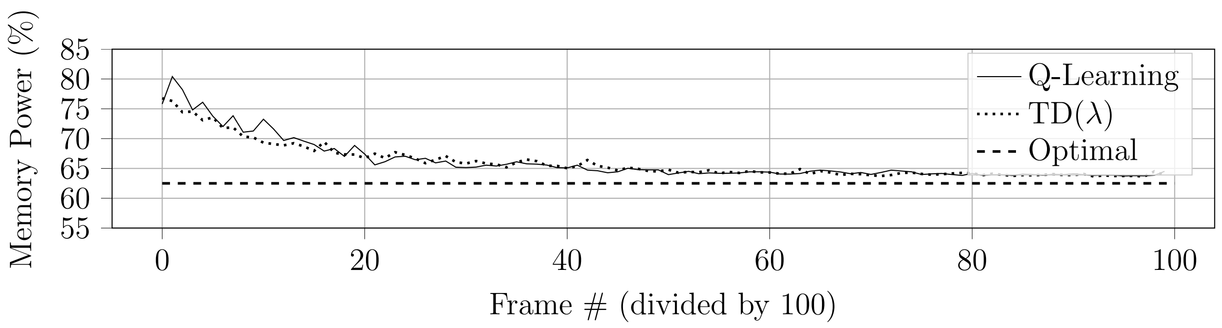
We develop a loadable kernel module (LKM) as middleware between the user application and CSRs. `malloc_approx()` calls from the applications are intercepted by the LKM, and use `mmap` to allocate a contiguous physical segment. The starting and ending addresses of the segment are written to additional CSRs. Whenever the user application loads/stores data, the MMU compares the memory address in hardware using CSRs to check if the data is in the non-critical segment.

Power Models

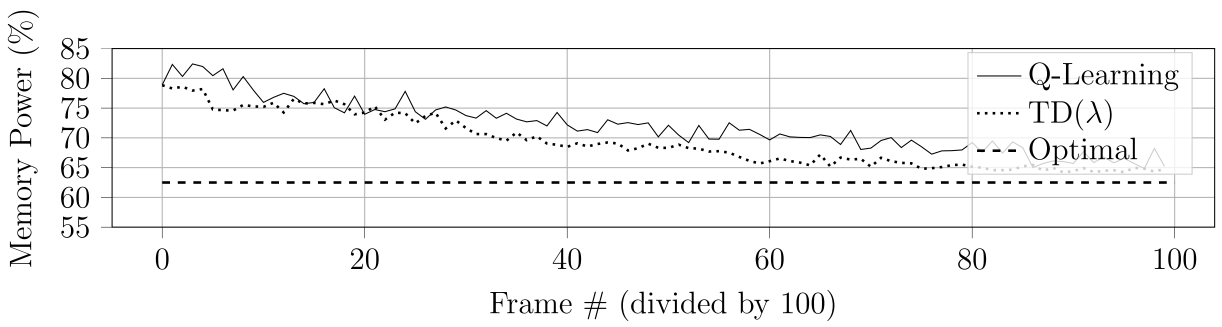
The evaluation platform does not come equipped with on-board power sensors for measuring active power consumption. Since the actual power consumption (and savings) depend on the executed application as well as the actual input, we use Sniper [19] simulations with McPAT [65], along with existing power models from literature [156, 70], to compute the power consumption at different knob settings. Figure 3.18 shows an overview of the complete tools-flow. Each new input to the application is simulated in Sniper, and McPAT is invoked to estimate the power and energy consumption of different system components without approximation. Power and energy consumption is then scaled according to the technology models to estimate different knob power values. For SRAM, we use the 65 nm node model proposed in [156]. For DRAM, we use the low-power DDR model proposed in Flikker [70]. DRAM is assumed to be partitioned into two sections: (1) 1/4 exact DRAM having a high refresh rate, and (2) 3/4 DRAM having a lower refresh rate based on the approximation knob. In order to calculate total DRAM power savings, we assume the usage profile for a typical low-power embedded scenario: 5% busy versus 95% in standby mode (self-refresh state) as assumed in Flikker. Although DRAM power consumption may vary based on factors like memory array partitioning, data access patterns, and current DRAM utilization, we do not consider these variations in this work. The power models do not consider ECCs, which could provide additional power savings.

3.5.5 Demonstrating SEAMS' energy savings and reduction of QoS violations

In this section, we demonstrate SEAMS' ability to learn directly from experience, without requiring any model of the environment's dynamics. These experiments are evaluated using `canny`.



(a) Configuration space = 64 states



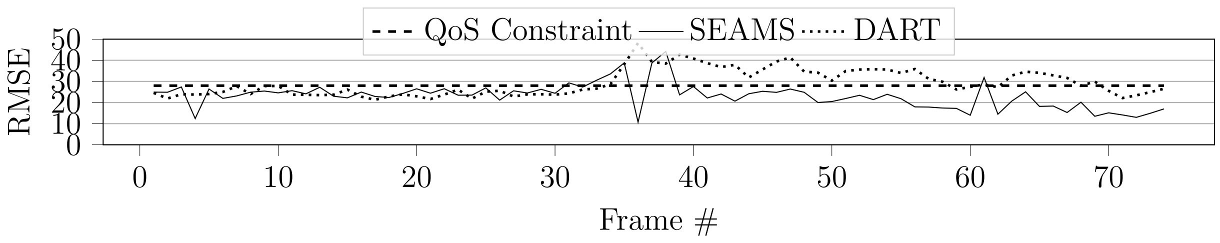
(b) Configuration space = 704 states

Figure 3.19: Power (normalized to exact execution) consumption achieved by different learning algorithms provided a goal to minimize power. Ideally, the policy should learn to reduce power consumption as quickly as possible toward the minimum (black dashed line).

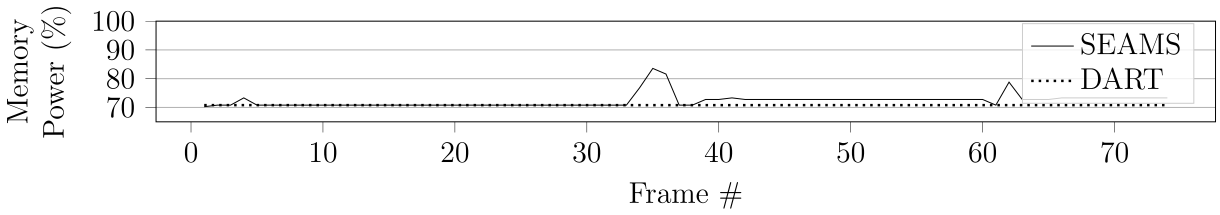
Policy Initialization

First, we evaluate SEAMS ability to learn an optimal policy to minimize energy from scratch. We compare two TD reinforcement learning algorithms: TD(λ) and Q-Learning. The primary difference between the methods is that TD(λ) uses bootstrapping. For both algorithms, we determine the learning parameters empirically using a simulated workload. Without any QoS constraints, approximation knobs should be set to the configuration corresponding to the lowest power. The goal of a policy should be to reach the optimal configuration as quickly as possible. Figure 3.19 compares the TD methods with the optimal configuration corresponding to the the lowest power (yielding maximum energy savings). The plots are averaged over 16 runs to remove the effect of any outliers.

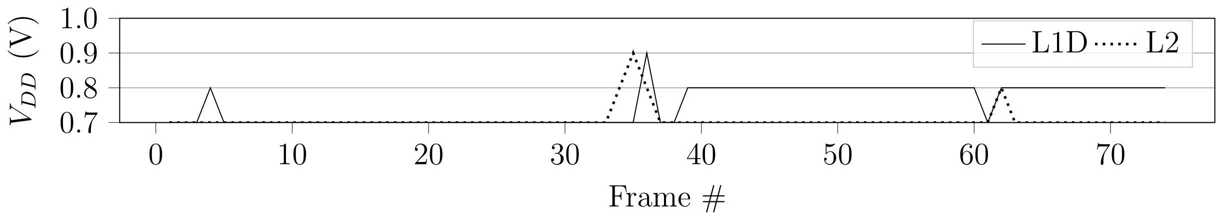
The x-axis in Figure 3.19 represents frames processed, and the y-axis represents average



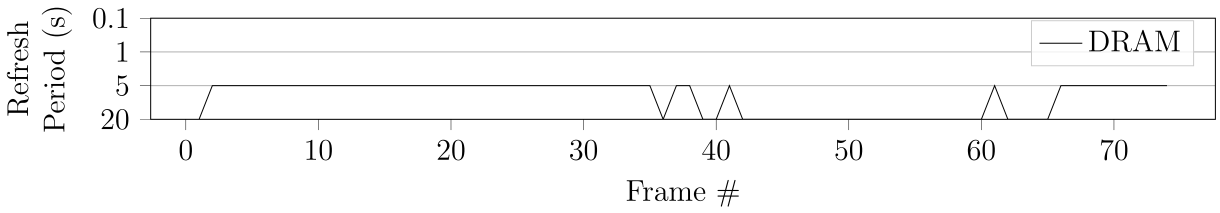
(a) Quality of Service



(b) Normalized Memory Power over exact computation



(c) L1 and L2 cache V_{DD} knobs



(d) DRAM refresh period knob

Figure 3.20: SEAMS self-optimizing power within quality constraint

memory power (normalized to the exact execution) for each episode for canny edge detection application [18]. We make two major observations. First, both algorithms can eventually converge to the optimal policy. Second, in Figure 3.19a, when the configuration space is small (i.e., we restrict the allowable knob settings), both Q-learning and TD(λ) converge at an equal rate. However, when the configuration space is increased (Figure 3.19b), TD(λ) can improve its policy faster than Q-learning because it uses short-term memory in the form of eligibility traces. The traces are used to update multiple state-action pairs based on the reward obtained, instead of just one state-action pair at every step. We conclude that with growing complexity in configuration knobs, TD(λ) is the better algorithm. Thus, for the rest of the experiments, we only use the TD(λ) algorithm in SEAMS.

Self-optimization

We study SEAMS' behavior for unexperienced inputs to show that SEAMS is capable of self-optimizing the approximation knobs in the memory hierarchy within the QoS budget specified by the application. We expose a controller with an established policy to varying inputs and compare it to state-of-the-art approximation management policy DART [156]. DART is a design-time technique that uses a branch and bound algorithm to consider the worst-case effects of all possible approximation knob configurations for a memory hierarchy. We train DART on a set of scenes used during the policy initialization phase. The goal is to honor the QoS constraint specified by the application while maximizing energy-efficiency. In our case, this means keeping the RMSE below a specified value. For a QoS constraint of 10 RMSE, DART statically sets the knobs as follows: L1 V_{DD}:0.8 V, L2 V_{DD}:0.8 V and DRAM T_{REF}:0.5 s. The energy/frame reported in all results is normalized with respect to the the knob configuration of L1 V_{DD}:1 V, L2 V_{DD}:1 V and DRAM T_{REF}:0.1 s.

In Figure 3.20a, a QoS constraint of 28 (RMSE) has been specified by the user, which is marked with a black dashed line. The three key observations are as follows: (1) Frame 32 is a key-frame where a scene change occurs. Neither of the tested policies have experienced this scene previously, and the scene requires a new configuration of knobs to continue to meet the QoS requirement. In frame 32, DART immediately violates the QoS constraint and continues to do so. SEAMS can take actions and reach a new configuration while remaining within the quality constraint. Initially, when SEAMS detects there is an overshoot, it penalizes the current action and takes action to reduce the QoS error. This leads to a conservative state, with more room for QoS relaxation. Subsequently, SEAMS increases the degree of approximation. In the subsequent control cycles SEAMS self-optimizes until it reaches a stable state. (2) Figure 3.20b shows the average power for each frame. There is no significant change in power with DART because it uses a fixed knob configuration at runtime. The normalized energy/frame required by SEAMS is 72.3 %, and normalized energy/frame required by DART is 70.8 %. (3) We define QoS overshoot as the area under curve for the regions of QoS violations during execution. For DART, QoS overshoot is 200 versus 50 for SEAMS. Thus, DART violates the QoS requirement 4× more than SEAMS (Figure 3.20a). This means that SEAMS can reduce QoS violations by 75 % with <5 % additional energy. These experiments demonstrate that SEAMS is self-optimizing, i.e., can continuously learn configurations that meet the QoS constraint when exposed to unknown inputs.

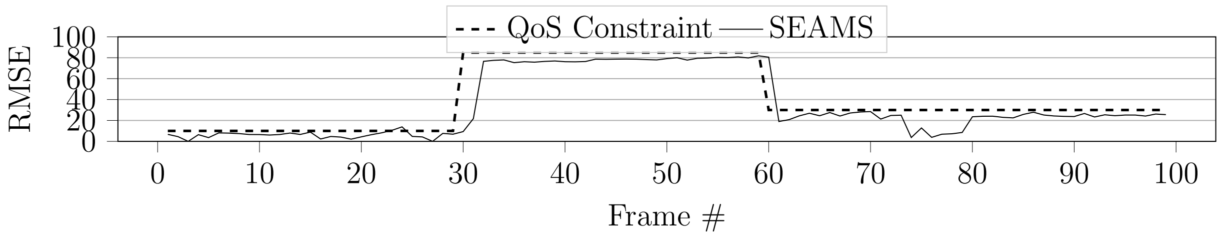
Coordination

We study SEAMS' behavior when exposed to varying quality constraints to show that SEAMS is capable of coordinating interdependent memory knobs. The goal is to adapt to new targets specified by the user and reconfigure the knobs to save more energy while processing the frames. Figure 3.21 shows how SEAMS behaves in the scenario described. The two key observations are as follows: (1) Figure 3.21a shows measured QoS compared to the dynamic QoS constraint. Initially, the QoS constraint is 10. At frame 30, it is relaxed and updated to 85, exposing an opportunity to conserve energy. Again, at frame 60, the constraint is changed to 30. SEAMS can self-adapt to find new configuration knobs that meet the constraints each time they are changed. (2) Figure 3.21b shows the normalized power for each frame. Initially, when the QoS constraint is 10, the memory power is around 80 %. When the QoS constraint is relaxed to 85 at frame 30, SEAMS can lower the memory power consumption by finding a new configuration and operating in that region until the constraint changes. Overall, the normalized energy/frame required by SEAMS is 75.9 %.

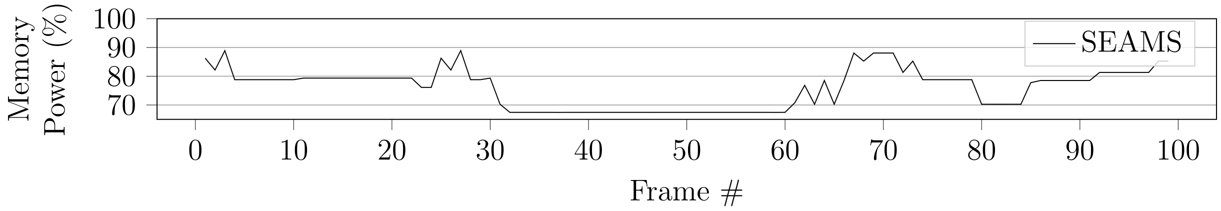
We conclude that SEAMS is capable of (1) self-adapting to new quality constraints specified by applications through coordination, and (2) continuously converging on optimal configurations.

Additional workloads

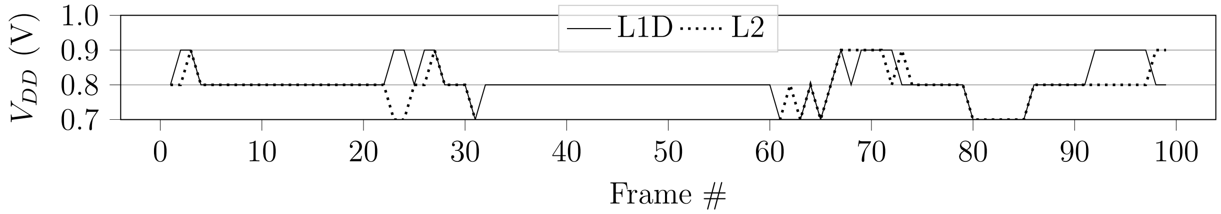
Figures 3.22a and 3.22c show SEAMS' result for `kmeans` with a normalized energy/frame of 76.4 %. Figures 3.22b and 3.22d show SEAMS' result for `blackscholes` with a normalized energy/frame of 70.9 %. We observe that even though the interdependent dynamics



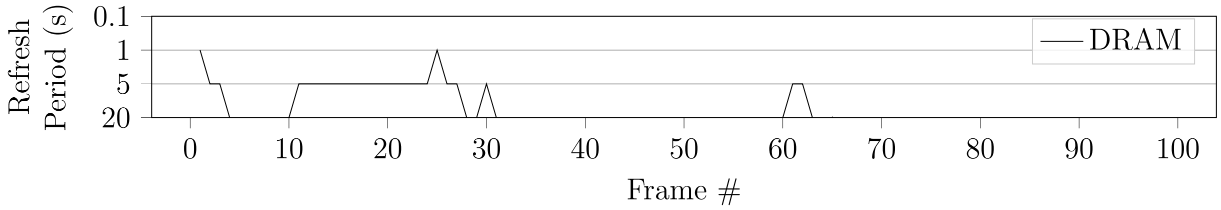
(a) Quality of Service



(b) Normalized Memory Power over exact computation



(c) L1 and L2 cache V_{DD} knobs



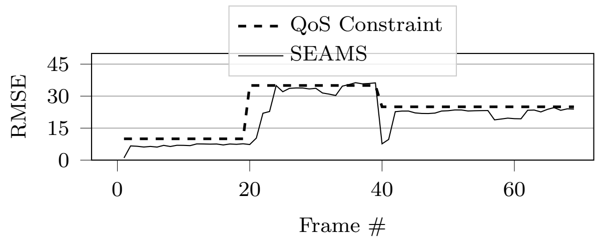
(d) DRAM refresh period knob

Figure 3.21: SEAMS self-adapting to user-specified quality constraints by coordination across the memory hierarchy.

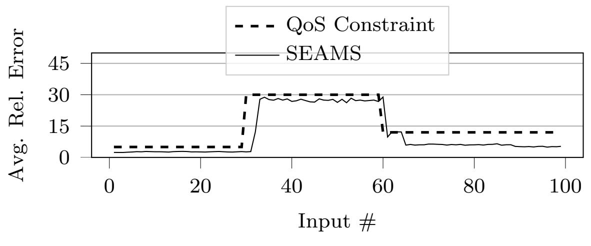
between all three layers of memory hierarchy and the achievable QoS and power are complex and application-dependent, SEAMS is able to meet the dynamic quality constraints by continuously finding new configurations corresponding to the new system goals.

SEAMS Overhead

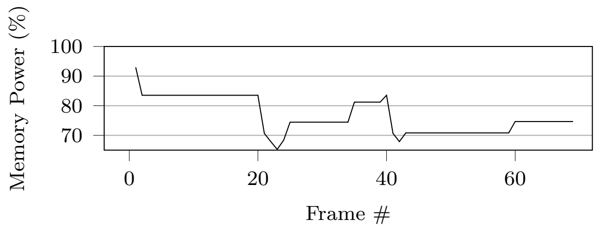
All runtime approximation strategies suffer from two primary sources of overhead: (1) calculating the QoS value, and (2) runtime management. To reap the benefits of approximation, SEAMS is not invoked for every input once the Q-values have been populated. In most cases, the quality monitor (e.g., the errors between pixels in canny edge detection) is embarrassingly parallel. If an additional core is available, the ground truth comparison can be performed in parallel. If SEAMS is invoked too frequently, the compute and energy overhead of the ground-truth comparison would not be justified. In Figure 3.23, we compare the QoS of k-means with different intervals of SEAMS invocation (marked with a tick). Any time the system goals change, SEAMS is invoked for all of the subsequent five frames to adapt to the new setting. We observe that SEAMS can adjust to new goals with reduced invocation



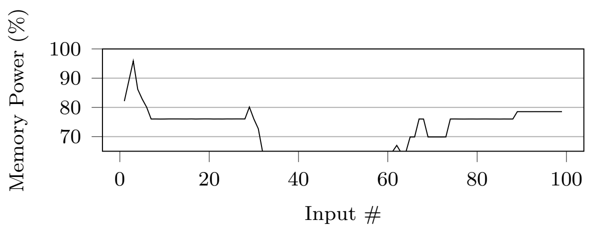
(a) `kmeans`:QoS



(b) `blackscholes`: QoS



(c) `kmeans`: Memory Power



(d) `blackscholes`: Memory Power

Figure 3.22: Additional workloads.

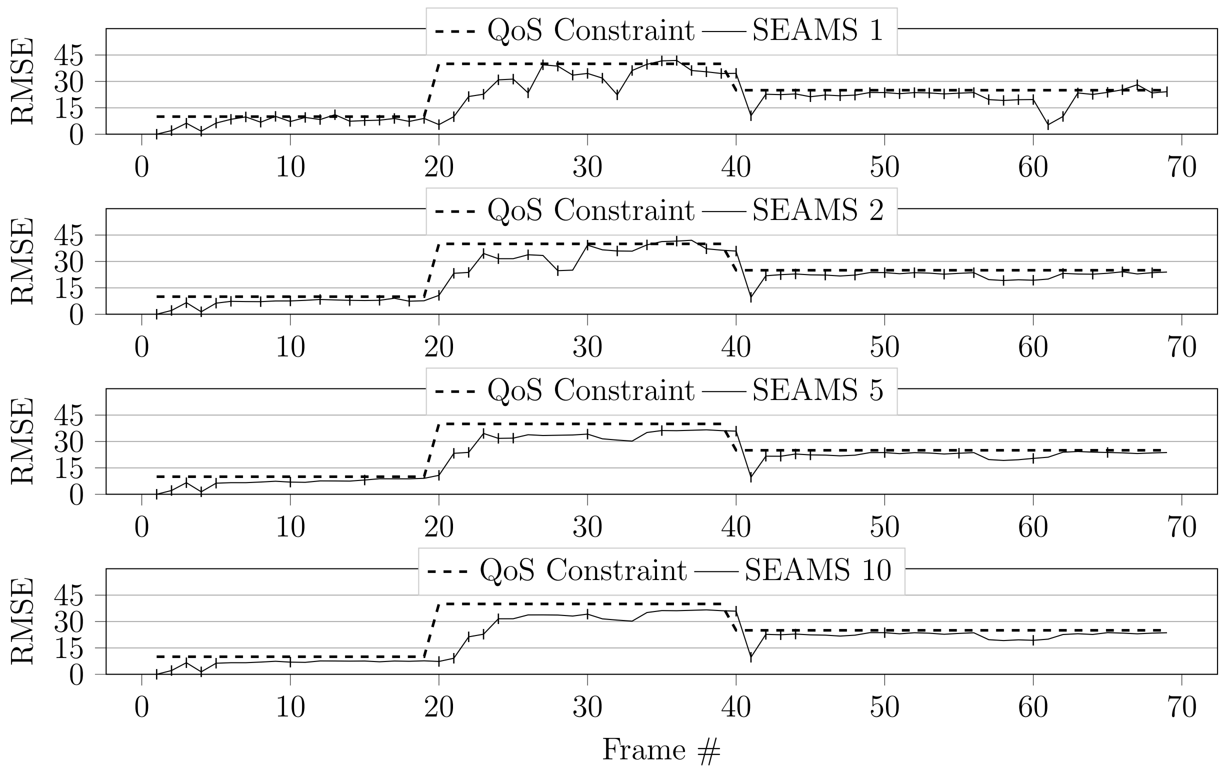


Figure 3.23: SEAMS QoS at different invocation intervals for `k-means`. Blue ticks indicate evaluation instances.

periods, at the risk of potentially missing self-optimization opportunities between invocations. Even ignoring regular invocation completely, SEAMS can be used in an event-driven manner (e.g., updated at design time when a new system is developed, at runtime when a new application is available for approximation, or when the goals (QoS constraints) change). State-of-the-art alternatives do not self-optimize for these situations for a full memory hierarchy.

In Figure 3.24, we compare the compute overhead and the power savings of SEAMS for various invocation periods. Based on these observations, we invoke SEAMS every five frames in all of our evaluations.

The overheads of the current version of SEAMS are not preventative for the presented architecture and execution scenario. However, the investigated architecture is unicore – in a many-core system, the proposed control structure will not scale well due to configuration

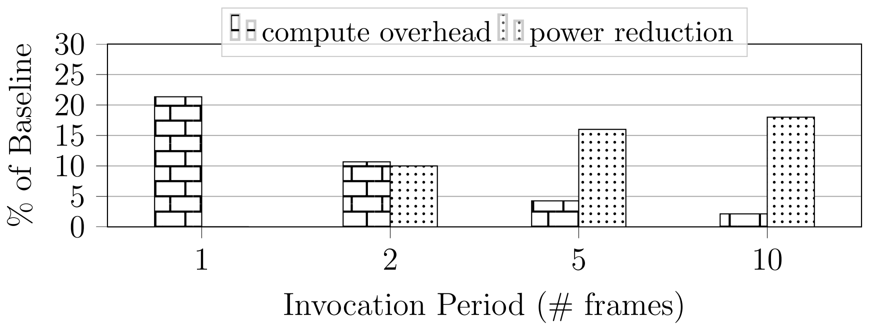


Figure 3.24: SEAMS overhead for different intervals

complexity if a single controller is responsible for configuring the entire memory system.

Similarly, the current reward calculation is based on the quality reports of a single application utilizing the approximate memory segments – multiple QoS applications running concurrently sharing approximate segments will complicate the controller. We believe a hierarchical or multi-agent architecture would effectively tackle the challenges of more complex systems and are topics for future investigation.

3.6 Discussion

In this chapter, we explored the concept of memory approximation and its challenges in the context of performance and energy optimization. Memory approximation offers a trade-off between quality of service (QoS) and performance or energy gains by relaxing the need for high-precision storage for certain data structures. However, maintaining the desired QoS at runtime requires additional mechanisms to monitor and control the memory approximation process.

To address these challenges, we introduced a formal control-theory based approach for memory approximation. This approach allows developers to specify a target QoS metric, and the system dynamically tunes the memory reliability knobs to guarantee the desired QoS. We described the development of a system model using System Identification theory and statistical black-box modeling to capture the effects of different approximation settings. Based on this model, we designed a controller that observes the application's behavior and automatically adjusts the knobs to maintain the QoS, even in the face of changing workloads and system variations.

Building upon this control-theory based approach, we presented SEAMS, a model-free method for tuning memory knobs without prior observation of the system. SEAMS enables the deployment of approximate memory systems without requiring design-time exploration of configuration knobs. It leverages self-adaptive and self-optimizing properties to learn optimal knob configurations for unknown applications, resulting in self-optimizing systems. SEAMS also enables coordination between multiple memory system knobs without explicit communication.

The chapter presented two case studies to illustrate the effectiveness of the proposed approaches. Case Study 1 focused on self-adaptive memory approximation, providing a detailed explanation of the formal control-theoretic approach, the system identification technique, and a comparison with a manual calibration scheme. Case Study 2 delved into the limitations of existing approximation techniques for full memory hierarchies and introduced SEAMS as a self-optimizing runtime manager. It explained the methodology of SEAMS, including the coordination of runtime decisions for interdependent knobs and subsystems, and demonstrated its effectiveness in reducing energy usage and QoS violations.

In summary, this chapter showcased the potential of formal control theory and self-adaptive/self-optimizing techniques in designing memory systems that balance the programmer's burden and guarantee the desired QoS. The presented approaches provide efficient and automated methods for memory approximation, enabling performance and energy optimizations in a variety of computing systems. In the next chapter we will study how memory management techniques can apply to operating systems layer.

Chapter 4

Operating System Abstraction Layer

Contemporary embedded systems rely on heterogeneous compute units with shared main memory to meet the diverse memory capacity and bandwidth requirements of modern applications, such as Nvidia Tegra, Nvidia Xavier, and AMD Accelerated Processing Unit. This shared memory architecture poses challenges for memory performance and energy efficiency, making memory a performance and energy bottleneck in emerging embedded system platforms.

To address this bottleneck, application developers often employ memory profilers to gain insights into the runtime requirements of their applications and attempt to reduce the memory footprint through code optimization. However, relying solely on static optimizations is not sufficient to fully exploit the dynamic nature of modern computing systems.

In the operating system abstraction layer, runtime resource management plays a crucial role in monitoring the system state and implementing policies to dynamically update the system configuration in order to achieve the system's goals, such as maximizing performance-per-unit-power. Traditionally, memory-related policies focus on exploiting memory bandwidth under-utilization at runtime by dynamically scaling the memory controller frequency to reduce power consumption without compromising performance.

However, embedded systems typically support numerous hardware and software knobs that impact system behavior. These knobs include scheduling applications across compute units, setting the frequency of heterogeneous compute units, and adjusting the degree of parallelism by varying the number of threads. Each of these knobs can influence the time required to process application data, and subsequently, affect energy consumption.

While memory profilers provide information such as the number of memory accesses (load/stores) and generate heatmaps indicating memory access density for the profiled application, the overhead of parsing and analyzing this detailed information at runtime can be impractical for high-frequency decision making by resource managers. Therefore, it becomes essential to monitor coarse-grained metrics that can assist runtime resource managers in efficiently determining the ideal system configuration.

In this chapter, we will focus on the operating system abstraction layer and explore the tools and techniques available to optimize memory performance.

Specifically, we will investigate reflective runtime manager called MARS to leverage computational self-awareness principles and efficiently utilize memory resources, enhance system performance, and minimize energy consumption.

4.1 Memory Management Techniques at the Operating System Layer

The operating system layer employs various memory management techniques to address the memory bottleneck and optimize memory performance and energy efficiency. This section explores several key techniques:

- **Runtime policies using Memory Bandwidth Utilization:** Dynamic Voltage and Frequency Scaling (DVFS) of the memory controller is a primary technique used to address the memory bottleneck [29]. By dynamically adjusting the frequency of the memory controller based on the observed memory bandwidth utilization, DVFS aims to strike a balance between performance and power consumption. Research studies have shown that DVFS can significantly reduce memory power consumption without compromising system performance [27]. Additionally, techniques such as MAR-CSE utilize an approximation equation based on the correlation of memory access rate and critical frequency to predict the voltage and frequency at runtime [66].

- **Runtime policies using reflection or prediction:** Reflection or prediction-based techniques are employed to make runtime policy decisions. CPU Dynamic Voltage and Frequency Scaling (CPU DVFS) has been extensively explored in the literature [151]. Operating systems provide various governors, such as ”ondemand,” ”performance,” and ”powersave,” to manage CPU frequency at runtime. Recent work utilizes reflective system models to predict system behavior when different frequencies are selected [32, 119], enabling informed decision-making to optimize system performance and energy consumption.

- **Runtime policies using Task Mapping:** Task mapping involves scheduling running applications and determining which threads run on which cores within the operating system scheduler. By strategically mapping tasks to appropriate compute resources, such as smaller power-efficient cores or larger cores with higher computational capacity, significant energy savings can be achieved. Combining task mapping with DVFS has been shown to yield substantial energy savings [153].

- **Runtime policies using Workload Information:** Recent work [107] focuses on considering the combined effect of application compute/memory intensity, thread synchronization contention, and non-uniform memory access patterns to develop a runtime energy management technique. This technique performs DVFS on CPU cores based on workload characteristics. However, it does not directly consider memory bandwidth utilization or dynamically change the memory controller frequency.

These memory management techniques at the operating system layer demonstrate the efforts to optimize memory performance and energy efficiency. In the following section, we will look at the potential for integration with computational self-awareness principles to achieve more intelligent memory management.

4.2 MARS Middleware

Real-time systems traditionally employ an *Observe*, *Decide*, and *Act* (ODA) feedback loop to manage goals and configure the system. In an ODA loop, observed behavior is compared to the desired behavior, and the discrepancy is fed to a policy for decision-making. The policy then invokes actions based on the results of the *Decide* stage.

MARS (Middleware for Adaptive and Reflective Systems) [91] is a cross-layer and multi-platform framework that supports the creation of resource managers for real-time systems. It enables the composition of system models and resource management policies in a flexible and coordinated manner.

The MARS framework incorporates a *Reflection* loop, which extends the traditional ODA logic by including past history and predictions to make intelligent decisions and enhance adaptivity.

MARS interacts with all layers of the system stack to orchestrate the management of resources effectively. It comprises four main components:

1. **Sensors and actuators:** These components collect sensed data, including performance counters (e.g., instructions executed, cache misses) and other sensory information (e.g., power, temperature), to assess the current system state and characterize workloads. Configurable knobs are also provided to enable the platform to adjust its configuration and optimize operating points or control tradeoffs.

2. **Resource management policies:** MARS allows users to define and implement decision-making engines known as resource management policies. These policies are created using generic interfaces, making them portable across different hardware platforms.

3. **Reflective system model:** The policies in MARS utilize reflective system models to make informed decisions. These models can represent either *models of policies* or *(sub)system models*. Policy models are used for coordinating decisions made within MARS or other components like the operating system. On the other hand, system models take policy decisions from the policy models and generate predicted system behavior.

4. **Policy manager:** The policy manager in MARS is responsible for reconfiguring the system by adding, removing, or swapping policies to better achieve the current system goal. It ensures that the appropriate policies are activated or deactivated based on the system's changing requirements and objectives.

The modular composition and flexible nature of MARS enable it to support new platforms, sensors, and actuators easily. By integrating the MARS middleware into the operating system layer, real-time systems can benefit from its adaptive and reflective capabilities, allowing for more efficient resource management and improved overall system performance.

4.3 Case Study 1: Workload Characterization for Runtime Memory Management

In 1965, Gordon E. Moore predicted that manufacturers would continue doubling the number of transistors on a chip approximately every two years [112]. The semiconductor industry has since developed by leaps and bounds, introducing billions of transistors on tiny dies. This growth in embedded compute resources has led to applications becoming increasingly demanding, generating data at rates that test the extremes of main memory latency and bandwidth. As a result, the performance and energy bottleneck in today's embedded system platforms is no longer the computation of data, but rather the transportation of that data to and from computation.

One of the current challenges for computing systems is that the computation is often performed far away from the data [90, 54]. While caching has traditionally been applied to address this challenge, the working set size of applications may exceed the cache capacity, leading to cache misses and the need for main memory accesses to fetch data for the processor. The movement of data from DRAM to the processor consumes a significant portion of the total system power [27], and cache misses can consume more than 50% of cycles [54].

Previous works have utilized memory bandwidth utilization to determine memory requirements and develop runtime policies to address memory-related energy consumption. For example, [27] proposes setting the memory controller frequency at the lowest frequency (e.g., 800MHz) when the memory is lightly loaded (memory bandwidth is under 2000 MB/s).

To further investigate the impact of memory controller frequency on energy consumption, Figure 4.1 illustrates the effect of memory controller frequency on total energy consumption for a variety of benchmarks executed on the Nvidia Jetson TX2 embedded platform [94]. The figure shows that the lowest frequency does not consistently lead to the minimum energy consumption across different workloads, as the extended execution time at lower frequencies can result in increased energy consumption.

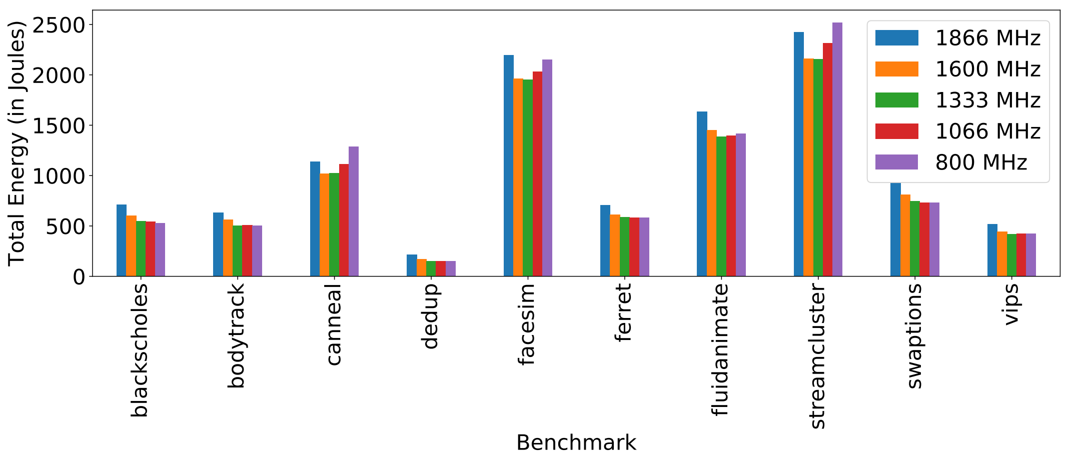


Figure 4.1: Total energy consumed as memory controller frequency is varied statically from 800MHz to 1866MHz for PARSEC workloads on the Jetson TX2. The frequency that results in the minimum energy spent varies.

To address these challenges, we propose a memory-driven application classification based on multiple dimensions to assist in developing policies for resource allocation strategies, specifically for DRAM. This classification combines the size of memory required (in MBs) for the working set of an application and the runtime DRAM bandwidth requirement (last-level cache miss rate) to determine an application's memory requirement. Both measurements can be made with minimal overhead and used at runtime to evaluate the dynamic requirements of applications.

In this section, we utilize this classification to determine a static memory controller frequency that minimizes the Energy-Delay Product (EDP). However, this classification can be used more generally to develop policies that dynamically configure the system based on the classes of applications currently running, considering their memory requirements. By incorporating this memory-driven application classification into memory management policies, we can enhance memory performance and energy efficiency for real-time systems.

4.3.1 Memory access pattern and working set size

Our approach combines the working set size and memory bandwidth by calculating their product (WBP). The objective is to find the operating frequency, which leads to the lowest energy-delay product (EDP).

Based on the EDP at different values of WBP, applications are categorized into one of three classes.

The WBP metric and the classification ranges are described next.

1. **Working Set Size**: Linux kernel 4.3 introduced idle page flags to track memory utilization. Once a process starts, the idle bits corresponding to all the virtual pages in that process are set to 1 to indicate that they have not been referenced. Whenever the process issues memory read/write requests, the idle bit corresponding to the virtual page is set to 0 by Linux kernel. A 0 implies that the page is not idle. We selected the window size as 200ms, however exploring more window durations and feasibility of adaptive windows remain as future work. Every 200ms, the number of 0 bits in the idle page flags are read and used to calculate the working set size as 0 as follows:

$$WorkingSetSize = \text{Number of active pages} \times \text{Size of each page}$$

2. **Memory Bandwidth**: ARM cores include logic to gather various statistics on the operation of the processor and memory system during runtime based on a Performance Monitoring Unit (PMU). The PMU provides hardware counters for different events, which are used to profile application behavior. The counter values of `L2_CACHE_REFILL` and `MEM_ACCESS` on each core are monitored to understand the memory traffic at run-time every 200ms (Sensing Window Length = 0.2s). Since the L2 is the last-level shared cache on the Jetson, the memory bandwidth can be calculated using these values as:

$$MemoryBandwidth = \frac{\sum_{i=1}^{activeCores} \texttt{L2\_CACHE\_REFILL}_{Core\_i} \times DataBusSize}{SensingWindowLength}$$

3. **Memory Power and System Power**: Nvidia drivers read the power measurements from onboard sensors connected by `I2C`. The separate domains for system power and memory power help isolate the energy consumed by the memory separate from the rest of the system.

4. **Latency/Delay**: The amount of time that the workload takes to complete is indicative of the compute and memory latency. The CPU governor is set to 'userspace' and the frequency of all CPU cores is set to maximum for all the experiments. Thus, changes in memory latency due to the change in memory controller frequency is reflected in the total execution time.

By considering these metrics, including working set size, memory bandwidth, memory power, system power, and latency, we can accurately characterize the memory access patterns and assess the dynamic requirements of applications. These metrics form the basis for our memory-driven application classification and enable the development of effective resource allocation policies at runtime.

Characterization of PARSEC benchmark suite

The PARSEC benchmark suite [130] is a popular suite for evaluating multiprocessor-based embedded platforms.

It consists of twelve workloads (nine applications and three kernels) from the recognition, mining, and synthesis (RMS) domain, as well as representative embedded system applications.

The workloads compose a diverse representative of working set size, degree of parallelism, off-chip traffic, and data-sharing that can be representative of the workloads executed on emerging embedded platforms.

Based on the working set, two broad classes of workloads are distinguished:

1. **Working set smaller than 16MB**: These applications do not generate much memory pressure because the working set can fit in the last-level shared cache. Example applications are `bodytrack` and `swaptions`.

2. **Working set larger than 16MB**: These applications have a very large working set generating more off-chip memory accesses to the DRAM. Example applications are `canneal`, `ferret`, `facesim`, `fluidanimate`. When the input size grows, the working set can even reach gigabytes due to algorithms that operate on large amounts of collected data.

Note that as the number of cores increases with the degree of parallelism, so does the bandwidth requirement.

`bodytrack` makes off-chip memory accesses in short, but bandwidth-intensive bursts.

When several instances of the same application execute concurrently, these short bursts limit the scalability.

The sampling of benchmarks from PARSEC demonstrates highly variable memory requirements that depend on multiple factors. The number of load/store operations, size of the last-level cache, and the number of parallel threads are some of the factors that affect the bandwidth requirement.

These various sources of memory pressure provide opportunities for resource managers to address the bottleneck at runtime for unpredictable workloads. Working set can effectively represent some of these dynamics, and we show that combining the working set size information with memory bandwidth at runtime can lead to efficient system configurations.

Our objective is to find the operating frequency that results in the lowest energy-delay product. To that effect, we are interested in finding changes in EDP across frequencies at different values of WBP.

This is done by experimentally observing the average EDP (over 10 applications) of PARSEC workloads on a target embedded system described in Section 4.3.3.

The average EDP at different values of WBP for PARSEC workloads obtained experimentally are presented in Figure 4.2.

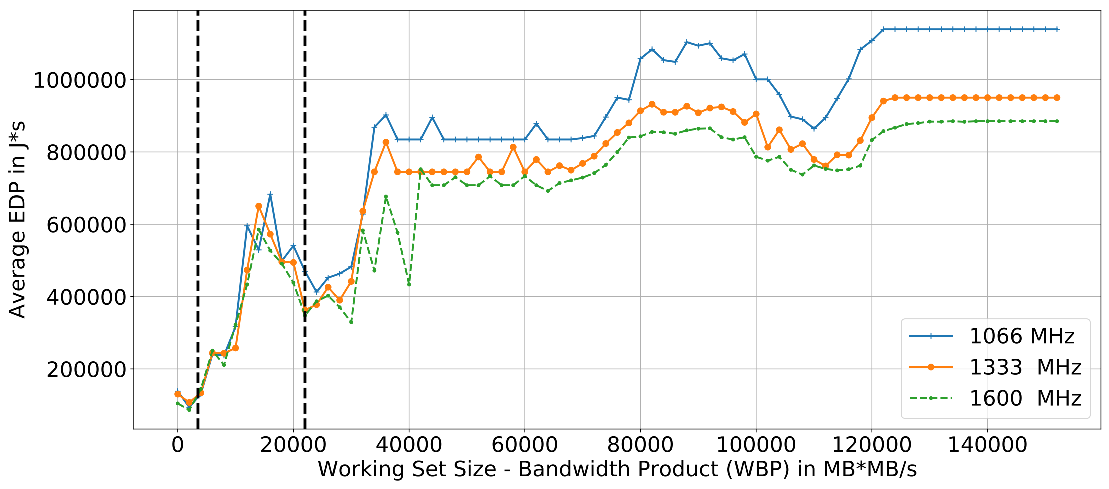


Figure 4.2: Change in average EDP (energy-delay product) of PARSEC workloads across frequencies at different values of WBP (working set size - bandwidth product).

The different colors represent different frequencies. The PARSEC applications operate in different regions of WBP throughout execution.

At low values of WBP, the EDP does not change with frequency. This is because the working set size of the application is small, and the memory requests are served from cache.

Thus, operating at lower frequency does not have any effect on the EDP.

As the WBP increases, memory requirement increases, and higher frequencies perform better. Frequencies lower than 1066 MHz (e.g., 800 MHz) have a very high latency whereas frequencies higher than 1333 MHz (e.g., 1866 MHz) consume too much power during the execution.

Thus, they never obtain optimal EDP for any of the workloads.

Hence, frequencies 800 MHz and 1866 MHz are not included in Figure 4.2.

Table 4.1: Classification of PARSEC workloads based on average WBP (working set size - memory bandwidth product) during runtime.

Workload	WBP (in MB^2/s)	Class	EMC Frequency
`dedup`	2100	C_1	1066 MHz
`swaptions`	3000	C_1	1066 MHz
`bodytrack`	3600	C_2	1333 MHz
`ferret`	3700	C_2	1333 MHz
`blackscholes`	5100	C_2	1333 MHz
`vips`	9700	C_2	1333 MHz
`fluidanimate`	16000	C_2	1333 MHz
`canneal`	22000	C_3	1600 MHz
`facesim`	62222	C_3	1600 MHz
`streamcluster`	119000	C_3	1600 MHz

4.3.2 Methodology for estimating the WBP metric

We classify applications based on their WBP profile.

For each class, the goal is to select a memory controller frequency that leads to the minimum EDP.

Two thresholds for WBP are selected based on the average application EDP profile at different frequencies.

In this work, the operating frequency is determined statically based on the average WBP profile of workloads.

However, we acknowledge the possibility of a runtime policy which checks the WBP at regular intervals to change the frequency during application execution.

The classification and thresholds are proposed as follows:

1. C_1 (Small memory footprint): $0MB^2/s \leq WBP < 3500MB^2/s$ These applications have a small working set size and a low memory bandwidth. The L2 cache is large enough to accommodate most of the requests to memory for this class of application. When running C_1 applications, **the system is configured at 1066 MHz**.

2. C_2 (Medium memory footprint): $3500MB^2/s \leq WBP < 22000MB^2/s$ These applications have a moderate memory requirement. The L2 cache cannot accommodate all the requests to the memory. The working set size is also considerable and generates requests which need to go to main memory. When running C_2 applications, **the system is configured at 1333 MHz**.

3. C_3 (Large memory footprint): $WBP > 22000MB^2/s$ These applications have a high memory requirement. Operating the system at 1600 MHz clearly gives the lowest EDP. The working set size for this class of application is large and spread out in different regions of memory. This incurs excessive cache misses and generates requests to the main memory. When running C_3 applications, **the system is configured at 1600 MHz**.

4.3.3 Evaluation of the proposed WBP-based memory management approach

Experimental Setup

We use Jetson TX2 [94] from Nvidia, an embedded System-On-Chip (SOC) platform, to evaluate our proposed technique.

The Jetson has heterogeneous compute cores (quad-core ARM Cortex A57 and dual-core Nvidia Denver2) distributed in two clusters along with an onboard 256-core Pascal GPU.

Each of the clusters resides in a separate frequency domain.

Jetson has a shared memory architecture, and all resources (CPU clusters, GPU) share the main memory.

Jetson TX2 has an 8GB 128-bit LPDDR4 memory and a 32GB eMMC 5.1 for onboard storage.

The Cortex cores come with: 48KB L1 instruction cache (I-cache) per core; 32KB L1 data cache (D-cache) per core. The Denver cores have 128KB L1 I-cache per core; 64KB L1 D-cache per core. All the cores share an L2 Unified Cache of 2MB.

Jetson uses an External Memory Controller (EMC) to manage the off-chip memory traffic.

The EMC has different operating frequencies ranging from `4800KHz` to `1866MHz`. The onboard ARM Cortex Real-time (R5) Boot and Power Management Processor (BPMP) changes the memory controller frequency through kernel drivers.

During all the experiments, the Denver cores are switched off, and the Cortex A57 cores are configured at the highest frequency.

Disabling CPU DVFA helps isolate the effect of EMC operating frequency.

Results

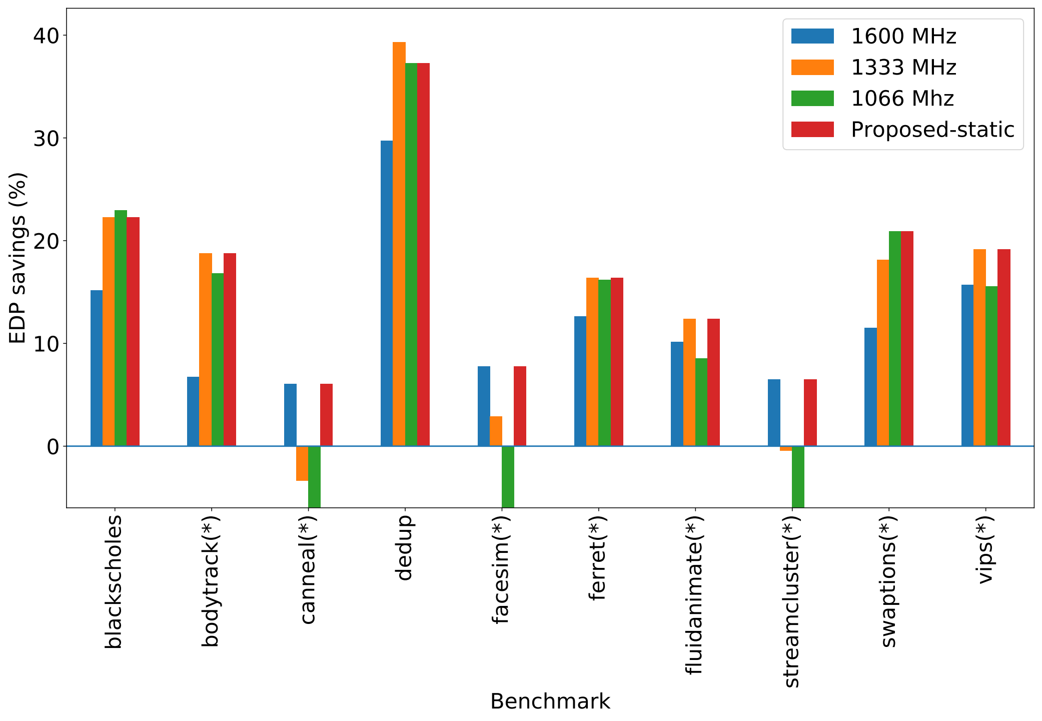


Figure 4.3: EDP reduction with proposed classification using WBP compared to static-frequency baselines. Applications marked with a * have an optimal static configuration with the proposed scheme.

In Section 4.3.1, PARSEC benchmarks are classified into three classes, and an ideal memory controller frequency for each class is proposed.

applications are executed with a fixed memory controller frequency based on class.

The results are presented in Figure 4.3.

We expect to see lower EDP with the proposed approach when compared to techniques which do not utilize any memory information when deciding the EMC Frequency.

The average WBPs for PARSEC benchmarks are presented in Table 4.1.

The working set size and memory bandwidth are measured using the Linux idle page tracker and `L2_CACHE_REFILL` PMU event counter every 200ms.

The proposed static configuration uses the EDP values obtained with this method.

The EDP value is compared with other static configurations (1600MHz, 1333MHz and 1066 MHz) which serve as baselines in Figure 4.3.

From the figure, we see that the proposed scheme can find the optimal configuration for eight out of ten PARSEC workloads.

The proposed scheme can achieve on average 16.7% (max: 37.3%) reduction in EDP when compared to the most aggressive memory controller frequency (1866 MHz).

The results are compared with an optimal scheme obtained by executing all workloads at all possible EMC frequencies and choosing the frequency that yields minimum EDP.

The optimal scheme can achieve an average reduction of 17.1% (max: 39.3%) EDP when compared to the most aggressive memory controller frequency (1866 MHz).

The static method of setting the memory controller frequency is not capable of setting the optimal knob for all the applications. Benchmarks like `blackscholes` exhibit dynamic memory accesses patterns, as shown in Figure 4.4. A static configuration is not sufficient to address these bursts of memory accesses. Therefore, it is necessary to perform DVFS adaptively during application execution. Additionally, if we consider a platform running multiple applications concurrently, the each application must be monitored and classified individually.

In the case study, we have introduced the WBP (Working Set Size - Memory Bandwidth Product) profiling metric, which combines two critical factors: (1) memory bandwidth uti-

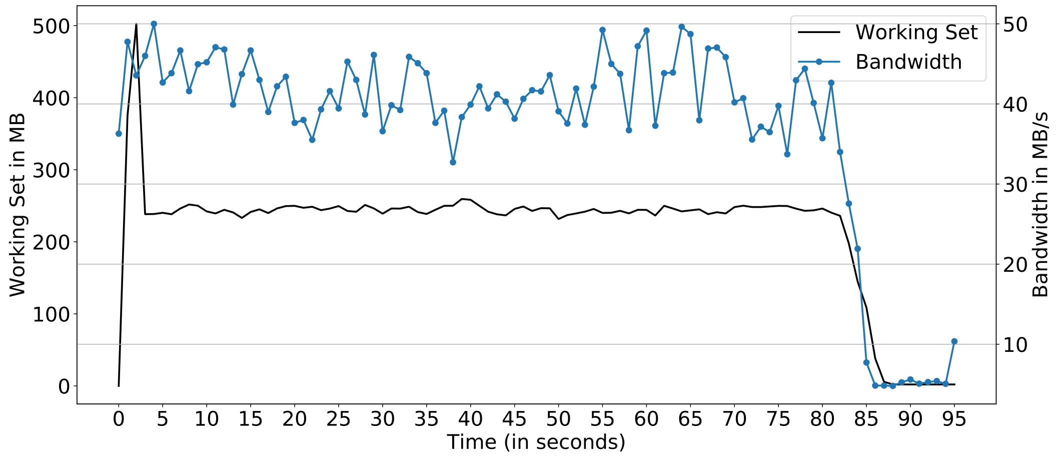


Figure 4.4: Runtime memory profile of `blackscholes`. Dynamic memory access pattern calls for the exploration of a runtime policy.

lization and (2) working set size of running workloads. By analyzing the changes in the energy-delay product (EDP) with frequency at different WBP values, we classify applications into three categories based on their memory requirements.

Our initial results demonstrate the promise of our approach, as the static configuration of the memory controller frequency correctly estimates the optimal frequency for 8 out of 10 PARSEC workloads. This classification technique offers a practical solution for managing memory-related energy consumption in embedded systems. The static scheme of setting the memory controller frequency for each class results in an average EDP savings of 16.7% and a maximum of 37.3%. These energy savings indicate the potential benefits of our approach in improving both performance and energy efficiency. While our current work utilizes fixed sensing windows of 200ms, it is worth noting that the optimal window size may vary for different workloads and scenarios. Future exploration of adaptive sensing window sizes, along with the integration of runtime policies using the proposed classification technique, can further enhance the effectiveness of our memory management approach.

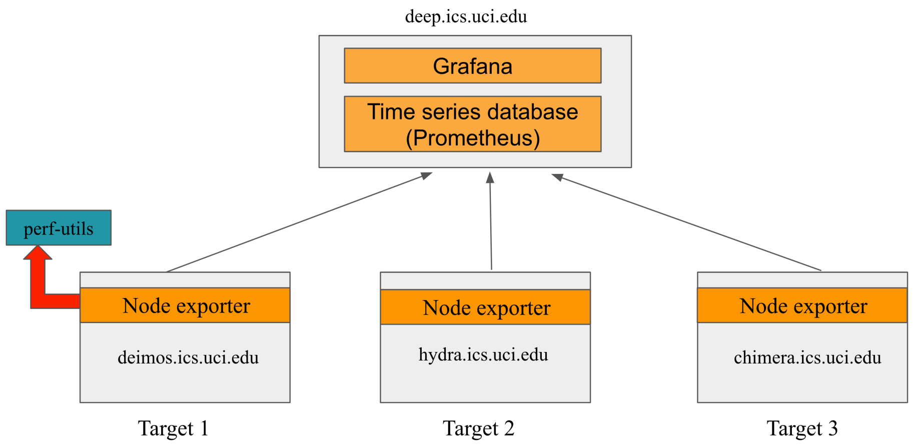


Figure 4.5: Overview of MARS 2.0 architecture [30].

4.4 MARS 2.0: Scalability and ML-based DVFS Boosting

To meet the growing demands of modern real-time workloads across a wide range of computing systems, MARS has undergone enhancements to extend its capabilities beyond embedded HMP. Figure 4.5 shows an instance of how MARS can be deployed in a cluster with telemetry monitors on each host, a distributed time-series database, and high-level policies. These advancements have enabled the framework to scale and adapt to larger-scale deployments, such as datacenters. We dive into the architecture components and present a case study next.

Telemetry Integration:

One significant enhancement is the integration of a telemetry daemon that continually gathers relevant performance metrics from multiple hosts. This telemetry daemon efficiently collects data from distributed systems, providing valuable insights into system behavior and resource utilization.

Prometheus Time-Series Database:

To efficiently store and manage this telemetry data, MARS leverages Prometheus [100], a powerful time-series database. Prometheus is designed to handle the high volume of time-stamped data points and provides efficient querying capabilities. It supports the Prom Query Language, allowing users to retrieve specific metrics or compose complex queries to extract meaningful insights from the collected data.

Grafana Visualization:

Additionally, MARS integrates with Grafana [21], a visualization tool that enables system designers and operators to gain valuable insights into the behavior and performance of the distributed system. Grafana facilitates the visualization and analysis of telemetry data, making it easier to monitor system performance over time and identify potential areas for optimization.

Offline Analysis and Insightful Models:

These new features are incorporated into a new version of MARS that enables simpler telemetry across networked systems and offline analysis of real-time telemetry data. This offline analysis capability allows system designers to generate insightful models based on historical performance data, aiding in proactive system optimization.

The evolution of MARS to version 2.0 signifies its adaptability to diverse computing environments, including datacenters, and its commitment to providing comprehensive insights into system behavior. This scalability and enhanced telemetry capabilities make MARS a valuable tool for optimizing memory performance and energy efficiency not only in embedded systems but also in larger-scale computing deployments.

4.4.1 Case Study 2: Expanding Datacenter Capacity with DVFS Boosting

The COVID-19 pandemic brought about unprecedented challenges to Meta's datacenter infrastructure [99]. A sudden surge in user demand, coupled with disruptions in server supply chains, created a capacity crisis that required innovative solutions. In this case study, we explore how Meta successfully expanded its datacenter capacity by leveraging CPU dynamic voltage and frequency scaling (DVFS) boosting.

4.4.2 Challenges for Scalable Deployment

Modern datacenters are often already operating at the limits of their power infrastructure. Enabling DVFS boosting to increase capacity posed the risk of exceeding these power constraints, leading to power capping and potential service disruptions.

To address this challenge, Meta's engineering team developed a comprehensive strategy to manage power consumption effectively while enabling boosting. While the MARS artifact was not directly used for this deployment, a similar software was used for the experiments. The experiments involved carefully monitoring power usage, optimizing power allocation, and implementing safeguards to prevent power overload. Meta's datacenter services exhibit a wide range of performance characteristics and resource requirements. Further, enabling DVFS boosting across this heterogeneous environment required a solution that could adapt to diverse service workloads.

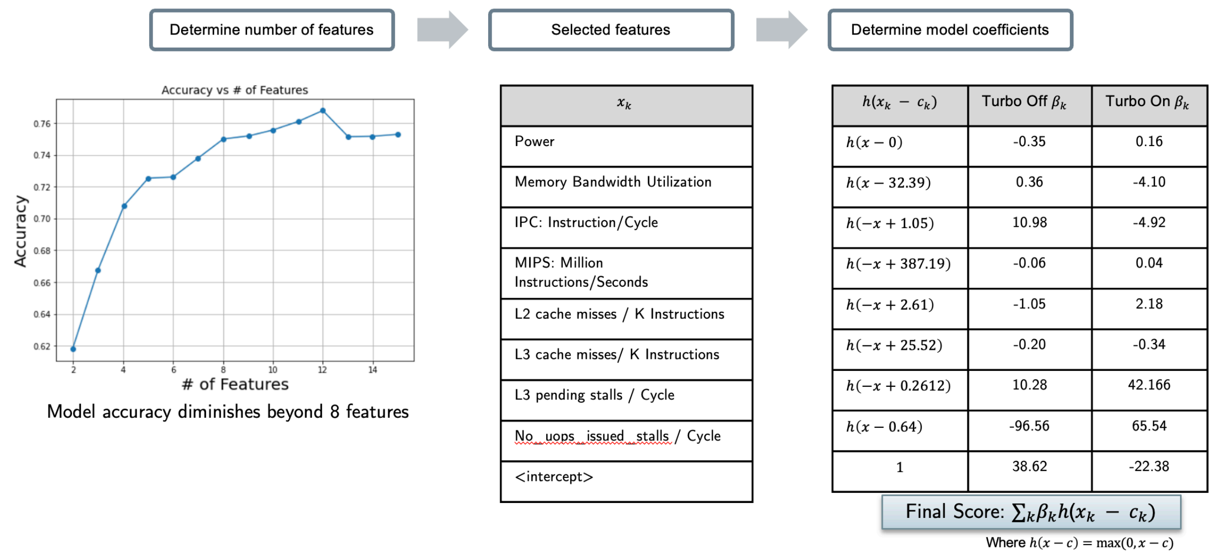
Model-based Workload Prediction for DVFS Boosting

It is crucial to selectively apply DVFS boosting to services that benefit most from it. The decision to enable boosting should consider factors like CPU core performance, IO and memory capacity, and network-bound characteristics. Generally, services fall under one of the following three categories:

1. **Direct capacity improvements**: CPU-bound services directly benefit for DVFS boosting as additional frequency from boosting helps them to deliver higher throughput. 65% of services at Meta are bottlenecked by CPU utilization (i.e., CPU-bound) and are easy candidates for running at higher CPU frequencies.
2. **Indirect capacity improvements**: IO- (6%) and Memory-bound (27%) services experience indirect benefits from DVFS boosting during high loads, leading to reduced tail latencies.
3. **No improvements**: These services (e.g., Network-bound services) do not benefit from DVFS boosting for capacity needs.

The first step in leveraging DVFS boosting was to identify the candidates that can genuinely benefit from DVFS boosting and classify them in the correct categories. With thousands of services running at the CPU sustainable frequency profile, the goal was to pinpoint those that could benefit from boosting to create additional capacity. To achieve this at scale without costly experimentation, a machine-learning-based heuristic was developed.

First, we collect a dataset using a telemetry service [26] that runs on each host to collect performance counters (details in Figure 4.6) and store them in a common data repository with a granularity of one minute. The `useful_mips` counter is used a proxy for application agnostic throughput metric. Classes were created to categorize services into low, medium,



Determine number of features
Selected features
Determine model coefficients
Accuracy vs # of Features
Accuracy
# of Features
Model accuracy diminishes beyond 8 features
Final Score: $\sum_k \beta_k h(x_k - c_k)$
Where $h(x-c) = \max(0, x-c)$

x_k
Power
Memory Bandwidth Utilization
IPC: Instruction/Cycle
MIPS: Million Instructions/Seconds
L2 cache misses / K Instructions
L3 cache misses/ K Instructions
L3 pending stalls / Cycle
No_uops_issued_stalls / Cycle
<intercept>

$h(x_k - c_k)$	Turbo Off β_k	Turbo On β_k
$h(x-0)$	-0.35	0.16
$h(x-32.39)$	0.36	-4.10
$h(-x+1.05)$	10.98	-4.92
$h(-x+387.19)$	-0.06	0.04
$h(-x+2.61)$	-1.05	2.18
$h(-x+25.52)$	-0.20	-0.34
$h(-x+0.2612)$	10.28	42.166
$h(x-0.64)$	-96.56	65.54
1	38.62	-22.38

Figure 4.6: Feature Selection and Model used in [99].

and high PPW classes, simplifying the prioritization of boosting based on total capacity impact in regions with tight power headroom.

Next, we develop a machine-learning-based heuristic that utilizes the micro-architectural counters and power as input vectors ($\mathbf{X}$) and classifies each service among the three classes as the target variable ($\mathbf{y}$). Production data from services with A/B test infrastructure was used for training, collecting features ($\mathbf{X}$) during the baseline configuration with DVFS boosting disabled. The open-source version of the multivariate adaptive regression splines technique [37] was chosen for modeling. This technique is well-suited for its ability to derive models in the presence of non-linearity, feature selection, and ease of interpretability and portability. Feature selection was crucial, with only 8 features selected from the 28 counters provided to the model. These features included power, memory bandwidth utilization, IPC, MIPS, L2 and L3 cache misses per kilo instructions, L3 pending stalls per cycle, and stalls due to no uops issued per cycle. Classes ($\mathbf{y}$) were assigned based on performance-per-watt ratios for A and B tests.

During validation and deployment, only samples with the CPU sustainable frequency configuration were used to predict low, medium, or high performance-per-watt. Cross-validation and accuracy requirements resulted in a model with 95% accuracy for the dataset used. The goal was not to identify all services benefiting from boosting but rather to enable boosting as much as possible to alleviate capacity constraints.

4.4.3 Memory Counters used and Insights

The 28 features used in MARS provided valuable insights beyond service optimizations. These features were collected when running with DVFS boosting disabled because candidate services were not initially using boosting. When analyzing the selected counters, several insights into performance-per-watt emerged:

- **IPC and MIPS**: High values of Instructions Per Cycle (IPC) and Millions of Instructions Per Second (MIPS) are positive indicators. They suggest that the CPU experiences minimal memory waiting time, implying that higher CPU frequency could enhance performance.
- **L2 and L3 Cache Misses**: Elevated values of L2 and L3 cache misses indicate a higher likelihood of CPU pipeline stalls. This serves as an indicator that the CPU is waiting for memory access, suggesting that boosting could be beneficial.
- **Memory Bandwidth Utilization**: Workloads with high memory bandwidth utilization tend to experience longer memory wait times. In such cases, boosting the CPU frequency might alleviate performance bottlenecks caused by memory access delays.
- **No uOps Issued Stalls**: Instances where no uops are issued due to stalls, possibly stemming from icache misses, result in wasted cycles. Importantly, faster CPU frequency does not appear to mitigate these wasted cycles.
- **Power**: Monitoring host power serves as a proxy for CPU activity. Higher CPU

activity levels may indicate busier CPU components, suggesting that increasing CPU frequencies could enhance overall performance.

These insights from the selected counters provide valuable guidance for optimizing system performance and power efficiency in datacenter operations. The implementation of DVFS boosting in Meta's datacenters during the pandemic yielded significant capacity improvements. By carefully managing power consumption and adapting boosting to service heterogeneity, Meta was able to create an additional 12 MW of capacity [99]. This boost in capacity was equivalent to building and populating half a datacenter's worth of rack supply.

Despite the increased power consumption, the deployment remained power-neutral, as it helped avoid the need for building new datacenters and purchasing additional servers. Moreover, the marginal cost of power usage was lower than the marginal cost of server hardware and datacenter construction. In conclusion, the case study demonstrates the effectiveness of DVFS boosting as a scalable and adaptable solution for expanding datacenter capacity.

4.5 Discussion

In this chapter, we have explored memory management techniques at the operating system layer in the context of modern data-centric systems. The growing demands of these systems, coupled with evolving architectures, have underscored the critical importance of memory performance and energy efficiency. Traditional static approaches and workload-specific optimizations have proven inadequate in addressing the dynamic nature of contemporary computing systems.

To address this challenge, we develop a software artifact called MARS to deploy self-aware memory management techniques discussed in Chapters 1 and 2 on linux based systems. Using MARS, we develop intelligent policies capable of dynamically adapting memory configurations, allocations, and access policies to optimize both performance and energy efficiency.

To illustrate the practical application of these principles, we presented two case studies. In Case Study 1, we explored memory-driven application classification, a technique that combines memory size requirements and runtime DRAM bandwidth needs to dynamically optimize memory controller frequencies. This approach not only demonstrated the potential of computational self-awareness but also highlighted its pertinence in addressing real-world challenges.

In Case Study 2, we delved into the realm of datacenter capacity expansion during the COVID-19 pandemic. Here, we witnessed how Meta effectively leveraged CPU DVFS boosting to overcome unprecedented challenges. This case study underscored the adaptability and scalability of DVFS boosting, emphasizing the significance of model-based proactive techniques in addressing the complex demands of modern datacenter environments.

We envision a future where these systems not only meet the demands of data-intensive applications but also maintain the application level quality constraints. In the upcoming chapters, we will continue our exploration of intelligent memory systems that adapt dynamically, optimize resource utilization efficiently in the application layer.

Chapter 5

Application Abstraction Layer

In this final chapter, we delve into the Application Abstraction Layer, where most of the application logic written and maintained by developers, for confronting memory management challenges in data-centric applications. With the growing demand for real-time, data-centric applications, effective management of memory resources is an area that warrants considerable attention. It is also in this layer we can consider service level quality constraints (such as latency, throughput) that are critical for a successful application.

5.1 Emerging data-centric architectures and end-to-end applications

In contemporary computational systems, a noticeable shift has occurred towards data-centric architectures, driven by the escalating complexity and volume of data that these systems are tasked with handling. Whether it's self-driving vehicles or large-scale language models, the trend is evident: these systems demand high-volume data processing, real-time analytics, and seamless integration across diverse components.

Data-centric architectures are meticulously designed to manage substantial data loads while accommodating complex algorithms, ranging from machine learning models to specialized processing routines. Coupled with an end-to-end approach, these architectures facilitate the seamless interaction of various system components, ensuring optimal system functionality and efficiency.

However, managing memory resources in these systems presents formidable challenges due to the concurrent execution of diverse tasks. Each task may generate and consume data at varying rates and volumes, potentially leading to system bottlenecks and inefficiencies.

5.2 Motivation for benchmarking and optimization of data-centric applications on embedded systems

Embedded systems are at the core of many data-centric applications, necessitating effective strategies for memory management, benchmarking, and optimization. Due to the real-time and diverse requirements of these systems, traditional benchmarking methods often fail to capture their unique complexities.

Embedded systems, like those in autonomous vehicles, rely on a mix of computational platforms, including CPUs, GPUs, and dedicated accelerators. Each component presents its own workload characteristics and memory requirements. Therefore, a comprehensive approach to benchmarking and optimization must accommodate this diversity.

Furthermore, these systems typically operate under strict power constraints, particularly in battery-powered devices. As such, optimization strategies must balance performance with power consumption. Efficient memory management plays a significant role in achieving this balance. This emphasizes the need for application level techniques for capturing the intricate

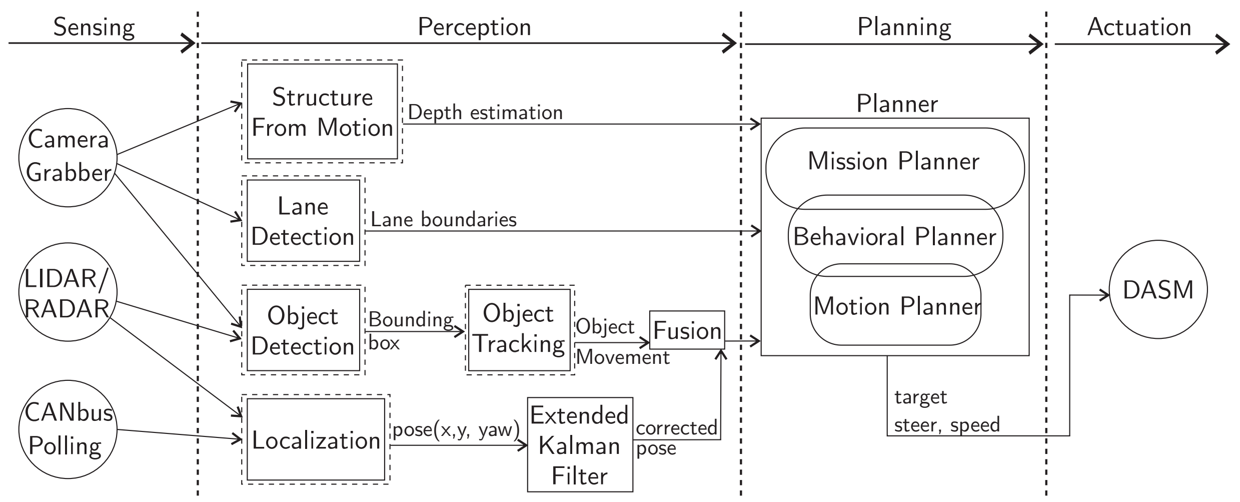


Figure 5.1: Software components and flow of information in a self-driving vehicle. Interfaces with sensors and actuators are marked with a circle.

complexities of data-centric applications on heterogeneous computing platforms. The Application Abstraction Layer, discussed in the subsequent sections, presents an understanding of application needs and behaviors, and provides policies for managing memory resources effectively. Thus, this layer can capture the holistic view of application requirements.

5.3 Case study 1: Chauffeur The First Open-Source End-to-End Benchmark Suite for Self-Driving Vehicles

Designing self-driving vehicles requires the orchestration of an ensemble of different self-driving workloads working in an end-to-end manner, guaranteeing functional and performance constraints. Figure 5.1 shows the task graph of a generic end-to-end self-driving software pipeline. Based on this representative task graph, we investigate and formalize the requirements of a benchmark suite for self-driving vehicles geared toward system designers and researchers.

A self-driving system must plan and follow a trajectory to reach a given destination using real-time sensor information[16, 118, 155]. Consider Figure 5.1 where we present a generic software pipeline for operating a self-driving vehicle. The pipeline shown here is implementation-independent and captures the logical computation blocks (tasks) and data flow pipeline (relation between tasks). We can divide the pipeline into four stages: Sensing, Perception, Planning, and Actuation. (1) Sensing is composed of three tasks associated with collecting real-time data from hardware sensors: Camera grabber, LIDAR/RADAR sensing, and CANbus polling. These tasks are responsible for reading raw sensor data connected using either (a) Automotive Ethernet [43], or (b) Controller Area Network (CAN bus). (2) The Perception stage uses the real-time raw information to extract relevant environmental knowledge for decision-making. The tasks involved are (a) depth estimation (Structure From Motion), (b) detection of lane boundaries (Lane Detection), (c) bounding and identification of surrounding objects (Object Detection), (d) tracking of detected objects, and (e) position estimation (Localization). (4) In the Planning stage, the planner uses the perception information to determine the future trajectory by setting the target steer and speed. (5) Finally, the Drivers Assistance System Module (DASM) uses CANbus to perform the hardware actuation and control the vehicle.

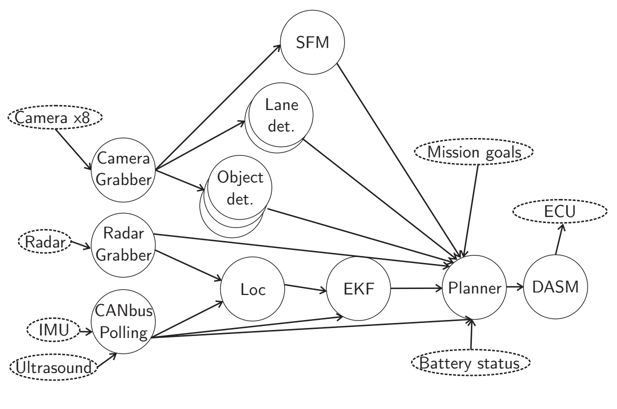
The generic self-driving software pipeline in Figure 5.1 can be instantiated in different ways. Vendor implementations vary in (a) the number and types of sensors, (b) the number of instances of software tasks, and (c) the underlying hardware on which the stack is running. For instance, Tesla's Full Self-Driving (FSD) computer [143], is equipped with eight cameras and uses a radar distance sensor along with multiple ultrasound sensors surrounding the car (Figure 5.2a) for autonomous transportation. On the other hand, self-driving warehouse robots have different requirements: 8 cameras surrounding the vehicle are excessive, and vendors often deploy a LIDAR sensor instead of cameras [146]; and Lane-Detection is no longer required. An example pipeline representing such a scenario is shown in Figure 5.2b. Although the task graph has a different instantiation, the end-to-end pipeline will still need to be designed effectively for the hardware on which it is running.

5.3.1 Requirements of a benchmark suite for self-driving vehicles

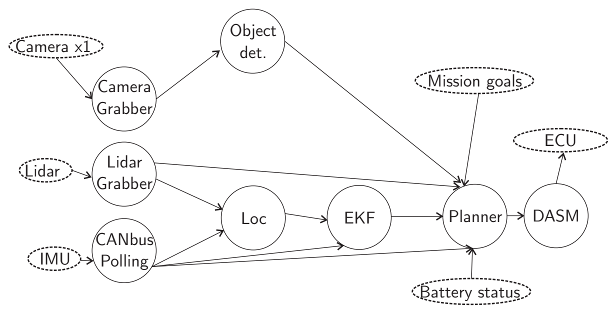
Designing self-driving vehicles requires cross-layer collaboration from researchers in diverse domains, including hardware designers, system software architects, and application developers. Running a full-stack end-to-end simulation is often time-consuming and infeasible on resource-constrained embedded boards. Instead, we focus on flexibly composing a computational pipeline of critical software components representing real autonomous driving workloads. To support the flexible exploration of end-to-end configurations, we identify the following requirements for a benchmark suite for self-driving vehicles:

Self-driving workloads

Self-driving workloads consist of sensor-driven, resource-intensive applications with real-time safety-critical requirements[96]. In Section 5.3.3, we describe the applications which constitute a typical self-driving scenario. Future systems require a redesign of the hardware



(a) Example instance for urban driving scenario.



(b) Example instance for warehouse robotics scenario.

Figure 5.2: Implementation-specific instances of the generic self-driving pipeline with different tasks.

and heterogeneous resources to meet these applications' real-time performance requirements.

Configurable end-to-end pipeline

The performance and efficiency of autonomous driving systems require researchers to analyze the end-to-end pipeline: sensing, perception, planning, and actuation. This requires studying different end-to-end pipelines. Very few works [50] [23] [154] [57] [158] have been able to study the system performance of end-to-end pipelines as the monolithic software stacks are too complex to deploy on an embedded platform. To the best of our knowledge, there is no configurable end-to-end pipeline that can be used to study self-driving workloads.

Embedded Runtime

A key challenge in designing self-driving vehicles is that resource-hungry applications consume a lot of power. This is critical for battery-powered vehicles where users need high mileage operation from a single charge. Several optimizations have been made for such platforms (e.g., using shared physical memory between the CPU and GPU). However, real-time performance requirements, energy constraints, and safety are extremely hard to satisfy simultaneously. While trading off performance for energy can cause problems in the vehicle's safe operation, having energy optimizations as an afterthought leads to a poor design. For practical deployment, it is essential to evaluate these workloads on resource-constrained platforms.

Heterogeneity

Integrated GPUs are already ubiquitous. Future systems trend is to meet the real-time performance requirements by accelerating the bottlenecks in applications either through GPUs or dedicated on-device hardware accelerators for specific kernels. Consequently, applications must support multiple, heterogeneous resources (e.g., CPU version, GPU version) and utilize any available resource at runtime. Fickenscher *et al.* in [35] have analyzed automated code generation for self-driving algorithms on these heterogeneous platforms with the help of Domain-Specific Language (DSL).

Diverse platforms

As applications embrace heterogeneity, vendors have developed various platforms to support their runtime. Notably, heterogeneous multi-cores augmented with graphics processing units (GPUs) as accelerators show promise to meet real-time self-driving workloads' performance requirements. NVIDIA strongly advocates such an approach through their series of embedded platforms like Jetson TX2 (integrated GPU with shared physical memory, not aimed for safety-critical applications) and Drive PX2 (discrete GPU, provides dual-modular redundancy features for safety-critical applications). A benchmark suite should not only be representative of the real-time workloads but also support these types of diverse embedded platforms.

Research support

Open-source benchmarks that are easy to download and run are critical to engage and support research in this area. Chauffeur would enable hardware designers and system researchers to go beyond full stack driving simulators that are difficult to run on low-resource boards. We provide the infrastructure to instrument and derive the performance implications on state-of-the-art systems efficiently.

5.3.2 Limitations of existing benchmark suites

Table 5.1 summarizes the status of existing benchmark suites, showing how they fail to meet all of the requirements discussed above, while showing the capabilities of Chauffeur; we expand on Table 5.1 below.

Traditional embedded benchmark suites such as PARSEC [14], MiBench [41], SPEC [48] include programs from different domains like computer vision, media processing, enterprise servers, animation physics. These benchmarking suites mainly focus on computer architecture and system/hardware design. However, they fail to capture the implications of highly data-intensive applications having stringent performance requirements that constitute a typical self-driving scenario. These benchmarks either focus solely either on CPU workloads (e.g., PARSEC) or on GPU workloads (e.g., Rodinia [22]). The narrow scope fails to capture the correct ratio of heterogeneous resource utilization in self-driving vehicles.

CAVBench is a benchmark suite targeted towards evaluating autonomous driving computing system performance [147] in a connected vehicle setting. It focuses on six workloads: SLAM, object detection, object tracking, battery diagnostics, speech recognition, and edge video analysis. CAVBench analyzes the execution time breakdown for each application and the Quality of Service (QoS) – Resource Utilization (RU) curve (QoS-RU curve). The QoS-RU curve is used to calculate the matching factor (MF) between the application and the computing platform on autonomous vehicles. CAVBench serves as an initial artifact to study edge computing systems for autonomous driving but fails to present a holistic view of the end-to-end self-driving scenario.

Autoware [55] is a popular open-source full-stack driving simulator that is expected to be deployed on autonomous vehicles. Autoware is based on Robot Operating System (ROS) and other well-established open-source software libraries. However, full-stack simulators typically require powerful platforms (e.g., recommended system for evaluating Autoware is

Table 5.1: Popular benchmark suites and the key challenges addressed. ($*$ = uniquely addressed by Chauffeur).

Features	**Traditional benchmarks** [14, 41, 48]	**CAVBench** [147]	**Autoware** [55, 56]	**Apollo** [1]	**Chauffeur**
Self-driving workloads		✓	✓	✓	✓
Configurable end-to-end pipeline					$*$
Embedded Runtime	✓		✓		✓
Heterogeneity		✓	✓	✓	✓
Diverse Platforms	✓				✓
Supports research	✓	✓	✓	✓	✓

an 8-core X866 CPU with 32GB of main memory, which is infeasible for embedded platforms). Researchers have redesigned Autoware to customize the software stack to run on NVIDIA DRIVE PX2 computing platform to study self-driving workloads [56]. However, such customizations restrict hardware designers to studying a single workload (one specific implementation) rather than different algorithms for the same task. Significant customization is required to port such a complex stack to emerging embedded platforms.

This problem is further exacerbated in **industry-standard** autonomous driving software frameworks like **Apollo**[5]. These frameworks are developed with the application (self-driving vehicle) in mind. However, operating such stacks on low-resource embedded hardware requires careful system design by researchers to account for architectural implications [67]. Hardware designers and system researchers find it cumbersome and time-consuming to set up an end-to-end driving stack to study hardware/architectural implications.

Chauffeur incorporates the features highlighted in Table 5.1 using a set of benchmarks aimed at hardware designers and system researchers. It comprises state-of-the-art representative applications from the domain of self-driving vehicles and targets low-resource embedded systems. Chauffeur is open-source and easy to download and run, enabling quick analysis of system implications of a configurable end-to-end self-driving pipeline. Chauffeur [80] is a benchmark suite comprising some essential applications for designing self-driving vehicles, with the ability to expand with additional applications. We categorize the (initial ten) applications in Chauffeur into four stages: (1) Sensing, (2) Perception, (3) Planning, and (4) Actuation. Sensing applications receive sensory information from different communication buses and share it with the rest of the pipeline. Perception applications collect the raw information shared by sensing applications and extract relevant knowledge used for decision making [98]. Perception provides a contextual understanding of the vehicle's environment (e.g., what the different objects are, location of the objects, road signs, traffic cones), and the vehicle's position with respect to the environment. The planning application develops and continuously updates the vehicle's trajectory to achieve the higher-level goals of the user (e.g., driving from Redmond to Seattle) while following the rules of the road. Finally, actuation applications control the vehicle and execute the planned actions generated by the previous stage. The applications defined in Chauffeur are not vendor/implementation-specific and apply to different self-driving use cases (e.g., autonomous driving versus warehouse robots).

5.3.3 Description of self-driving application categories

Camera Grabber

Visual sensors (e.g., CMOS camera) are a vital component to enable perception about the environment in self-driving vehicles. Typically, visual sensors generate a lot of data and require a high-bandwidth communication bus. This requirement is incredibly stringent when interfacing multiple cameras simultaneously, as shown in Figure 5.2a. Car manufacturers use automotive ethernet to transfer such high volume data with very low latency and meet real-time performance requirements. The sensing application *camera grabber* is a representative workload for such processes. The camera grabber is responsible for receiving the packets from the network and then placing them on the main memory for the following stages.

LIDAR/RADAR

Although cameras are reliable and relatively cheap to produce, perception solely using camera data is non-trivial as it relies on black-box neural networks. Traditionally, car manufacturers use detection and ranging sensors (e.g., LIDAR, RADAR) to detect surrounding objects and calculate distances. Distance sensors assist in hazard detection and range-finding in features like adaptive cruise control (ACC) and automatic emergency braking (AEB). These sensors are imperative during adverse weather and lighting conditions and are still prevalent in modern vehicles.

CAN bus Polling

The Controller Area Network (CAN bus) is another integral interface found in automobiles that automobile engineers use to interface with the vehicle's hardware. CAN bus is typically used for low-volume data transfers with high reliability. It serves the following purposes: (1) reading the current status of the vehicle (e.g., Odometer value), (2) interfacing with CAN bus sensors (e.g., Inertial Measurement Unit (IMU)), and (3) controlling the steer and speed of the vehicle.

Structure From Motion (SFM)

SFM is a perception application that aims to reconstruct three-dimensional structures from a sequence of two-dimensional moving images [44]. SFM uses the subsequent images to triangulate the 3D position of objects in the environment.

Lane Detection

Lane detection is a perception application that aims to detect road boundaries (lane line markings) and estimate the vehicle pose with respect to the detected lines using visual sensors on the vehicle. The application includes the localization of the road, the determination of the relative position between vehicle and road, and the analysis of the vehicle's heading direction [8].

Object Detection

Vision-based object detection is a perception application that is one of the primary prerequisites for self-driving vehicles. Distance and ranging sensors (e.g., LIDAR, RADAR) alone are not sufficient to meet the requirements of self-driving. For example, RADAR sensors, albeit working in adverse environmental conditions, do not produce a high precision output. On the other hand, information from LIDAR sensors, albeit extremely precise, are too complicated to process and prohibitively expensive. Modern object detection applications use neural networks along with visual sensor data to identify objects in the area surrounding the vehicle by drawing bounding boxes and classifying the object inside each bounding box.

Object Tracking

Given some objects of interest marked in a frame, the object tracking application locates the objects in subsequent frames in the video [46]. Object tracking is a part of the perception stage, and it tracks objects as they move in the environment. It also allows the self-driving vehicle to estimate the motion of objects and predict how they will move in the subsequent frames.

Table 5.2: Applications in a typical self-driving vehicle; highlighting inputs and outputs and how different applications are related.

Applications	**Stage**	**Input**	**Output**
Camera Grabber	Sensing	Packets over Automotive Ethernet	$\textcircled{S1}$ Image frames in the shared memory
LIDAR/RADAR	Sensing	Packets over Automotive Ethernet	$\textcircled{S2}$ Point cloud in the shared memory
CAN bus pooling	Sensing	Messages (Frames) over CAN bus	$\textcircled{S3}$ Sensed information in shared memory
Structure From Motion	Perception	$\textcircled{S1}$	$\textcircled{P1}$ Depth estimation
Lane Detection	Perception	$\textcircled{S1}$	$\textcircled{P2}$ Lane Boundaries
Object Detection	Perception	$\textcircled{S1}$, $\textcircled{S2}$	$\textcircled{I1}$ Bounding Box
Object Tracking	Perception	$\textcircled{I1}$	$\textcircled{I2}$ Object movement
Localization	Perception	$\textcircled{S2}$	$\textcircled{I3}$ position and orientation pose(x, y, yaw)
Extended Kalman Filter	Perception	$\textcircled{I3}$, $\textcircled{S3}$	$\textcircled{I4}$ Corrected pose
Fusion	Perception	$\textcircled{I2}$, $\textcircled{I4}$	$\textcircled{P3}$ Fused object and vehicle location
Path planner	Planning	$\textcircled{P1}$, $\textcircled{P2}$, $\textcircled{P3}$	$\textcircled{A1}$ Spatio-temporal trajectory
DASM	Actuation	$\textcircled{A1}$	Steer, Brake

Localization

Localization is a perception application that works closely with the environmental perception using visual sensors to identify the vehicle's position within the perceived environment. Global Positioning System (GPS) is the most commonly used localization system used in the vehicle industry. Although cheap and easily accessible, GPS suffers from poor reliability and accuracy and is not a good candidate for localization applications in self-driving vehicles [62]. Researchers have developed advanced sensors (e.g., RADAR, LIDAR, Visual sensors) that can further be fused to provide more robust, accurate, and reliable localization used in modern vehicles.

Extended Kalman Filter (EKF)

The result of localization (using distance sensor like RADAR) is prone to drift over time and can be noisy. A Kalman filter is an excellent candidate for combining the distance information with other vehicle-status information (e.g., IMU, Odometer, GPS) and handling such disturbances. Kalman filter fuses multiple data sources and performs continuous prediction (for missing data) and correction (for drifting data) on the localization results. The EKF application is the non-linear implementation of the Kalman filter [108].

Fusion

The fusion task helps combine object information with the car's location on the fly. The information is sourced by pre-processing raw data from different sensors (e.g., cameras, different types of RADAR, LIDAR) and fused synchronously. The fusion process involves transformation of different coordinate systems and updating the environment maps periodically in real-time. These tasks have inherent data parallelism and are good targets for hardware acceleration [36].

Path Planner

The planning stage is responsible for understanding higher-level goals from the user (e.g., Navigate from Seattle to Redmond) and convert them to purposeful decisions to achieve the higher-level goals while avoiding obstacles [98]. The complexity of this stage compels a hierarchical design by partitioning the software into layers: (1) Mission planning, (2) Behavioral planning, and (3) Motion planning. *Mission planning* computes the global trajectory based on the current location and the target destination along with stops and which roads can be taken to achieve this application (e.g., avoid freeways). *Behavioral planning* generates local objectives (e.g., Change lanes, overtake) to interact with other agents on the path and follow the rules of the road. *Motion planning* takes the local objectives and generates the control plan to actuate the steering and speed. Although most recent works [113, 124, 6] follow some implementation of the planner hierarchy, the exact partitioning within the planner often varies between implementation.

Drivers Assistance System Module (DASM)

The DASM application performs the final actuation stage in the pipeline. The local objectives of the path planner (through motion planner) are realized through a PID controller. The PID controller actuates the speed and steering based on the velocity and angle commands and can automatically use the odometer and IMU feedback to maintain the targets set by the path planning stage.

One of the goals that we pursue with the Chauffeur benchmark suite is to identify the system implications of the representative applications on emerging embedded platforms. We offer insights obtained by profiling the applications under different configurations (e.g., integrated

Table 5.3: Implementations used in Chauffeur.

Implementation	**Application**	**Dataset**	**Data Size and Type**
ROS	Camera Grabber	application-specific	variable
ROS	LIDAR/RADAR	application-specific	variable
OpenMVG [89]	Structure From Motion	provided	360° 5376x2688 color images
Jetson Inference [93]	Object Detection	cuda-lane-detection	30FPS h.264 1280x720
darknet-ros [15]	Object Detection	KITTI Odometry Dataset [38]	1392x512 color images
lidar-tracking [97]	Object Detection & Tracking	KITTI Odometry Dataset	3D Velodyne point cloud 100k points per frame
LaneNet [73, 148]	Lane Detection	tusimple-benchmark	1280x720 color images
cuda-lane-detection[52]	Lane Detection	provided	1280x720 30FPS h.264 video
FLOAM [42]	Localization	KITTI Odometry Dataset	3D Velodyne point cloud 100k points per frame
orb-slam-3 [17]	Localization	KITTI Odometry Dataset	1392x512 greyscale images
EKF [114]	Extended Kalman Filter	provided	radar and lidar pose estimates
Hybrid A* [61]	Path planner	provided	2D obstacle map

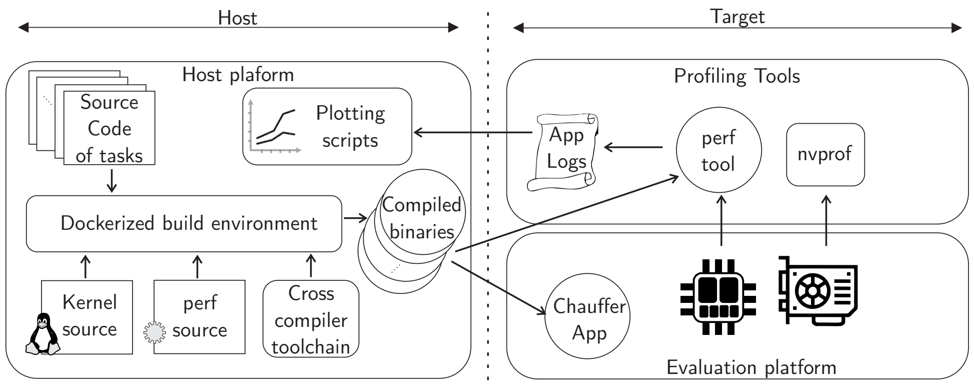


Figure 5.3: Tool flow for using Chauffeur suite.

vs shared main memory, different degrees of parallelism). We believe this will serve as a good starting point for researchers to design computing platforms for future self-driving vehicles.

5.3.4 Characteristics of Chauffeur: representative workloads, and performance evaluation

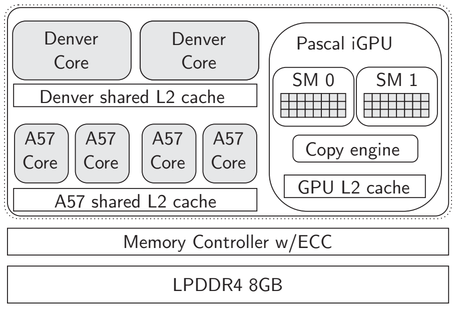
A significant challenge faced by system designers when using existing software stacks to analyze end-to-end performance bottlenecks and exploration of different platform configurations is the complexity of configuring and running them on embedded platforms. Chauffeur overcomes these barriers through a tool flow that enables researchers to analyze and evaluate different platforms quickly. We describe the Chauffeur tool flow and present an example of this tool flow for evaluating end-to-end performance evaluation.

Tool Flow

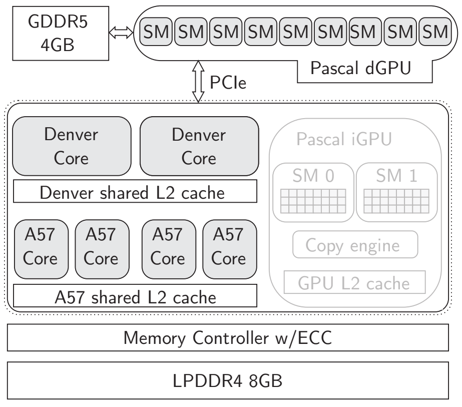
Figure 5.3 illustrates the Chauffeur tool flow across the host and target platforms. Users have two options to compile applications for the target platform: (1) directly compile the source on the board, or (2) use cross-compilation. We provide a dockerized build environment that builds the source code of applications and includes necessary dependencies. The application binaries are then deployed on the target platform and profiled using different tools. We use *perf* for IPC and CPU performance counters, *nvprof* and *NSight Systems* for GPU profiling. The tools are used to understand the implications of the end-to-end pipeline on the system. We include the scripts used for compiling, executing, and profiling as part of the repository. The compilation, deployment, and profiling steps are currently not completely automated and must be performed in a step-by-step fashion, manually, as explained in the repository.

Sample Experimental Evaluation platforms

We illustrate the use of Chauffeur to comparatively evaluate application execution on two exemplar embedded hardware platforms from NVIDIA: (a) NVIDIA Jetson TX2 platform, and (b) NVIDIA Drive PX2 platform. These emerging embedded platforms are widely adopted in many self-driving use cases (e.g., warehouse robotics, Tesla's Autopilot Hardware 2.0). Due to the widespread adoption of these platforms, prototype design and product performance evaluation are effortless as researchers can compare different policies against the same hardware.



(a) Jetson TX2 Architecture (figure adapted from [2]). Parker SoC includes integrated GPU (iGPU) with shared main memory.



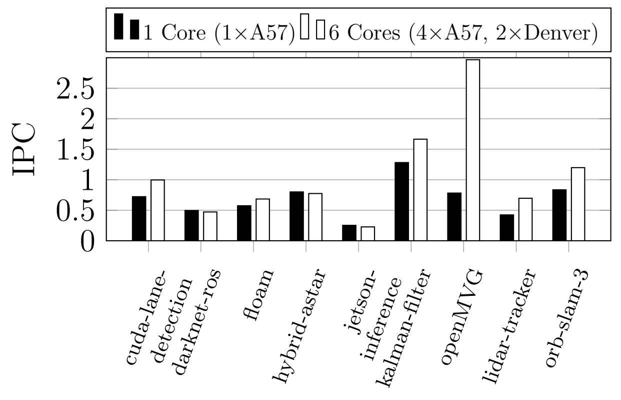
(b) Drive PX2 Architecture: Parker SoC with discrete Pascal GPU connected with PCIe. iGPU is not used for experiments.

Figure 5.4: Architecture of exemplar NVIDIA evaluated platforms.

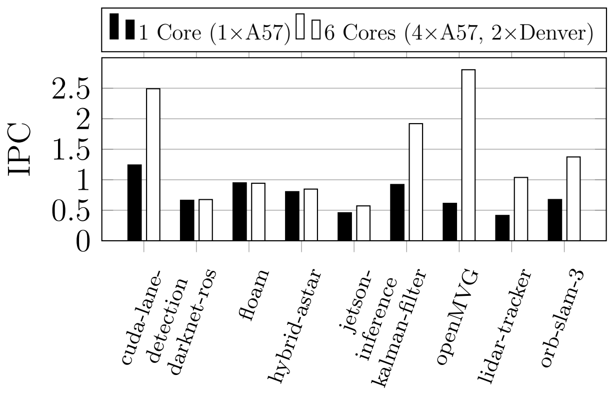
Figure 5.4a shows the architecture of the Jetson TX2. The TX2 consists of a Parker system on a chip (SoC) with two super Denver (NVidia proprietary) cores and four big A57 (ARM) cores. The Parker SoC includes an integrated Pascal GPU (iGPU) with two Streaming Multiprocessors (128 cores each). The CPU and GPU share 8GB LPDDR4 main memory. The Linux version used is 4.9.140-tegra, and the CUDA runtime library version is 10.0. Figure 5.4b shows the architecture of the Drive PX2. Like the TX2, the PX2 has a Parker SoC with two super Denver cores and four big A57 cores. We do not use the on-chip Pascal iGPU (grayed-out in the figure) in our experiments, as the Drive PX2 has a more powerful discrete GPU (dGPU). As a result, the entire 8GB LPDDR4 main memory is dedicated to the CPU. The dGPU consists of nine Streaming Multiprocessors (128 cores each), and is connected to the Tegra SoC using a PCIe bus. The dGPU also has a dedicated 4GB of GDDR5 memory. The Linux version used is 4.9.80-rt61-tegra, and the CUDA runtime library version is 9.2. Although the Drive PX2 supports dual modular redundancy by providing two instances of the described hardware architecture, we do not use the second Parker SoC/dGPU for our experiments. We run all experiments in maximum performance mode by disabling the dynamic frequency scaling of CPU cores and GPU.

The experimental platforms encompass diverse memory layout, and GPU compute capability. The shared main memory of Jetson TX2 creates two challenges at runtime: (1) memory space is a limiter for parallel applications when the end-to-end pipeline is considered, (2) memory contention between CPU and GPU workloads creates a memory bandwidth bottleneck. The discrete GPU on the Drive PX2 can alleviate challenges (1) and (2) in some cases. Still, the cost of memory copies before launching and after finishing kernel execution may outweigh the benefits. We explore some of these challenges by analyzing the Chauffeur applications that form the end-to-end self-driving pipeline.

To demonstrate the utility and flexibility of Chauffeur, we characterize Chauffeur applications and observe the performance implications of these Chauffeur applications on the two



(a) Jetson TX2.



(b) Drive PX2.

Figure 5.5: Comparison of *instructions per cycle* (IPC) (averaged across all cores) across different NVIDIA embedded platforms.

exemplar NVIDIA embedded platforms, to generate takeaways that can guide the exploration of different end-to-end self-driving pipelines. First, we report the utilization of the compute micro-architecture. Then, we identify the resource bottleneck for each application. Finally, we present the power breakdown among the different resources. Identification of performance and power bottlenecks of Chauffeur applications serve as guidance for future optimizations.

IPC of Chauffeur applications

Our first goal is to observe how well applications can utilize platform CPU's micro-architecture. We use *instructions per cycle* (IPC) to show on average how many instructions the CPU can retire in each clock cycle. Improving IPC directly translates to lowering execution time for the application, critical for performance in self-driving applications. The Cortex A57 cores in our exemplar NVIDIA platforms (Figure 4) contain a 3-wide decoder front-end for fetching instructions. Meanwhile, the Denver (”super”) cores are implemented by NVIDIA and have a 7-wide decoder width. The maximum theoretical IPC for A57 cores is 3, and Denver cores are 7. Applications can exhibit a low IPC for various reasons: (a) the processor pipeline is not able to fetch enough instructions for the execution stage, (b) incorrect speculations, or (c) not enough resources to retire instructions (e.g., not enough cores, high cache misses). Typically IPC > 1 indicates that the application is instruction-bound (bottle-necked by code execution on CPU cores). IPC < 1 shows some resource (e.g., Memory, GPU) is stalling code execution, and further investigation is required to identify the bottle-necked resource.

Figure 5.5 shows the observed IPC of Chauffeur applications. The periodic applications are operated in a data-ready mode to discount any idle periods. Figure 5.5a is the result of execution on Jetson TX2 and Figure 5.5b is the result of execution on Drive PX2. We make the following observations: (1) average IPC for six cores (1.4) $>$ average IPC for one core (0.8). While this confirms the intuition that applications, in general, perform better with more cores, we explore effective speedup from parallelization in more detail in Section 5.3.4. (2) Some applications (e.g., openMVG, kalman-filter, orb-slam-3) have much better IPC when the super cores are enabled. Therefore, it is better to map them to super cores instead of big cores. (3) Object detection applications suffer from the lowest average IPC across all core configurations (average jetson-inference IPC is 0.42, average darknet-ros IPC is 0.61).

Takeaways:*(1) We need to investigate the degree of parallelism of high IPC applications (cuda-lane-detection, openMVG) (2) For the applications with a low IPC but running on a GPU, we need a full system analysis for identifying if GPU is the actual bottleneck. (3) For the remaining applications, we need to look into the memory access behavior.*

Effective speedup from parallelism

Applications with large inputs and working sets are typically good candidates for exploiting parallelism. We explore Chauffeur applications' ability to exploit CPU parallelism by increasing the number of online CPU cores. Figure 5.6 shows the results in terms of speedup.

We make the following observations: (1) All applications gain speedup when increasing from 1 to 2 cores. For applications that are not explicitly parallelized (e.g., hybrid-star, jetson-inference, kalman-filter), this benefit comes from multicore execution thanks to reduced contention even in single-threaded implementations. (2) Certain applications (e.g., kalman filter, lidar-tracker) perform better when supercores are enabled (increasing from 4 to 5 cores). They can benefit from supercores' powerful floating-point units or larger L1 cache; (3) Some applications (e.g., OpenMVG, lidar-tracker) show linear speedup by increasing from 1 to 6 cores. This results from parallelization (OpenCV multi-threaded APIs for lidar-tracker and OpenMP for OpenMVG). (4) openMVG (SFM) experiences up to 3.3× speedup on Jetson TX2 and up to 3.9× speedup on Drive PX2 when compared to a single-core execution. The application operates on large images, is multi-threaded, and data-parallel. Hence it benefits a lot from multiple cores. However, the current implementation does not leverage the GPU. (5) jetson-inference (Object detection) experiences no speedup on the Jetson TX2 and up to 1.3× speedup on the Drive PX2. Although it spends 52% time executing the decoder thread on CPU, a large number of *memcpy* operations limits the degree of parallelism. (6) darknet-ros (Object detection) does not show any speedup with an increasing number of cores. We conclude it is not core bound. Candidate backend bottlenecks for darknet-ros include memory and GPU. (7) floam (Localization) is a compute-intensive application with the current implementation running only on CPU. The execution time to process one input on Drive PX2 is around ≈150ms and is not affected much by increasing cores. However, on Jetson TX2, we see it starts poorly (≈250ms for one core), improves with more cores (≈106ms for four cores), but performs worse with super cores. We performed a finer-grained analysis and found the source: a high number of branch mispredictions in super cores (≈16 Million/s) as opposed to big cores (≈13 Million/s).

Takeaways: (1) *floam (Localization) and openMVG (Structure from Motion) implementations are CPU (core) bound in the micro-architecture pipeline. (2) Further study is required for the remaining perception applications to identify their bottleneck. (3) Bigger cores with*

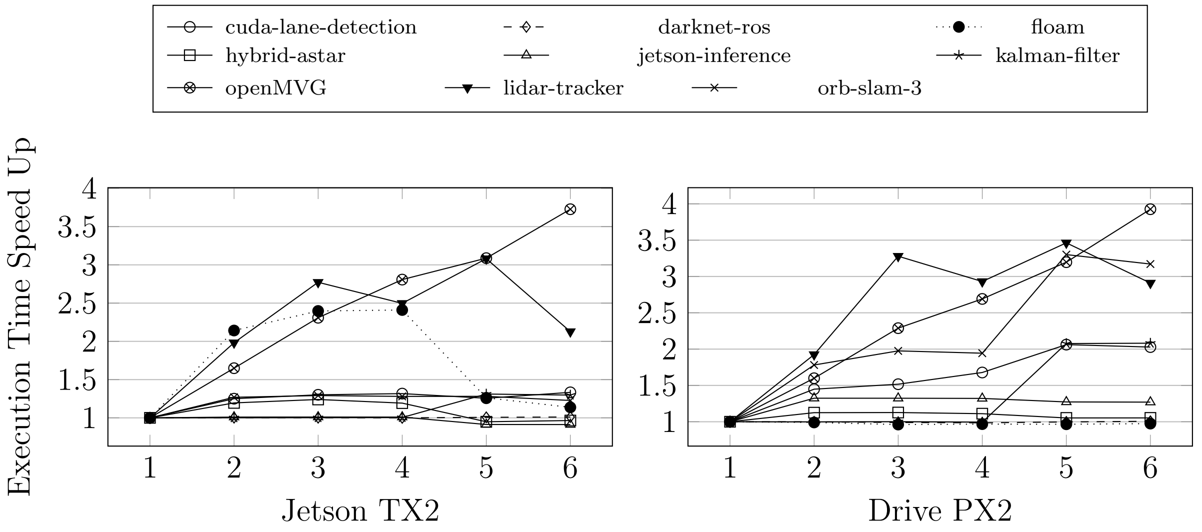


Figure 5.6: Speed-up of application execution time with increasing number of online cores.

wider instruction decode paths are not always a better choice for running applications, as bad speculations in modern CPUs can cause severe degradation in performance.

Main memory access by CPU

Table 5.4 shows the memory access characteristics of Chauffeur applications. We conduct these experiments on Jetson TX2 will all six cores active using MARS framework [91]. We make the following observations: (1) Although applications issue many memory requests, most accesses are served by the cache hierarchy (L1 and L2 cache). Thus, memory access rates are much higher than main memory bandwidth rates. (2) openMVG has the highest

Table 5.4: Comparison of memory access and main memory (DRAM) bandwidth (B/W) of Chauffeur applications on the Jetson TX2. Measured numbers are only from CPU performance counters and do not consider memory traffic from GPU. Unit is *million-transfers/sec (MT/s)*

Transfer (MT/s)	idle	cuda-lane-detection	darknet-ros	floam	hybrid-astar	jetson-inference	kalman-filter	lanenet-lane-detection	lidar-tracking	OpenMVG	orb-slam-3
Avg Access	9	717	120	534	499	199	1348	537	934	1852	1346
Peak Access	55	2703	944	1703	1318	1110	2820	2353	1512	7225	2088
Avg B/W	0	1	1	4	1	3	0	1	10	15	18
Peak B/W	0	19	14	25	16	17	13	17	21	89	67

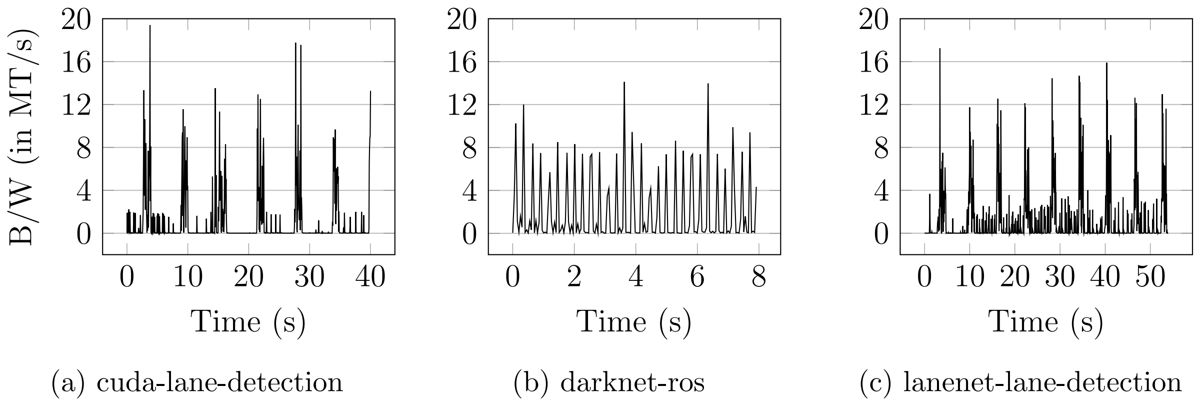


Figure 5.7: Main-memory access pattern of selected Chauffeur applications from CPU cores. Applications demonstrate memory accesses phases.

memory access rate and main memory peak bandwidth utilization. The high number of memory requests is a result of data-parallelism (as shown in Section 5.3.4) as openMVG exploits all cores. The CPU cores' fast and parallel computation on large inputs leads to numerous memory accesses, which increases the main memory bandwidth. (3) Several applications report extremely low average main memory bandwidth. This can be explained by looking at the memory access patterns over time (Figure 5.7). These applications have phases of memory accesses, causing a surge of memory bandwidth requirement followed by a computation phase (in CPU/GPU). **Takeaways:** *Memory access patterns are highly dynamic and very hard to model at design time. Runtime policies need to observe the memory access patterns and avoid overlapping memory phases between applications to reduce contention when running the end-to-end pipeline.*

Power profile of applications

The average instantaneous power is a direct representation of utilization of the onboard resources and demonstrates opportunities for future optimizations. Figure 5.8 shows the instantaneous power breakdown of Chauffeur applications on the Jetson TX2. We could not provide a comparative study due to the lack of power sensors in the Drive PX2 platform. We

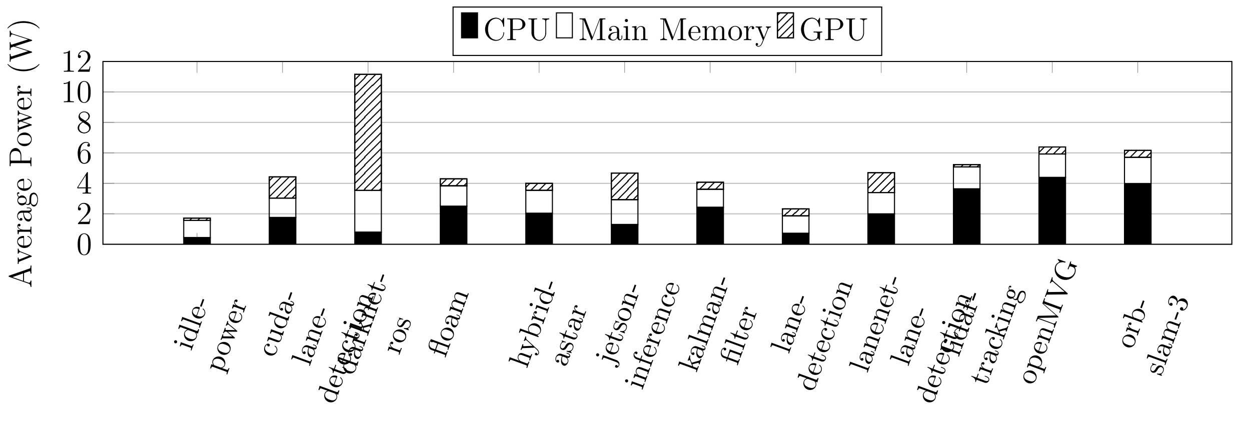


Figure 5.8: Power breakdown for Jetson TX2 using onboard I2C power sensors

make the following observations: (1) darknet-ros reports the highest GPU power consumption of 7.6W on the iGPU. (2) openMVG reports the highest CPU power of consumption of 4.4W on the iGPU. This is expected, as we observed in Section 5.3.4 that openMVG exhibits a high degree of parallelism (up to 3.7× speedup) with the increase in cores, and the reported results are for six cores. (3) Memory requires a more profound investigation for optimizing power. Chauffeur applications are a good target for accuracy/power tradeoffs, as some sensors might be more relevant than others depending on the scenario.

Takeaways: *(1) darknet-ros justifies hardware acceleration. (2) openMVG justifies GPU acceleration. (3) Approximate memory techniques [79], and efficient memory management techniques [74, 12] should be explored to reduce memory power.*

Chauffeur serves as a self-driving benchmark suite, focusing on end-to-end application performance. It provides a comprehensive evaluation of system behavior, identifying bottlenecks in performance, power, and memory across a wide range of data-centric applications. By examining the intricacies of system behavior and considering different types of data with varying criticality, Chauffeur demonstrates the universal relevance of an adaptable Application Abstraction Layer. The insights gained from Chauffeur enable the development of strategies for efficient memory management, ensuring optimal system performance in resource-limited embedded platforms.

5.4 Case study 2: OAsys : Self-Optimizing Autoscaler for ML Inference Serving Systems: A Holistic Approach

Datacenter operators must continuously ensure adequate compute capacity exists for increasing machine learning (ML) demands. The physical infrastructure that forms the backbone of datacenters requires a tremendous amount of energy to operate, and construction of new datacenters often extends over multiple years. Thus, datacenter operators employ forecast-based autoscaling strategies to sustain the rapid growth of ML workloads.

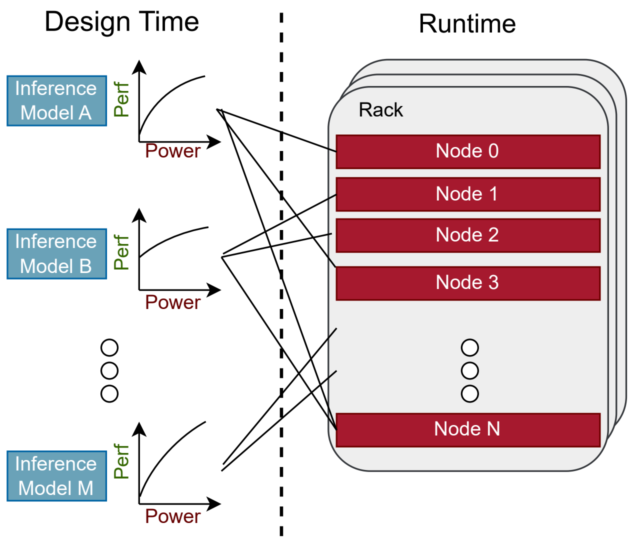


Figure 5.9: Datacenter operators rely on autoscaling of compute resources based on ML inferencing demands.

Autoscaling exploits the unique latency demands of concurrent workloads. While some workloads are latency-critical at specific times of the day, others can tolerate higher latency and may be scheduled during off-peak hours. To manage this complex landscape operators rely on autoscalers to adjust the number of nodes allocated to each workload based on elastic compute demands, optimizing both performance and resource allocation (Figure 5.9).

Traditional autoscaling approaches, however, focus solely on dynamically adjusting the number of hosts based on load patterns, and use a static system configuration for each host. Without dynamic runtime adaptation, the system configuration leads to suboptimal resource allocation and low throughput. For instance, a fixed batch size or resource configuration (e.g., CPU Turbo Mode) may result in underutilized resources and reduced throughput, as the system fails to fully exploit the available processing power. On the other hand, physical infrastructure poses additional constraints on the available power budgets: operating the system at full capacity might exceed allocated power budget, leading to performance degradation or even tripping circuit breakers.

We propose a novel approach to autoscaling called OASys using off-policy model-free reinforcement learning. Our main contributions are: (1) A multi-agent distributed reinforcement learning algorithm that considers both the application performance and infrastructure efficiency. We collect traces from a state-of-the-art datacenter infrastructure to train the off-policy algorithm. (2) Demonstration that by dynamically adapting system configurations based on workload characteristics and power budgets, we achieve higher throughput while meeting stringent performance requirements and minimizing energy costs.

OASys is thus demonstrated as a promising solution for efficient resource management in modern MLaaS inference serving systems.

5.4.1 Challenges of ML Inferencing with Traditional Autoscaler

In this section we demonstrate the intricate interplay between batch size, Quality of Service (QoS), and power control mechanisms that datacenter operators must carefully navigate in the context of ML inference as a service (IaaS).

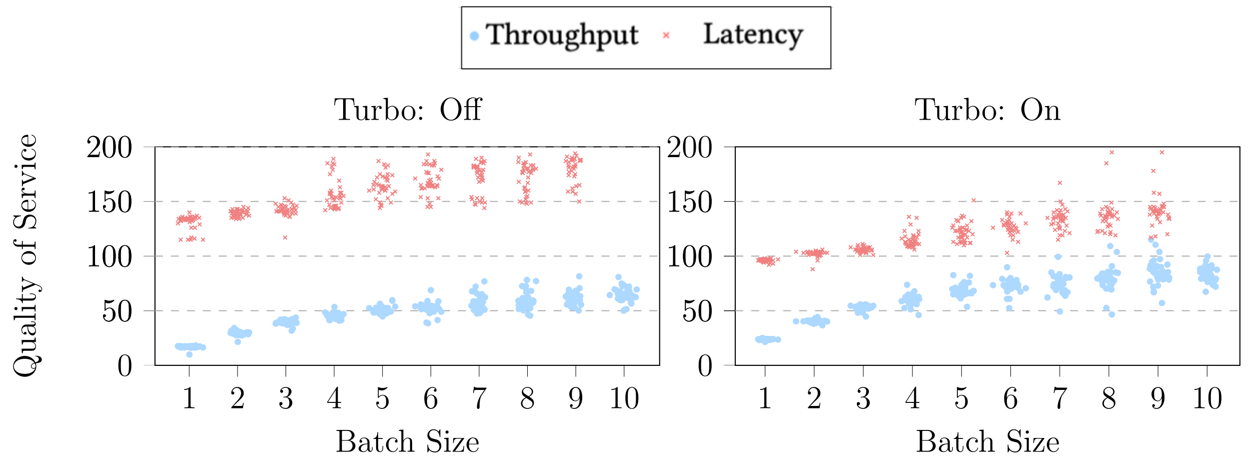


Figure 5.10: Effect of Batch size on Quality of Service. Enabling Turbo increases throughput and decreases latency.

Application-level Constraints

In ML workloads, the batch size refers to the number of input data points processed simultaneously during inference tasks. The selection of an appropriate batch size has implications on various aspects of workload performance, resource utilization, and QoS [141]. Application QoS encompasses the level of service quality and performance that the datacenter must deliver to meet user expectations and application requirements. Modern ML Frameworks (e.g., Torchserve) support dynamic batch sizes that operators use to meet QoS objectives based on demand and nature of workload. In Figure 5.10, we examine the impact of varying batch sizes on the QoS of an ML inference model (AlexNet [60]). The x-axis represents an incremental increase in batch size, with each step denoting an increase of 1. The y-axis quantifies QoS, measured in terms of both latency (red) in ms and throughput (blue) in inferences per second. We used the experimental setup detailed in Section 5.4.3 to collected the data. We make two key observations: (1) as batch size increases, throughput experiences a noticeable improvement, indicating enhanced system efficiency for parallel processing; (2) however, this increase in batch size also results in a corresponding rise in latency, which may impact realtime or low-latency applications negatively. Thus, operators must carefully select the appropriate batch size to strike a balance between optimizing throughput and minimizing latency in ML inference systems. It is worth noting that when latency is not a constraint, GPUs prove to be exceptionally well-suited for achieving massively parallel throughput [58], particularly valuable in the context of ML training. However, our focus in this work is on CPU-based ML inferencing for latency-critical tasks that cannot be deferred to off-peak hours.

CPU Turbo and Associated Risks

Host-level mechanisms (such as CPU Turbo [72], DVFS boosting [106]) can improve performance of CPU-bound applications. Figure 5.10 shows the effect of Turbo on QoS. By harnessing the extra power headroom, applications can simultaneously improve throughput (creating additional capacity for datacenters) and minimize latency. However, the effectiveness of turbo mode depends on the specific characteristics of the workload. Memory-bound workloads may experience only marginal improvements and could potentially waste the extra power, while CPU-bound workloads are more likely to benefit from enhanced QoS. Nonetheless, uncontrolled use of turbo without considering the rack-level dynamics of the datacenter power hierarchy [49] can cause service degradation, power capping, and even tripping of breakers in the worst case.

Achieving a delicate balance between power control mechanisms, QoS requirements, and dynamic batch size adjustments is a complex challenge that remains an open issue. As standard practice, many datacenter operators deploy autoscalers without altering system configurations in an attempt to address these challenges. We propose a model-free policy for configuring systems. This allows us to provide runtime adaptivity using real system traces without the necessity of observing actual power failures in datacenters, which would be required for generating data-driven models of datacenter power dynamics.

5.4.2 Partially Observable Markov Decision Process

In this section, we formally define the problem as a partially observable Markov Decision Process [45]. We consider a single rack denoted which comprises of N nodes. On these nodes, there exist M inference models running concurrently. **Objective**: Maximize the inferences per second served by all nodes within the rack while adhering to two crucial constraints:

1. Latency Constraint: The maximum latency for each inference model m should not be exceeded. This constraint remains fixed as determined by the application developer.
2. Power Budget Constraint: The total power consumption of the rack must not exceed the specified power budget.

Partially Observable MDP (POMDP)

To tackle this problem, we model each node as a POMDP [68] agent. Each agent has: (1) Observation, O_i, representing the current characteristics of workload on the rack. (2) Action, A_i, denoting the decision taken by the agent in response to the current observation. To choose actions, each agent i uses a parameterized policy $\pi_{\theta_i} : \mathcal{O}_i \times \mathcal{A}_i \rightarrow [0, 1]$. The problem is not fully observable as, although each agent can observe other agents' workloads at runtime, they cannot know the performance achieved by other nodes, or the actions (i.e., turbo mode, batch size) taken by other agents. We use the following notations throughout the problem formulation:

- N: The number of nodes in the rack, where each node is an agent.
- M: The set of inference models executing on rack, with $1 \leq |M| \leq N$. Specifically, $M = m_1, m_2, \ldots, m_{|M|}$.

Observation (O_i) for each agent

The observation at each node includes the following:

1. Index i: The index of the node within the rack.
2. X: A mapping of inference models to nodes is represented by a vector of size N where each element is an integer representing the model assigned to that node. Specifically, $\mathbf{X} = [x_1, x_2, \ldots, x_n]$, and each element $x_i \in M$, and represents which inference model is running on node i. This information is generated by the autoscaler.
3. Latency (l_i): Denotes the (99th percentile) tail latency of the inference process on node i.

Action (A_i) for each agent

The action vector for each node (i) includes the following:

1. Turbo Mode (T_i): A binary action for each node (i) indicating whether to enable ($T_i = 1$) or disable ($T_i = 0$) turbo mode.
2. Batch Size (B_i): An action for each node (i) that sets the batch size for the inference model running on that node. This action involves selecting a value from a predefined set of discrete options.

Reward design (R)

The reward function of each agent aims to balance the optimization of throughput, and adhere to latency constraints of the model served on the node.

1. Throughput Reward without Latency Constraints:

   To maximize throughput, we simply minimize each node's 99th percentile latency l_i without considering any constraints:

   $$\min \sum_{i \in N} l_i$$

   This may seem counter intuitive, however, Figure 5.10 shows that throughput increases along with tail latency as batch size increases. This is because parallelism improves total inference requests served at the cost of potentially underserved individual requests. We verified the effectiveness of our reward minimizing tail latency by comparing it to a reward maximizing speedup based on throughput of each node, and found that the latency reward had the desired effect on throughput without the burden of additional observation.

2. Throughput Reward with Latency Constraints

   To guarantee that latency constraints are satisfied on each node i, we impose the following constraint:

   $$l_i \leq L_m, \quad \text{where } m \text{ is the model on node } i$$

   If the latency constraint is met for an inference model on a node ($l_i \leq L_m$), the reward is equal to the latency measured for that model and node ($R_i = l_i$). If the latency constraint is not met ($l_i > L_m$), the reward is determined as the difference between the maximum allowable latency (L_m) and the actual latency (l_i):

   $$R_i = L_m - l_i$$

The objective that maximizes throughput while enforcing latency constraints:

$$\max \sum_{i \in N} R_i$$

where the reward of each node is:

$$R_i = \begin{cases} l_i, & \text{if } l_i \leq L_m \\ L_m - l_i, & \text{if } l_i > L_m \end{cases}$$

3. Power Constraints

Rack-level power constraint can be enforced by:

$$\sum_{n \in N} p_n \leq P_r, \quad \forall n \in N$$

This constraint ensures that the sum of the power of all nodes p_n in the rack does not exceed the rack's power budget P_r. However this requires power monitoring from all the nodes.

To simplify the constraint, we limit the power by indirectly enforcing a limit on the number of nodes with turbo mode enabled:

$$\sum_{n \in N} T_n \leq \text{MaxTurboNodes}$$

In the simplified constraint, $\sum_{n \in N} T_n$ represents the total count of nodes in the rack where turbo mode is enabled (indicated by $T_n = 1$). Thus, we limit the number of nodes with turbo mode enabled at any given time. MaxTurboNodes is determined during the design of datacenter power distribution. This simplification allows for a more direct

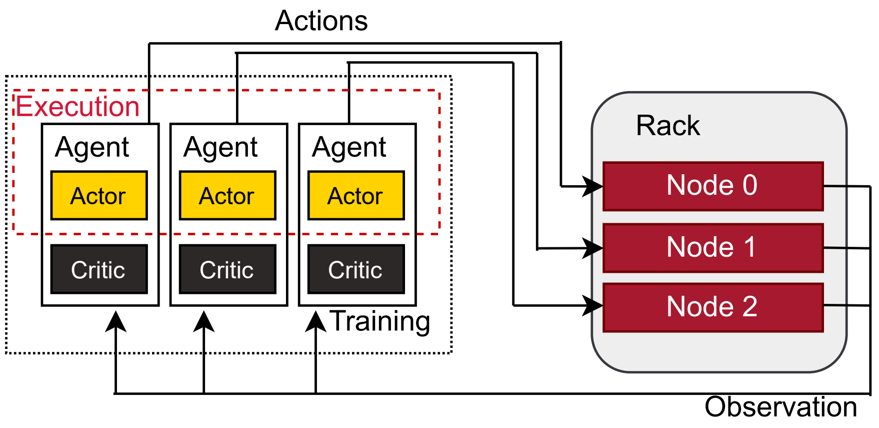


Figure 5.11: Overview of how OASys agents interact with the datacenter rack for both training and deployment.

and manageable way to control power consumption by limiting the number of nodes in high-power turbo mode.

Instead of requiring power monitoring from all nodes, this constraint requires monitoring turbo from all nodes. However, we have established that this information is not observable. To address this, we use actor-critic multi-agent reinforcement learning to train policies that indirectly learn to honor the global power constraint.

Actor-Critic based OASys Algorithm

To address the optimization problem with node-level latency and rack-level power constraints, we deploy OASys: an approach based on a multi-agent actor-critic algorithm (shown in Figure 5.11). OASys enables distributed decision-making among the POMDP agents while considering both latency and power budget constraints, and consists of two key components: the Actor and the Critic.

Actor

In the multi-agent environment, each agent is a decentralized Actor responsible for selecting actions for each node to maximize the expected cumulative reward. In our case, an Actor network outputs action probabilities for each agent based on their individual states. Specifically, the Actor's action selection process involves two main actions for each agent:

- Turbo Mode Action (T_i): The Actor decides whether to enable or disable turbo mode for each node, represented as a binary action ($T_i = 1$ for enable, $T_i = 0$ for disable).
- Batch Size Action (B_i): The Actor selects an appropriate batch size for the inference model running on each node from a predefined set of discrete options.

The Actor network is trained using policy gradients to maximize the expected cumulative reward. The expectation of the policy gradient of each agent i is $J(\theta_i) = E[R_i]$, which can be written as follows:

$$\nabla_{\theta_i} J(\theta_i) = E_{s \sim p^\pi, a \sim \pi_\theta}[\nabla_{\theta_i} \log \pi_i(a_i|s_i) Q^\pi(x, a_1, \ldots, a_N)] \tag{5.1}$$

Critic

The centralized Critic evaluates the quality of the actions selected by each Actor according to the state and action, and feeds back to the actor network, so that the actor can adjust its strategy according to the quality, and strive for better performance next time. The critic network can receive the information of all agents and make better evaluations for all agents. In addition to the traditional role of evaluating the actions' impact on maximizing throughput and meeting latency constraints, the Critic also considers the power budget of the rack. The Critic network takes as input the global states and actions, and estimates the expected cumulative reward. It provides feedback to each Actor by assigning a value (Q-value) to the selected actions, based on the state and actions of all actors. The Critic is trained to minimize the mean squared error between predicted Q-values and actual rewards.

Algorithm 4 OASys Algorithm

```
1: **Input**: Decentralized parameterized policy $\pi(a_i|s_i,\theta_i)$ for each agent $i$
2: **Input (training only)**: Centralized parameterized critic $Q(s,a_1,a_2,\ldots,a_N,\phi)$
3: **Input (training only)**: Step sizes $\alpha_{\theta_i} > 0$, $\alpha_\phi > 0$ for each agent $i$
4: if training then
5:     Initialize decentralized policy parameters $\theta_i \in \mathbb{R}^{d_{\theta_i}}$ for each agent $i$ (e.g., set to 0)
6:     Initialize centralized Q-function parameters $\phi \in \mathbb{R}^{d_\phi}$ (e.g., set to 0)
7: end if
8: while true do                                                    ▷ Agents run continuously
9:     if runtime then                                                ▷ Agent is deployed
10:        while !autoscaler do      ▷ only invoke the agents when the autoscaler changes
   workload or constraints
11:            wait
12:        end while
13:        for each agent $i$ do
14:            $a_i = \max \pi(a_i|s_i,\theta_i)$                   ▷ Select action with max reward
15:        end for
16:    else                                                                ▷ Training
17:        Initialize state $s = (s_1, s_2, \ldots, s_N)$               ▷ Joint state of all agents
18:        $I \leftarrow 1$
19:        while $s$ is not terminal do              ▷ Iterate until $s$ violates no constraints
20:            for each agent $i$ do
21:                $a_i \sim \pi(a_i|s_i,\theta_i)$                    ▷ Sample action $a_i$ for agent $i$
22:            end for
23:            $a = (a_1, a_2, \ldots, a_N)$                           ▷ Joint action of all agents
24:            Execute joint actions $a$, observe joint state $s'$
25:            $r_i$ is the reward received by agent $i$
26:            $Q(s,a,\phi) \leftarrow Q(s,a,\phi) + \alpha_\phi \left(\sum_{i=1}^{N} r_i + \gamma Q(s', \pi(s',\theta_1), \pi(s',\theta_2), \ldots, \pi(s',\theta_N), \phi) - Q(s,a,\phi)\right)$
   ▷ Update global critic based on all agents
27:            for each agent $i$ do
28:                $\theta_i \leftarrow \theta_i + \alpha_{\theta_i} I \nabla_{\theta_i} \ln \pi(a_i|s_i,\theta_i) \nabla_a Q(s,a,\phi)|_{a=a_i}$      ▷ Update each agent's
   policy (actor) based on critic
29:            end for
30:            $I \leftarrow \gamma I$
31:            $s \leftarrow s'$
32:        end while
33:    end if
34: end while
```

Algorithm

Algorithm 4 shows the OASys logic for both training and deploying the agents. Using profiling data collected on the target datacenter system for each target inference model (See Section 5.4.3), the policy is first generated offline. Each training episode contains consistent system state, i.e., fixed workload and latency constraints (lines 16-17). Episodes do not terminate until all constraints, power and latency, are met across all nodes (line 19). For each step within an episode, first each agent's actor selects an action according to its policy (lines 20-21). Those actions are carried out, and based on the subsequent system behavior and reward, the critic is updated (lines 23-26). Then each agent's policy is updated using the critic (lines 27-28). In summary, the critic uses the global system state and reward to tune each agent's individual actor. After sufficient training, the actor will provide expected reward for each action given current state. We use these actors at runtime to select the best action each time the autoscaler changes either the workload or latency constraints (lines 9-14).

By jointly optimizing actions for throughput, latency, and power, OASys aims to strike a balance between maximizing inference speedup, adhering to latency and power constraints, ultimately distributing power most effectively within the rack.

5.4.3 OASys as a Self-Optimizing Autoscaler

We demonstrate that OASys can (1) Dynamically adapt to varying latency constraints by configuring the batch size and only using turbo mode when needed. (2) Given a group of machines running ML workloads, improve throughput by 55% when compared to a traditional autoscaler.

We first generate a profile for 4 representative ML models: alexnet, densenet, vgg16, and

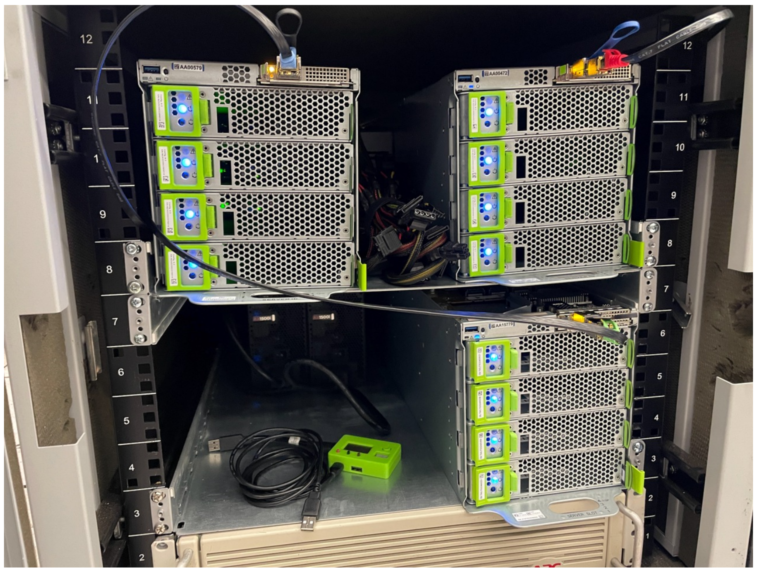


Figure 5.12: 12 nodes (3 sleds × 4 nodes) of Yosemite V3 Open Compute Project (OCP) used for experiments. These servers are representative of datacenter racks.

resnet-18 based on a real representative server cluster from Open Compute Project (OCP). The server cluster (shown in Fig. 5.12) is based on Intel Xeon Platinum 8321HC CPUs with 26 physical (52 logical) cores, and 96GB DDR4, running CentOS Stream 8. The base single core frequency is 1.4 GHz and the turbo boost frequency is 3 GHz. We run TorchServe to provide a mechanism for scalable model serving using http requests. Based on the collected traces we simulate the behavior of mixing different workloads using the OASys approach. Finally, we use the learned policy to evaluate the benefits of using OASys compared to traditional autoscaler policies. The inferences are performed on randomly selected images from ImageNet dataset [28].

Dynamic Latency Constraints

To show OASys can meet dynamic latency constraints while freeing compute resources for other nodes when constraints are relaxed, we do a case study of 4 different ML models (alexnet, densenet, vgg16 and resnet-18). Each model runs on a separate node and uses separate policies that were learned using OASys as described in Section 5.4.2. Figure 5.13

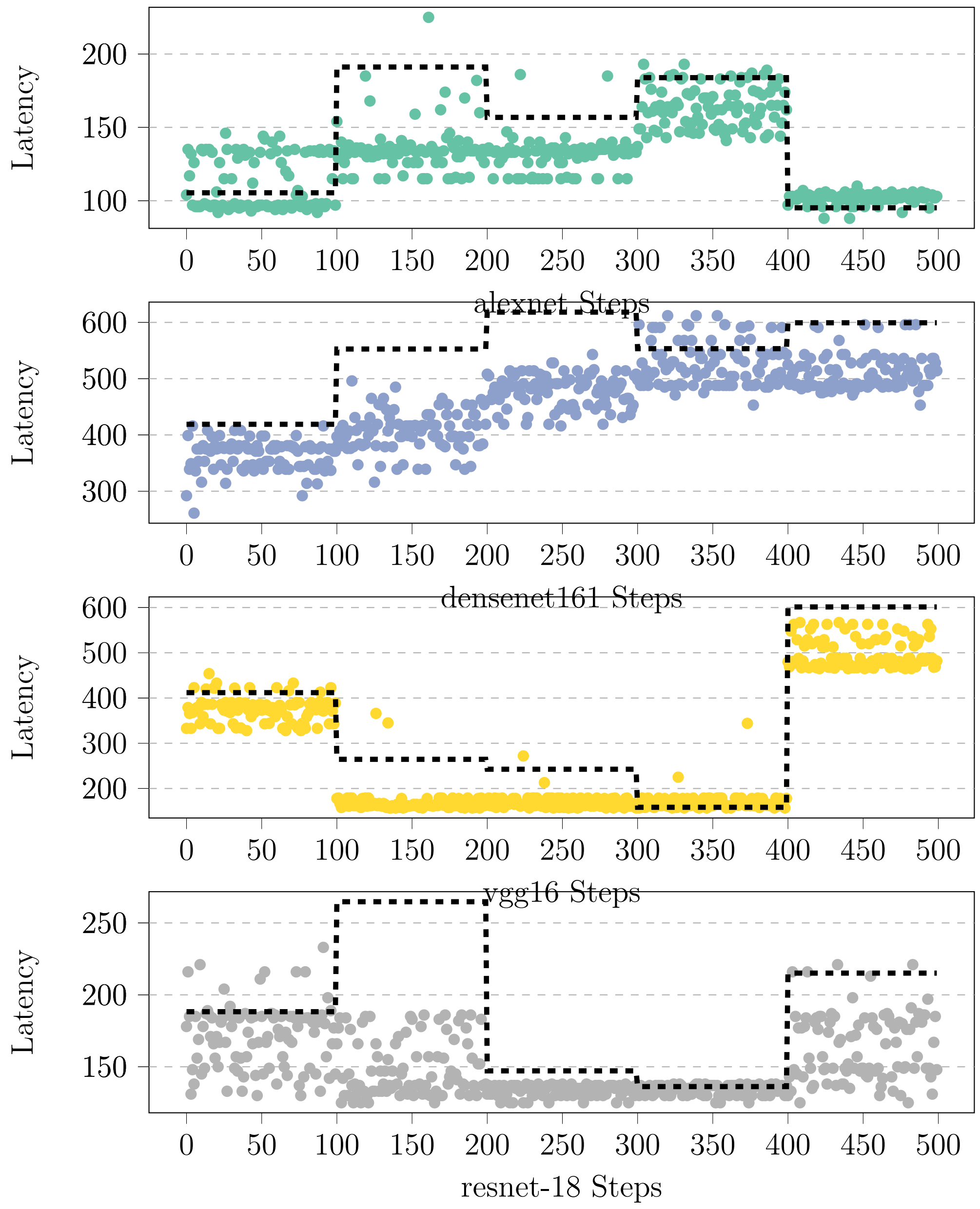


Figure 5.13: Four nodes running different ML inference models with OASys and can meet dynamic latency constraints.

shows the achieved per-inference latency and constraint of different ML models. The X axis is steps, where each step corresponds to polling the current QoS metrics (latency and throughput) from Torchserve. We currently poll the metrics every minute for the purpose of fine-grained observability, but only invoke the OASys policy whenever autoscaler changes the allocation of machines to determine the updated batch size and turbo settings that is currently set to every 100 steps. The Y axis is observed latency (in colored data points) and the latency constraint (in black dashed line) as determined the autoscaler. In practice the latency constraint can be determined by looking at the diurnal load patterns and relaxing the constraint during peak demands if the workloads are not latency-critical. Note that we randomly change the constraint every 100 steps to highlight the efficacy of OASys to adapt to any constraint. We make 2 major observations. ***First, OASys can dynamically adapt to different latency constraints set by the autoscaler.*** When latency constraint is relaxed, OASys approach automatically selects configurations that are near the constraint limit. This is done using larger batch size to increase throughput and reserving the turbo for other workloads that need it more. ***Second, we observe that there are few observations that exceed the budget.*** This happens because the performance of neural network inferencing inherently comes with variance. Moreover, we randomly select images from the ImageNet dataset and the performance of the ML workloads also varies based on the input. Thus, in some cases due to the variance, the observed latency exceeds, however we can account for the variance when designing the reward function. Increasing the negative reward corresponding to missed latencies (in Section 5.4.2) will learn policies that are more strict. We conclude that OASys can learn to meet dynamic latency constraints as determined by autoscaler by self-optimizing both system level (turbo) as well as application level (batch size) configurations.

In the next section, we will investigate into the additional throughput we get from OASys.

Throughput Improvements under Turbo Constraint

To show the benefits of using OASys to boost throughput (and hence create additional datacenter capacity), we examine mixes of workloads working together in the rack and measure the achieved throughput using OASys as compared to autoscalers with static configurations. We summarize the workloads along with their constraints in Table 5.5. The table has six different workload mixes, labeled as Mix 1 through Mix 6. Each workload mix consists of four different deep learning workloads, namely alexnet, densenet161, vgg16, and resnet-18. The parentheses indicate the latency (execution time) for each workload in milliseconds (ms). We evaluate OASys by running each mix 10 times under 3 policies: (1) Baseline: Autoscaler that runs all hosts with Turbo off. (2) Baseline Turbo: Autoscaler that runs all hosts with Turbo on. Note that we can never deploy this in practice because it will violate the power constraints and cause infrastructure failure. (3) OASys: Self-Optimizing that can holistically configure both application level (batch-size) as well as system level (turbo) knobs. Figure 5.14 shows the observed throughput when running different mixes with the policies mentioned above. The X axis is the workload mix (from Table 5.5). The Y axis is the average throughput (measured in inferences per seconds) as reported by Torchserve. We make the following observations. First, average throughput for Baseline policy is 412 inferences/second, average throughput for Baseline Turbo policy is 515 and average throughput for OASys is 797. ***Thus, compared to Baseline Turbo, OASys can provide on average OASys* 55% *more throughput.*** This creates additional capacity for running more workloads. Second, ***OASys can always provide more throughput compared to Baseline.*** The highest throughput is 1257 corresponding to Mix 1. This exploits the fact that Baseline autoscalers cannot dynamically change the batch sizes, but OASys can. Third, there are some cases (e.g., Mix 4) where Baseline Turbo (enabled turbo for all nodes) can outperform OASys. This happens due to the fact that Baseline Turbo does not consider turbo limits so can switches on Turbo for all the workloads at once: this can cause tripping

Table 5.5: Workload mixes used to evaluate OASys

Mix	Workload (Latency)
0	alexnet (213ms), densenet161 (523ms), vgg16 (488ms), resnet-18 (219ms)
1	alexnet (193ms), densenet161 (422ms), vgg16 (181ms), resnet-18 (142ms)
2	alexnet (95ms), densenet161 (210ms), vgg16 (489ms), resnet-18 (259ms)
3	alexnet (197ms), densenet161 (255ms), vgg16 (437ms), resnet-18 (165ms)
4	alexnet (152ms), densenet161 (511ms), vgg16 (309ms), resnet-18 (267ms)
5	alexnet (186ms), densenet161 (354ms), vgg16 (242ms), resnet-18 (188ms)

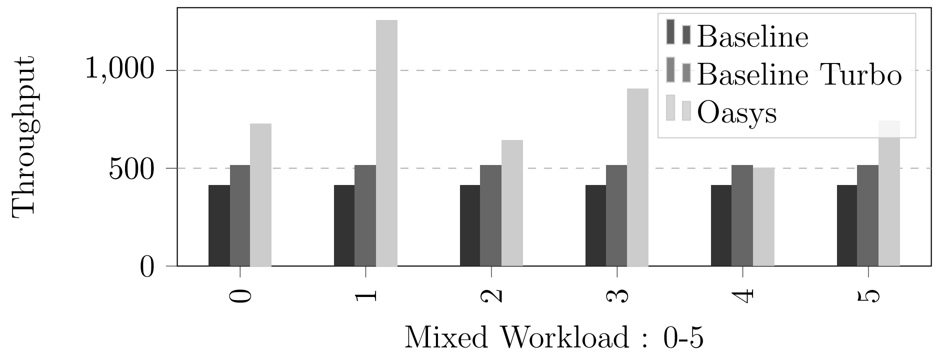


Figure 5.14: Throughput improvements compared to traditional autoscaler

of circuit breakers. However, ***OASys in all the scenarios learns to limit turbo based on the power constraint.*** We restrict the turbo to only 70% nodes during the training process in Section 5.4.2 for safe operation. Thus, we conclude that OASys can increase throughput while enforcing rack-level power limits, enabling operators to efficiently utilize datacenter resources.

5.5 Discussion

Throughout this chapter, we have emphasized the pivotal role of the Application Abstraction Layer in the context of self-aware memory management. It serves as the critical interface between developers and memory management systems, allowing for the fine-tuning of memory resources based on application-specific constraints. Whether it's the demanding workloads of self-driving vehicles or the relentless growth of machine learning applications, data-centricity is the driving force behind modern computing. As such, effective memory management is not a one-size-fits-all endeavor but requires an intimate understanding of each application's unique requirements.

Our exploration has revealed that memory demands within data-centric applications are dynamic and diverse. Tasks within these applications may exhibit varying memory footprints and execution patterns, making static memory configurations obsolete. Computational self-awareness offers a dynamic solution to this challenge, enabling developers to extract the maximum potential from emerging architectures. The chapter delved into two distinct case studies, Chauffeur and OASys, to illustrate the practical application of these principles.

Case Study 1: Chauffeur - The First Open-Source End-to-End Benchmark Suite for Self-Driving Vehicles The generic end-to-end self-driving software pipeline, as illustrated in Figure 5.1, encompasses a series of tasks, each crucial to the vehicle's operation. From real-time data sensing to perception, planning, and actuation, these tasks demand efficient memory management to guarantee seamless and safe operation. Our findings underscore the need for adaptive memory management within the Chauffeur benchmark suite, as the diverse tasks involved exhibit varying memory requirements and execution patterns. Through the integration of computational self-awareness principles, we propose dynamic memory configurations as a way to optimize performance while ensuring energy efficiency for enhancing self-driving vehicle technology.

Case Study 2: OASys - Efficient Resource Management for MLaaS Inference Serving Systems Within the realm of datacenter operations, we investigated the pressing challenge of sustaining compute capacity for machine learning (ML) workloads. Datacenter operators are confronted with the daunting task of ensuring efficient resource allocation to meet the burgeoning demands of ML applications while adhering to strict power constraints. Our novel approach, OASys, leverages off-policy model-free reinforcement learning to dynamically adapt system configurations based on workload characteristics and power budgets. This innovative solution addresses the shortcomings of traditional autoscaling approaches by optimizing both performance and resource allocation.

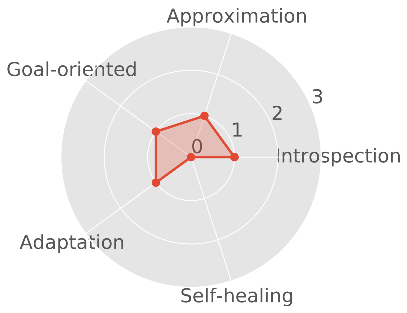
The learnings from Chauffeur and OASys demonstrates the potential of self-aware memory management revolutionize resource management in modern MLaaS inference serving systems, offering a promising avenue for enhancing efficiency and reducing energy costs. In summary, through our case studies, we've showcased the practical application of computational self-awareness in addressing the complex memory management challenges inherent in emerging data-centric applications. As we look ahead, the integration of self-aware memory management within the Application Abstraction Layer holds the promise of driving performance improvements and energy savings across diverse computing domains, ultimately shaping the future of memory management in a data-centric world.

Chapter 6

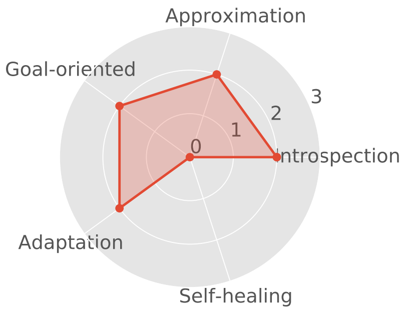
Conclusions

This thesis presents the exploration of computational self-awareness (CSA) principles within memory management to overcome the limitations of traditional approaches and optimize system performance, reliability, power efficiency, and application constraints.

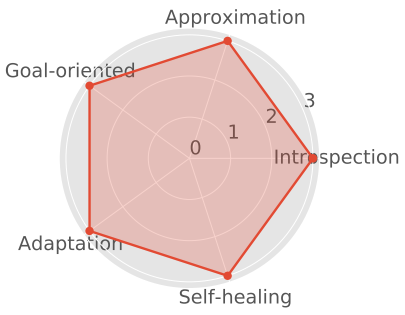
Memory management is a critical aspect of computing systems, and the rapid evolution of computing architectures and the demands of data-intensive applications have posed significant challenges in achieving efficient memory performance and energy efficiency. Traditional approaches based on static configurations and workload-specific optimizations are no longer



(a) Traditional



(b) My Thesis



(c) Comprehensive self-awareness

Figure 6.1: Self-aware Memory Subsystem: (a) past, (b) present and (c) future.

sufficient to address the dynamic nature of modern computing systems. Figure 6.1 shows the gradual progress from traditional to state-of-art, as well as future directions in self-aware memory systems as decsribed in Chapter 2. By incorporating self-awareness into memory management, I have developed systems that can adapt to changing workloads, optimize resource utilization, and align their operations with user or application goals.

Through the exploration of computational self-awareness in memory management, I made several important contributions. First, I identified the self-* properties relevant to memory management and demonstrated how they can be applied to address the challenges of efficient memory performance and energy efficiency. Second, I integrated self-awareness principles at different layers of the memory subsystem, enabling comprehensive optimization and coordination of memory management operations. This comprehensive approach facilitates the optimization and coordination of memory management operations across different memory layers, including on-chip caches, off-chip main memory, and device variations. By extending self-awareness to different layers, the system can dynamically allocate resources, balance trade-offs between different memory types, and ensure optimal performance even in the face of hardware variations. Third, I investigated the concept of memory approximation and its challenges, emphasizing the need for dynamic adaptation and error mitigation techniques. Finally, I demonstrated the effectiveness and versatility of self-aware memory management in different scenarios, as showcased by the end-to-end driving (e.g., in Chauffeur) and data-centers (e.g., in OASys) case studies.

By leveraging self-awareness principles and techniques, the thesis demonstrates the potential for significant improvements in memory performance, energy efficiency, and reliability. Our research has shown that self-aware memory management systems can dynamically adjust memory configurations, allocation policies, and access patterns to optimize resource utilization and meet application goals. Through extensive experimental evaluation, we have validated the effectiveness of our self-aware memory management techniques and demonstrated their benefits in terms of performance improvement and energy savings. Further exploration can focus on the development of intelligent self-aware memory management strategies that leverage advanced machine learning algorithms, control theory, and scalable techniques. Additionally, the integration of self-awareness techniques with emerging deep learning accelerators, such as Google TPU, holds promise for further enhancing memory performance and energy efficiency. The insights gained from this thesis contribute to the advancement of self-aware memory management and pave the way for future research and development in this field.

Bibliography

[1] M. Alcon, H. Tabani, L. Kosmidis, E. Mezzetti, J. Abella, and F. J. Cazorla. Timing of autonomous driving software: Problem analysis and prospects for future solutions. In *Proc. IEEE RTAS*, pages 267–280, 2020.

[2] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith. Gpu scheduling on the nvidia tx2: Hidden details revealed. In *Proc. IEEE RTSS*, pages 104–115, 2017.

[3] A. Ansari, S. Feng, S. Gupta, and S. Mahlke. Enabling ultra low voltage system operation by tolerating on-chip cache failures. In *Proceedings of the ACM/IEEE international symposium on Low power electronics and design*, pages 307–310, New York, NY, USA, 2009. Association for Computing Machinery.

[4] A. Ansari, S. Feng, S. Gupta, and S. A. Mahlke. Enabling ultra low voltage system operation by tolerating on-chip cache failures. In *Proceedings of the 2009 International Symposium on Low Power Electronics and Design, 2009, San Fancisco, CA, USA, August 19-21, 2009*, pages 307–310, 2009.

[5] Apollo. An open autonomous driving platform, source-code and manuals. `https://github.com/ApolloAuto/apollo`, 2019.

[6] S. Aradi. Survey of deep reinforcement learning for motion planning of autonomous vehicles. *IEEE TITS*, pages 1–20, 2020.

[7] F. Arnaud, A. Thean, M. Eller, M. Lipinski, Y. Teh, M. Ostermayr, K. Kang, N. Kim, K. Ohuchi, J. Han, et al. Competitive and cost effective high-k based 28nm cmos technology for low power applications. In *IEEE International Electron Devices Meeting (IEDM)*, pages 1–4. IEEE, 2009.

[8] A. A. Assidiq, O. O. Khalifa, M. R. Islam, and S. Khan. Real time lane detection for autonomous vehicles. In *Proc. CCCE*, pages 82–88, 2008.

[9] Aurangzeb and Eigenmann. Harnessing Parallelism in Multicore Systems to Expedite and Improve Function Approximation. In *LCPC*, 2016.

[10] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of Programming Language Design and Implementation*, pages 198–209, New York, NY, USA, 2010. Association for Computing Machinery.

[11] J. Balkind, K. Lim, F. Gao, J. Tu, D. Wentzlaff, M. Schaffner, F. Zaruba, and L. Benini. Openpiton+ ariane: The first open-source, smp linux-booting risc-v system scaling from one to many cores. In *Third Workshop on Computer Architecture Research with RISC-V, CARRV*, volume 19. CARRV, 2019.

[12] S. Bateni, Z. Wang, Y. Zhu, Y. Hu, and C. Liu. Co-optimizing performance and memory footprint via integrated cpu/gpu memory management, an implementation on autonomous driving platform. In *Proc. IEEE RTAS*, pages 310–323, 2020.

[13] V. Bhalodia. SCALE DRAM subsystem power analysis. Master's thesis, Massachusetts Institute of Technology, 2005.

[14] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite. In *Proc. of PACT*. ACM Press, 2008.

[15] M. Bjelonic. YOLO ROS: Real-time object detection for ROS. `https://github.com/leggedrobotics/darknet_ros`, 2016–2018.

[16] H. Caesar, V. Bankiti, A. H. Lang, S. Vora, V. E. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan, and O. Beijbom. nuscenes: A multimodal dataset for autonomous driving. In *Proc. IEEE/CVF CVPR*, pages 11621–11631, 2020.

[17] C. Campos, R. Elvira, J. J. G. Rodriguez, J. M. M. Montiel, and J. D. Tardos. Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam. *IEEE T-RO*, page 1–17, 2021.

[18] J. Canny. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, PAMI-8:679–698, 1986.

[19] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11:1–25, 2014.

[20] A. Carroll, G. Heiser, et al. An analysis of power consumption in a smartphone. In *Proceedings of Annual Technical Conference*, volume 14, page 21, USA, 2010. USENIX Association.

[21] M. Chakraborty and A. P. Kundan. *Grafana: Monitoring Cloud-Native Applications*, pages 187–240. Apress, Berkeley, CA, 2021.

[22] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. IEEE IISWC*, pages 44–54, 2009.

[23] Z. Chen and X. Huang. End-to-end learning for lane keeping of self-driving cars. In *Proc. IEEE IV*, pages 1856–1860, 2017.

[24] K. Cho, Y. Lee, Y. H. Oh, G.-c. Hwang, and J. W. Lee. edram-based tiered-reliability memory with applications to low-power frame buffers. In *Proceedings of International Symposium on Low Power Electronics and Design*, pages 333–338, New York, NY, USA, 2014. Association for Computing Machinery.

[25] M. Cho, J. Schlessman, W. Wolf, and S. Mukhopadhyay. Accuracy-aware sram: a reconfigurable low power sram architecture for mobile multimedia applications. In *Asia and South Pacific Design Automation Conference*, 2009.

[26] B. Coutinho. Dynolog: Open source system observability. Facebook Developers Blog, 2022. Accessed on 2023-12-02.

[27] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th International Conference on Autonomic Computing, ICAC 2011, Karlsruhe, Germany, June 14-18, 2011*, pages 31–40, 2011.

[28] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE CVPR*, pages 248–255, 2009.

[29] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. Memscale: Active low-power modes for main memory. *SIGARCH Comput. Archit. News*, 39(1):225–238, Mar. 2011.

[30] B. Donyanavard, N. Dutt, B. Maity, P. Malani, and T. Mück. MARS: A framework for runtime monitoring, modeling, and management of realtime systems. In *2023 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2023.

[31] B. Donyanavard, T. Mück, A. M. Rahmani, N. Dutt, A. Sadighi, F. Maurer, and A. Herkersdorf. Sosa: Self-optimizing learning with self-adaptive control for hierarchical system-on-chip management. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 685–698, New York, NY, USA, 2019. Association for Computing Machinery.

[32] B. Donyanavard, T. Mück, S. Sarma, and N. Dutt. Sparta: Runtime task allocation for energy efficient heterogeneous many-cores. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES '16, pages 27:1–27:10, New York, NY, USA, 2016. ACM.

[33] R. Eigenmann et al. Harnessing parallelism in multicore systems to expedite and improve function approximation. In *Languages and Compilers for Parallel Computing*, pages 88–92, Cham, 2017. Springer International Publishing.

[34] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *Proceedings of the 17th international conference on Architectural Support for Programming Languages and Operating Systems*, pages 301–312, New York, NY, USA, 2012. Association for Computing Machinery.

[35] J. Fickenscher, F. Hannig, and J. Teich. Dsl-based acceleration of automotive environment perception and mapping algorithms for embedded cpus, gpus, and fpgas. In *Proc. of ARCS*, pages 71–86, 2019.

[36] J. Fickenscher, J. Schlumberger, F. Hannig, J. Teich, and M. E. Bouzouraa. Cell-based update algorithm for occupancy grid maps and hybrid map for adas on embedded gpus. In *Proc. of DATE*, pages 443–448, 2018.

[37] J. H. Friedman. Multivariate Adaptive Regression Splines. *The Annals of Statistics*, 19(1):1 – 67, 1991.

[38] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun. Vision meets robotics: The kitti dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013.

[39] T. Goldbrunner, T. Wild, and A. Herkersdorf. Memory access pattern profiling for streaming applications based on matlab models. In *28th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 32–38. IEEE, 2018.

[40] B. Grigorian, N. Farahpour, and G. Reinman. Brainiac: Bringing reliable accuracy into neurally-implemented approximate computing. In *IEEE 21st International Symposium on High Performance Computer Architecture*, pages 615–626. IEEE, 2015.

[41] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proc. IEEE IISWC*, pages 3–14, 2001.

[42] W. Han and Y. Zhang. Fast loam (lidar odometry and mapping). `https://github.com/wh200720041/floam`, 2019.

[43] P. Hank, S. Müller, O. Vermesan, and J. Van Den Keybus. Automotive ethernet: in-vehicle networking and smart mobility. In *Proc. DATE*, page 1735–1739. EDA Consortium, 2013.

[44] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision.* Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[45] M. Hausknecht and P. Stone. Deep recurrent q-learning for partially observable mdps, 2017.

[46] D. Held, S. Thrun, and S. Savarese. Learning to track at 100 fps with deep regression networks. In *Proc. ECCV*, pages 749–765, 2016.

[47] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems.* John Wiley & Sons, 2004.

[48] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[49] C.-H. Hsu, Q. Deng, J. Mars, and L. Tang. Smoothoperator: Reducing power fragmentation and improving power utilization in large-scale datacenters. *SIGPLAN Not.*, 53(2):535–548, mar 2018.

[50] W. Jang, H. Jeong, K. Kang, N. Dutt, and J. C. Kim. R-tod: Real-time object detector with minimized end-to-end delay for autonomous driving. In *Proc. IEEE RTSS*, pages 191–204, 2020.

[51] J. Jiao. Heap: A holistic error assessment framework for multiple approximations using probabilistic graphical models. *Electronics*, 9:373, 2020.

[52] Jonaspfab and Danielebp. Cuda implementation of a hough transform based lane detection algorithm. `https://github.com/jonaspfab/cuda-lane-detection`, 2019.

[53] M. Jung, E. Zulian, D. M. Mathew, M. Herrmann, C. Brugger, C. Weis, and N. Wehn. Omitting refresh: A case study for commodity and wide i/o drams. In *Proceedings of the 2015 International Symposium on Memory Systems*, page 85–91, New York, NY, USA, 2015. Association for Computing Machinery.

[54] 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*, ISCA '15, pages 158–169, New York, NY, USA, 2015. ACM.

[55] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada. An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68, 2015.

[56] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *Proc. ACM/IEEE ICCPS*, pages 287–296, 2018.

[57] J. Kim, A. Rohrbach, T. Darrell, J. Canny, and Z. Akata. Textual explanations for self-driving vehicles. In *Proc. ECCV*, pages 563–578, 2018.

[58] Y. Kim, Y. Choi, and M. Rhu. Paris and elsa: An elastic scheduling algorithm for reconfigurable multi-gpu inference servers. In *Proceedings of DAC*, New York, NY, USA, 2022. Association for Computing Machinery.

[59] S. Koppula, L. Orosa, A. G. Yağlıkçı, R. Azizi, T. Shahroodi, K. Kanellopoulos, and O. Mutlu. Eden: Enabling energy-efficient, high-performance deep neural network inference using approximate dram. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 166–181, New York, NY, USA, 2019. Association for Computing Machinery.

[60] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6), may 2017.

[61] K. Kurzer. Path planning in unstructured environments : A real-time hybrid a* implementation for fast and deterministic path generation for the kth research concept vehicle. Master's thesis, KTH, Integrated Transport Research Lab, ITRL, 2016.

[62] S. Kuutti, S. Fallah, K. Katsaros, M. Dianati, F. Mccullough, and A. Mouzakitis. A survey of the state-of-the-art localization techniques and their potentials for autonomous vehicle applications. *IEEE TOTJ*, 5(2):829–846, 2018.

[63] M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang. Input Responsiveness: Using Canary Inputs to Dynamically Steer Approximation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016.

[64] L. Lennart. Black-box Models from Input-output Measurements. In *Proceedings of the IEEE Instrumentation and Measurement Technology Conference*, 2001.

[65] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, New York, NY, USA, 2009. Association for Computing Machinery.

[66] W. Liang, S. Chen, Y. Chang, and J. Fang. Memory-aware dynamic voltage and frequency prediction for portable devices. In *The Fourteenth IEEE Internationl Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2008, Kaohisung, Taiwan, 25-27 August 2008, Proceedings*, pages 229–236, 2008.

[67] S. C. Lin, Y. Zhang, C. H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars. The architectural implications of autonomous driving: Constraints and acceleration. *ACM SIGPLAN Notices*, 53(2):751–766, 2018.

[68] M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the Eleventh International Conference on International Conference on Machine Learning*, ICML'94, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[69] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: Saving DRAM Refresh-power Through Critical Data Partitioning. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[70] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: saving dram refresh-power through critical data partitioning. In *Proceedings of the 16th international conference on Architectural support for programming languages and operating systems*, pages 213–224, New York, NY, USA, 2011. Association for Computing Machinery.

[71] L. Ljung. *System Identification: Theory for the User.* Prentice-Hall, Inc., 1999.

[72] D. Lo and C. Kozyrakis. Dynamic management of turbomode in modern multi-core chips. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 603–613, 2014.

[73] Y. Luo. Lanenet-lane-detection. `https://github.com/MaybeShewill-CV/lanenet-lane-detection`, 2020.

[74] B. Maity, B. Donyanavard, and N. Dutt. Self-aware Memory Management for Emerging Energy-efficient Architectures. In *2020 11th International Green and Sustainable Computing Workshops (IGSC)*, pages 1–8, 2020.

[75] B. Maity, B. Donyanavard, A. Surhonne, A. Rahmani, A. Herkersdorf, and N. Dutt. Seams: Self-optimizing runtime manager for approximate memory hierarchies. *ACM Trans. Embed. Comput. Syst.*, 20(5), jul 2021.

[76] B. Maity, B. Donyanavard, N. Venkatasubramanian, and N. Dutt. Workload characterization for memory management in emerging embedded platforms. In *Analysis, Estimations, and Applications of Embedded Systems*, pages 65–76, Cham, 2023. Springer Nature Switzerland.

[77] B. Maity, M. Shoushtari, A. M. Rahmani, and N. Dutt. Self-adaptive memory approximation: A formal control theory approach. *IEEE Embedded Systems Letters*, 12:33–36, 2019.

[78] B. Maity, M. Shoushtari, A. M. Rahmani, and N. Dutt. Simulation Infrastructure and System Dynamics of Quality Configurable Memory. *CECS Tech. Rep. 19-03*, 2019.

[79] B. Maity, M. Shoushtari, A. M. Rahmani, and N. Dutt. Self-adaptive memory approximation: A formal control theory approach. *IEEE Embedded Systems Letters*, 12(2):33–36, 2020.

[80] B. Maity, S. Yi, D. Seo, L. Cheng, S.-S. Lim, J.-C. Kim, B. Donyanavard, and N. Dutt. Chauffeur: Benchmark suite for design and end-to-end analysis of self-driving vehicles on embedded systems. *ACM Trans. Embed. Comput. Syst.*, 20(5s), 2021.

[81] M. Masadeh, O. Hasan, and S. Tahar. Using machine learning for quality configurable approximate computing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1575–1578. IEEE, 2019.

[82] M. Masadeh, O. Hasan, and S. Tahar. Machine Learning-Based Self-Compensating Approximate Computing. *arXiv e-prints*, page arXiv:2001.03783, 2020.

[83] A. Merkel and F. Bellosa. Memory-Aware Scheduling for Energy Efficiency on Multicore Processors. In *Proc. HotPower*, 2008.

[84] J. S. Miguel, M. Badr, and N. E. Jerger. Load value approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 127–139, USA, 2014. IEEE Computer Society.

[85] K. Moazzemi, B. Maity, S. Yi, A. M. Rahmani, and N. Dutt. Hessle-free: Heterogeneous systems leveraging fuzzy control for runtime resource management. *ACM Trans. Embed. Comput. Syst.*, 18(5s), 2019.

[86] A.-M. Monazzah, M. Shoushtari, A. Rahmani, and N. Dutt. QuARK: Quality-configurable Approximate STT-MRAM Cache by Fine-grained Tuning of Reliability-Energy Knobs. In *IEEE/ACM International Symposium on Low Power Electronics and Design*, 2017.

[87] A. M. H. Monazzah, M. Shoushtari, S. G. Miremadi, A. M. Rahmani, and N. Dutt. Quark: Quality-configurable approximate stt-mram cache by fine-grained tuning of reliability-energy knobs. In *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6. IEEE, 2017.

[88] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin. SNNAP: Approximate computing on programmable SoCs via neural acceleration. In *IEEE 21st International Symposium on High Performance Computer Architecture*, pages 603–614. IEEE, 2015.

[89] P. Moulon, P. Monasse, R. Perrot, and R. Marlet. Openmvg: Open multiple view geometry. In *Proc. Springer RRPR*, pages 60–74, 2016.

[90] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, pages 129–, Washington, DC, USA, 2003. IEEE Computer Society.

[91] T. Mück, B. Donyanavard, B. Maity, K. Moazzemi, and N. Dutt. Mars: Middleware for adaptive reflective computer systems, 2021.

[92] NVIDIA. *CUPTI :: CUDA Toolkit Documentation*, 2014. https://docs.nvidia.com/cuda/cupti/index.html.

[93] NVIDIA. Hello ai world nvidia jetson. `https://github.com/dusty-nv/jetson-inference`, 2021.

[94] Nvidia Jetson TX2 Architecture. Available at `https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/`.

[95] F. Oboril, A. Shirvanian, and M. B. Tahoori. Fault tolerant approximate computing using emerging non-volatile spintronic memories. In *34th IEEE VLSI Test Symposium, VTS 2016, Las Vegas, NV, USA, April 25-27, 2016*, page 1, 2016.

[96] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. Berg, and S. Wang. An evaluation of the nvidia tx1 for supporting real-time computer-vision workloads. In *Proc. IEEE RTAS*, pages 353–364, 2017.

[97] P. Palanisamy. Multiple-Object-Tracking-from-Point-Clouds. `https://doi.org/10.5281/zenodo.3559186`, 2019.

[98] S. D. Pendleton, H. Andersen, X. Du, X. Shen, M. Meghjani, Y. H. Eng, D. Rus, and M. H. Ang. Perception, planning, control, and coordination for autonomous vehicles. *Machines*, 5(1), 2017.

[99] L. Piga, I. Narayanan, A. Sundarrajan, M. Skach, Q. Deng, B. Maity, M. Chakkaravarthy, A. Huang, A. Dhanotia, and P. Malani. Expanding Datacenter Capacity with DVFS Boosting: A Safe and Scalable Deployment Experience. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24)*, 2024.

[100] Prometheus. Monitoring system & time series database. `https://prometheus.io/`. Last accessed 2023-01-02.

[101] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *42nd Annual IEEE/ACM international symposium on microarchitecture (MICRO)*, pages 14–23, New York, NY, USA, 2009. Association for Computing Machinery.

[102] A. Raha, S. Sutar, H. Jayakumar, and V. Raghunathan. Quality configurable approximate dram. *IEEE Transactions on Computers*, 66(7):1172–1187, 2017.

[103] E. A. Rambo, B. Donyanavard, M. Seo, F. Maurer, T. Kadeed, C. B. de Melo, B. Maity, A. Surhonne, A. Herkersdorf, F. Kurdahi, N. Dutt, and R. Ernst. The self-aware information processing factory paradigm for mixed-critical multiprocessing. *IEEE Transactions on Emerging Topics in Computing*, 10(1):250–266, 2022.

[104] E. A. Rambo, T. Kadeed, R. Ernst, M. Seo, F. Kurdahi, B. Donyanavard, C. B. de Melo, B. Maity, K. Moazzemi, K. Stewart, S. Yi, A. M. Rahmani, N. Dutt, F. Maurer, N. A. V. Doan, A. Surhonne, T. Wild, and A. Herkersdorf. The information processing factory: A paradigm for life cycle management of dependable systems. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis Companion*, CODES/ISSS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[105] A. Ranjan, S. Venkataramani, X. Fong, K. Roy, and A. Raghunathan. Approximate storage for energy efficient spintronic memories. In *Proceedings of DAC*. ACM, 2015.

[106] M. Rapp, M. B. Sikal, H. Khdr, and J. Henkel. Smartboost: Lightweight ml-driven boosting for thermally-constrained many-core processors. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 265–270, 2021.

[107] B. K. Reddy, E. W. Wächter, B. M. Al-Hashimi, and G. V. Merrett. Workload-aware runtime energy management for HPC systems. In *2018 International Conference on High Performance Computing & Simulation, HPCS 2018, Orleans, France, July 16-20, 2018*, pages 292–299, 2018.

[108] M. I. Ribeiro. Kalman and extended kalman filters: Concept, derivation and properties. *Institute for Systems and Robotics*, 43:46, 2004.

[109] M. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman. Monitoring and debugging the quality of results in approximate programs. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 399–411, New York, NY, USA, 2015. Association for Computing Machinery.

[110] F. Sampaio, M. Shafique, B. Zatt, S. Bampi, and J. Henkel. Approximation-aware Multi-Level Cells STT-RAM cache architecture. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 79–88. IEEE, 2015.

[111] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate storage in solid-state memories. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, page 25–36, New York, NY, USA, 2013. Association for Computing Machinery.

[112] R. R. Schaller. Moore's law: past, present and future. *IEEE Spectrum*, 34(6):52–59, June 1997.

[113] W. Schwarting, J. Alonso-Mora, and D. Rus. Planning and decision-making for autonomous vehicles. *Annual Review of Control, Robotics, and Autonomous Systems*, 1(1):187–210, 2018.

[114] J. Shannon. Extended kalman filter. `https://github.com/jeremy-shannon/CarND-Extended-Kalman-Filter-Project`, 2019.

[115] M. Shoushtari, A. BanaiyanMofrad, and N. Dutt. Exploiting Partially-Forgetful Memories for Approximate Computing. *IEEE Embedded Systems Letters*, 2015.

[116] M. Shoushtari, A. BanaiyanMofrad, and N. Dutt. Exploiting partially-forgetful memories for approximate computing. *IEEE Embedded Systems Letters*, 7:19–22, 2015.

[117] C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan. Relaxing non-volatility for fast and energy-efficient stt-ram caches. In *IEEE 17th International Symposium on High Performance Computer Architecture*, pages 50–61. IEEE, 2011.

[118] X. Song, P. Wang, D. Zhou, R. Zhu, C. Guan, Y. Dai, H. Su, H. Li, and R. Yang. Apollocar3d: A large 3d car instance understanding benchmark for autonomous driving. In *Proc. IEEE/CVF CVPR*, pages 5452–5462, 2019.

[119] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. Green governors: A framework for continuously adaptive dvfs. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, IGCC '11, pages 1–8, Washington, DC, USA, 2011. IEEE Computer Society.

[120] A. Sriraman and A. Dhanotia. Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale. In *Proc. ASPLOS*, 2020.

[121] R. St. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger. General-purpose code acceleration with limited-precision analog computation. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, pages 505–516. IEEE, 2014.

[122] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[123] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning.* MIT press, Cambridge, MA, USA, 2018.

[124] A. Taha and N. AbuAli. Route planning considerations for autonomous vehicles. *IEEE ComMag*, 56(10):78–84, 2018.

[125] M. T. Teimoori, M. A. Hanif, A. Ejlali, and M. Shafique. AdAM: Adaptive approximation management for the non-volatile memory hierarchies. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 785–790. IEEE, 2018.

[126] A. A. *et al.* Self-aware computing. *MIT Tech. Rep.*, 2009.

[127] A. M. *et al.* General purpose computing on low-power embedded GPUs: Has it come of age? In *Proc. SAMOS*, 2013.

[128] B. D. *et al.* Intelligent Management of Mobile Systems through Computational Self-Awareness. *arXiv:2008.00095 [cs.AR]*, 2020.

[129] B. D. *et al.* *Reflecting on Self-Aware Systems-on-Chip*, pages 79–95. Springer International Publishing, Cham, 2021.

[130] C. B. *et al.* The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proc. of PACT*, 2008.

[131] C. W. *et al.* DLAU: A Scalable Deep Learning Accelerator Unit on FPGA. *IEEE TCAD*, 2017.

[132] E. I. *et al.* Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *Proc. ISCA*, 2008.

[133] F. F. *et al.* Energy-Efficient Virtual Machines Consolidation in Cloud Data Centers Using Reinforcement Learning. In *Proc. PDP*, 2014.

[134] F. P. *et al.* Memory-Aware Green Scheduling on Multi-core Processors. In *Proc. ICPPW*, 2010.

[135] F. Z. *et al.* On Self-Adaptation, Self-Expression, and Self-Awareness in Autonomic Service Component Ensembles. In *Proc. SASO*, 2011.

[136] K. B. *et al.* Self-Aware Cyber-Physical Systems. *ACM TCPS*, 2020.

[137] M. E. T. *et al.* Memory MISER: Improving Main Memory Energy Efficiency in Servers. *IEEE TC*, 2009.

[138] R. G. K. *et al.* Machine Learning for Design Space Exploration and Optimization of Manycore Systems. In *Proc. ICCAD*, 2018.

[139] S. H. *et al.* A flexible low-power machine learning accelerator for healthcare applications. In *Proc. ICSICT*, 2016.

[140] S. K. *et al. Self-Aware Computing Systems.* Springer, 2017.

[141] U. G. *et al.* Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In *Proceedings of ISCA*, ISCA '20, page 982–995. IEEE Press, 2020.

[142] Y. *et al.* A multi-agent hybrid cognitive architecture with self-awareness for homecare robot. In *Proc. ICCSE*, 2014.

[143] The Verge. Tesla fsd chip. `https://www.theverge.com/2019/4/22/18511594/tesla-new-self-driving-chip-is-here-and-this-is-your-best-look-yet`, 2019.

[144] R. Venkatagiri, K. Ahmed, A. Mahmoud, S. Misailovic, D. Marinov, C. W. Fletcher, and S. V. Adve. gem5-approxilyzer: An open-source tool for application-level soft error analysis. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 214–221. IEEE, 2019.

[145] J. Wang and B. H. Calhoun. Minimum supply voltage and yield estimation for large srams under parametric variations. *IEEE Trans. VLSI Syst. 2011*, 19(11), 2011.

[146] Y. Wang, W. Chao, D. Garg, B. Hariharan, M. Campbell, and K. Q. Weinberger. Pseudo-lidar from visual depth estimation: Bridging the gap in 3d object detection for autonomous driving. In *Proc. IEEE/CVF CVPR*, pages 8445–8453, 2019.

[147] Y. Wang, S. Liu, X. Wu, and W. Shi. Cavbench: A benchmark suite for connected and autonomous vehicles. In *Proc. IEEE/ACM SEC*, pages 30–42, 2018.

[148] Z. Wang, W. Ren, and Q. Qiu. Lanenet: Real-time lane detection networks for autonomous driving, 2018.

[149] A. Waterman, Y. Lee, D. Patterson, K. Asanovic, and V. I. U. level Isa. The risc-v instruction set manual. *Volume I: User-Level ISA', version*, 2, 2014.

[150] C. J. C. H. Watkins and P. Dayan. Q-learning. In *Machine Learning*, volume 8, pages 279–292. Springer Science and Business Media LLC, 1992.

[151] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

[152] C. C. White III and D. J. White. Markov decision processes. *European Journal of Operational Research*, 39:1–16, 1989.

[153] D. Wu, B. M. Al-Hashimi, and P. Eles. Scheduling and mapping of conditional task graph for the synthesis of low power embedded systems. *IEE Proceedings - Computers and Digital Techniques*, 150(5):262–, Sep. 2003.

[154] H. Xu, Y. Gao, F. Yu, and T. Darrell. End-to-end learning of driving models from large-scale video datasets. In *Proc. IEEE/CVF CVPR*, pages 2174–2182, 2017.

[155] J. Xue, J. Fang, T. Li, B. Zhang, P. Zhang, Z. Ye, and J. Dou. Blvd: Building a large-scale 5d semantics benchmark for autonomous driving. In *Proc. IEEE ICRA*, pages 6685–6691, 2019.

[156] R. Yarmand, M. Kamal, A. Afzali-Kusha, and M. Pedram. Dart: A framework for determining approximation levels in an approximable memory hierarchy. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28:273–286, 2019.

[157] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran. Axbench: A multiplatform benchmark suite for approximate computing. *IEEE Design & Test*, 34:60–68, 2017.

[158] S. Yi, T. Kim, J. Kim, and N. Dutt. Energy-efficient adaptive system reconfiguration for dynamic deadlines in autonomous driving. In *IEEE ISORC*, pages 96–104, 2021.

[159] D. You and K. S. Chung. Dynamic voltage and frequency scaling framework for low-power embedded GPUs. *IET Electronics Letters*, 2012.

[160] F. Zaruba and L. Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27, 2019.

[161] H. Zhang, S. Zhao, A. Pattnaik, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das. Distilling the essence of raw video to reduce memory usage and energy at edge devices. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 657–669, New York, NY, USA, 2019. Association for Computing Machinery.

[162] B. Zimmer, S. O. Toh, H. Vo, Y. Lee, O. Thomas, K. Asanovic, and B. Nikolic. SRAM Assist Techniques for Operation in a Wide Voltage Range in 28-nm CMOS. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 59:853–857, 2012.