SEAMS: Self-Optimizing Runtime Manager for Approximate Memory Hierarchies

BISWADIP MAITY, University of California, Irvine, USA

BRYAN DONYANAVARD and ANMOL SURHONNE, Technical University of Munich, Germany

AMIR RAHMANI, University of California, Irvine, USA

ANDREAS HERKERSDORF, Technical University of Munich, Germany

NIKIL DUTT, University of California, Irvine, USA

Memory approximation techniques are commonly limited in scope, targeting individual levels of the memory hierarchy. Existing approximation techniques for a full memory hierarchy determine optimal configurations at design-time provided a goal and application. Such policies are rigid: they cannot adapt to unknown workloads and must be redesigned for different memory configurations and technologies. We propose SEAMS: the first self-optimizing runtime manager for coordinating configurable approximation knobs across all levels of the memory hierarchy. SEAMS continuously updates and optimizes its approximation management policy throughout runtime for diverse workloads. SEAMS optimizes the approximate memory configuration to minimize energy consumption without compromising the quality threshold specified by application developers. SEAMS can (1) learn a policy at runtime to manage variable application **quality of service (QoS)** constraints, (2) automatically optimize for a target metric within those constraints, and (3) coordinate runtime decisions for interdependent knobs and subsystems. We demonstrate SEAMS' ability to efficiently provide functions (1)–(3) on a RISC-V Linux platform with approximate memory segments in the on-chip cache and main memory. We demonstrate SEAMS' ability to save up to 37% energy in the memory subsystem without any design-time overhead. We show SEAMS' ability to reduce QoS violations by 75% with < 5% additional energy.

CCS Concepts: • **Computer systems organization** → *Embedded systems*; **Processors and memory architectures**; • **Theory of computation** → Reinforcement learning;

Additional Key Words and Phrases: Approximate computing, memory hierarchy, model-free control, RISC-V

ACM Reference format:

Biswadip Maity, Bryan Donyanavard, Anmol Surhonne, Amir Rahmani, Andreas Herkersdorf, and Nikil Dutt. 2021. SEAMS: Self-Optimizing Runtime Manager for Approximate Memory Hierarchies. *ACM Trans. Embedd. Comput. Syst.* 20, 5, Article 48 (July 2021), 26 pages.
<https://doi.org/10.1145/3466875>

The authors would like to thank DAAD for supporting this research under Grant no. 57440917. This work was also partially supported by NSF grant no. CCF-1704859.

Authors' addresses: B. Maity, A. Rahmani, and N. Dutt, Donald Bren School of Information and Computer Sciences, University of California, Irvine, Irvine, CA 92697; emails: {maityb, a.rahmani}@uci.edu, dutt@ics.uci.edu; B. Donyanavard, Technical University of Munich, Munich, Germany; email: bdonyanavard@sdsu.edu; A. Surhonne and A. Herkersdorf, Chair of Integrated Systems, Technical University of Munich, Theresienstr. 90, Munich 80290, Germany; emails: {anmol.surhonne, herkersdorf}@tum.de.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2021 Copyright held by the owner/author(s).

1539-9087/2021/07-ART48 \$15.00

<https://doi.org/10.1145/3466875>

1 INTRODUCTION

Intelligently trading off performance and energy in battery-powered systems is crucial as applications become increasingly resource-intensive. Application profiling reveals memory accesses as one of the most significant performance and energy bottlenecks [8]. Approximate memory is an effective way to alleviate the energy bottleneck in memory for applications that can tolerate output errors. Inexact memory load/store operations potentially improve energy consumption, leakage power, latency, or lifetime by relaxing the need for high-precision storage for some data structures.

Approximation techniques for different memory technologies with configurable degrees of approximation are well-known [10, 21, 32, 38–40], but typically limited to a subsection of the memory hierarchy. A holistic solution must manage approximation knobs across the entire memory hierarchy (e.g., V_{DD} for SRAM caches, t_{REF} for DRAM main memory), from cache to main memory, in order to fully exploit memory approximation opportunities. Furthermore, runtime dynamic reconfiguration of approximation knobs is required to fully leverage the performance/energy tradeoff for varying workloads while honoring application goals, e.g., **quality of service (QoS)** targets, and system constraints, e.g., minimize energy consumption.

Current approximation techniques using runtime dynamic reconfiguration depend on design-time modeling of workloads (application along with input) to determine the optimal knob configurations for a specific system configuration [19, 33, 44, 50], making the techniques application- and platform-specific. Design-time workload profiling requires observability, and introduces significant overhead to utilize approximate memory for each new memory technology. Runtime dynamic reconfiguration across the memory hierarchy requires coordination between multiple knobs (e.g., L1 V_{DD} , L2 V_{DD} , and DRAM t_{REF}). Coordination is challenging because knobs in one layer can affect other subsystems directly or indirectly (e.g., write errors in L1 affect reads in higher layers).

Consider the system shown in Figure 1. 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 [17, 50]. More flexible solutions have been proposed to provide dynamic configuration of knobs at runtime, but require identifying workload-specific system dynamics at design-time [23, 44]. Requiring *a priori* 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.

To address these challenges, we develop SEAMS, a model-free method to tune memory knobs without prior observation of the system (memory-hierarchy or workload). SEAMS eases the design of systems with approximate memory by enabling deployment without requiring design-time exploration of configuration knobs. SEAMS' methodology is independent of the underlying memory technologies and works regardless of the nature of knobs. Once deployed, SEAMS can learn optimal knob configurations for unknown applications, resulting in self-optimizing systems.

The main contributions of this article are as follows:

- (1) Enable **self-optimization** of multi-level memory approximation knobs through SEAMS, a runtime resource manager using reinforcement learning. Self-optimization is demonstrated by minimizing energy consumption while honoring a QoS constraint at runtime for unknown workloads.
- (2) Enable **coordination** between multiple memory system knobs without explicit communication. Coordination is demonstrated through dynamic runtime reconfiguration of multiple

Fig. 1. Runtime management of approximation knobs using output quality monitoring.

knobs by continuously evaluating different subsystem configurations (e.g., $\uparrow$ L1 knob, $\downarrow$ L2 knob vs. $\downarrow$ L1 knob, $\uparrow$ L2 knob).

- (3) An approximate memory management approach that is (a) technology agnostic, (b) application-independent, and (c) can easily be applied to any multi-level memory hierarchy.
- (4) **Experimental case study:** A software implementation of SEAMS is evaluated using an FPGA board with a modified RISC-V processing core to validate the approach.

We believe SEAMS will enable quick adoption of approximation using a variety of memory nodes.

2 MOTIVATION

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.

2.1 Case Study

Consider the system equipped with an approximate memory subsystem in Figure 1. 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 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 2 shows QoS observed for the system in Figure 1 for different applications and configurations of L1 and L2 approximation knobs.¹ The dots' size represents the QoS (i.e., number of errors; smaller is better). We observe the effect of configuration knobs on two applications:

¹The 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 5.

Fig. 2. 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.

- **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 2(a).
- **Application B:** The Canny-edge detection application as described in Section 5.2. 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 2(b) and (c), respectively.

We make three key observations: First, we observe that configurations achieving a target QoS vary both *within* and *between* applications. In Figure 2, we define a feasible region (dashed 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 2(a) (Application A), and Figure 2(b) (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 2(b) (input B1) and (c) (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.

2.2 Approximation: State-of-the-art

2.2.1 Open-Loop Control. A common approximation strategy is to use design-time techniques to find optimal knob configurations [19, 27, 31, 41]. 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 [19, 44, 50]. Thus, open-loop control techniques are application-specific and not portable to new systems.

2.2.2 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 [16, 29, 35, 53]. 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.

2.3 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 *a priori* knowledge are attractive candidates to find optimal solutions through runtime observation. Reinforcement learning [43] 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 BACKGROUND AND RELATED WORK

Approximate memory subsystems have been widely explored [28]. Identifying non-critical sections of data is an integral step toward running an approximate application. Allowing faults in the critical data sections leads to crashes and requires additional recovery mechanisms. Identifying non-critical data sections can either be done automatically [9, 36, 46], or through explicit programming language support [2]. Various memory approximation strategies can be implemented based on memory hardware technology given non-critical data is identified.

Table 1. Examples of Approximate Memory Technology Knobs

	Technology	Memory Type	Technology Knobs	Knob Objective	Reference
Cache	SRAM	Volatile	(*) Operating Voltage (V_{DD})	Energy savings	ASPLOS'12 [14], ESL'15 [39]
	STT-RAM	Non-Volatile	Read Voltage (V_{read}) Write Pulse Duration (t_{write})	Energy savings	HPCA'11 [40], CASES'15 [37], ISLPED'17 [30]
Main Memory	DRAM	Volatile	(*) Data Refresh Period (t_{REF}) Operating Voltage (V_{DD}) Row Activation Delay (t_{RCD})	Energy savings Reduce Latency	ASPLOS'11 [21], ISLPED'14 [10], MICRO'19 [19]
	PCM	Non-Volatile	Data Comparison Write (Th)	Energy savings Increase lifetime	MICRO'09 [32], MICRO'13 [38]

(*) used to evaluate SEAMS' performance in Section 6.

3.1 Approximate Memory Technologies

Table 1 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 1.

3.1.1 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 6, 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 [1], 70 nm [11], 28 nm [3, 54]). We use a model for a 6T SRAM at 65 nm node from [50] for comparison with related memory approximation work. Figure 3(a) shows the bit error rate for reads/writes corresponding to relative power supply voltage for the model under consideration.

3.1.2 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 for issuing refresh commands regularly to the DRAM. The frequent refresh operations (e.g., sending an AUTO-REFRESH command every 7.81 μ s 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 [21, 34] 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 Flicker [21] 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 Flicker DRAM model as our

(a) Bit error rate for a 6T SRAM cell with varying V_{DD} values in 65 nm. Data from [50].

(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 Flicker [21].

Fig. 3. Error and Power models used for (a) L1/L2 cache and (b) DRAM.

approximate DRAM memory technology with configurable refresh rate for the non-critical section of the memory array. Flicker is also used by a state-of-the-art memory approximation manager [50], allowing for comparison. Although the exact refresh rate required for reliable operation at runtime is affected by temperature, device capacity, and cell layout [18], these variations are not considered in this work.

3.2 Configuring Approximate Memory

Several methods have been proposed to tune memory approximation knobs. Table 2 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 [19], 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. [23] propose a solution to maintain a quality target at runtime using classical control theory. Quality configuration tracking is modeled as a formal quality-control problem, and black-box modeling 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 [44], 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.

Table 2. Memory Approximation Approaches and The Key Challenges Addressed
 (* = Uniquely Addressed by SEAMS)

Features	EDEN [19]	Control Theory [23]	AdAM [44]	DART [50]	SEAMS
Technology Independent		✓		✓	✓
Memory Hierarchy			✓	✓	✓
Application Agnostic					*
Coordination					*
Self-Adaptivity		✓			✓
Self-Optimization					*
Model-Independence	✓			✓	✓
Real System Evaluation	✓				✓

In DART [50], 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 *a priori* 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 [25, 26]. 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 [12, 22].

SEAMS incorporates the features highlighted in Table 2 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.

4 SEAMS METHODOLOGY

Figure 4 presents the SEAMS realization of the logical architecture described in Figure 1, 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 1. 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 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 [14] is another example of a microarchitecture design that efficiently supports ISA extensions for disciplined

Fig. 4. Overview of SEAMS system architecture.

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 5.3). 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 [49]: (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 [42].

4.1 Design Methodology

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

4.1.1 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. The state vector can be summarized as follows:

- (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.

4.1.2 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).

4.1.3 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

$$\begin{aligned} reward_P &= 1 - \frac{Power}{max_{Power}}, \\ reward_P &\in \{x | 0 \leq x \leq 1\}, \end{aligned} \quad (1)$$

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

$$\begin{aligned} reward_Q &= -\frac{Q - Q_{threshold}}{max_Q}, \\ reward_Q &\in \{x | -1 \leq x \leq 1\}, \end{aligned} \quad (2)$$

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.

Fig. 5. SEAMS taking actions against the environment, and the environment returns observations (updated state) and reward.

Table 3. Example Instance of Memory Hierarchy Knob Used to Evaluate SEAMS

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

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

$$R = \begin{cases} reward_P, & \text{if } Q \leq Q_{threshold} \\ reward_Q, & \text{otherwise.} \end{cases} \quad (3)$$

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

4.2 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 TD learning [43] to learn directly from experience without a model of the environment's dynamics. Figure 5 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 [48] 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 $TD(\lambda)$ [42]. SEAMS uses the $TD(\lambda)$ 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 5). 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 1.

ALGORITHM 1: TD(λ) algorithm [43] for determining SEAMS policy.

```

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

```

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 ($\alpha = 0.6$, $\gamma = 0.1$, and $\lambda = 0.95$) 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.

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 is repeatedly used to explore the state-action pairs and populate the value function $Q(s, a)$. This set of inputs forms an *episode* which groups the controller-environment interactions into smaller sequences. After the policy is initialized and SEAMS is exposed to new inputs, the 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 6.5. At the end of each episode, some information is updated, which is explained in lines [4 and 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 $Q(S', A^*)$.
- 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' .

4.3 SEAMS Case Study

Figure 4 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 uncore 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 and 1.0 V in increments of 0.1 V [50]. The main memory refresh periods are 0.1 s, 1 s, 5 s, and 20 s [21]. 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 6 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 5 and the Canny edge detection application (described in Section 5.2). 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.

Fig. 6. 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.

Reward R is calculated based on Equation (3). 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 1.

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 [21, 50].

5 EXPERIMENTAL SETUP

Figure 7 shows an overview of our evaluation platform. Prior work on configurable memory approximation uses simulation to evaluate methods [24]. 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 [5], an open-source framework designed to enable scalable architecture research prototypes. We use Openpiton with a single Ariane [52] 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 4.

5.1 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

Fig. 7. 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.

Table 4. System Configuration Used for SEAMS Evaluation

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

observable bit error rates [21, 50], and introduce bit errors through fault injection to emulate the effect of approximation knobs.

5.1.1 SRAM Fault Injection. The RISC-V specification defines separate addresses for CSRs associated with each hardware thread [47]. 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 7. 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-specific values described in Section 3.1. The knob settings from the CSRs are propagated to the ② 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 utilizes 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 7 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 FIs 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.

5.1.2 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-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 cells² 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.

5.2 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 5 summarizes the applications used for SEAMS' case study. (1) Canny edge detection [6] algorithm operates on a video stream and marks the edges in each frame. (2) k-means is a machine-learning application [51] that partitions three-dimensional input points (RGB pixels) into six different clusters. (3) blackscholes [51] 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 proposed [15, 45] 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 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

²The bit error rates are sourced from [21] for temperatures under 48°C.

Table 5. Applications Used for SEAMS' Evaluation Along with Their Inputs and QoS

Application	Domain	Input Size	Quality Metric
canny [6]	Image-Processing	352×288 (Grayscale)	Image Diff (RMSE)
k-means [51]	Machine Learning	426×240 (RGB)	Image Diff (RMSE)
blackscholes [51]	Finance	4 K entries	Avg. Relative Error

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 the 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 [51]. 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 [4, 13]. 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 6.5. If additional cores are available, ground-truth comparison can be performed in parallel. In our uncore setup, ground-truth comparison incurs unavoidable overhead during the initial exploration phase.

5.3 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.

5.4 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 [7] simulations with McPAT [20], along with existing power models from literature [21, 50], to compute the power consumption at different knob settings. Figure 8 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.

Fig. 8. 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.

For SRAM, we use the 65 nm node model proposed in [50]. For DRAM, we use the low-power DDR model proposed in Flikker [21]. 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.

6 PERFORMANCE EVALUATION

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.

6.1 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 9 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 9 represents frames processed, and the y -axis represents average memory power (normalized to the exact execution) for each episode for canny edge detection application [6]. We make two major observations. First, both algorithms can eventually converge to the optimal policy. Second, in Figure 9(a), 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 9(b)), TD(λ) can improve its policy faster than Q-learning

(a) Configuration space = 64 states

(b) Configuration space = 704 states

Fig. 9. 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).

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.

6.2 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 [50]. 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 10(a), 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 has experienced this scene previously, and the scene requires a new configuration of knobs to continue to meet the QoS requirement. In

Fig. 10. SEAMS self-optimizing power within quality constraint.

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 10(b) 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 the 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 $\times$ more than SEAMS (Figure 10(a)). 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.

Fig. 11. SEAMS self-adapting to user-specified quality constraints by coordination across the memory hierarchy.

6.3 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 11 shows how SEAMS behaves in the scenario described. The two key observations are as follows: (1) Figure 11(a) 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 11(b) 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

Fig. 12. Additional workloads.

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.

6.4 Additional Workloads

Figures 12(a) and (c) show SEAMS' result for kmeans with a normalized energy/frame of 76.4 %. Figures 12(b) and (d) show SEAMS' result for blackscholes with a normalized energy/frame of 70.9 %. We observe that even though the interdependent dynamics 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.

6.5 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 13, 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 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.

Fig. 13. SEAMS QoS at different invocation intervals for k -means. Blue ticks indicate evaluation instances.

Fig. 14. SEAMS overhead for different intervals.

In Figure 14, 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 uncore—in a many-core system, the proposed control structure will not scale well due to configuration 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.

7 CONCLUSION

In this article, we address the challenge of designing a technology-agnostic runtime manager for approximate memory hierarchies. To this end, we propose SEAMS, the first self-optimizing, model-free runtime manager for tuning approximation knobs in a unified multi-layer memory hierarchy. SEAMS achieves acceptable QoS for diverse workloads while minimizing energy consumption. SEAMS uses TD learning to learn directly from experience without a model of the environment's dynamics. We develop an experimental case study to evaluate SEAMS on a modified RISC-V processing core. We demonstrate the efficacy of SEAMS for configuration knobs when (1) there is no quality constraint, and the system objective is to minimize energy; (2) there is a quality constraint, and unknown inputs require reconfiguration of the knobs; and (3) the user dynamically changes the QoS constraints. SEAMS can automatically control approximate memory hierarchies regardless of memory technology or application. Due to these advantages, we believe that SEAMS has the potential for developing extremely energy-efficient systems across memory devices, parameters, and technologies.

REFERENCES

- [1] Amin Ansari, Shuguang Feng, Shantanu Gupta, and Scott Mahlke. 2009. 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*. Association for Computing Machinery, New York, NY, 307–310.
- [2] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. 2011. Language and compiler support for auto-tuning variable-accuracy algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 85–96.
- [3] F. Arnaud, A. Thean, M. Eller, M. Lipinski, Y. W. Teh, M. Ostermayr, K. Kang, N. S. Kim, K. Ohuchi, J. P. Han, et al. 2009. Competitive and cost effective high-k based 28nm CMOS technology for low power applications. In *IEEE International Electron Devices Meeting (IEDM)*. IEEE, 1–4.
- [4] Woongki Baek and Trishul M. Chilimbi. 2010. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, 198–209.
- [5] Jonathan Balkind, Katie Lim, Fei Gao, Jinzheng Tu, David Wentzlaff, Michael Schaffner, Florian Zaruba, and Luca Benini. 2019. 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*. CARRV.
- [6] John Canny. 1986. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8 (1986), 679–698.
- [7] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An Evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)* 11 (2014), 1–25.
- [8] Aaron Carroll, Gernot Heiser, et al. 2010. An analysis of power consumption in a smartphone. In *Proceedings of Annual Technical Conference*, Vol. 14. USENIX Association, 21.
- [9] Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference*. Association for Computing Machinery, New York, NY, 1–9.
- [10] Kyungsang Cho, Yongjun Lee, Young H. Oh, Gyoo-cheol Hwang, and Jae W. Lee. 2014. eDRAM-based tiered-reliability memory with applications to low-power frame buffers. In *Proceedings of International Symposium on Low Power Electronics and Design*. Association for Computing Machinery, New York, NY, 333–338.
- [11] Minki Cho, Jason Schlessman, Wayne Wolf, and Saibal Mukhopadhyay. 2009. Accuracy-aware SRAM: A reconfigurable low power SRAM architecture for mobile multimedia applications. In *Asia and South Pacific Design Automation Conference*.
- [12] Bryan Donyanavard, Tiago Mück, Amir M. Rahmani, Nikil Dutt, Armin Sadighi, Florian Maurer, and Andreas Herkersdorf. 2019. 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*. Association for Computing Machinery, New York, NY, 685–698.

- [13] Rudolf Eigenmann et al. 2017. Harnessing parallelism in multicore systems to expedite and improve function approximation. In *Languages and Compilers for Parallel Computing*. Springer International Publishing, Cham, 88–92.
- [14] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Architecture support for disciplined approximate programming. In *Proceedings of the 17th international Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, 301–312.
- [15] Thomas Goldbrunner, Thomas Wild, and Andreas Herkersdorf. 2018. Memory access pattern profiling for streaming applications based on MATLAB models. In *28th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS'18)*. IEEE, 32–38.
- [16] Beayna Grigorian, Nazanin Farahpour, and Glenn Reinman. 2015. BRAINIAC: Bringing reliable accuracy into neurally-implemented approximate computing. In *IEEE 21st International Symposium on High Performance Computer Architecture*. IEEE, 615–626.
- [17] Jiajia Jiao. 2020. HEAP: A holistic error assessment framework for multiple approximations using probabilistic graphical models. *Electronics* 9 (2020), 373.
- [18] Matthias Jung, Éder Zulian, Deepak M. Mathew, Matthias Herrmann, Christian Brugger, Christian Weis, and Norbert Wehn. 2015. Omitting refresh: A case study for commodity and wide I/O DRAMs. In *Proceedings of the 2015 International Symposium on Memory Systems*. Association for Computing Machinery, New York, NY, 85–91.
- [19] Skanda Koppula, Lois Orosa, A. Giray Yağlıkcı, Roknoddin Azizi, Taha Shahroodi, Konstantinos Kanellopoulos, and Onur Mutlu. 2019. 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*. Association for Computing Machinery, New York, NY, 166–181.
- [20] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. 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*. Association for Computing Machinery, New York, NY, 469–480.
- [21] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. 2011. Flicker: Saving DRAM refresh-power through critical data partitioning. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, 213–224.
- [22] Biswadip Maity, Bryan Donyanavard, and Nikil Dutt. 2020. Self-aware memory management for emerging energy-efficient architectures. In *2020 11th International Green and Sustainable Computing Workshops (IGSC'20)*. 1–8.
- [23] Biswadip Maity, Majid Shoushtari, Amir M. Rahmani, and Nikil Dutt. 2019. Self-adaptive memory approximation: A formal control theory approach. *IEEE Embedded Systems Letters* 12 (2019), 33–36.
- [24] Biswadip Maity, Majid Shoushtari, Amir M. Rahmani, and Nikil Dutt. 2019. Simulation infrastructure and system dynamics of quality configurable memory. CECS Technical Report 19-03.
- [25] Mahmoud Masadeh, Osman Hasan, and Sofiene Tahar. 2019. Using machine learning for quality configurable approximate computing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE'19)*. IEEE, 1575–1578.
- [26] Mahmoud Masadeh, Osman Hasan, and Sofiene Tahar. 2020. Machine learning-based self-compensating approximate computing. arXiv:2001.03783.
- [27] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. 2014. Load value approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 127–139.
- [28] Sparsh Mittal. 2016. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)* 48 (2016), 1–33.
- [29] Kasra Moazzemi, Biswadip Maity, Saehanseul Yi, Amir M. Rahmani, and Nikil Dutt. 2019. HESSLE-FREE: Heterogeneous systems leveraging fuzzy control for runtime resource management. *ACM Transactions on Embedded Computer Systems* 18, 5s, Article 74 (2019).
- [30] Amir Mahdi Hosseini Monazzah, Majid Shoushtari, Seyed Ghassem Miremadi, Amir M. Rahmani, and Nikil Dutt. 2017. 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'17)*. IEEE, 1–6.
- [31] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. 2015. SNNAP: Approximate computing on programmable SoCs via neural acceleration. In *IEEE 21st International Symposium on High Performance Computer Architecture*. IEEE, 603–614.
- [32] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. 2009. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. Association for Computing Machinery, New York, NY, 14–23.
- [33] A. Raha and V. Raghunathan. 2018. Approximating beyond the processor: Exploring full-system energy-accuracy tradeoffs in a smart camera system. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 12 (2018), 2884–2897.

- [34] A. Raha, S. Sutar, H. Jayakumar, and V. Raghunathan. 2017. Quality configurable approximate DRAM. *IEEE Transactions on Computers* 66, 7 (2017), 1172–1187.
- [35] Michael Ringenburg, Adrian Sampson, Isaac Ackerman, Luis Ceze, and Dan Grossman. 2015. 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*. Association for Computing Machinery, New York, NY, 399–411.
- [36] Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. 2014. ASAC: Automatic sensitivity analysis for approximate computing. In *Proceedings of the SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*. Association for Computing Machinery, New York, NY, 95–104.
- [37] Felipe Sampaio, Muhammad Shafique, Bruno Zatt, Sergio Bampi, and Jörg Henkel. 2015. Approximation-aware Multi-Level Cells STT-RAM cache architecture. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'15)*. IEEE, 79–88.
- [38] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. 2013. Approximate storage in solid-state memories. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, New York, NY, 25–36.
- [39] Majid Shoushtari, Abbas BanaiyanMofrad, and Nikil Dutt. 2015. Exploiting partially-forgetful memories for approximate computing. *IEEE Embedded Systems Letters* 7 (2015), 19–22.
- [40] Clinton W. Smullen, Vidyabhushan Mohan, Anurag Nigam, Sudhanva Gurumurthi, and Mircea R. Stan. 2011. Relaxing non-volatility for fast and energy-efficient STT-RAM caches. In *IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 50–61.
- [41] Renée St. Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. 2014. General-purpose code acceleration with limited-precision analog computation. In *Proceedings of the 41st Annual International Symposium on Computer Architecture*. IEEE, 505–516.
- [42] Richard S. Sutton. 1988. Learning to predict by the methods of temporal differences. *Machine Learning* 3 (1988), 9–44.
- [43] Richard S. Sutton and Andrew G. Barto. 2018. *Introduction to Reinforcement Learning*. MIT Ppress, Cambridge, MA.
- [44] Mohammad Taghi Teimoori, Muhammad Abdullah Hanif, Alireza Ejlali, and Muhammad Shafique. 2018. AdAM: Adaptive approximation management for the non-volatile memory hierarchies. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 785–790.
- [45] Radha Venkatagiri, Khaliq Ahmed, Abdulrahman Mahmoud, Sasa Misailovic, Darko Marinov, Christopher W. Fletcher, and Sarita V. Adve. 2019. 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'19)*. IEEE, 214–221.
- [46] Radha Venkatagiri, Abdulrahman Mahmoud, Siva Kumar Sastry Hari, and Sarita V. Adve. 2016. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–14.
- [47] Andrew Waterman, Yunsup Lee, David Patterson, and Krste Asanović. 2014. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2*, Technical Report No. UCB/EECS-2014-54. University of California at Berkeley.
- [48] Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-learning. In *Machine Learning*, Vol. 8. Springer Science and Business Media LLC, 279–292.
- [49] Chelsea C. White III and Douglas J. White. 1989. Markov decision processes. *European Journal of Operational Research* 39 (1989), 1–16.
- [50] Roohollah Yarmand, Mehdi Kamal, Ali Afzali-Kusha, and Massoud Pedram. 2019. DART: A framework for determining approximation levels in an approximable memory hierarchy. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28 (2019), 273–286.
- [51] Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. 2017. AxBench: A multiplatform benchmark suite for approximate computing. *IEEE Design & Test* 34 (2017), 60–68.
- [52] Florian Zaruba and Luca Benini. 2019. 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).
- [53] Haibo Zhang, Shulin Zhao, Ashutosh Pattnaik, Mahmut T. Kandemir, Anand Sivasubramaniam, and Chita R. Das. 2019. 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*. Association for Computing Machinery, New York, NY, 657–669.
- [54] Brian Zimmer, Seng Oon Toh, Huy Vo, Yunsup Lee, Olivier Thomas, Krste Asanovic, and Borivoje Nikolic. 2012. SRAM assist techniques for operation in a wide voltage range in 28-nm CMOS. *IEEE Transactions on Circuits and Systems II: Express Briefs* 59 (2012), 853–857.

Received November 2020; revised February 2021; accepted May 2021