Self-aware Memory Management for Emerging Energy-efficient Architectures

(Invited Paper)

Biswadip Maity
Department of Computer Science
University of California, Irvine
Irvine, USA
maityb@uci.edu

Bryan Donyanavard
Ericsson Research
Stockholm, Sweden
bryan.donyanavard@ericsson.com

Nikil Dutt
Department of Computer Science
University of California, Irvine
Irvine, USA
dutt@uci.edu

***Abstract*—With the advent of GPUs and application-specific accelerators in embedded platforms, data-intensive applications have exacerbated the memory performance and energy bottleneck. Memory requirements and usage patterns vary widely in emerging architectures, and resource contention manifests differently based on the instance of the architecture. Workload-specific and system-specific optimizations for energy-efficient architectures are impractical due to the fast-evolving landscape of computer applications and hardware. We discuss how to apply self-awareness principles to design an energy-efficient memory subsystem, and the different degrees of self-awareness such a system can achieve. We apply these principles on approximate memory systems and observe energy savings of 10.5% for on-chip L1 cache. We believe this is a rich area for research and outline some future opportunities for using self-awareness in emerging energy-efficient architectures.**

***Index Terms*—Computational Self-awareness, Memory Management, Approximate Computing.**

I. Introduction

Modern applications are increasingly data-centric, producing and processing terabytes of data, resulting in bottlenecks in memory and storage subsystems. Innovations in storage devices (e.g., NAND flash, DRAMs, PCRAMs) have helped maintain Moore's law trajectory. However, memory bandwidth requirements of modern applications (e.g., video transcoding, machine learning) are increasing faster than memory technology is advancing.

This problem is further exacerbated in embedded systems where data movement remains a significant performance and energy bottleneck. Figure 1 illustrates an example of an emerging embedded architecture with heterogeneous processing elements. The data-movement can occur within and across different elements, and main-memory accesses remain one of the biggest bottlenecks. Current efforts in computer architecture research support platform heterogeneity in three common ways. (1) Sophisticated out-of-order heterogeneous cores with highly optimized micro-architecture pipelines. These architectures fail to fully exploit the parallelism inherent in some applications (e.g., image processing), pose significant challenges for effective resource scheduling to prevent noisy neighbors from hogging shared memory bandwidth, and fail to adapt to dynamic application behavior. (2) GPUs with small but numerous cores that are used to achieve data-driven efficiency. GPU SIMD engines can exploit inherent application parallelism; however, application developers often lack the hardware expertise to optimize data movements, resulting in excessive memory contention. (3) Application-specific solutions (e.g., application-specific circuits to accelerate bottleneck tasks). Application-specific processing elements suffer from large orchestration overhead to set up and tear down the region of interest.

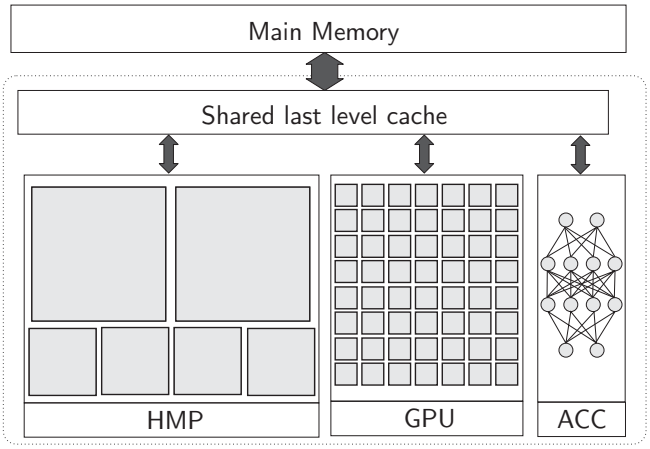


Fig. 1: Example of an embedded platform with heterogeneous processing elements on the chip. The processing elements shown here are Heterogeneous Multiprocessor (HMP), Graphics Processing Unit (GPU), and Accelerator (ACC). The processing elements share a main-memory, which is a significant performance and energy bottleneck in emerging architectures.

Memory requirements and usage patterns vary widely in emerging architectures, and depend on the architecture implementation. Furthermore, active developments in device technologies [21], [8] and rapid changes in computing architectures make workload/architecture-specific optimizations impractical. A promising approach to provide the required flexibility is to

This work was supported in part by NSF under Grant CCF-1704859.

978-1-6654-1552-1/20/$31.00 ©2020 IEEE

use intelligent management via computational self-awareness principles, in which systems can learn from experience and adapt to changes. The growing literature in computational self-awareness proposes several mechanisms for a system to build models through observation, and use the knowledge gained to make efficient decisions at runtime [33]. However, designing intelligent management requires a description of the desired system properties, and a reference architecture that engineers can use to provide self-awareness capabilities in the system. We discuss how self-awareness properties can be advantageously applied to the memory subsystem and demonstrate the utility through a memory approximation use-case.

II. Systems and the 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.

A. 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. 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 [10] 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 [52]. 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 [19]. 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 [48] to deep learning [23] 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.* [5] 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.

B. 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. 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 [46]. Online learning techniques (e.g., reinforcement learning) can also be leveraged to automatically learn policies specific to workloads, reducing the burden on system designers [15]. We now review some representative

efforts in both design-time, as well as run-time optimizations that exploit ML techniques.

1) Design-time Optimizations: The design-time objective is to evaluate different system configurations across varied dimensions of power, performance, temperature, reliability, and estimate behavior of real hardware. Design-space exploration requires careful profiling of representative workloads on available hardware and observation (or simulation) of system configurations for all objective metrics of interest (e.g., performance, power). Researchers develop machine learning techniques as part of design-time optimization frameworks to explore configuration space strategically [46], [45], [7]. Similar techniques have been applied in the memory subsystem. Sen and Imam [4] present ML techniques for developing various memory-response models that can instantly provide a predicted response corresponding to any new memory configuration to help design hybrid main memories. Navarro *et al.* [44] develop an ML methodology for cache memory design by predicting the optimal cache reconfiguration for any given application, based on its dynamic instructions. Hasemi *et al.* [39] demonstrate the ability to learn memory access patterns through recurrent neural networks.

2) Run-time Optimizations: Once a system is deployed with the configuration obtained from design-space exploration, several factors (e.g., device variability, dynamic workloads, aging) require further adaptation at runtime. These adaptations can be performed with the help of (1) 'knobs': adjustable parameters in the systems and applications (e.g., core operating frequency, task mapping), and (2) 'sensors': provide telemetry information to aid runtime decisions (e.g., temperature, power, applications FPS). Runtime optimizations for memory subsystems require policies that are aware of the underlying technology and architecture properties, observe the manifestation of memory behavior at runtime, and prioritize system goals accordingly. Several works propose such policies in the memory subsystem for runtime adaptivity [42], [36], [35]. More recently, intelligent computing systems use machine learning techniques to manage critical hardware resources at runtime efficiently [1]. Broadly, ML-based runtime-optimizations can be categorized into two bins: (1) model-based control through prediction [14], and (2) model-free control through decisions [15]. Prior research has explored the use of intelligent agents within the memory subsystem. Ipek *et al.* in [27] presents a reinforcement-learning-based method in the memory-controller for the complex problem of DRAM scheduling. In Kleio [50], Doudali *et al.* presents a page scheduler with machine intelligence for applications that execute over hybrid memory systems. Huang *et al.* [37] develop a reinforcement-learning-based joint task offloading and bandwidth allocation for multi-user mobile edge computing scenarios.

ML-based techniques for system design rely on black-box models with very limited interpretability, and cannot be reasoned about. On the other hand, self-awareness in computing systems [49] allows us to develop an intelligent system by building reasoning on top of the black-box models along with other properties like *adaptation* and *self-healing*. Computational self-awareness principles draw on a large body of work from different communities such as psychology, neuroscience, autonomic computing, machine learning, artificial intelligence, and multi-agent systems. The ability to reason using the models allows the system to *introspect*: a fundamental property of a self-aware computational system as described in the next section. Self-awareness properties can be used to leverage the most out of the available resources and improve the performance, energy-efficiency, reliability, and fast adoption of emerging computing systems. Therefore, we now look at the characteristic properties of computational self-awareness and how they can be applied to a memory subsystem.

III. Computational Self-awareness

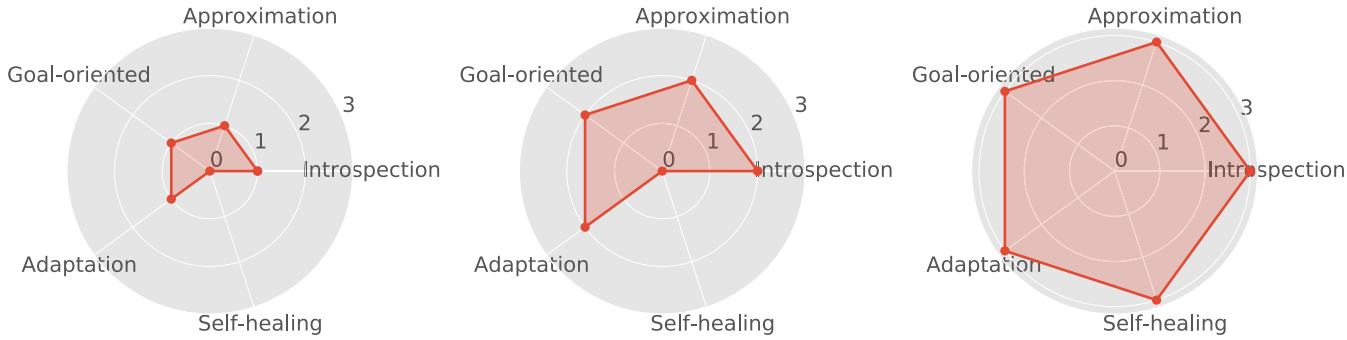
The growing literature in self-aware computing systems suggests that intelligent systems are being designed and deployed that can learn and adapt at runtime. It draws its roots from the fields of psychology and neuroscience and integrates interdisciplinary research.

Kounev *et al.* [49] 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.

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 [30] versus in robotics [51]. Agarwal *et al.* [6] examine the fundamental properties that pertain to self-aware computation: introspection, approximation, goal orientation, adaptation, and self-healing. Bellman *et al.* [33] 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*: the ability to observe the system and environment during execution, reflect on its behavior, and learn. Platform and application-level telemetry are utilized as sensors to observe the behavior of the memory subsystem at runtime. The sensors are spread across multiple abstraction layers: device, hardware, kernel, vendor-library, and application layer.

  Figure 3 shows the different abstraction layers with corresponding sensors. Examples of sensors in the memory subsystem: cache miss rate at various levels, main-memory bandwidth, main-memory latency, working set



(a) Traditional (b) Ongoing work (c) Comprehensive self-awareness

Fig. 2: Self-aware Approximate Memories: (a) past, (b) present and (c) future. Values corresponding to each property are explained in Section V.

of processes, numbers of errors in data transmissions, or memory access, CUPTI [3] sensors for CUDA GPU kernels, and application-level QoS (e.g., FPS).

- *Approximation*: automatically choosing the level of precision required for the execution of a task. Due to constrained resources (e.g., energy, memory bandwidth) in computing systems, it is essential to utilize the least amount of precision to accomplish the task at hand and not waste more resources than necessary. As seen in Section II, data movement is a significant performance and energy bottleneck in emerging architectures. Thus, researchers have explored knobs for emerging memory technologies to tradeoff accuracy of memory load/stores to achieve higher performance and energy-efficiency [11], [12], [40]. We discuss approximation for memory subsystem in detail in Section V.
- *Goal orientation*: attempt to meet user's or application's goal while optimizing under the constraints. Goals encapsulate a user's requirements without any specification about how to achieve them (e.g., processing at-least 30 FPS in a video application, maintain a maximum dissipation power to avoid shutdown). Emerging many-core systems are highly complex and require thorough orchestration of different goals across the computing abstraction stack to satisfy constraints [9]. These goals are dynamic and change with time, as well as across different abstraction layers. In the memory subsystem, a goal can be to limit the maximum allocated bandwidth for main-memory access for efficient performance isolation in multi-core platforms [32]. For an energy-efficient system, there are multiple ways to formulate the goal of the memory subsystem. For example: (1) meet the application's QoS while minimizing the energy consumption, (2) if the application does not report QoS, then the goal can be formulated as minimizing the energy consumed per unit of work executed by the system.
- *Adaptation*: Ability of the computing-system to dynamically change the operating configuration using decisions at runtime. A self-adaptive system: (1) analyzes the observed information about system state and environment at runtime, (2) computes the difference between the observed information and current goals, and (3) tunes the knobs and selects a different operating point in the operating space of knobs if necessary to reach the goal. Through adaptivity, a system can realize the goals currently defined by a user/application. A key challenge in selecting the values of actuation knobs is the size of the associated design space. The effect of knobs is not always predictable, and dynamic interactions across knobs often result in expected behavior. In a memory subsystem, examples of knobs include memory-controller schedule, bandwidth reservation, dynamic voltage and frequency scaling (DVFS) of memory-controller, and load/store accuracy.
- *Self-healing*: ensure correct operation in the face of unexpected errors or incorrect emergent behavior. Self-healing systems introspect to observe errors at runtime and perform appropriate actions to adapt to the errors. Although it is a special case of adaptation, *self-healing* is still defined as a separate property because emerging architectures can be used for safety-critical applications (e.g., autonomous driving, pacemaker) [24], [25], [26]. A cross-layer representation of faults can be modeled using a Resilience Articulation Point (RAP) model [13] where *faults* in physical sources (e.g., process variation, temperature) causes *errors* in the form of bit flips in the memory, which can corrupt the stored data, ultimately causing the application to *fail*. Furthermore, multi-layer analysis can improve memory resilience [41], [43]. Self-healing systems need to implement appropriate mechanisms to detect and recover from such situations.

Since the notion of self-awareness can be applied to various domains, the associated properties and their definitions vary both within and across domains. Following the definition in [6], we restrict our discussions to these five critical properties for computational self-awareness to see some examples in the memory-subsystem; however, other self-* properties remain to be explored.

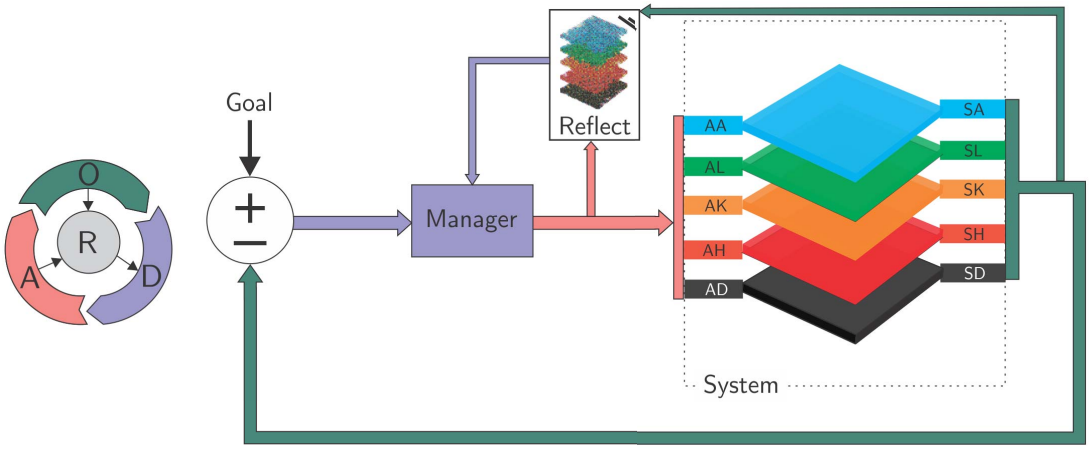


Fig. 3: Observe-Decide-Act (ODA) loop in an embedded system with reflection (R) for self-aware management [17]. The layers shown are application (blue), vendor libraries (green), kernel (orange), hardware (red), and device (black). Each layer has sense (S) and act (A) capabilities.

IV. Implementation through Reflection

Implementation of self-aware systems require fundamentally new design methodologies [33] to (1) continuously monitor the state of the system and the environment, (2) reason about the current state through reflection and learning from past experiences, and (3) dynamically adapt to new goals. We exploit the Observe-Decide-Act (ODA) loop to systematically implement some properties of computational self-awareness described in Section III to alleviate the memory energy-efficiency bottleneck. Figure 3 demonstrates a reference architecture for realizing self-awareness by adding *reflection* in the ODA loop [17].

A. Observe

In the observation step, telemetry information is collected about the system along with the environment. It is one of the pillars of *introspection* (the other being *reflection*). The collected information is then either (1) used directly by the decision step or (2) used to construct static/dynamic models along with episodic history (discussed in the following subsection), which is then utilized by the decision making process.

Figure 3 follows a similar approach for cross-layer sensing (shown with a dark-green arrow), where information is collected from across the abstraction layers for efficiently managing the memory subsystem. Prior works [31], [32] utilize main-memory bandwidth information to efficiently manage memory subsystems by determining runtime requirements and develop dynamic policies to configure system knobs (e.g., memory-controller frequency, bandwidth reservation) accordingly.

A singular metric like bandwidth utilization is not always sufficient: policies for a range of workload scenarios require insight into an application's memory access pattern and working set size. Memory profilers provide fine-grained information such as the memory access pattern for the entire virtual address space, call-stack information, or load/store density of different memory regions. However, parsing this detailed information frequently at runtime induces excessive overhead. In [19], Maity *et al.* perform an initial study based on a metric (WBP) that combines the working-set-size and main-memory bandwidth to characterize data-centric applications based on their memory access patterns. WBP can be estimated with low overhead, and the combined metrics provide insight that runtime policies can use to decide the desired system configuration for specific workload scenarios. Early results show that a static configuration devised with this metric yields an optimal memory-controller frequency 80% of the time for PARSEC workloads [22], demonstrating the promise of this approach.

B. Reflect and Decide

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 [16] 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 3, a self-model of the system is being used to performed the *reflection* (shown in white box).

The runtime manager (violet box in Figure 3) is responsible for closing the loop by making decisions about the system under control. Runtime resource management through decision making is a well-researched area. Several efforts have been undertaken for energy-efficient management of the memory

TABLE I: 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.

subsystem and can broadly be categorized into: (1) heuristic-based [2], [29], (2) control-theory-based [34], [38], and (3) machine-learning-based [27], [28]. 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.

C. Act

Once the decision-making engine determines the updated operating configuration, the next step is to change the knobs of the system. Several knobs already exist in a system that effect the energy-efficiency of the memory subsystem: per-cluster CPU DVFS, GPU frequency, memory-controller frequency, scheduling memory requests in the controller, active tasks in the heterogeneous processing elements. With emerging architectures (e.g., RISC-V, on-chip accelerators), new knobs are exploding the design space, making the runtime decision process of selecting optimal operating points even harder. Moreover, due to the shared memory in embedded systems, one knob's effect on another can be unpredictable due to memory contention. To explore the effect of novel actuators in the memory subsystem, Maity *et al.* [20] implement device-agnostic memory approximation knobs using the Sniper simulator [18]. We discuss memory approximation in more detail in Section V as it is essential in computational self-awareness.

V. Use case: Approximate Memories

Modern data-centric applications often contain large data sections that do not need to adhere to an all-or-nothing correctness model. Inherently, these applications are resilient to certain imprecision levels, which leads to efficient utilization of underlying resources. Exploiting imprecision opportunity is crucial in designing energy-efficient memory systems as applications with models consisting of trillions of parameters and terabytes of storage become a reality and continue to grow.

Approximation during memory load/store/hold operations takes us one step closer to a self-aware computing system, aiming to utilize the least amount of precision required to accomplish the system/application goals. Traditionally, application programmers are burdened with the difficult task of setting memory approximation knobs to achieve the desired quality of service. An approach based on self-awareness principles can potentially alleviate the manual task of tuning the knobs with an intelligent memory manager as shown in Figure 4. In [20], a control-theory-based approach where developers only specify a target QoS metric is proposed. The system uses a formal control-theoretic approach to tune the memory reliability knobs of a quality-configurable memory to guarantee the desired QoS. Before the system is deployed, a system model is identified for the memory components and their behavior given different approximation settings using a statistical black-box modeling technique. Using the system model, a controller is designed that observes the application's behavior at fixed epochs and tunes knobs automatically to deliver the desired QoS in the face of workload and system variation. We use this runtime memory-approximation exemplar to demonstrate the realization of self-awareness properties in Table I. Traditional, state-of-art, and future directions in self-aware approximate memories are shown in Figure 2.

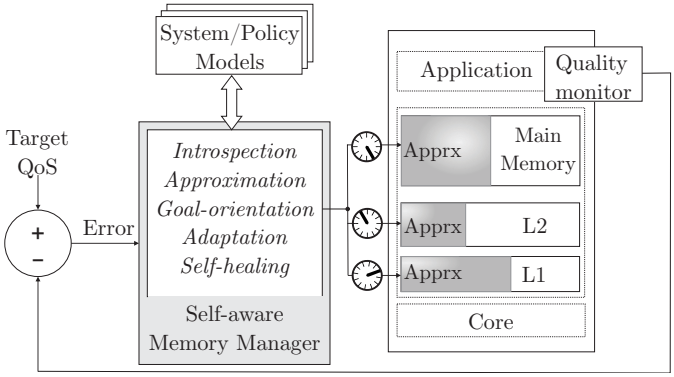


Fig. 4: Self-aware Memory Management using output quality monitoring.

Preliminary results from [20] show that self-adaptive memory approximation using formal control theory can alleviate the programmer's burden of manual knob tuning for on-chip memory approximation. Energy savings of 10.5% are

achieved for the L1 data cache using this method. Recent work [47] for determining approximation knobs using a design-time exploration indicates up to 54% and 22% power consumption improvements for the SRAM cache and the DRAM memory, respectively, highlighting opportunity for improvement in future self-aware approximate memory systems.

VI. Conclusion

We discuss the challenges of energy-efficiency in emerging architectures and how they can be addressed using self-awareness principles, particularly in the context of memory management. Self-awareness properties can be applied for designing an energy-efficient memory subsystem and the different degrees of self-awareness the system can achieve. While machine learning-based black-box methods are commonly used today, they lack interpretable reasoning and cannot fully leverage the available system resources. Through *reflection* in self-awareness, system managers are able to reason using models when making decisions about system configuration. We demonstrate the utility of computational self-awareness with a approximate memory use-case, in which our preliminary results demonstrate 10.5% energy savings for L1 data cache. The use of computational self-awareness principles is a promising and exciting direction of research for memory management that has great potential for improving the performance, energy-efficiency, reliability, and fast adoption of emerging computer architectures and newer memory substrates.

References

[1] J. F. Martinez and E. Ipek. Dynamic Multicore Resource Management: A Machine Learning Approach. *IEEE Micro*, 2009.

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

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

[4] S. Sen and N. Imam. Machine Learning Based Design Space Exploration for Hybrid Main-Memory Design. In *Proc. MEMSYS*, 2019.

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

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

[7] A. Deshwal *et al.* MOOS: A Multi-Objective Design Space Exploration and Optimization Framework for NoC Enabled Manycore Systems. *ACM TECS*, 2019.

[8] A. Ghosh *et al.* Communication - Impact of Electrode Chemistry on the Non-Volatile Performance of Lithium Niobite Memristors for Neuromorphic Computing. *ECS JSS*, 2020.

[9] A. M. Rahmani *et al.* HDGM: Hierarchical Dynamic Goal Management for Many-Core Resource Allocation. *IEEE ESL*, 2018.

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

[11] A. Raha *et al.* Quality Configurable Approximate DRAM. *IEEE TC*, 2017.

[12] A. Sampson *et al.* Approximate storage in solid-state memories. In *Proc. MICRO*, 2013.

[13] Andreas Herkersdorf *et al.* Resilience Articulation Point (RAP): Cross-layer dependability modeling for nanometer system-on-chip resilience. *Microelectron. Reliab.*, 2014.

[14] B. Donyanavard *et al.* SPARTA: Runtime task allocation for energy efficient heterogeneous manycores. In *Proc. CODES+ISSS*, 2016.

[15] B. Donyanavard *et al.* SOSA: Self-Optimizing Learning with Self-Adaptive Control for Hierarchical System-on-Chip Management. In *Proc. MICRO*, 2019.

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

[17] B. Donyanavard *et al. Reflecting on Self-Aware Systems-on-Chip*. Springer International Publishing, Cham, 2021.

[18] B. Maity *et al.* Simulation Infrastructure and System Dynamics of Quality Configurable Memory. *CECS Tech. Rep. 19-03*, 2019.

[19] B. Maity *et al.* Workload Characterization for Memory Management in Emerging Embedded Platforms. In *Proc. IESS*, 2019.

[20] B. Maity *et al.* Self-Adaptive Memory Approximation: A Formal Control Theory Approach. *IEEE ESL*, 2020.

[21] B. Zivasatienraj *et al.* Temporal versatility from intercalation-based neuromorphic devices exhibiting 150 mV non-volatile operation. *AIP JAP*, 2020.

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

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

[24] E. A. Rambo *et al.* The Information Processing Factory: A Paradigm for Life Cycle Management of Dependable Systems. In *Proc. CODES+ISSS*, 2019.

[25] E. A. Rambo *et al.* The Information Processing Factory: Organization, Terminology, and Definitions. *arXiv:1907.01578 [cs.DC]*, 2019.

[26] E. A. Rambo *et al.* The Self-Aware Information Processing Factory Paradigm for Mixed-Critical Multiprocessing. *IEEE TETC*, 2020.

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

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

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

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

[31] H. David *et al.* Memory Power Management via Dynamic Voltage/Frequency Scaling. In *Proc. ICAC*, 2011.

[32] H. Yun *et al.* MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Proc. RTAS*, 2013.

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

[34] K. Moazzemi *et al.* HESSLE-FREE: Heterogeneous Systems Leveraging Fuzzy Control for Runtime Resource Management. *ACM TECS*, 2019.

[35] L. Bathen *et al.* VaMV: Variability-aware Memory Virtualization. In *Proc. DATE*, 2012.

[36] L. Bathen *et al.* ViPZonE: OS-Level Memory Variability-Driven Physical Address Zoning for Energy Savings. In *Proc. CODES+ISSS*, 2012.

[37] L. Huang *et al.* Deep reinforcement learning-based joint task offloading and bandwidth allocation for multi-user mobile edge computing. *Digital Communications and Networks*, 2019.

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

[39] M. Hashemi *et al.* Learning Memory Access Patterns. *arXiv:1803.02329 [cs.LG]*, 2018.

[40] M. Shoushtari *et al.* Exploiting Partially-Forgetful Memories for Approximate Computing. *IEEE ESL*, 2015.

[41] M. Shoushtari *et al.* Special session: quality-configurable memory hierarchy through approximation. In *Proc. CASES*, 2017.

[42] N. Dutt *et al.* Variability-aware memory management for nanoscale computing. In *Proc. ASP-DAC*, 2013.

[43] N. Dutt *et al.* Multi-Layer Memory Resiliency. In *Proc. DAC*, 2014.

[44] O. Navarro *et al.* A Machine Learning Methodology for Cache Memory Design Based on Dynamic Instructions. *ACM TECS*, 2020.

[45] R. G. Kim *et al.* Machine Learning and Manycore Systems Design: A Serendipitous Symbiosis. *IEEE Computer*, 2018.

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

[47] R. Yarmand *et al.* DART: A Framework for Determining Approximation Levels in an Approximable Memory Hierarchy. *IEEE TVLSI*, 2019.

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

[49] S. Kounev *et al. Self-Aware Computing Systems*. Springer, 2017.

[50] T. Doudali *et al.* Kleio: A Hybrid Memory Page Scheduler with Machine Intelligence. In *Proc. HPDC*, 2019.
[51] Yu Du *et al.* A multi-agent hybrid cognitive architecture with self-awareness for homecare robot. In *Proc. ICCSE*, 2014.
[52] D. You and K. S. Chung. Dynamic voltage and frequency scaling framework for low-power embedded GPUs. *IET Electronics Letters*, 2012.