Cohmeleon: Learning-Based Orchestration of Accelerator Coherence in Heterogeneous SoCs

Joseph Zuckerman
jzuck@cs.columbia.edu
Department of Computer Science,
Columbia University
New York, New York, USA

Davide Giri
davide_giri@cs.columbia.edu
Department of Computer Science,
Columbia University
New York, New York, USA

Jihye Kwon
jihyekwon@cs.columbia.edu
Department of Computer Science,
Columbia University
New York, New York, USA

Paolo Mantovani*
paolo@cs.columbia.edu
Department of Computer Science,
Columbia University
New York, New York, USA

Luca P. Carloni
luca@cs.columbia.edu
Department of Computer Science,
Columbia University
New York, New York, USA

ABSTRACT

One of the most critical aspects of integrating loosely-coupled accelerators in heterogeneous SoC architectures is orchestrating their interactions with the memory hierarchy, especially in terms of navigating the various cache-coherence options: from accelerators accessing off-chip memory directly, bypassing the cache hierarchy, to accelerators having their own private cache. By running real-size applications on FPGA-based prototypes of many-accelerator multi-core SoCs, we show that the best cache-coherence mode for a given accelerator varies at runtime, depending on the accelerator's characteristics, the workload size, and the overall SoC status.

Cohmeleon applies reinforcement learning to select the best coherence mode for each accelerator dynamically at runtime, as opposed to statically at design time. It makes these selections adaptively, by continuously observing the system and measuring its performance. Cohmeleon is accelerator-agnostic, architecture-independent, and it requires minimal hardware support. Cohmeleon is also transparent to application programmers and has a negligible software overhead. FPGA-based experiments show that our runtime approach offers, on average, a 38% speedup with a 66% reduction of off-chip memory accesses compared to state-of-the-art design-time approaches. Moreover, it can match runtime solutions that are manually tuned for the target architecture.

CCS CONCEPTS

• **Computer systems organization** → **System on a chip**; *Reconfigurable computing*; *Heterogeneous (hybrid) systems*; • **Computing methodologies** → **Reinforcement learning**.

KEYWORDS

cache coherence, hardware accelerators, q-learning, system-on-chip

1 INTRODUCTION

Modern computing systems of all kinds increasingly rely on heterogeneous system-on-chip (SoC) architectures, which combine general-purpose processor cores with many domain-specific hardware accelerators [10, 21, 27, 52, 57, 71, 89]. In the case of smartphones, for example, major vendors devote most of their SoC area
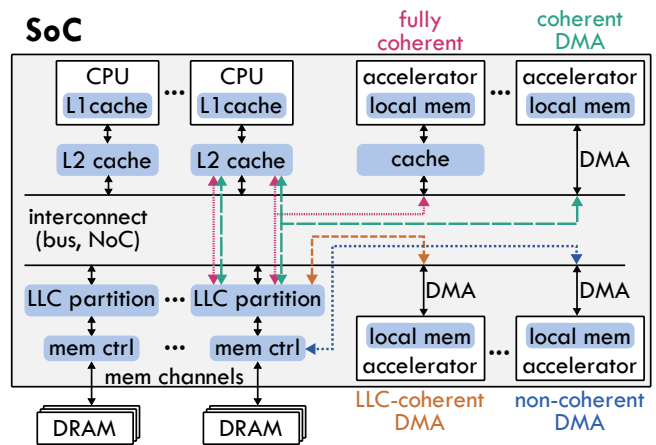

Figure 1: Memory hierarchy of a heterogeneous SoC with partitioned memory space and multiple DRAM controllers. The colored lines show the different ways accelerators can interact with the memory hierarchy.

to a growing number of specialized hardware blocks [40, 70]. A multitude of accelerators have been designed for many different application domains [12, 17, 31, 39, 47, 60, 62, 65, 66, 76, 81, 86, 90].

This work is focused on *fixed-function loosely-coupled* accelerators (LCAs), a very common category of accelerators that includes, for instance, the NVIDIA Deep Learning Accelerator (NVDLA) – part of the NVIDIA Tegra Xavier SoC [27, 60]. By following the loosely-coupled approach, these hardware accelerators are designed independently from the processor cores, lie outside the core on the system interconnect, are invoked through a device driver, execute coarse-grained tasks with no need for fine-grained synchronization, and are shared among multiple cores on an as-needed basis [16, 23]. Fixed-function accelerators are not programmable (i.e. they do not execute instructions), but they can be highly configurable.

For key computational tasks, hardware specialization delivers order-of-magnitude gains in energy-efficient performance compared to software execution on general-purpose processors [41].

---

*Paolo Mantovani is now with Google Research.

These gains are due to a combination of specialization and parallelism, but they also require efficient memory subsystems [25]. Indeed, the integration of LCAs requires managing their interaction with the SoC memory hierarchy, which is typically designed around the processor cores. Figure 1 highlights the memory hierarchy of a generic heterogeneous SoC, where a multi-level cache hierarchy supports the processor cores' operation. Complex SoCs may have a partitioned last-level cache (LLC) as well as multiple DRAM controllers and channels to increase the off-chip bandwidth and better distribute the traffic [4, 54, 64]. LCAs interact directly with the SoC memory hierarchy to load data into their private scratchpad and to store results back to memory [23, 52, 63].

Most LCAs process data in patterns that are very specific to the particular algorithm they implement. Since these patterns are predictable, designers exploit them to specialize the microarchitectures of the accelerator scratchpad and datapath [74]. As a result, while some accelerators have irregular memory access patterns that resemble those of general-purpose cores, many others benefit from a different memory hierarchy, more tailored to their needs [1, 35, 63, 72].

The literature proposes many different modes for the interaction between LCAs and the memory hierarchy, ranging from accelerators accessing off-chip memory directly, bypassing the cache hierarchy, to accelerators having their own private cache [6, 35, 36, 45]. Even though fixed-function LCAs do not require fine-grained synchronization like programmable accelerators (e.g. GPUs) or tightly-coupled accelerators normally do, they may still benefit from accessing on-chip cached data and from using hardware-based coherence provided by the cache hierarchy as an alternative to software-based coherence mechanisms, such as cache flushes. By extending existing classifications, we identify four main *cache-coherence modes* for accelerators.

Then, we show that the best coherence mode varies at runtime, based on the communication properties of the specific accelerator, its invocation parameters (e.g. the size of the task it executes), and the overall system status (e.g. the number of accelerators active in parallel). Thus, selecting an accelerator's cache-coherence mode statically at design-time yields suboptimal performance with respect to a runtime approach. This is especially true for complex SoCs, where many concurrent applications consisting of multiple threads that invoke multiple LCAs in parallel to execute coarse-grained tasks. The on-chip traffic may vary considerably depending on which accelerators are running and what parameters they have been configured with.

For these reasons, we make the case that SoC architectures should support multiple cache-coherence modes for accelerators as well as their runtime selection. Accordingly, we propose COHMELEON, a learning-based approach for selecting the best coherence mode for fixed-function LCAs dynamically at runtime, as opposed to statically at design time. COHMELEON continuously observes the system and measures its performance by means of a few hardware monitors and performance counters that are commonly found in SoC architectures. With this information, COHMELEON trains a reinforcement learning model to select the cache-coherence mode for each fixed-function LCA invocation. Our learning model has the flexibility to target multiple optimization objectives simultaneously, like execution time and off-chip memory accesses. COHMELEON is

fully transparent to application programmers, and it adds a negligible overhead to the accelerator invocations. Our approach does not require any prior knowledge about the accelerators in the SoC and it is easily applicable to different SoC architectures. COHMELEON requires minimal hardware support, provided that the target architecture has support for multiple cache-coherence modes.

To evaluate COHMELEON, we realized a set of complex FPGA-based prototypes of SoC architectures. Each SoC has multiple LLC partitions and multiple DRAM controllers for parallel access to main memory. Using the FPGA as an experimental infrastructure allows us to run real-size applications on top of Linux SMP, without losing accuracy. We developed a set of multi-threaded applications with broad coverage in terms of accelerator parallelism and workload sizes. The applications invoke a variety of open-source accelerators as well as a traffic-generator that reproduces the communication characteristics of a wide range of fixed-function accelerators.

Our experiments show that COHMELEON, on average, gives a speedup of 38%, while reducing the off-chip memory accesses by 66% compared to state-of-the-art design-time solutions. We also show that COHMELEON, without any prior knowledge on the target architecture, can match the performance of a runtime algorithm manually tuned for it.

To build our SoC prototypes we leveraged ESP, an open-source platform for agile design and prototyping of heterogeneous SoCs [9, 20, 54]. We enhanced the ESP cache hierarchy, hardware monitors, and accelerator invocation API.

In summary, we make the following contributions:

• We classify the main accelerator cache-coherence modes for heterogeneous SoCs (Section 2).

• We make the case for the runtime selection of the cache-coherence mode of each accelerator (Section 3).

• We propose and evaluate COHMELEON, a learning-based approach that transparently selects at runtime the best cache-coherence mode for each accelerator, without any prior knowledge of the target architecture (Sections 4, 6).

• The implementation of COHMELEON, its FPGA-based experimental infrastructure (Section 5), and our enhancements to ESP are released as part of the open-source ESP project[20].

2 COHERENCE MODES

The literature on cache coherence for accelerators has proposed several ways of maintaining coherence in heterogeneous systems. These solutions range from managing the coherence fully in hardware to managing it fully in software, or with a hybrid of the two methods [43, 48, 77].

By extending existing classifications in literature [7, 23, 35, 36], we identify four main types of cache coherence for accelerators from a system perspective. We refer to these as *accelerator cache-coherence modes*, which are defined independently from the specific cache-coherence protocol implemented by a given cache hierarchy. All modes *always* keep data coherent, but they do so with different combinations of software-based and hardware-based coherence. The modes naming defines the degree of hardware coherence (non-coherent, LLC-coherent, coherent), and at what level the accelerator accesses the memory hierarchy: access to a private cache (cache access) or direct memory access on the system interconnect (DMA).

	non-coh DMA	LLC-coh DMA	coh DMA	fully-coh
Chen et al. [16]	✓			
Cota et al. [23]	✓	✓		
Fusion [45]			✓	✓
gem5-aladdin [6, 22, 72]	✓	✓		✓
Spandex [1]				✓
ESP [35, 36]	✓	✓		✓
NVDLA [60]	✓			
Buffets [63]	✓			
Kurth et al. [46]	✓			
Cavalcante et al. [13]			✓	
BiC [29]	✓			
Cohesion [44]		✓		
ARM ACE/ACE-Lite [3]	✓		✓	✓
Xilinx Zynq [56, 59, 68]	✓		✓	
Power7+ [8]			✓	
Wirespeed [30]			✓	
Arteris Ncore [11]			✓	✓
CAPI [79]			✓	
OpenCAPI [61]			✓	
CCIX [14, 15]			✓	✓
Gen-Z [32]	✓			
CXL [24]			✓	✓

Table 1: Accelerator coherence modes in literature.

Figure 1 depicts the accelerator interaction with the memory hierarchy for each cache-coherence mode, while Table 1 lists the literature where these modes appear. The literature, including industry, has clearly not converged on a single coherence mode for LCAs. Each mode brings different benefits that can make it optimal depending on the situation.

Non-Coherent DMA. The accelerator does not have a private cache, and its memory requests bypass the cache hierarchy and access main memory directly. In this approach, coherence is implicitly managed by software. If the accelerator data is allocated in a cacheable memory region, a flush of the caches is required before invoking the accelerator to make sure main memory contains the updated version of the data. Some solutions allocate the accelerator data in virtual memory for an efficient and transparent data sharing between processors and accelerators (e.g. gem5-Aladdin [72]). Others reserve a contiguous physical memory region for the accelerator data to avoid dealing with virtual address translation for the accelerator (e.g. NVDLA [60]).

LLC-Coherent DMA. The accelerator does not have a private cache, and its memory requests are sent directly to the LLC. The accelerator is kept coherent with the LLC, but not with the private caches of the processor cores. Therefore, only the private caches must be flushed before the accelerator execution. Since the accelerator does not have a private cache, it normally sends memory requests that have no notion of coherence, as, for example, when using the AMBA AXI interface [3]. Then, the system around the accelerator enforces the LLC-coherence. For this reason, enabling the LLC-coherent mode requires a bridge to map the non-coherent accelerator requests to the system's cache protocol. LLC-coherent DMA has been demonstrated in literature on top of a MESI cache hierarchy [6, 35, 36, 75].

Coherent DMA. Similarly to LLC-coherent DMA, the accelerator does not have a private cache, and its memory requests are sent directly to the LLC. However, in this case, the cache hierarchy maintains full hardware coherence. Therefore, no cache flush is needed, and the LLC recalls or invalidates data in the private caches as needed. Also in this case, the accelerators are normally non-coherent, requiring a bridge to map their messages to the coherent system interconnect. All the entries in Table 1 with support for coherent DMA have their own version of this bridge. For instance, ARM introduced the accelerator coherency port (ACP) in the AXI protocol specification to allow the connection of non-coherent accelerators to the coherent system interconnect [3, 77]. Coherent DMA, also referred to as *I/O coherence* [11, 13, 24], is a common solution for cache-coherent chip-to-chip interconnects, as in the case of connecting an accelerator to a CPU through a PCIe link [14, 61, 79].

Fully-coherent (Coherent Cache Access). A fully-coherent accelerator is equipped with a private cache, to which it sends memory requests, instead of sending them directly on the system interconnect like in the case of DMA. The coherence is handled fully in hardware, exactly as for general-purpose processors. There are various available cache-coherence protocols for the accelerator's private cache. Beside standard protocols like MESI and MOESI [35, 36, 72], other options are Fusion [45], DeNovo [18], or GPU-like coherence protocols [58]. Spandex acknowledges this diversity and introduces a flexible coherence interface that enables systems with heterogeneous cache-coherence protocols [1].

3 MOTIVATION

Accelerators are heterogeneous by nature and this diversity is reflected in their communication properties [23, 52]. They manifest a wide range of memory access patterns, from long streaming bursts to single-word irregular accesses. While an accelerator may read the same input data multiple times, another may store intermediate results or write the output in place. Some are highly computation-bound, requiring a very low communication bandwidth. Others are communication-bound and benefit from an increased memory bandwidth. Furthermore, the same accelerator may show a wide range of communication properties at runtime based on how it is configured when invoked; some accelerators may even contain multiple engines, each optionally selected at runtime. Another important factor is the input size. If the accelerator's data is small enough to fit in its local memory, it only needs to be loaded once from the memory hierarchy; otherwise, many memory transactions are needed. These varieties of communication properties and runtime variability contribute to the complexity of optimizing the interactions between accelerators and the memory hierarchy.

As mentioned in Section 1, our work is focused on fixed-function loosely-coupled accelerators that execute coarse-grain tasks. Normally, the synchronization between processors and accelerators and across multiple accelerators does not occur at fine granularity (e.g. a cache line), but rather at the level of an entire task, which corresponds to the invocation of a particular accelerator. Even without fine-grained synchronization, the cache hierarchy can still be useful. The primary benefit is accessing an on-chip copy of the data (in case of a cache hit), as opposed to directly accessing off-chip memory. The caches can be especially useful when accelerators
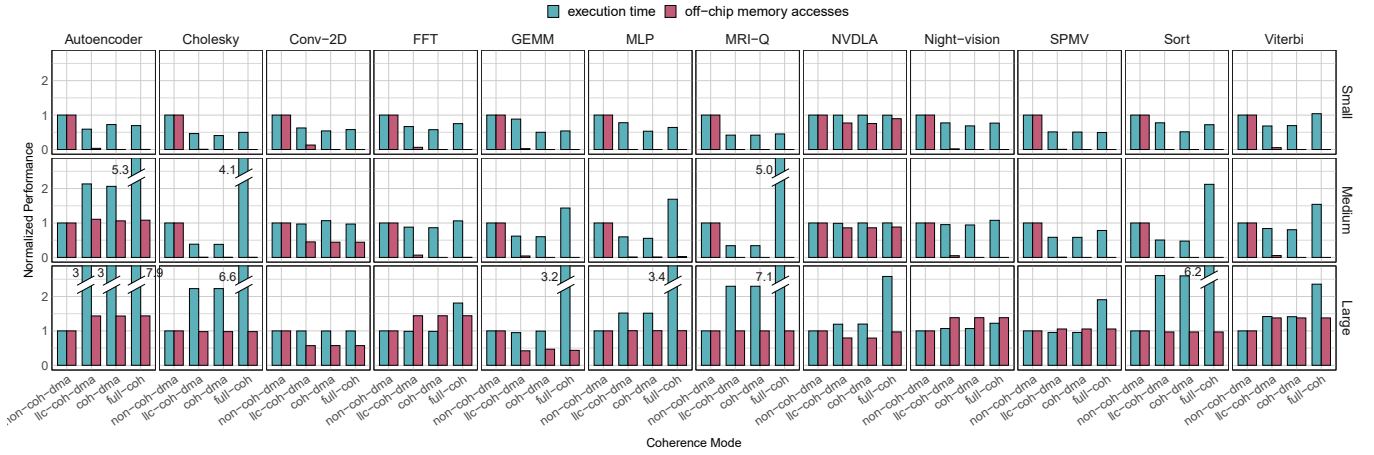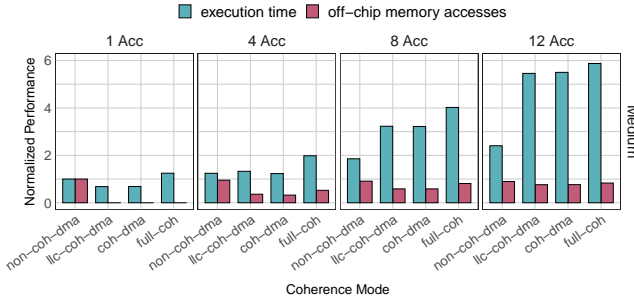

Figure 3: Accelerators running in parallel.

to being at least three times faster than all the other modes while incurring 30% less memory accesses (*Large* workload). While a few accelerators behave similarly, others, like GEMM, never have the non-coherent mode as the best option.

The modes that do not require the device driver to flush the caches have a smaller invocation overhead and may benefit from cached data. Hence, they tend to perform best for smaller workload sizes (e.g. MLP Small/Medium). The non-coherent mode becomes more effective for larger workloads, which do not fit in the caches and make the invocation overhead negligible. For large workloads, in the case of some accelerators, the non-coherent DMA has fewer off-chip accesses than the other modes (e.g. FFT Large). In fact, for large workload sizes, the modes that use the caches can incur thrashing, i.e. the miss rate is very high and causes a large amount of cache evictions. As the workload size increases beyond 4MB, this phenomenon is bound to increase in favor of the non-coherent mode, which bypasses the caches.

The non-coherent DMA mode always accesses off-chip data directly, but when the accelerator requests long bursts it can sustain higher throughput than the modes using the caches. Hence, it can have better performance even with more memory accesses (e.g. Cholesky Large).

Parallel Accelerator Execution. Modern SoCs frequently contain up to tens of hardware accelerators, and applications executing in parallel can cause multiple hardware accelerators to run simultaneously. We thus study the performance degradation due to concurrent execution of multiple accelerators and report the results of our study in Figure 3. We chose medium-sized (256KB) workloads for each of accelerator types and ran multiple experiments with 1, 4, 8 and 12 accelerators executing concurrently. We built an SoC with 12 accelerators, with 3 instances of FFT, Night-vision, Sort, and SPMV. Each accelerator is invoked multiple times in a row from a separate software thread. For each set of experiments, we averaged the performance of each accelerator over multiple executions and we normalized it against the non-coherent DMA results for the single accelerator execution. Then, we averaged the normalized performance of all four accelerators to produce the results in Figure 3.

As the number of active accelerators increases, the non-coherent mode appears to suffer the least, recording an execution time loss of up to 2.4× with 12 accelerators, whereas the value of off-chip accesses stays constant. The other modes, which could benefit from cached data in the case of standalone accelerators, see an increase of memory accesses due to contention on the caches. Coherent DMA suffers the worst slowdown: 8× higher execution time when

12 accelerators run concurrently compared to the single accelerator case.

The Case for a Learning-Based Approach. The results presented in this section and summarized in Figure 2 and Figure 3 highlight that no fixed coherence solution is close to optimal for heterogeneous SoCs integrating multiple types of fixed-function LCAs. Furthermore, these results suggest that deriving an analytic solution in the form of a heuristic would require a tremendous effort to search a huge design space that consists of a large number of variables to consider. For instance, the penalty of each particular type of data transfer may differ depending on the type of coherent interconnect of the system; the number of available accelerators may vary; the number of concurrent accelerator invocations can change over time; the size of the workload might not be fixed; updates to the system software could impact scheduling priorities.

Our motivating results, collected with the implementation of the 4 coherence modes provided by ESP, are aligned with the findings of the state-of-the-art literature presented in Table 1, which are based on a variety of different coherence modes implementations.

These considerations led us to the development of COHMELEON: the first learning-based approach to the orchestration of accelerator coherence in heterogeneous SoCs. As we describe the implementation of COHMELEON in Section 4 and evaluate its advantages in Section 6, we show how the learning approach can eliminate the need for a human-in-the-loop and avoid a lengthy design-space exploration that would have to be repeated whenever some of the state-space variables change in the system. Furthermore, our learning approach can optimize concurrently a multi-objective reward function, which leads to improved performance with fewer accesses to external memory.

4 COHMELEON

We now present COHMELEON, our solution for runtime orchestration of accelerator coherence. COHMELEON's ability to adaptively reconfigure the memory hierarchy of a heterogeneous SoC is built upon features that span various levels of the hardware-software stack. We propose a general framework for runtime coherence selection, highlighting the necessary hardware and software requirements. Then, we show how reinforcement learning can be applied to this framework to enable online and continuous learning. Finally, we discuss the implementation of our approach in a particular SoC platform.

4.1 Runtime Reconfiguration

Our general framework is divided into four main phases. During each accelerator invocation we first *sense* the conditions of the SoC using a minimal set of status variables. This information is used to *decide* which coherence mode to apply to an accelerator invocation following a given policy. This decision is immediately *actuated*, and then its performance is *evaluated* using hardware monitors when the accelerator execution completes.

(1) Sense: We aim at achieving system introspection with respect to accelerator performance under cache coherence modes by making the system capable of continuously observing its own operation. We propose a lightweight software layer to keep track of important system variables. Since tracking the complete state of an SoC is intractable, we identified the following set of parameters in

order to capture a compact "snapshot" of the system while keeping the overhead of doing so small:

• Number of active accelerators.
• Coherence mode of each active accelerator.
• Memory footprint (workload size) of active accelerators.

In Section 3, we saw that the number of active accelerators and the workload size affected performance. Furthermore, the coherence and memory footprint of other active accelerators are relevant because the active coherence choices can cause resource contention, and the total amount of active data impacts the efficacy of caches.

(2) Decide: The coherence decision follows a *policy*, a set of rules that dictate the runtime decision given the current state. We consider various policies, including those generated by our reinforcement learning module and several baseline policies.

(3) Actuate: We choose to configure an accelerator's coherence once per invocation, since this is a natural point of synchronization, and fixed-function accelerators tend to have uniform behavior throughout an invocation. In addition, selecting the coherence mode at invocation time incurs no overhead, because it happens concurrently to the application-specific configuration of the accelerator.

How the actuation of coherence mode is performed depends on the underlying SoC implementation, but the important prerequisite is that the coherence mode can be changed at runtime. Changing the coherence mode requires hardware support for multiple policies, as opposed to a static choice of coherence mode for each accelerator or the entire SoC. COHMELEON does not necessarily require support for all four coherence modes; it makes the selection based on the options that are available. In general, support for particular coherence modes can be decoupled from the accelerator design by using mechanisms provided by the surrounding SoC. Hence, designers need not worry about coherence when creating accelerators for a system that utilizes COHMELEON.

(4) Evaluate: To evaluate the quality of an accelerator's invocation, we identify four metrics to observe:

• Total execution time, including any overhead due to accelerator invocation (i.e device driver, cache flushes).
• Off-chip memory accesses during the invocation.
• Total cycles in which the accelerator is actively executing.
• Total cycles in which the accelerator is communicating (issuing a request or awaiting a response) with memory.

Total execution time and off-chip memory accesses are obvious performance attributes. We choose the other two metrics to account for compute-bound accelerators. For such accelerators, the total execution time may not change even if the memory system performs better, but the ratio of communication cycles to total cycles would get smaller.

While execution time can be measured in software, the remaining values must be measured in hardware. Hence, we propose the addition of a lightweight hardware monitoring system that can be integrated easily into any SoC. Our monitoring system continuously exposes these values to software, thus allowing for the evaluation of an accelerator's performance to inform an intelligent selection of coherence modes.

4.2 Reinforcement Learning Module

In reinforcement learning (RL), an autonomous agent learns behaviors through trial-and-error interactions with the environment [42,

Figure 4: Overview of the proposed learning module.

80]. At each step, the agent perceives the *state* (sense), chooses an *action* (decide), takes the action (actuate), and receives a *reward* at the beginning of the next step (evaluate). The objective is to find an optimal *policy* that determines which action to choose at each state so that the expected reward is maximized.

For the runtime reconfiguration problem of COHMELEON, we propose a variant of Q-learning, a widely adopted RL approach that does not require any model of the environment [84, 85]. This approach has a number of advantages for the target problem. First, it enables an automatic discovery of a coherence-selection policy during regular SoC operation. Second, it requires neither human expertise of the target SoC architecture nor offline-training data. Third, RL allows continuous updating of the coherence-selection policy. As an SoC runs new workloads and encounters new system states, a RL module can update its policy to improve the performance. Finally, we can optimize the system with respect to multiple objectives by formulating the learning *reward* accordingly.

A Q-learning agent maintains a *Q-table* that stores, for each state-action pair, a *Q-value* that represents the expected reward of taking that action from that state. The agent can use many strategies to select an action from each state, including an *epsilon-greedy* approach. At each step, this approach selects either a random action, with probability ϵ, or the action with the highest Q-value based on the current state. This encourages both *exploration* of the action space and *exploitation* of the knowledge gained from prior trials. After each action is taken, the reward is evaluated at the next step, and the Q-value corresponding to that state-action pair is updated from the previous value with a *learning rate* α.

State Attribute	Description	Values
Fully coh acc	Total number of active fully coherent accelerators	0, 1, 2+
Non coh acc per tile	Avg no. of non-coh accelerators communicating with each memory partition needed by the target accelerator invocation	0, 1, 2+
To LLC per tile	Avg no. of accelerators accessing each LLC partition needed by the target accelerator invocation	0, 1, 2+
Tile footprint	Avg utilization of each partition of the cache hierarchy needed by the target accelerator invocation	≤L2, ≤LLC Slice, > LLC Slice
Acc footprint	Memory footprint of the target accelerator invocation	≤L2, ≤LLC Slice, > LLC Slice

Table 3: State space S: a state $s \in S$ is a 5-tuple of the following attributes.

Model definition. Figure 4 shows an overview of the proposed learning module for coherence selection. The problem is modeled with the following states, actions, and rewards.

(1) States: Based on the results in the motivation section and on prior work [7, 37], we define the state space S with the following 5 attributes: *Fully-Coherent-Acc, Non-Coh-Acc-per-Tile, To-LLC-per-Tile, Tile-Footprint,* and *Acc-Footprint*. Each attribute can have one of three possible values, as described in Table 3. A state $s \in S$ is a 5-tuple of these attributes, and hence, $|S| = 3^5 = 243$. The value of the state s for any accelerator invocation is used to index the Q-table of Figure 4.

(2) Actions: The action set A contains the 4 coherence modes: *non-coherent, LLC-coherent-DMA, coherent-DMA,* and *fully-coherent*. Thus, the coherence Q-table contains $|S| \times |A| = 243 \times 4 = 972$ entries. The action step sets the coherence configuration register of the given accelerator tile.

(3) Rewards: As shown in Figure 4, when an accelerator is to be invoked, the RL module senses the state s and looks up the Q-table to determine the best action (or, randomly chooses an action) a. Then, the accelerator is invoked with the selected action. After it completes execution, its reward is computed based on the performance of both the accelerator invocation and the memory system during the invocation. To define the reward function, we define three measurements of an invocation. For the i-th invocation of an accelerator k:

• $exec(k, i)$ is the *scaled execution time* - the total execution time divided by the footprint of the invocation.

• $comm(k, i)$ is the *communication ratio* - the number of accelerator communication cycles divided by the total number of execution cycles.

• $mem(k, i)$ is the *scaled off-chip memory access count* during the invocation - the total number of memory accesses divided by the footprint of the computation.

Then, we define the three component functions:

$$R_{exec}(k, i) = \frac{\min_{j \leq i} [exec(k, j)]}{exec(k, i)}$$

$$R_{comm}(k, i) = \frac{\min_{j \leq i} [comm(k, j)]}{comm(k, i)}$$

$$R_{mem}(k, i) = 1 - \frac{mem(k, i) - \min_{j \leq i} [mem(k, j)]}{\max_{j \leq i} [mem(k, j)] - \min_{j \leq i} [mem(k, j)]}$$

For the *exec* component, we store the best scaled execution time for each accelerator thus far. We can see that smaller execution times maximizes the ratio in R_{exec}. As previously mentioned, we utilize the *comm* part to account for compute-bound accelerators. If

the memory system performs better, this ratio will be lower. Again, we see that R_{comm} is maximized for smaller values of the communication ratio. R_{mem} takes a different form because accelerator invocations may cause zero off-chip memory accesses. Using both the maximum and minimum scaled access counts, the presented equation maps higher memory-access counts near zero and lower counts near one. Finally, the reward $R(s, a; k, i)$ for the i-th invocation of accelerator k with action a from state s is

$$R(s, a; k, i) = x \cdot R_{exec}(k, i) + y \cdot R_{comm}(k, i) + z \cdot R_{mem}(k, i)$$

where x, y, and z are constant weights that can be tuned.

Training. At the beginning of training, all entries in the Q-table are set to zero. The table is updated in the following manner. Whenever an accelerator is invoked, the state is recorded. After the accelerator completes execution, the reward is computed and is used to update the Q-table for the recorded state and chosen action, as follows:

$$Q(s, a) \leftarrow (1 - \alpha) \times Q(s, a) + \alpha \times R(s, a)$$

where $R(s, a)$ is the reward that results from taking action a out of state s, and α is the learning rate.

4.3 Implementation

As shown in Table 1, there is already a number of architectures in literature that support multiple coherence modes for accelerators (e.g. ARM ACE/ACE-Lite, gem5-aladdin) and are amenable for hosting COHMELEON. We implemented and evaluated COHMELEON as part of a comprehensive FPGA-based infrastructure that we developed to study accelerator coherence-mode selections in many-accelerator SoCs. Our implementation is based on ESP, an open-source platform for agile SoC design [54]. We chose ESP because it supports the runtime selection of three of the coherence modes we identified in Section 2 and has a suite of fixed-function LCAs. Furthermore, ESP allows for rapid prototyping of many-accelerator SoCs on FPGA and is open source.

The ESP SoC platform automates the integration of processor cores and accelerators into a grid of tiles connected by a 2D-mesh multi-plane NoC. There are four main types of tiles: processor tile, accelerator tile, memory tile for the communication with main memory, and auxiliary tile for peripherals (e.g. UART or Ethernet) or system utilities (e.g. the interrupt controller). In this work the processor tile contains the SPARC 32-bit LEON3 processor core [19] from Cobham Gaisler. All components are connected by ESP's network-on-chip, which counts 6 32-bit physical planes, with one cycle latency between neighboring routers. Each memory tile has a link to main memory with bandwidth of 32 bits per cycle. The

ESP cache hierarchy matches the one represented in Figure 1 and is distributed across processor tiles, which include an L2 private cache, and memory tiles. Each memory tile hosts a partition of the LLC, a DRAM controller, and a dedicated channel to the corresponding partition of off-chip main memory. In addition, the accelerator tile can optionally integrate a private cache to enable the fully-coherent mode. With the exception of the optional private cache, ESP's support for the runtime selection from multiple coherence modes adds negligible area to the SoC. Furthermore, coherence is handled by the "socket" surrounding the accelerator, so the accelerators are designed with no notion of coherence.

(1) Sense: We implemented our introspective SoC status tracking in the ESP accelerator invocation API, which is a set of user-space functions for invoking loosely-coupled accelerators from software applications [54]. We defined new global structures containing the number of active accelerators, their footprints, and the chosen coherence mode. When an accelerator is invoked – and when it returns control to software – this structure is updated accordingly.

(2) Decide: The decision-making for a runtime coherence *policy* is also implemented in the back-end of the ESP API. Here, we outline several possible policies to compare to our RL approach. The *Random* policy randomly chooses a coherence mode for each accelerator invocation at runtime. The *Fixed* policy statically selects the same coherence mode for each accelerator invocation, mimicking a design-time decision. This represents nearly all previous work. The *Fixed* policy can either be *homogeneous*, where every accelerator operates with the same coherence mode, or *heterogeneous*, where the coherence mode can be chosen independently for each accelerator. In the heterogeneous case, we choose a coherence mode based on profiling the accelerator's performance in each mode while sweeping the footprint of the workload on different invocations. The *fixed-heterogeneous* policy is used as a comparison to prior design-time solutions that select a fixed coherence mode for each accelerator [6, 7].

Next, we present an introspective, manually-tuned algorithm, that chooses the coherence mode based on the status of the system. We designed this algorithm to minimize the runtime for accelerators in an ESP SoC. We used data from tens of thousands of accelerator invocations, combined with knowledge of ESP's implementation of the coherence modes, to produce a highly optimized policy, shown as Algorithm 1. The *manual* algorithm is used as a comparison to prior approaches that use static heuristics to select a coherence mode at runtime [37]. If applied to SoC architectures other than our target (i.e. ESP) this algorithm would need manual tuning and possibly some major adjustments. For instance, an SoC that uses a more-optimized coherence protocol for accelerators than MESI could benefit from an increased reliance on the fully-coherent mode, but this manual algorithm would not select it more frequently.

In contrast, the RL module we presented in Section 4.2 generates its own coherence-selection policy. In learning epochs where the agent "chooses" to explore the state space, it follows a random policy. When attempting to maximize the reward, however, the model selects the coherence mode with the highest Q-value from the current state. The Q-table thus dictates the coherence decision given the current state, but unlike other policies, it is *adaptive* and can change with the addition of new information.

Algorithm 1 Manually-tuned coherence mode selection.

---

if footprint ≤ EXTRA_SMALL_THRESHOLD **then**
    coh ← FULLY-COH
else if footprint ≤ CACHE_L2_SIZE **then**
    if active_coh_dma > active_fully_coh **then**
        coh ← FULLY-COH
    else
        coh ← COH-DMA
    end if
else if footprint + active_footprint > CACHE_LLC_SIZE **then**
    coh ← NON-COH
else
    if active_non_coh ≥ 2 **then**
        coh ← LLC-COH-DMA
    else
        coh ← COH-DMA
    end if
end if

---

(3) Actuate: Each coherence decision is actuated by the accelerator device driver's writing to a memory-mapped register in the accelerator tile that controls the mechanism by which the accelerator communicates with memory. Because accelerators share cacheable memory with processors in ESP, the LLC-coherent-DMA and non-coherent-DMA modes require software-managed cache flushes, as described in Section 2, before the accelerator can begin executing. It is possible to handle the flushes in a way that is completely transparent to the programmer, which is the case in ESP. In fact, COHMELEON actuates the coherence mode with a single line of code. Using a custom MESI directory-based and NoC-based protocol, ESP supports runtime selection for all of the coherence policies, excluding coherent-DMA. We extended the protocol to support coherent-DMA by issuing recalls from the LLC to a private cache when the private cache holds data that is the target of a DMA request. By adding support for the coherent-DMA model, we did not introduce any area overhead in the accelerator tiles.

(4) Evaluate: We developed a new hardware monitoring system to measure our chosen metrics of accelerator performance. In each tile, we added a set of memory-mapped registers to a pre-existing APB interface. Each register is connected to logic that increments its value when the corresponding phenomenon occurs. These monitors are distributed across all tiles but are accessible from software through a single contiguous region of the I/O address space using standard Linux system calls, such as `mmap` and `ioremap`.

We access the hardware monitors from the accelerator device driver. The accelerator cycle counters, which are reset at the beginning of its execution, are read at the end of the invocation. Memory access counters are read before and after each invocation to determine the change, potentially accounting for overflow. We note that when multiple accelerators are communicating with a memory controller, we cannot know the exact number of memory accesses caused by each accelerator without additional hardware support to track which accelerator's transactions cause misses or evictions in the LLC. Instead, in order to minimize COHMELEON's required hardware support, we chose to approximate the number of off-chip memory accesses for a particular accelerator. Our approximation relies on the assumption that larger workloads will, in general, trigger more memory accesses. This works particularly well for the non-coherent mode, since all data must be brought in from off chip, and the other modes when running workloads that do not fit in the caches. Furthermore, more transactions likely mean more cache

| | SoCs w/ Traffic Generator | | | | Case Study SoCs | | |
	SoC0	SoC1	SoC2	SoC3	SoC4	SoC5	SoC6
Accelerators	12	7	9	16	11	8	9
NoC size	5x5	4x4	4x4	5x5	5x4	4x4	4x4
CPUs	4	2	4	4	2	1	1
DDRs	4	4	2	4	4	4	2
LLC part.	512kB	256kB	512kB	256kB	256kB	256kB	256kB
Total LLC	2MB	1MB	1MB	1MB	1MB	1MB	512kB
L2 cache	64kB	32kB	32kB	64kB	32kB	32kB	32kB

Table 4: Parameters of the evaluation SoCs.

misses and evictions. We thus define the memory accesses caused
by accelerator k at a memory controller m as:

$$ddr(k, m) = ddr\_total(m) \times \frac{footprint(k, m)}{\sum_{acc \in A} footprint(acc, m)}$$

where A is the set of active accelerators, $ddr\_total(m)$ is the ob-
served change in off-chip memory accesses at memory controller
m, and $footprint(acc, m)$ is the memory footprint of accelerator acc
allocated to memory controller m. By this definition, each accelera-
tor's share of the total memory accesses will be proportional to its
active memory footprint. Better approximations are likely possible
with some knowledge of accelerator characteristics, but we chose
this approximation to keep COHMELEON accelerator-agnostic.

5 EVALUATION STRATEGY

To evaluate COHMELEON, we realized multiple SoC prototypes and
test applications that serve as full-system case studies.

Traffic-Generator. From the viewpoint of the rest of the SoC,
an accelerator can be characterized by its patterns of communica-
tion with the memory hierarchy. After analyzing many accelerators,
we derived a list of basic accelerator properties that influence these
patterns. Then, we designed a traffic-generator that is configurable
with respect to these properties, allowing us to efficiently study
the diverse set of communication patterns that are expressed by
accelerators. The parameters of the traffic-generator are access
pattern (streaming, strided, or irregular), DMA burst length, com-
pute duration, data reuse factor, read-to-write ratio, stride length
(for strided patterns), access fraction (for irregular patterns), and
in-place storage.

SoCs. We implemented FPGA-based prototypes of four different
SoCs utilizing the traffic-generator, and three Case Study SoCs
utilizing the same set of accelerators from Section 3. For these
designs, Table 4 reports the key parameters, which we vary in order
to validate that our contributions can generalize to different SoC
configurations. All accelerators are equipped with a private cache,
except five of the accelerators of SoC3, which could not be included
due to resource constraints of the chosen FPGA.

SoC4 has one instance of each of the 11 accelerators of Table 2,
thus representing the modern SoC trend of invoking many het-
erogeneous accelerators while running multiple applications in
parallel.

SoC5 targets the domain of collaborative autonomous vehicles
by embedding key accelerators [73]. Two FFT and two Viterbi
accelerators support encoding and decoding for vehicle-to-vehicle
(V2V) communication. Two 2D Convolution (Conv-2D) and two
Matrix Multiplication (GEMM) accelerators support inference for
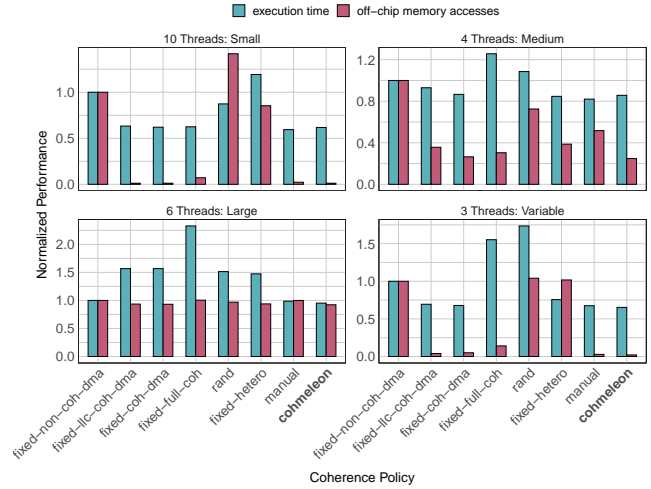convolutional neural networks (CNNs) for object recognition and

**Figure 5: Performance of four phases of the evaluation ap-
plication, varying threads count and workload sizes.**

labeling. For this purpose the Conv-2D and GEMM accelerators
embed bias addition, pooling and activation capabilities.

SoC6 targets the computer vision domain by providing three
instances of an image classification pipeline composed of three
accelerators [34]: night-vision, which has a 4-stage pipeline for
processing dark images, autoencoder for denoising images, and
MLP for image classification.

Applications. We developed a multithreaded application to in-
voke the traffic-generator in many different ways. The application
consists of a set of *phases* that are each meant to represent a real
application. A phase consists of a number of threads. A thread con-
sists of a single dataset and a "chain" of accelerators, configured
with different parameters, that operate serially on that dataset -
the output of one accelerator is the input to the next. Optionally,
each thread can loop over the accelerator invocations. The appli-
cation phases and parameters are specified using a configuration
file. Through our experiments, we ensured that the instances of
the application vary in terms of the number of accelerator threads
running in parallel, the workload sizes in use, and the configuration
parameters of each traffic-generator.

Furthermore, we developed multithreaded evaluation applica-
tions specific to each Case Study SoC, with a structure similar to the
applications for the traffic-generators SoCs: organized in phases and
designed to stress multiple operating modes and workload sizes for
each accelerator. For each SoC, the application invokes pipelines of
accelerators as appropriate for the target domain. For example, the
night-vision, autoencoder and MLP work in a chain to undarken,
denoise, and then classify the input images. On SoCs containing
multiple copies of the same accelerators, our evaluation applications
can parallelize the workload to improve the throughput. For these
applications, we define the following characterization of workload
sizes: *Small (S)* when smaller than accelerator's L2 cache; *Medium
(M)* if smaller than one partition of the LLC; *Large (L)* when smaller
than the aggregate LLC; *Extra-Large (XL)* if larger than the LLC.

We use ESP's solution for the allocation of accelerator data [53].
Data is allocated in big Linux pages, so that it results in a relatively
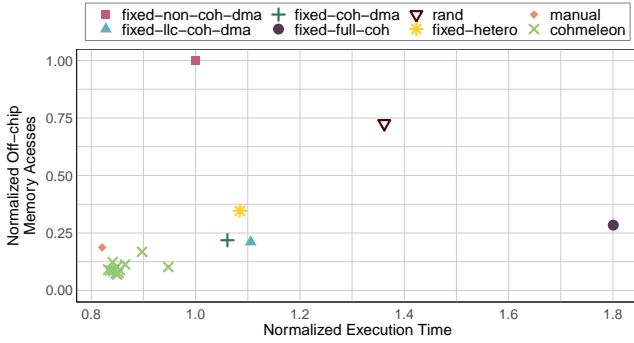small page table that can normally fit in the accelerator tile's TLB.

Figure 6: The impact of the reward function on SoC0.

Figure 7: Breakdown of coherence decisions.

The TLB is loaded at the beginning of the accelerator invocation and it provides a miss-free address translation. This solution allows for true data sharing between CPUs and accelerators with no need for data copies or contiguous physical allocations. The overhead of loading the TLB and address translation is included in all results.

Experimental Setup. We deployed the full SoCs on a Xilinx Ultrascale XCVU440 FPGA. The LEON3 cores (soft-cores) in the SoC run Linux SMP, on top of which we executed the evaluation applications for each of the above SoCs.

COHMELEON learns online at runtime, i.e. there is no offline training. For each SoC we run a randomly configured instance of the evaluation application. The learning coefficients are initialized to $\epsilon = 0.5$ and $\alpha = 0.25$ and decay linearly to 0 over a selected number of iterations. Once the learning model has converged, we disable further updates and evaluate its performance on a different instance of the application.

To compare performance across policies we measured the total execution time and off-chip memory accesses for each phase of the applications, including any overhead of the accelerator invocations, such as cache flushes. Because the phases vary in terms of the number of accelerators working in parallel and workload sizes, we can compare how the different policies perform in many situations.

6 EXPERIMENTAL RESULTS

Phase Analysis. We first present the results of four selected phases of the evaluation application running on SoC0 (Figure 5). These phases differ in the number of threads and workload sizes in order to highlight both the variation of performance in policies and the coverage that our evaluation applications provides. All results are normalized to the values of the *Fixed non-coherent DMA* policy.

As the workload size and number of threads vary, we observe variations in the relative performance of the *Fixed homogeneous* policies, consistent with the motivation results in Section 3. On the other hand, both the manually-tuned algorithm and COHMELEON match or improve upon the best execution time of the other policies in all phases. Meanwhile, the *Fixed heterogeneous* policy is unable to achieve this result due to the variation in workload size and system status. The manually-tuned algorithm achieves similar execution times as COHMELEON in all phases, but COHMELEON achieves that performance with fewer off-chip memory accesses.

Design-Space Exploration of Reward Function. Next, we experiment with different reward functions on SoC0, only varying the values of the three x, y, z coefficients of R_{exec}, R_{comm}, and R_{mem}. We train COHMELEON for 50 iterations of the evaluation
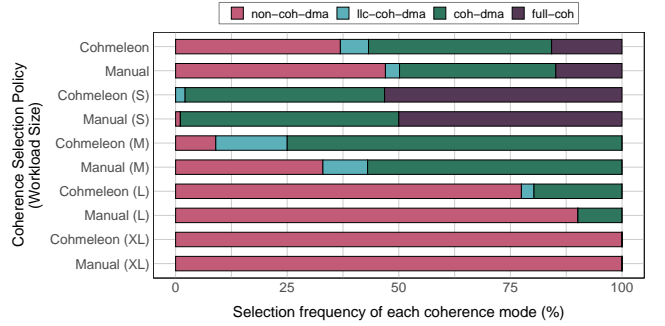
application with each reward function, and then test each model, as well as the other baseline policies, on a different instance of the application. For each policy, we normalize the performance of each phase to the *Fixed non-coherent DMA* policy. In Figure 6, we plot the geometric mean of the normalized execution time and off-chip memory accesses over all phases. First, we highlight the large cluster of COHMELEON data points in the bottom left of the chart. COHMELEON is capable of matching the execution time of the manually-tuned algorithm, while achieving the best value for off-chip memory accesses. COHMELEON's flexibility in optimizing for multiple objectives clearly allows for the discovery of policies that are near-optimal over multiple performance metrics. However, we notice that while COHMELEON generates *Pareto-optimal* data points, the cluster of points does not present much variation. Thus, we cannot trade off an improvement in execution time with a reduction of off-chip memory accesses or vice versa. Indeed, off-chip memory accesses contribute to a relevant part of the execution time of communication-bound accelerators.

We trained 15 different models, and most perform quite similarly. In fact, only two points, which correspond to reward functions that weighed heavily (> 90%) for off-chip memory accesses, have significantly worse performance. The remaining points have extensive variation in the reward function. For instance, two of the Pareto-optimal points use reward functions that give the following percentage weights to execution time, communication ratio, and off-chip memory accesses, respectively: (a) 67.5, 7.5, 25 and (b) 12.5, 12.5, 75. We conclude that, for this particular architecture, near-optimal performance can be achieved with a wide variety of reward functions without significant trial-and-error.

Coherence Breakdown. In Figure 7, we present the breakdown of coherence decisions by COHMELEON and the manually-tuned algorithm. We report both the total breakdown and the breakdown for each category of workload size. Across all invocations, we see a heavy reliance on the *coherent DMA* (coh-dma) and *non-coherent DMA* (non-coh-dma) modes. Broadly, COHMELEON seems to learn a policy that results in a breakdown of coherence modes similar to that of the manually-tuned algorithm. However, across all categories (except XL workloads), COHMELEON relies less upon the *non-coherent DMA* and more upon *coherent DMA* and *LLC-coherent DMA* (llc-coh-dma) than the manual algorithm. The *non-coherent DMA* mode typically results in the highest number of off-chip memory accesses for workloads that fit on-chip. Due to its bi-objective reward, COHMELEON tries to avoid this selection in such a situation.
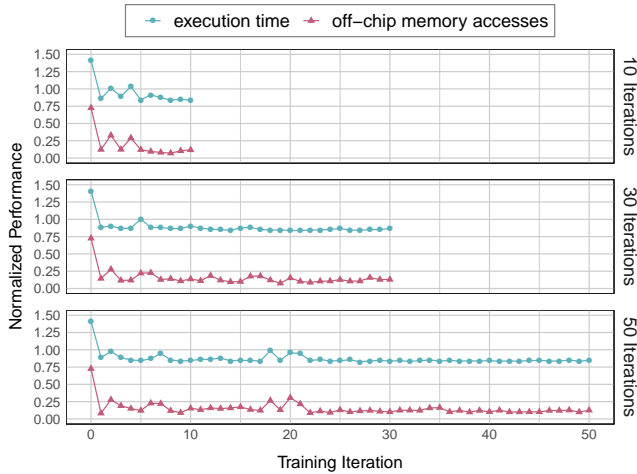
Figure 8: Performance over training iterations.

Training Time. To evaluate the effects of training time on COHMELEON, we run a series of experiments in which we evaluate the performance of the RL model after each training iteration. We alternate the training of COHMELEON on one iteration of an instance of the evaluation application with testing the resulting model on a different instance of the evaluation application. Both instances of the application contain several hundred accelerator invocations and are designed to be as diverse as possible in terms of operating conditions. We repeat this experiment for 10, 30, and 50 total iterations. Each trial initializes ϵ (exploration rate) to 0.5 and α (learning rate) to 0.25. Each value is decayed linearly to 0 over the course of training, thus making the decay rates different for each number of iterations. Figure 8 shows the performance after each iteration, reported as the geometric mean over all phases of the performance normalized to the *Fixed non-coherent DMA* policy. Iteration 0 is the performance of an untrained model, equivalent to the *Random* policy. We see a quick drop-off in execution time and off-chip memory accesses after just one training iteration for all models. A training iteration includes over 300 accelerator invocations, which provide for enough exploration to immediately learn an improved policy. We see some oscillation in the results for the next few iterations, as the models continue to explore the state space with different actions. All models reach approximately the same performance at the end of training. We conclude that ten iterations are sufficient to achieve near-optimal results.

Additional SoCs. We repeat our experiments on eight different SoC configurations, utilizing the SoCs from Table 4 to verify that COHMELEON is effective with different architectural parameters. On each configuration, we use a reward function of 67.5% execution time, 7.5% communication ratio, and 25% off-chip memory accesses and train for 10 iterations of the corresponding application. These results are reported in Figure 9.

First, we reuse the SoC0 layout, but this time we use one set of accelerators with streaming access patterns and another set of accelerators that have irregular access patterns. We observe that the *Fixed non-coherent DMA* has the best execution time among the fixed homogeneous policies for streaming accelerators, whereas for irregular accelerators the *Fixed LLC coherent DMA* and *Fixed coherent DMA* policies achieve better execution time with fewer

off-chip memory accesses. This clearly shows once again that the communication patterns can affect the optimal coherence choice. In both of these configurations, COHMELEON and the manual algorithm achieve the best execution time, while COHMELEON slightly lowers off-chip memory accesses.

We also utilize SoC1, SoC2, and SoC3, each configured with a set of accelerators with mixed properties. We see the same ordering among the fixed homogeneous policies, but with some substantial differences in their relative performance. (i.e. *Fixed fully-coherent* has roughly 2.2× the execution time of *Fixed non-coherent DMA* on SoC1, but only 1.5× on SoC3). The performance of COHMELEON and the manual algorithm remains the best across these SoCs.

Finally, we perform the same experiments on the Case Study SoCs, i.e. SoC4, SoC5, and SoC6. On SoC5 and SoC6, we observe that there is much less variability among the performances of the fixed modes. Because the accompanying applications only invoke accelerators in ways that are appropriate for the corresponding real-world application, there is less diversity in terms of the characteristics of the applications. COHMELEON still achieves optimal or near-optimal performance across these SoCs. In contrast, the manual algorithm has suboptimal performance on SoC5, showing it is not capable of generalizing optimally to all SoCs.

Given these observations, it is clear that these SoC configurations have different communication properties due to their set of accelerators and architectural parameters. Nonetheless, we see that COHMELEON can achieve the minimal or near-minimal value for both execution time and off-chip memory accesses across all experiments. COHMELEON improves as there is more diversity in performance across the coherence modes. This is intuitive, as there are opportunities to improve performance by exploiting an intelligent decision. Across all SoC configurations, COHMELEON gives an average speedup of 38% with a 66% reduction in off-chip memory accesses when compared to the five fixed policies.

Cohmeleon Overhead. We measured the fraction of the total execution time caused by COHMELEON's status tracking, computation, and decision-making, which is included in all prior results. For small (16KB) workloads, the overhead is between 3 and 6% of the total execution time. This value drops as the workload size increases and the accelerators have longer execution times. For large (4MB) workloads, the overhead is safely below 0.1%, a negligible value.

7 RELATED WORK

Comparing Accelerator Coherence Modes. There are only a few studies that compare the cache-coherence options for accelerators, as we did in Section 3. Kumar *et al.* propose a fully-coherent approach based on a timestamp-based coherence protocol [55] and compare it with classic fully-coherent and coherent-DMA solutions [45]. Shao *et al.* investigate the non-coherent and fully-coherent modes [72]. Cota *et al.* evaluate LLC-coherent and non-coherent accelerators [23]. These works focus on the simulation of simple SoCs with a few accelerators. We prototype NoC-based SoCs with up to 16 accelerators using an FPGA-based setup to run complex real-size applications that manage multiple accelerators on top of Linux. The setup of Giri *et al.* is similar to ours, but they did not evaluate all four cache-coherence modes [35, 36].
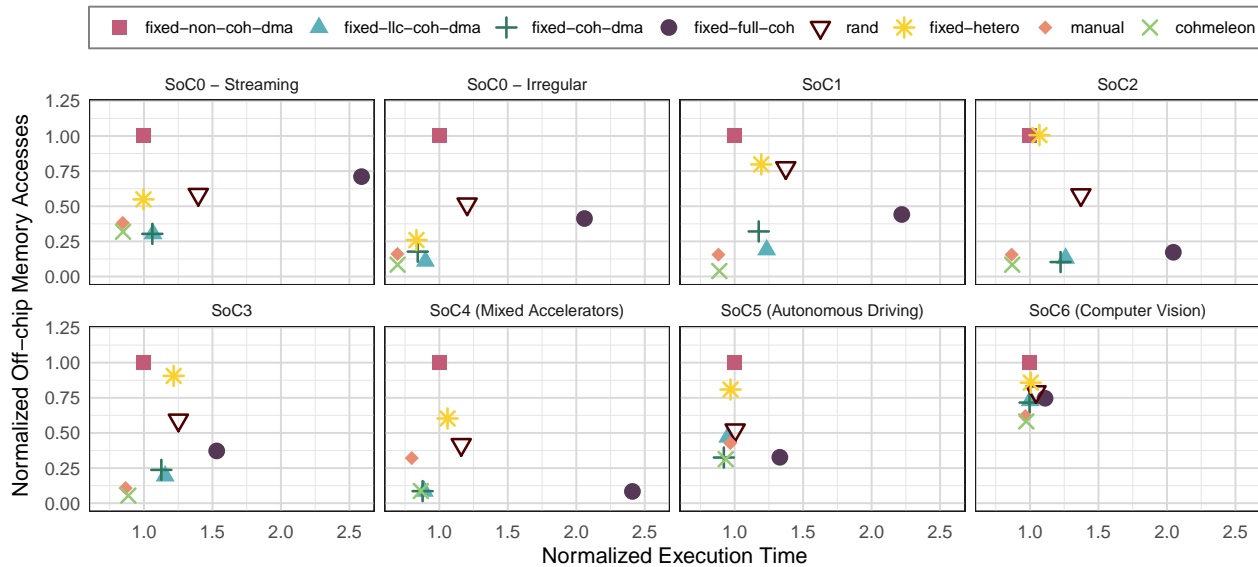
Figure 9: Performance across SoCs with different accelerators and architectural parameters.

Heterogeneous Accelerator Coherence Modes. Bhardwaj *et al.* propose a machine learning approach to assign an optimal coherence mode to each accelerator at design time [6, 7]. Giri *et al.* propose a manually-designed algorithm for deciding the cache-coherence mode at runtime, based on the system status [37]. These approaches do not handle all four cache-coherence modes, are not updated online, and require specific tuning for the target architecture; further, the latter is not a learning-based approach.

Multi-chip Accelerator Coherence. Cache coherence is relevant also for systems where the accelerators live on their own chip and communicate with a host processor core via an I/O interface, such as PCIe. Industry examples of cache-coherent chip-to-chip interconnect for accelerators include CCIX [14, 15], CXL [24], IBM CAPI [79], OpenCAPI [61], Arteris NCore [11] and ARM Core-Link [2]. Similarly to the single-chip case, they offer multiple options for handling the accelerators' cache coherence. Hence, our approach could be applied also to multi-chip systems.

Cache Bypassing. The coherence modes classified in Section 2 differ based on the degree of hardware coherence and cache bypassing. The cache bypassing for fixed-function LCAs has task granularity and doesn't require modifications to the cache hierarchy. Differently, because of the programmable nature of GPUs, the literature proposes a variety of GPU-specific cache bypassing techniques with instruction granularity and that require modification either to the compiler or the cache hierarchy [49, 50, 83, 87, 88].

RL for SoC Control Problems. Although cohmeleon is the first work using online learning to orchestrate accelerator coherence, many online learning methods have been proposed in various application domains. Liu *et al.* propose a dynamic power manager based on Q-learning that does not require any prior knowledge of the workload [51]. Gupta *et al.* present a deep Q-learning approach to dynamically manage the type, number, and frequency of active cores in SoCs [38]. Zheng *et al.* propose an energy-efficient NoC design with DVFS (dynamic voltage and frequency scaling) and a per-router Q-learning agent for selecting voltage/frequency

values [91]. Besides Q-learning and RL, other machine-learning approaches have also been proposed for system optimization [26, 28].

8 CONCLUSIONS

We showed that the performance of fixed-function LCAs in SoC architectures benefits from runtime reconfiguration of their cache-coherence mode. cohmeleon applies reinforcement learning to automatically and adaptively select the optimal cache-coherence mode at the time of each accelerator's invocation. It operates in a way that is transparent to the programmer, with negligible overhead, and without any knowledge about the target accelerators and architecture. We released cohmeleon and its FPGA-based experimental infrastructure as part of the open-source ESP project [20].

ACKNOWLEDGMENTS

A ARTIFACT APPENDIX

A.1 Abstract

To build and evaluate COHMELEON, we leveraged ESP, an open-source platform for agile design and prototyping of heterogeneous SoCs. We integrated COHMELEON in ESP, and we released it in a fork of ESP on GitHub. COHMELEON will be merged into the main ESP repository on GitHub before MICRO 2021.

We collected the results presented in the paper by running real-size applications on FPGA-based prototypes of many-accelerator multi-core SoCs designed with ESP. The main steps required to evaluate COHMELEON on FPGA include: high-level synthesis (HLS) of accelerators, FPGA bitstream generation for full SoCs, software build, deployment of the SoCs and software on an FPGA board, and execution of the experiments on the FPGA by running application software on top of the Linux operating system. This document contains the instructions for reproducing the experiments presented in the paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Reinforcement learning algorithm for selecting the coherence mode of each accelerator at run-time based on the continuous monitoring of the system.
- **Program:** Multiple multi-threaded applications containing many accelerator invocations. These applications present a wide range of accelerator parallelism and workload sizes.
- **Compilation:** Automated cross-compilation for the SPARC processors in the SoC prototypes on FPGA. The compilation includes Linux and the test programs.
- **Run-time environment:** SoC prototypes on FPGA: Linux 5.1 SMP. Host machine for using ESP: either Docker or one of the OS supported by ESP, i.e. CentOS 7, Ubuntu 18.04 or Red Hat Enterprise Linux 7.8.
- **Hardware:** proFPGA UltraScale XCVU440 Prototyping Board.
- **Metrics:** Execution time and off-chip memory accesses.
- **Output:** CSV files with detailed results for each accelerator invocation.
- **Experiments:** Accelerators running in isolation with different coherence modes (Fig. 2), accelerators running in parallel with different coherence modes (Fig. 3), comparison of COHMELEON to other solutions (design-time, run-time) on 7 different SoCs (Fig. 5,6,9), results with varying training iterations (Fig. 8).
- **How much disk space required (approximately)?:** Up to 64 GB if reproducing all experiments in the paper.
- **How much time is needed to prepare workflow (approximately)?** Setting up ESP takes 2-3 hours assuming all required tools are already installed. The generation of each FPGA bitstream takes up to 5 hours due to the large size of the target FPGA.
- **How much time is needed to complete experiments (approximately)?** Around 2 hours per experiment.
- **Publicly available?** Yes: complete source code and expected results.
- **Code licenses (if publicly available)?** Apache 2.0.
- **Archived (provide DOI)?:** Yes: 10.5281/zenodo.5150725.

A.3 Description

A.3.1 How to access. We integrated COHMELEON in ESP and we released it on GitHub (https://github.com/jzuckerman/esp/tree/cohmeleon) and on Zenodo (https://doi.org/10.5281/zenodo.5150725). The most relevant directories and files for the integration and evaluation of COHMELEON are the following:

- `accelerators/`: Accelerators used in the experiments.
- `soft/common/apps/examples/`: Applications for the case study SoCs and the motivation section results.
- `socs/`: SoC design folders. The SoCs used for the experiments are labeled with the prefix `profpga-xcvu440-`. All *Make* commands should be issued from these design folders. Each folder comes with scripts and configuration files for running the experiments.
- `results/`: Expected results in CSV format.
- `soft/common/drivers/linux/libesp/rl.h`: Core implementation of the COHMELEON reinforcement-learning algorithm.
- `rtl/sockets/csr/esp_tile_csr.vhd`: Core implementation of the ESP hardware monitoring system used by COHMELEON.
- `soft/common/drivers/linux/esp/esp.c`: Changes to the ESP device driver for accelerators to enable COHMELEON.
- `soft/common/drivers/linux/libesp/libesp.c`: Changes to the ESP API for accelerator invocation to enable COHMELEON.

A.3.2 Hardware dependencies. proFPGA UltraScale XCVU440 Prototyping Board (https://www.profpga.com/products/fpga-modules-overview/virtex-ultrascale-based/profpga-xcvu440) with four DDR4 daughter cards, a gigabit Ethernet interface daughter card, and a multi-interface daughter card. COHMELEON will also work with any of the other FPGA boards supported by ESP. We used the XCVU440 board because it has the largest FPGA and is the only one with support for up to 4 memory channels.

A.3.3 Software dependencies. The software dependencies of ESP are described in the "How to setup" guide (https://esp.cs.columbia.edu/docs/setup/setup-guide/), specifically in the sections "Software packets", "CAD tools", "Environment variables", and "Docker" for users interested in using the ESP Docker image. In terms of CAD tools, evaluating COHMELEON requires only Xilinx Vivado 2019.2 and Cadence Stratus HLS 20.24 (other versions of Stratus should work too).

A.4 Installation

The installation of ESP is described in the "How to setup" guide (https://esp.cs.columbia.edu/docs/setup/setup-guide/), specifically in the sections "ESP repository", "Software toolchain", and "Docker". Evaluating COHMELEON requires cloning the fork of ESP listed in Section A.3.1 and checking out the `cohmeleon` branch. Additionally, only the toolchain for the LEON3 processor needs to be installed.

A.5 Experiment workflow

We encourage anyone attempting to use COHMELEON to first familiarize themselves with ESP. The resources available on the ESP website (https://esp.cs.columbia.edu/resources/) include several hands-on tutorial guides, the recordings of conference tutorials, and an overview paper.

(1) **Run HLS**. Generate the RTL implementation of the accelerators. From any of the `socs/proFPGA-xcvu440-*` folders, run `make <acc_name>-hls` for all the following accelerators: `cholesky_stratus`, `conv2d_stratus`, `fft_stratus`, `gemm_stratus`, `mriq_stratus`, `nightvision_stratus`, `sort_stratus`, `svhn_autoenc_hls4ml`, `svhn_mlp_hls4ml`, `synth_stratus`, `vitdodec_stratus`.

(2) **Generate FPGA Bitstream**. In each `socs/proFPGA-xcvu440-*` SoC design folder, run the following:
- `make esp-config`: Apply a predefined SoC configuration, which can be visualized by running `make esp-xconfig`.
- `make vivado-syn`: Generate the FPGA bitstream. This step can take several hours.

There is a design folder for each of the 7 SoCs used for the experiments:

- `profpga-xcvu440-12synth` : SoC0
- `profpga-xcvu440-7synth` : SoC1
- `profpga-xcvu440-9synth` : SoC2
- `profpga-xcvu440-16synth` : SoC3
- `profpga-xcvu440-10acc` : SoC4, also used for accelerator execution in isolation experiment (Fig. 2)
- `profpga-xcvu440-autonomous-driving` : SoC5
- `profpga-xcvu440-computer-vision` : SoC6
- `profpga-xcvu440-12acc` : Used for accelerator execution in parallel experiment (Fig. 3), subsequently referred to as SoC7

(3) **Build Software**.
- `make linux`: Compile Linux for the Leon3 processor, and create Linux image for the experiments on FPGA.
- `make apps-cohmeleon`: Compile the applications required for the COHMELEON experiments, and copy all executables, scripts, and data to the root-file system that gets included in the Linux image (`soft-build/leon3/sysroot`).
- `make linux`: Always re-run this target after modifying the content of the root file system.

(4) **Deploy on FPGA**.
- `make fpga-program`: Program the FPGA with the SoC bitstream.
- `make fpga-run-linux`: Boot Linux on the SoC deployed on FPGA. Once the boot reaches the login prompt, log in with the username `root` and the password `openesp`.

Deploying the bitstream and compiled software on the FPGA, as well as running the experiments described in the next steps, requires proper connections to the FPGA. The main communication link is Ethernet, which is used for transferring the Linux image onto the FPGA. After booting Linux on the FPGA, the same Ethernet link enables the use of `ssh` and `scp`. In addition, it is possible to establish a connection the UART port in order to monitor the terminal outputs of the programs executing on the FPGA. Detailed instructions on the FPGA setup can be found in the ESP "How to: design a single-core SoC" tutorial (https://esp.cs.columbia.edu/docs/singlecore/singlecore-guide/), specifically in the sections "Debug link configuration", "UART interface", and "SSH".

(5) **Run Experiments**. After the login, navigate to the following directories to launch the experiment scripts for each SoC. Scripts and outputs are described in Section A.6.
- `profpga-xcvu440-<N>synth` :
  `/applications/test/`
- `profpga-xcvu440-10acc` :
  `/examples/multiacc/`
  `/examples/single_acc_coh`
- `profpga-xcvu440-autonomous-driving` :
  `/examples/auton_driving/`
- `profpga-xcvu440-computer-vision` :
  `/examples/comp_vision/`
- `profpga-xcvu440-12acc` :
  `/examples/parallel_acc_coh`

Output files of each experiment can be copied from the FPGA using `scp`.

(6) **Process Data**. We provide scripts to process the experiments' outputs and produce the results presented in the paper, as described in Section A.6.

A.6 Evaluation and expected results

We provide the following scripts to execute the experiments presented in the paper:

- `single_acc_coh.sh`: Reproduce the results in Fig. 2. Output file: `single_acc_coh.csv`. (SoC4)
- `parallel_acc_coh.sh`: Reproduce the results in Fig. 3. Output file: `parallel_acc_coh_devices.csv`. (SoC7)
- `cohmeleon_eval.sh`: Reproduce the results in Fig. 5, 7, 9. Output file: `<app_name>_phases.csv`, which reports the execution time and memory accesses for each phase of the application. These files can be passed to the script `process_results.py` to produce the results of Fig. 9. Selected phases can be used to produce the results from Fig. 5. The coherence mode selected for each accelerator invocation is reported, along with statistics, in `<app_name>_devices.csv`. This can be used to reproduce the results in Fig. 7. (SoCs 0, 1, 2, 3, 4, 5, 6)
- `training_time.sh` : This script trains COHMELEON over the course of 10, 30, and 50 iterations, and records the performance after each iteration. The script outputs 3 files, `training_10.csv`, `training_30.csv`, and `training_50.csv`. These results can be passed to the script `process_training.py` to extract the results for Fig. 8. (SoC0)

The results for each figure in the paper are included in the `results` folder of the repository. There is one CSV file for each figure.

A.7 Experiment customization

The reward function can be tuned by changing the coefficients in the last line before the return of `calculate_reward()` in the `rl.h` header file. After recompiling the applications and redeploying Linux on the FPGA, the new model parameters can be tested with the `cohmeleon_eval.sh` script. Repeating these steps for multiple sets of coefficients will produce the results in Fig. 6.

REFERENCES

[1] Johnathan Alsop, Matthew Sinclair, and Sarita Adve. 2018. Spandex: A Flexible Interface for Efficient Heterogeneous Coherence. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 261–274.

[2] ARM. [n. d.]. CoreLink Interconnect. https://developer.arm.com/ip-products/system-ip/corelink-interconnect.

[3] ARM. 2020. AMBA AXI and ACE Protocol Specification. https://developer.arm.com/documentation/ihi0022/h.

[4] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. 2016. OpenPiton: An Open Source Manycore Research Framework. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 217–232.

[5] Kevin Barker, Thomas Benson, Dan Campbell, David Ediger, Roberto Gioiosa, Adolfy Hoisie, Darren Kerbyson, Joseph Manzano, Andres Marquez, Leon Song, Nathan Tallent, and Antonino Tumeo. 2013. *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual.* Pacific Northwest National Laboratory and Georgia Tech Research Institute.

[6] Kshitij Bhardwaj, Marton Havasi, Yuan Yao, David M. Brooks, José Miguel Hernández-Lobato, and Gu-Yeon Wei. 2020. A Comprehensive Methodology to Determine Optimal Coherence Interfaces for Many-Accelerator SoCs. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*. 145–150.

[7] K. Bhardwaj, M. Havasi, Y. Yao, D. M. Brooks, J. M. H. Lobato, and Gu-Yeon Wei. 2019. Determining Optimal Coherency Interface for Many-Accelerator SoCs Using Bayesian Optimization. *IEEE Computer Architecture Letters* 18, 2 (2019), 119–123. https://doi.org/10.1109/LCA.2019.2910521

[8] B. Blaner, B. Abali, B. M. Bass, S. Chari, R. Kalla, S. Kunkel, K. Lauricella, R. Leavens, J. J. Reilly, and P. A. Sandon. 2013. IBM POWER7+ processor on-chip accelerators for cryptography and active memory expansion. *IBM Journal of Research and Development* 57, 6 (2013), 3:1–3:16.

[9] Luca P. Carloni. 2016. The Case for Embedded Scalable Platforms. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*. 17:1–17:6.

[10] Luca P. Carloni, Emilio G. Cota, Giuseppe Di Guglielmo, Davide Giri, Jihye Kwon, Paolo Mantovani, Luca Piccolboni, and Michele Petracca. 2019. Teaching Heterogeneous Computing with System-Level Design Methods. In *Proc. of WCAE*.

[11] Loyd Case. 2016. Easing Heterogeneous Cache Coherent SoC Design using Arteris Ncore Interconnect IP. *The Linley Group* (2016).

[12] Jared Casper and Kunle Olukotun. 2014. Hardware Acceleration of Database Operations. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. 151–160. https://doi.org/10.1145/2554688.2554787

[13] Matheus Cavalcante, Andreas Kurth, Fabian Schuiki, and Luca Benini. 2020. Design of an Open-Source Bridge between Non-Coherent Burst-Based and Coherent Cache-Line-Based Memory Systems. In *Proceedings of the International Conference on Computing Frontiers (CF)*. 81–88. https://doi.org/10.1145/3387902.3392631

[14] CCIX Consortium. 2018. CCIX Base Specification 1.0. https://www.ccixconsortium.com/library/specification/.

[15] CCIX Consortium. 2019. An Introduction to CCIX. https://www.ccixconsortium.com/wp-content/uploads/2019/11/CCIX-White-Paper-Rev111219.pdf.

[16] Y. Chen, J. Cong, M. A. Ghodrat, M. Huang, C. Liu, B. Xiao, and Y. Zou. 2013. Accelerator-rich CMPs: From concept to real hardware. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*. 169–176. https://doi.org/10.1109/ICCD.2013.6657309

[17] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2016), 127–138.

[18] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C. Chou. 2011. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 155–166. https://doi.org/10.1109/PACT.2011.21

[19] Cobham Gaisler. [n. d.]. LEON3 Processor. www.gaisler.com/index.php/products/processors/leon3.

[20] Columbia SLD Group. 2019. ESP Release. www.esp.cs.columbia.edu.

[21] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. 2014. Accelerator-rich Architectures: Opportunities and Progresses. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*.

[22] Thanh Cong and Francois Charot. 2021. Design Space Exploration of Heterogeneous-Accelerator SoCs with Hyperparameter Optimization. In *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 338–343.

[23] Emilio G. Cota, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. 2015. An Analysis of Accelerator Coupling in Heterogeneous Architectures. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*.

[24] CXL Consortium. 2020. Compute Express Linx 2.0 White Paper. https://www.computeexpresslink.org/resource-library.

[25] William J. Dally, Yatish Turakhia, and Song Han. 2020. Domain-Specific Hardware Accelerators. *Communication of ACM* 63, 7 (2020), 48–57.

[26] Yi Ding, Nikita Mishra, and Henry Hoffmann. 2019. Generative and Multi-phase Learning for Computer Systems Optimization. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 39–52.

[27] Michael Ditty, Ashish Karandikar, and David Reed. 2018. Nvidia's Xavier SoC. In *Hot Chips: A Symposium on High Performance Chips*.

[28] 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 IEEE/ACM International Symposium on Microarchitecture*. 685–698.

[29] C. F. Fajardo, Z. Fang, R. Iyer, G. F. Garcia, S. E. Lee, and L. Zhao. 2011. Buffer-Integrated-Cache: A cost-effective SRAM architecture for handheld and embedded platforms. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*. 966–971.

[30] Hubertus Franke, Jimi Xenidis, Claude Basso, Brian M. Bass, Sandra S. Woodward, Jeffrey D. Brown, and Charles L. Johnson. 2010. Introduction to the Wire-Speed Processor and Architecture. *IBM Journal of Research and Development* 54, 1 (2010), 3:1–3:11.

[31] D. Fujiki, S. Wu, N. Ozog, K. Goliya, D. Blaauw, S. Narayanasamy, and R. Das. 2020. SeedEx: A Genome Sequencing Accelerator for Optimal Alignments in Subminimal Space. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 937–950. https://doi.org/10.1109/MICRO50266.2020.00080

[32] Gen-Z Consortium. 2020. Gen-Z Specification 1.1. https://genzconsortium.org/specifications/.

[33] Davide Giri, Kuan-Lin Chiu, Guy Eichler, Paolo Mantovani, and Luca P. Carloni. 2021. Accelerator Integration for Open-Source SoC Design. *IEEE Micro* 41, 4 (2021), 8–14. https://doi.org/10.1109/MM.2021.3073893

[34] Davide Giri, Kuan-Lin Chiu, Giuseppe Di Guglielmo, Paolo Mantovani, and Luca P. Carloni. 2020. ESP4ML: Platform-Based Design of Systems-on-Chip for Embedded Machine Learning. In *Proceedings of the IEEE Conference on Design, Automation, and Test in Europe (DATE)*.

[35] Davide Giri, Paolo Mantovani, and Luca P. Carloni. 2018. Accelerators & Coherence: An SoC Perspective. *IEEE Micro* 38, 6 (2018), 36–45.

[36] Davide Giri, Paolo Mantovani, and Luca P. Carloni. 2018. NoC-Based Support of Heterogeneous Cache-Coherence Models for Accelerators. In *Proceedings of the International Symposium on Networks-on-Chip (NOCS)*. 1:1–1:8.

[37] Davide Giri, Paolo Mantovani, and Luca P. Carloni. 2019. Runtime Reconfigurable Memory Hierarchy in Embedded Scalable Platforms. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*. 719–726.

[38] Ujjwal Gupta, Sumit K Mandal, Manqing Mao, Chaitali Chakrabarti, and Umit Y Ogras. 2019. A deep Q-learning approach for dynamic management of heterogeneous processors. *IEEE Computer Architecture Letters* 18, 1 (2019), 14–17.

[39] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 243–254. https://doi.org/10.1109/ISCA.2016.30

[40] Mark D. Hill and Vijay Janapa Reddi. 2020. Accelerator-level Parallelism. arXiv:cs.DC/1907.02064

[41] Mark Horowitz. 2014. Computing's energy problem (and what we can do about it). In *Digest of Technical Papers of the International Solid-State Circuits Conference (ISSCC)*. 10–14.

[42] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. 1996. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research* 4 (1996), 237–285. https://doi.org/10.1613/jair.301

[43] John H. Kelm, Daniel R. Johnson, William Tuohy, Steven S. Lumetta, and Sanjay J. Patel. 2010. Cohesion: A Hybrid Memory Model for Accelerators. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 429–440.

[44] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel. 2011. Cohesion: An Adaptive Hybrid Memory Model for Accelerators. *IEEE Micro* 31, 1 (2011), 42–55. https://doi.org/10.1109/MM.2011.8

[45] Snehasish Kumar, Arrvindh Shriraman, and Naveen Vedula. 2015. Fusion: Design Tradeoffs in Coherent Cache Hierarchies for Accelerators. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 733–745.

[46] Andreas Kurth, Wolfgang Rǔnninger, Thomas Benz, Matheus Cavalcante, Fabian Schuiki, Florian Zaruba, and Luca Benini. 2020. An Open-Source Platform for High-Performance Non-Coherent On-Chip Communication. arXiv:2009.05334. arXiv:cs.AR/arXiv:2009.05334

[47] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 461–475.

[48] Jacob Leverich, Hideho Arakida, Alex Solomatnikov, Amin Firoozshahian, Mark Horowitz, and Christos Kozyrakis. 2008. Comparative Evaluation of Memory

Models for Chip Multiprocessors. *ACM Trans. Archit. Code Optim.* 5, 3, Article 12 (Dec. 2008), 30 pages.

[49] Ang Li, Gert-Jan van den Braak, Akash Kumar, and Henk Corporaal. 2015. Adaptive and Transparent Cache Bypassing for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. Association for Computing Machinery, New York, NY, USA, Article 17, 12 pages. https://doi.org/10.1145/2807591.2807606

[50] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. 2015. Locality-Driven Dynamic GPU Cache Bypassing. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15)*. Association for Computing Machinery, New York, NY, USA, 67âĂŞ77. https://doi.org/10.1145/2751205.2751237

[51] Wei Liu, Ying Tan, and Qinru Qiu. 2010. Enhanced Q-Learning Algorithm for Dynamic Power Management with Performance Constraint. In *Proceedings of the IEEE Conference on Design, Automation, and Test in Europe (DATE)*. 602–605.

[52] Michael J. Lyons, Mark Hempstead, Gu-Yeon Wei, and David Brooks. 2012. The Accelerator Store: A Shared Memory Framework for Accelerator-based Systems. *ACM Transactions on Architecture and Code Optimization (TACO)* (2012).

[53] Paolo Mantovani, Emilio G. Cota, Christian Pilato, Giuseppe Di Guglielmo, and Luca P. Carloni. 2016. Handling Large Data Sets for High-performance Embedded Applications in Heterogeneous Systems-on-chip. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis of Embedded Systems (CASES)*. 3:1–3:10.

[54] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G Cota, Michele Petracca, Christian Pilato, and Luca P Carloni. 2020. Agile SoC development with open ESP. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[55] Sang Lyul Min and Jean-Loup Baer. 1992. Design and analysis of a scalable cache coherence scheme based on clocks and timestamps. *IEEE Transactions on Parallel and Distributed Systems* 1 (1992), 25–44.

[56] Seung Won Min, Sitao Huang, Mohamed El-Hadedy, Jinjun Xiong, Deming Chen, and Wen-mei Hwu. 2019. Analysis and Optimization of I/O Cache Coherency Strategies for SoC-FPGA Device. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 301–306. https://doi.org/10.1109/FPL.2019.00055

[57] Mobileye (an Intel Company). 2018. Towards Autonomous Driving. https://s21.q4cdn.com/600692695/files/doc_presentations/2018/CES-2018-final-MBLY.pdf. CES.

[58] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood. 2020. *A Primer on Memory Consistency and Cache Coherence: Second Edition*. Morgan & Claypool.

[59] Rikin J Nayak and Jaiminkumar B Chavda. 2018. Comparison of accelerator coherency port (ACP) and high performance port (HP) for data transfer in DDR memory Using Xilinx ZYNQ SoC. In *Information and Communication Technology for Intelligent Systems (ICTIS 2017) - Volume 1*. Springer, 94–102.

[60] NVIDIA. 2017. NVIDIA Deep Learning Accelerator (NVDLA). www.nvdla.org.

[61] OpenCAPI Consortium. 2016. OpenCAPI 4.0 Specifications. https://opencapi.org/technical/specifications/.

[62] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 27–40. https://doi.org/10.1145/3079856.3080254

[63] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W. Keckler, Christopher W. Fletcher, and Joel Emer. 2019. Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 137–151. https://doi.org/10.1145/3297858.3304025

[64] D. Petrisko, F. Gilani, M. Wyse, D. C. Jung, S. Davidson, P. Gao, C. Zhao, Z. Azad, S. Canakci, B. Veluri, T. Guarino, A. Joshi, M. Oskin, and M. B. Taylor. 2020. BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs. *IEEE Micro* 40, 4 (2020), 93–102. https://doi.org/10.1109/MM.2020.2996145

[65] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. 58–70. https://doi.org/10.1109/HPCA47549.2020.00015

[66] S. Rahman, N. Abu-Ghazaleh, and R. Gupta. 2020. GraphPulse: An Event-Driven Hardware Accelerator for Asynchronous Graph Processing. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 908–921. https://doi.org/10.1109/MICRO50266.2020.00078

[67] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. Machsuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 110–119.

[68] Mohammadsadegh Sadri, Christian Weis, Norbert Wehn, and Luca Benini. 2013. Energy and performance exploration of accelerator coherency port using Xilinx

[69] ZYNQ. In *Proceedings of the FPGAworld Conference*.

[69] Benjamin Carrion Schafer and Anushree Mahapatra. 2014. S2cbench: Synthesizable systemc benchmark suite for high-level synthesis. *IEEE Embedded Systems Letters* 6, 3 (2014), 53–56.

[70] Yakun Sophia Shao and David Brooks. 2015. *Research Infrastructures for Hardware Accelerators*. Morgan & Claypool.

[71] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. 2019. Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 14–27. https://doi.org/10.1145/3352460.3358302

[72] Yakun Sophia Shao, Sam Likun Xi, Vijayalakshmi Srinivasan, Gu-Yeon Wei, and David Brooks. 2016. Co-Designing Accelerators and SoC Interfaces Using gem5-Aladdin. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Article 48, 12 pages.

[73] E Sisbot, Augusto Vega, Arun Paidimarri, John-David Wellman, Alper Buyuktosunoglu, Pradip Bose, and David Trilla. 2019. Multi-Vehicle Map Fusion using GNU Radio. *Proceedings of the GNU Radio Conference* 4, 1 (2019).

[74] Stephanie Soldavini and Christian Pilato. 2021. A Survey on Domain-Specific Memory Architectures. *arXiv preprint arXiv:2108.08672* (2021).

[75] Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool.

[76] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang. 2020. MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 766–780. https://doi.org/10.1109/MICRO50266.2020.00068

[77] Ashley Stevens. 2011. Introduction to AMBA® 4 ACE and big.LITTLE Processing Technology. *ARM White Paper, CoreLink Intelligent System IP by ARM* (2011).

[78] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).

[79] Jeffrey Stuecheli, Bart Blaner, C. R. Johns, and M. S. Siegel. 2015. CAPI: A Coherent Accelerator Processor Interface. *IBM Journal of Research and Development* (2015).

[80] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. MIT press.

[81] Thierry Tambe, Coleman Hooper, Lillian Pentecost, Tianyu Jia, En-Yu Yang, Marco Donato, Victor Sanh, Paul N. Whatmough, Alexander M. Rush, David Brooks, and Gu-Yeon Wei. 2021. EdgeBERT: Sentence-Level Energy Optimizations for Latency-Aware Multi-Task NLP Inference. arXiv:cs.AR/2011.14203

[82] Shelby Thomas, Chetan Gohkale, Enrico Tanuwidjaja, Tony Chong, David Lau, Saturnino Garcia, and Michael Bedford Taylor. 2014. CortexSuite: A synthetic brain benchmark suite. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 76–79.

[83] Yingying Tian, Sooraj Puthoor, Joseph L. Greathouse, Bradford M. Beckmann, and Daniel A. Jiménez. 2015. Adaptive GPU Cache Bypassing. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs (GPGPU-8)*. Association for Computing Machinery, New York, NY, USA, 25âĂŞ35. https://doi.org/10.1145/2716282.2716283

[84] Christopher J.C.H. Watkins and Peter Dayan. 1992. Q-learning. *Machine Learning* 8, 3/4 (1992), 279–292. https://doi.org/10.1023/a:1022676722315

[85] Christopher John Cornish Hellaby Watkins. 1989. *Learning from Delayed Rewards*. Ph.D. Dissertation. King's College, Cambridge, UK.

[86] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. 2014. Q100: The Architecture and Design of a Database Processing Unit. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 255–268. https://doi.org/10.1145/2541940.2541961

[87] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. 2013. An efficient compiler framework for cache bypassing on GPUs. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 516–523. https://doi.org/10.1109/ICCAD.2013.6691165

[88] Xiaolong Xie, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. 2015. Coordinated static and dynamic cache bypassing for GPUs. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 76–88. https://doi.org/10.1109/HPCA.2015.7056023

[89] Xilinx. 2018. Adaptable Intelligence: The Next Computing Era. Keynote, Hot Chips Symposium.

[90] P. Yao, L. Zheng, Z. Zeng, Y. Huang, C. Gui, X. Liao, H. Jin, and J. Xue. 2020. A Locality-Aware Energy-Efficient Accelerator for Graph Mining Applications. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 895–907. https://doi.org/10.1109/MICRO50266.2020.00077

[91] Hao Zheng and Ahmed Louri. 2019. An energy-efficient network-on-chip design using reinforcement learning. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*.