ArchGym: An Open-Source Gymnasium for Machine Learning Assisted Architecture Design

Srivatsan Krishnan
srivatsan@seas.harvard.edu
Harvard University
Cambridge, Massachusetts, USA

Amir Yazdanbaksh
ayazdan@google.com
Google Research, Brain Team
Mountain View, California, USA

Shvetank Prakash
sprakash@g.harvard.edu
Harvard University
Cambridge, Massachusetts, USA

Jason Jabbour
jasonjabbour@g.harvard.edu
Harvard University
Cambridge, Massachusetts, USA

Ikechukwu Uchendu
iuchendu@g.harvard.edu
Harvard University
Cambridge, Massachusetts, USA

Susobhan Ghosh
susobhan_ghosh@g.harvard.edu
Harvard University
Cambridge, Massachusetts, USA

Behzad Boroujerdian
behzadboro@utexas.edu
UT Austin/Harvard University
Cambridge, Massachusetts, USA

Daniel Richins
drichins@utexas.edu
UT Austin
Austin, Texas, USA

Devashree Tripathy
devashreetripathy@iitbbs.ac.in
IIT Bhubaneswar/Harvard University
Bhubaneswar, Odisha, India

Aleksandra Faust
faust@google.com
Google Research, Brain Team
Mountain View, California, USA

Vijay Janapa Reddi
vj@eecs.harvard.edu
Harvard University
Cambridge, Massachusetts, USA

ABSTRACT

Machine learning (ML) has become a prevalent approach to tame the complexity of design space exploration for domain-specific architectures. While appealing, using ML for design space exploration poses several challenges. First, it is not straightforward to identify the most suitable algorithm from an ever-increasing pool of ML methods. Second, assessing the trade-offs between performance and sample efficiency across these methods is inconclusive. Finally, the lack of a holistic framework for fair, reproducible, and objective comparison across these methods hinders the progress of adopting ML-aided architecture design space exploration and impedes creating repeatable artifacts. To mitigate these challenges, we introduce ArchGym, an open-source gymnasium and easy-to-extend framework that connects a diverse range of search algorithms to architecture simulators. To demonstrate its utility, we evaluate ArchGym across multiple vanilla and domain-specific search algorithms in the design of a custom memory controller, deep neural network accelerators, and a custom SoC for AR/VR workloads, collectively encompassing over 21K experiments. The results suggest that with an unlimited number of samples, ML algorithms are equally favorable to meet the user-defined target specification if its hyperparameters are tuned thoroughly; no one solution is necessarily better than another (e.g., reinforcement learning vs. Bayesian methods). We coin the term "*hyperparameter lottery*" to describe the relatively probable chance for a search algorithm to find an optimal design provided meticulously selected hyperparameters. Additionally, the ease of data collection and aggregation in ArchGym facilitates research in ML-aided architecture design space exploration. As a case study, we show this advantage by developing a proxy cost model with an RMSE of 0.61% that offers a 2,000-fold reduction in simulation time. Code and data for ArchGym is available at https://bit.ly/ArchGym.

CCS CONCEPTS

• **Computer systems organization → Architectures**; • **Computing methodologies → Reinforcement learning**; **Machine learning algorithms**; **Bio-inspired approaches**.

KEYWORDS

Machine learning, Machine Learning for Computer Architecture, Machine Learning for System, Reinforcement Learning, Bayesian Optimization, Open Source, Baselines, Reproducibility

1 INTRODUCTION

Hardware customization [18, 20, 29, 46, 47, 68, 89, 99, 103] has played a pivotal role in realizing the potential of machine learning in various applications [4, 17, 23, 48, 56, 70, 102]. The stagnation of Moore's law and the increasing demand for compute efficiency [12, 19, 83, 90] have propelled the field towards pursuing extreme domain-specific customization. While intriguing, this direction poses several challenges. The immense number of design parameters across the compute stack leads to a combinatorial explosion of the search space [37, 89, 108]. Within this space, numerous infeasible design points further complicate optimization [37, 57]. Additionally, the diversity of the application landscape and the unique characteristics of the search space across the compute stack challenge the performance of conventional optimization methods. To address these challenges, both industry [70, 85, 98] and academia [37, 41, 51, 52, 86, 89, 97] have turned towards ML-driven optimization to meet stringent domain-specific requirements. Although prior work has demonstrated the benefits of ML in design optimization, the lack of reproducible baselines hinders fair and objective comparison across different methods.

First, *selecting the most suitable algorithms and gauging the role of hyperparameters and their efficacy is still inconclusive.* There are a wide range of ML/heuristic methods, from random walker [106] to reinforcement learning (RL) [96], that can be employed for design space exploration (DSE). For example, recent work has applied Bayesian [84, 92], data-driven offline [89], and RL [51] optimization methods for architecture parameter exploration of Deep Neural Network (DNN) accelerators. While these methods have shown noticeable performance improvement over their choice of baselines, it is not evident whether the improvements are because of the choice of optimization algorithms or hyperparameters. To ensure reproducibility and facilitate widespread adoption of ML-aided architecture design space exploration, it is imperative to outline a systematic benchmarking methodology.

Second, while simulators have been the backbone of architectural innovations, *there is an emerging need to address the trade-offs between accuracy, speed, and cost in architecture exploration.* The accuracy and performance estimation speed widely varies from one simulator to another, depending on the underlying modeling details (e.g. cycle-accurate [15, 65] → transaction-level simulator [67, 95] → analytical model [6, 10, 58, 78, 104] → ML-based proxy models [36, 54, 69, 97]). While analytical or ML-based proxy models are nimble by virtue of discarding low-level details, they generally suffer from high prediction error. Also, due to commercial licensing, there can be a strict limits on the number of samples collected from a simulator [33]. Overall, these constraints exhibit distinct performance vs. sample efficiency trade-offs, affecting the choice of optimization algorithm for architecture exploration. Therefore, it is challenging to delineate *how to systematically compare the effectiveness of various ML algorithms under these constraints.*

Lastly, *rendering the outcome of DSEs into meaningful artifacts such as datasets is critical for drawing insights about the design space.* It is commonly known that the landscape of ML algorithms is rapidly evolving and some ML algorithms [59] need data to be useful. Solely in the RL domain, we have witnessed the emergence of several algorithmic formulations (*e.g.* PPO [88], SAC [34], DQN [71], DDPG [62]) solving a variety of problems. In parallel, recent efforts have employed offline RL [59] methods to amortize the cost of data collection. In this rapidly evolving ecosystem, it is consequential to ensure *how to amortize the overhead of search algorithms for architecture exploration.* It is not apparent, nor systematically studied how to leverage exploration data while being agnostic to the underlying search algorithm.

To alleviate these challenges, we introduce ArchGym (See Fig. 1), an open-source gymnasium to analyze and evaluate various ML-driven methods for design optimization. ArchGym reinforces using the same interface between search algorithms and performance models (e.g. architecture simulator or proxy cost models), enabling effective mapping of variety of search algorithms. This interface also forms the scaffold to develop baselines for comparison and benchmarking of search algorithms, as they continue to evolve and grow. Furthermore, ArchGym provides an infrastructure to collect and share datasets in a reproducible and accessible manner, which organically advances the understanding of the underlying design spaces and improves the status quo search algorithms.

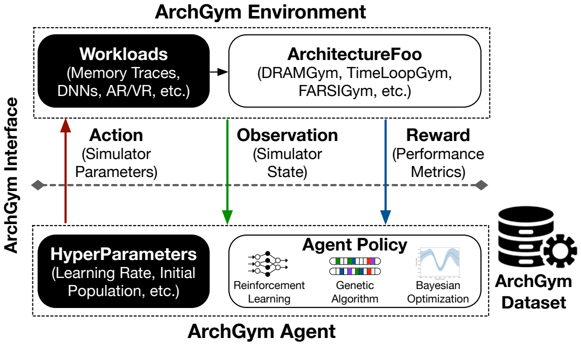


Figure 1: ArchGym comprises two main components: the 'ArchitectureFoo' environment and the 'Agent'. ArchitectureFoo encapsulates the cost model, which can be a simulator (e.g., DRAMSys [95]), a roofline model (e.g., FARSI [10]), an analytical model (e.g., MASTERO [58]), or even real hardware. Similarly, the second component, Agent, is an abstraction of a policy and hyperparameters (see Section 4). With a standardized interface that connects these two components, ArchGym provides a unified framework for evaluating different machine learning-based search algorithms fairly while also saving the exploration data as the ArchGym Dataset. By using ArchGym, researchers and practitioners can compare and evaluate the performance of different algorithms in a consistent and systematic manner.

Table 1: Prior works using ML for optimizing architectural components.

Prior Work	ML-Method	Architecture Component
DRL-NOCS [64]	Reinforcement Learning	Network-on-Chips
GAMMA [52]	Genetic Algorithm	ML Accelerator Mapping
PRIME [57]	Data-Driven Offline Learning	ML Accelerator Datapath
Reagen. et. al. [84]	Bayesian Optimization	NN HW-SW Co-Design
FAST [108]	Linear Combinatorial Swarms	NN HW-SW Co-Design
Ipek. et. al. [43]	Reinforcement Learning	DRAM Memory Controller
Zhang. et. al. [109]	Ant Colony Optimization	Circuit Parameters
Compiler Gym [21]	Reinforcement Learning	Compiler Optimization
ArchGym (This Work)	**ML * (BO, GA, RL, ACO ...)**	**Architecture* (DRAM, SoC, Mapping etc)**

We perform more than 21,600 experiments corresponding to around 1.5 billion simulations across four architectural design space exploration problems, (1) DRAM memory controller, (2) DNN accelerator, (3) SoC design, and (4) DNN mapping. We use five commonly used search algorithms and comprehensively sweep their associated hyperparameters. Our evaluation shows that there are significant variations in the final performance of these algorithms. For instance, we observe a statistical spread in the performance of different search algorithms of up to 90%, 20%, and 40% for DRAM memory controller, DNN Accelerator, and SoC design, respectively.[1] Including outliers, each algorithm yields at least *one configuration* that achieves the *best* objective across different design spaces.

These observed variations is the results are primarily a consequence of hyperparameter selection, as yet an open research problem [107]. The choice of optimal hyperparameter values depends

[1] We measure the statistical spread by reporting the interquartile range.

on the characteristics of the ML algorithm as well as the underlying domain. However, commonly used hyperparameter tuning techniques [8, 28, 74] introduce another layer of complexity. That is, identifying the optimal hyperparameters for architecture DSE remains non-trivial, an improbable task akin to winning a lottery, requiring significant amount of resources. This "*hyperparameter lottery*" describes the comparable chance of an algorithm to attain an optimal solution.[2] Finally, in contrast to common wisdom, our analyses suggest that the evaluated search methods are equally favorable across different design space exploration problems. Below we summarize the main contributions of our work:

- We design and open source the ArchGym framework for ML-aided architecture design space exploration, enabling systematic evaluation and objective comparison of search algorithms.
- Leveraging this framework, we show that, contrary to common wisdom, the evaluated search algorithms are all equally favorable for architecture design space exploration, no one algorithm (e.g. RL or Bayesian methods or GA) is necessarily more promising.
- We argue that to fairly compare ML algorithms, it is crucial to take into account the cost of hyperparameter optimization, such as access to hardware simulator samples. Without proper evaluation metrics, the effectiveness of search algorithms can be misleading to realize the potential of ML-aided design.
- We release a set of curated datasets that are useful for building high-fidelity proxy cost models. Such proxy cost models are often orders of magnitude faster compared to conventional cycle-accurate simulators, mitigating the trade-off between speed and accuracy in architecture exploration.
- Building off the intuition that increasing dataset size improves accuracy, we show that adding diversity, enabled by ArchGym, can reduce the average root mean square error by up to 42×.

2 BACKGROUND AND RELATED WORK

Though ML can be used for many classes of optimization problems, in this paper, we center our study to architecture design space exploration problems. Architecture DSE corresponds to a class of problems that uses search algorithms to navigate the architecture parameter design space. This generally forms a prohibitively large search space, and as a result an intractable problem for manual search. Hence, architects commonly employ heuristics or ML-aided search algorithms to navigate the space in the pursue of efficient designs. One of the common metrics to asses the efficiency of search algorithms is the number of requisite samples before reaching an optimal solution. The search algorithm iteratively suggests parameter values for a given workload (or set of workloads). The fitness (i.e., *how good a particular parameter selection is*) of these selections is determined by a cost model. For architecture DSE, this cost model can be a time- and resource-consuming cycle-accurate simulators or a relatively fast inaccurate analytical models.

While an exhaustive search may be feasible when number of parameters is modest, such approach is not practical even in automated DSE frameworks [22, 40, 94]. A growing body of work has used analytical models [53, 75], sampling techniques [91, 105], and statistical simulation [25, 76, 82] to navigate large search spaces.

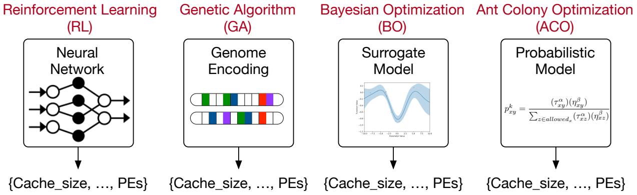

Reinforcement Learning (RL)
Genetic Algorithm (GA)
Bayesian Optimization (BO)
Ant Colony Optimization (ACO)
Neural Network
Genome Encoding
Surrogate Model
Probabilistic Model
$p^k_{xy} = \frac{(\tau^\alpha_{xy})(\eta^\beta_{xy})}{\sum_{z\in allowed_x}(\tau^\alpha_{xz})(\eta^\beta_{xz})}$
{Cache_size, ..., PEs}
{Cache_size, ..., PEs}
{Cache_size, ..., PEs}
{Cache_size, ..., PEs}

Figure 2: Agent and its policy are used to determine optimal parameter selection. For example, in an RL agent, the policy is typically a neural network. In Genetic algorithms (GA), the policy is a genome. In BO, the policy is a surrogate model; in ACO, the policy is a probabilistic model. The policies in each of these agents determine the parameter selection. For example, in micro-architectural resource allocation problems, parameters can be any micro-architectural parameters such as cache size or PE counts.

However, the continued increase in configurable architecture parameters [89, 108] is expanding the design space [50], straining conventional search methods.

To mitigate this, ML algorithms have been widely employed in searching architectural parameters with inspiring results as some are highlighted Table 1. Bayesian optimization (BO) [9, 84, 110] is a popular technique for tuning architectural parameters. However, since the time complexity of BO is cubic to the number of samples [66, 101], it is yet to be seen if BO can successfully converge on extremely large design spaces [51, 57, 108]. Alternatively, single agent RL is used for data path and mapping optimization for DNN accelerators [51], identifying quantization levels [26], and commercial chip floor planning [70]. However, it is commonly-known that RL algorithms are sample inefficient [13, 44] and susceptible to variations in hyperparameter initialization [45]. It is evident that both the machine learning and hardware communities could benefit from developing stronger introspection skills in order to create more resilient and dependable RL optimization algorithms.

3 DESIGN AND IMPLEMENTATION

Our primary objectives in ArchGym is to apply ML approaches for architecture design space exploration, fairly compare their performance, and provide an infrastructure for collecting exploration datasets. Additionally, we aim to understand the relevant trade-offs associated with different ML algorithms.

ArchGym consists of three main components, namely *Gym Environment*, *Agents*, and *Interface Signals*. Figure 1 outlines a high-level view of these components in ArchGym. Our proposed search framework is designed to be modular and flexible to exchange the architectural problem under study as well as ML agents in a straightforward manner. To achieve this, we wrap each architectural component into an ArchGym environment. The architectural cost model can be a cycle-accurate simulator, analytical model, an ML-based cost model, or alternatively silicon hardware. There are bi-directional interface signals from the ArchGym environment to the agents. Each agent, irrespective of its type (e.g., Bayesian optimization, reinforcement learning, ant colony optimization), uses the exact same interface signals to interact with ArchGym environments.

[2] We refer to a design as optimal as long as it meets all user-defined criteria for a target hardware, for example latency $< L$.

Table 2: The similarity between various ML-based search algorithms. ArchGym uses these similarities to integrate the different algorithms into a standard, unified interface.

Q*n*	Search Algorithms				Requirements
	Reinforcement Learning	Bayesian Optimization	Ant-Colony Optimization	Genetic Algorithms	
Q1	Policy will determine the actions	Surrogate model will determine the actions	Pheromones will determine the actions	Genome will determine the actions	Reward/Fitness is needed
Q2	Reward is used as feedback	Fitness is used as feedback	Fitness value will determine pheromone concentration	Fitness value will determine which candidates need to reproduce	State of the simulator/observation is needed to understand how good/bad it is.
Q3	Hyperparameters in RL algorithm	Acquisition Function	Stochastic model will determine when to explore vs exploit	Mutation/Cross-over operation	Intrinsic to Agent

3.1 Environment

Each environment is an encapsulation of the architecture cost model along with the target workload(s). The architecture cost model determines the cost of running the workload, given a set of architecture parameters. For example, the cost can be latency, throughput, area, energy, or any other combinations of user-defined performance metrics. Fig. 1 outlines various components in the environment. We expound each part in the following.

Architecture cost model. Depending upon the architecture under study, 'ArchitectureFoo' (see Fig. 1) can be replaced with representative architecture cost model. This is a placeholder for an architecture cost model in which a user intends to apply ML methods for design space exploration. For instance, if the user wants to use ML for architecture design space exploration of a memory controller, the ArchitectureFoo would be replaced by *DRAMGym*, which encapsulates the DRAM architecture cost model. Similarly, if the user wants to employ ML for design space exploration of DNN accelerators, ArchitectureFoo would be replaced by *TimeloopGym* which encapsulates Timeloop [78]. Finally, for a complex SoC for AR/VR workloads, we can encapsulate FARSI [10] as a *FARSIGym* to provide the SoC cost model.

Target Workload(s). Workloads are the integral components in ArchGym. Each workload representation can be diverse and vary significantly depending upon the component the user intends to optimize. For instance, in the case of optimizing memory controllers, the workload could be memory access traces of a particular application. Likewise, for DNNs, the workload can be represented as a graph or information about various layers. Similarly, an AR/VR workload can be represented as a task graph where each task can be mapped to different IPs in an SoC.

3.2 Agent

We define 'Agent' as an encapsulation of the machine learning algorithm used for search. An ML algorithm consists of 'hyperparameters' and a guiding 'policy'. The hyperparameter is intrinsic to an algorithm which can significantly influences its performance. A policy, on the other hand, determines how the agent selects a parameter iteratively to optimize its target objective. To further reinforce this abstraction, we seed our infrastructure with five agents from recently developed search algorithms, namely, Ant-Colony Optimization [31], Bayesian Optimization [72], Genetic Algorithms [16], Random Walker agent [106], and Reinforcement Learning [96] to solve the same set of architectural design space exploration problems. Figure 2 demonstrates the four agents that we develop and integrate in ArchGym. The random walker algorithm [106], which is not shown in this figure, is merely a random search with a random number generator as its policy.

As shown in Table 1, these algorithms are commonly used in hardware design at different abstraction levels. For example, BO has been used in efficient accelerator design space exploration [84] and in selecting the best coherency interfaces for hardware accelerators on SoCs [9]. RL has been used for DNN architecture design [51], chip floor planning [70], and exploring router-less NoC designs [63]. GA has been used in design space exploration of heterogeneous multiprocessor embedded systems [73] and behavioral IPs [87]. ACO has been applied to solving long-standing compiler optimization problems [93] and has been used in high-level synthesis design space exploration [30].

3.3 Interface

An Agent's role in architecture design space exploration is to output a set of parameter selections according to its policy. This parameter selection acts as an input signal to the ArchGym environment. Likewise, the ArchGym's environment outputs the state information of the architecture cost model and a feedback signal back to the agent. We need interface signals to facilitate these communications between the agent and the ArchGym. Three main signals are needed: `action`, `observation`, `reward` (Fig. 1). They all use these signals to optimize their target objective regardless of the agent type.

ArchGym uses the OpenAI gym interface to expose the parameters to all machine learning algorithms. The `action` is used as the interface to relay the agents actions to the environment. Likewise, the `observation` is used as the interface to relay the state information of the environment back the agent. Additionally, the `reward` is the feedback signal for the agent's architecture parameter selection. In the case of RL, this signal is called the reward signal. Likewise, this reward signal is called fitness in other agents, such as Bayesian optimization, ACO, and GA. The agent uses this `reward` to fine-tune its policy to make better parameter selections in the future to optimize its target objective.

Since all agents interact with ArchGym environment through the same interface, we can additionally record each interaction. We call these interactions trajectories. These trajectory recordings can be used to construct standardized datasets for training sample inefficient algorithms, offline algorithms [57], or constructing cost models for architectural simulators (See Section 6).

Table 3: Summary of the Gym environments in experimental setup. Objective is to optimize reward via optimal actions.

Gym Environment	Simulator	Workload	Action	Observation	Reward
DRAMGym	DRAMSys [95]	Memory Trace (i.e., Streaming Access Pattern, Random Access Pattern)	Memory Controller Parameters (i.e, RefreshPolicy, RequestBufferSize, etc.)	`<latency, power, energy>`	$r_x = \frac{X_{\text{target}}}{\lvert X_{\text{target}} - X_{\text{obs}} \rvert}$
TimeloopGym	Timeloop [78]	Convolutional Neural Network (i.e., AlexNet, MobileNet, ResNet-50)	Accelerator Parameters (i.e., NumPEs, WeightsSPad_BlockSize, etc.)	`<latency, energy, area>`	$r_x = \frac{X_{\text{target}}}{\lvert X_{\text{target}} - X_{\text{obs}} \rvert}$
FARSIGym	FARSI [10]	AR/VR Audio & Image Processing Task Dependency Graph (i.e., Audio Decoder, Edge Detection)	System On-Chip Parameters (i.e., PE_Type, NoC_BusWidth, etc.)	`<power, performance, area>`	$distance\ to\ budget = \sum_m \alpha * \frac{(D_m - B_m)}{B_m}$ $m \in \{Performance, Power, Area\}$
MaestroGym	Maestro [58]	DNN Workloads (i.e., ResNet18, MobileNet, etc.)	DNN Mapping (i.e., L1 and L2 mapping, etc.)	`<runtime, throughput, energy, area>`	$r_x = \frac{1}{X_{\text{target}}}$

3.4 Dataset Generation

One of the unique features of ArchGym is that it can be used to collect standardized datasets across all agents for the same architecture design space exploration task. Using a standardized interface (Section 3.3), ArchGym can log the information exchanges between all agents and the environment into popular dataset exchange formats like TFDS [77], RLDS [81]. Over time, accruing these datasets for similar architecture design space explorations enables new variants of data-driven offline learning algorithms [89]. Furthermore, as we demonstrate in Section 7, the diversity in these datasets (from different agents) can create high-fidelity proxy cost models to replace the slow architectural cost models (e.g., cycle-accurate simulators). We strongly believe a community-wide adoption of ArchGym methodology can create useful datasets for tackling fundamental problems surrounding data scarcity from the architecture cost model and can significantly speed up architecture design space exploration.

4 INTEGRATION OF NEW ALGORITHMS

In this section, we provide an intuitive explanation of the design and implementation of ArchGym for ML-aided design space exploration. This approach allows us to better understand the similarities between different ML search algorithms and serves as a foundation for adding novel ML-based search algorithms to ArchGym.

The goal of any intelligent agent is to determine the optimal parameter selection to maximize (or minimize) the objective. During the agents' training process (optimization phase), it has to fine-tune its policy. To achieve these goals, each agent answers the following questions: ***(Q1)****: How does an agent determine a particular parameter (action)?*; ***(Q2)*** *How does an agent use feedback for selecting a particular parameter and fine-tune the policy?*; ***(Q3)*** *How does the agent balance between exploration and exploitation?* We tabulate the answer to these questions for the four agents shown in Table 2.

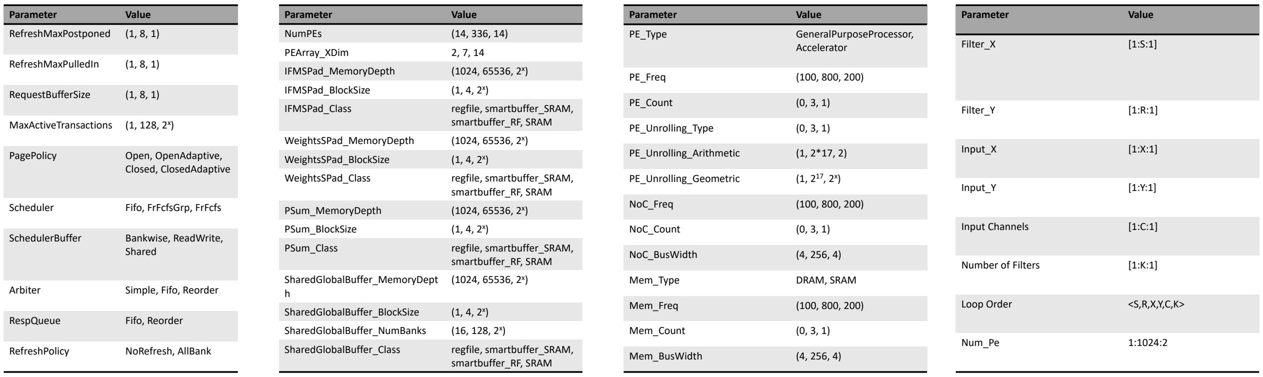
Given that each agent approaches the same questions differently, we distill the information exchanged by all these agents to answer **Q1**, **Q2**, and **Q3** to standardize the interface. Such standardization of interfaces provides baselines to objectively compare all the agents in a fair and reproducible manner.

(Q1) How does each agent select a parameter? In the Q1 scenario, each agent has a policy that allows it to take intelligent parameter selection. As shown in Fig. 2, RL uses the neural network policy to determine the action. In BO, the agent's surrogate model determines the actions. Likewise, in ACO, the agent has a simple probabilistic model as a function of pheromone concentration, guiding the ant agent to take a particular action. Lastly, in the case of GA, the genome of each population determines the actions. Hence, from an information exchange point of view, each agent outputs actions (parameters) while the policy being intrinsic to the agent. Moreover, since each search algorithm is iterative, the agent needs to output actions at each step/iteration.

(Q2) How does the agent receive feedback after selecting a parameter? Once the agent selects a parameter, it is relayed to the environment. The environment performs the simulation and outputs a feedback signal that the agent uses to fine-tune its policy. In RL, the reward signal is a function of the optimization objective (e.g., minimizing latency, area, etc.). In BO, ACO, and GA, it receives fitness metrics to evaluate the selected parameter set and fine-tune the policy accordingly. For instance, in an ACO agent, the pheromone concentration is modulated by this fitness. Similarly, in BO, the surrogate model is refined. Likewise, GA uses it to determine the fittest agent for natural selection, which will result in a new genome. Hence, from an information exchange point of view, each agent receives a reward/fitness metric, which is consequently used to fine-tune the policy and take more intelligent parameter selection.

(Q3) How does the agent balance between exploration and exploitation? In a large parameter design space, an intelligent agent should ensure that it shouldn't get stuck at local maxima or minima. To learn this, it needs to explore other regions smartly, even though there is a higher chance that it would be sub-optimal. The efficacy of each agent in determining the optimal parameter set depends upon how the agent balances exploration and exploitation. For this purpose, each agent has a set of hyperparameters that influences how an agent performs exploration and exploitation. For example, in GA, there are two, namely probability of mutation and crossover, which allow the GA agent to explore. Similarly, ACO has a probabilistic model which switches between selecting random actions or choosing a parameter set that has a higher pheromone concentration. When the ACO agent chooses a random action, it facilitates exploration. Likewise, in BO, there is an acquisition function [72] that facilitates exploration. Lastly, in RL, many methods, such as adding noise to NN policy and regularization, etc., are employed to facilitate exploration in reinforcement learning. Hence, in summary, how agent balances exploration and exploitation is often an integral part of the agent. Therefore, each agent should expose its hyperparameters during the agent's initialization.

We adopt the OpenAI gym [5] interface to integrate all the necessary information exchange to and from the agents. While OpenAI gym is limited to RL, Arch-Gym supports many other agents thanks to our observations that all agents can be systematically be broken down into Q1, Q2 and Q3. For example, in Q1, each agent uses the policy to determine a set of parameters. Furthermore, the gym environment provides a `step()` interface through which



Parameter	Value
RefreshMaxPostponed	(1, 8, 1)
RefreshMaxPulledIn	(1, 8, 1)
RequestBufferSize	(1, 8, 1)
MaxActiveTransactions	(1, 128, 2^x)
PagePolicy	Open, OpenAdaptive, Closed, ClosedAdaptive
Scheduler	Fifo, FrFcfsGrp, FrFcfs
SchedulerBuffer	Bankwise, ReadWrite, Shared
Arbiter	Simple, Fifo, Reorder
RespQueue	Fifo, Reorder
RefreshPolicy	NoRefresh, AllBank

(a) DRAM Memory Controller

Parameter	Value
NumPEs	(14, 336, 14)
PEArray_XDim	2, 7, 14
IFMSPad_MemoryDepth	(1024, 65536, 2^x)
IFMSPad_BlockSize	(1, 4, 2^x)
IFMSPad_Class	regfile, smartbuffer_SRAM, smartbuffer_RF, SRAM
WeightsSPad_MemoryDepth	(1024, 65536, 2^x)
WeightsSPad_BlockSize	(1, 4, 2^x)
WeightsSPad_Class	regfile, smartbuffer_SRAM, smartbuffer_RF, SRAM
PSum_MemoryDepth	(1024, 65536, 2^x)
PSum_BlockSize	(1, 4, 2^x)
PSum_Class	regfile, smartbuffer_SRAM, smartbuffer_RF, SRAM
SharedGlobalBuffer_MemoryDepth	(1024, 65536, 2^x)
SharedGlobalBuffer_BlockSize	(1, 4, 2^x)
SharedGlobalBuffer_NumBanks	(16, 128, 2^x)
SharedGlobalBuffer_Class	regfile, smartbuffer_SRAM, smartbuffer_RF, SRAM

(b) Eyeriss-Like Accelerator

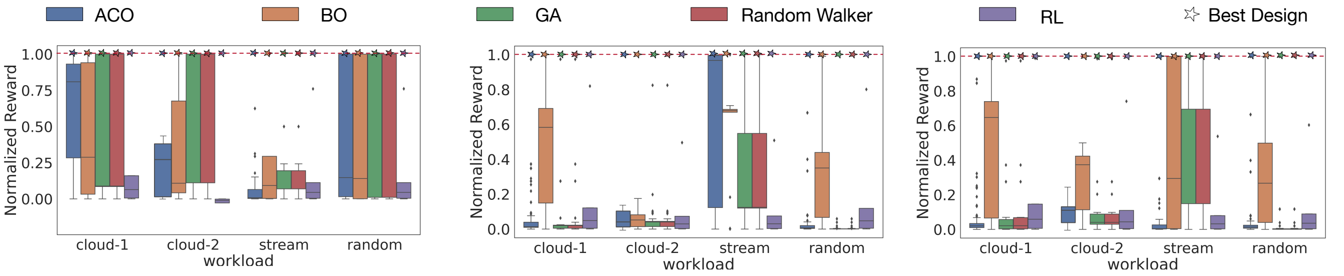
Parameter	Value
PE_Type	GeneralPurposeProcessor, Accelerator
PE_Freq	(100, 800, 200)
PE_Count	(0, 3, 1)
PE_Unrolling_Type	(0, 3, 1)
PE_Unrolling_Arithmetic	(1, 2*17, 2)
PE_Unrolling_Geometric	(1, 2^{17}, 2^x)
NoC_Freq	(100, 800, 200)
NoC_Count	(0, 3, 1)
NoC_BusWidth	(4, 256, 4)
Mem_Type	DRAM, SRAM
Mem_Freq	(100, 800, 200)
Mem_Count	(0, 3, 1)
Mem_BusWidth	(4, 256, 4)

(c) System On-Chip Architecture

Parameter	Value
Filter_X	[1:S:1]
Filter_Y	[1:R:1]
Input_X	[1:X:1]
Input_Y	[1:Y:1]
Input Channels	[1:C:1]
Number of Filters	[1:K:1]
Loop Order	<S,R,X,Y,C,K>
Num_Pe	1:1024:2

(d) DNN Mapping

Figure 3: Architecture DSE problem from (a) component-level , (b) accelerator level, and (c) SoC level, and (d) Mapping. A mixture of numerical and categorical parameters are learned. Numerical parameters are specified in tuple format: (min, max, step). The total search space for DRAMGym, TimeloopGym, FARSIGym, and MASTEROGym are 1.9e7, 2e14, and 1.6e17, 1e24 (for VGG16 second layer) respectively.



(a) Low power

(b) Low latency

(c) Joint optimization of low latency and power

Figure 4: Hyperparameter lottery across different target objectives: low-power, low latency, and joint optimization of latency and power for DRAMGym environment. ACO, BO, GA, RL refers to ant colony optimization, Bayesian optimization, genetic algorithm, and reinforcement learning, respectively. The design that achieves the maximum reward is the best design and denoted by a star symbol.

the agent's action (parameter) can be encapsulated. In the case of Q2, the `step()` also returns a feedback signal (e.g., reward/fitness) which the agents can use to fine-tune its policy. Lastly, in the case of Q3, each agent's hyperparameters are innate to the agent's initialization. Moreover, in certain RL algorithms (e.g., DDPG [62]), noise is added to the parameters (i.e., action) to allow further exploration. These noise-induced parameters can be passed through the `step()`.

5 EXPERIMENTAL SETUP

In this section we describe the simulator, workload, and agent implementations. Table 3 summarizes the key aspects of each simulator, such as workloads, actions, observation, and rewards. In Figure 3, we provide the parameter set for each ArchGym environment and elaborate on our experimental setup. Briefly, we use four environments to demonstrate the utility of ArchGym and their details (architecture, parameters, workloads).

Simulators. We developed gym environments for four simulators to demonstrate ArchGym's generalizability: DRAMGym uses DRAMSys [49], TimeloopGym uses Timeloop [78] for DNN accelerators, FARSIGym uses FARSI [10] for complex SoCs, and MaestroGym uses Maestro [58] for DNN mapping as the simulator.

Workloads. For exploring DRAM memory controller designs, we use the memory traces provided within DRAMSys [95]. Likewise, for evaluating several candidate SoC designs, we use AR/VR workloads that come prepackaged with the FARSI simulator [10]. We use Pytorch2Timeloop [78] to convert the CNN models to a format that Timeloop accepts. Likewise, we use different CNN models available in Maestro to evaluate the optimal mapping.

Agents. For each agent, we took existing open-source implementations and modified the interfaces for our architectural design space problem. For ACO, we adopted Deepswarm [14], which implements ant colony swarm intelligence. The skopt Python library [1] provides an implementation of GA Optimization. We use ACME [39] research framework for RL. Our BO implementation was repurposed from the Scikit-opt [2] Python library. For random walker implementation we used Numpy [35].

6 EVALUATION

We demonstrate how ArchGym evaluates ML algorithms for architecture design space exploration in four hardware architecture components: a DRAM memory controller, a DNN hardware accelerator, an SoC, and DNN mapping problem. We then show how ML algorithm performance varies under different sample efficiency constraints. Finally, we demonstrate that the dataset collected from standardized interface from all ML algorithms can aid in constructing a high-fidelity proxy model of the simulator.

The insights from ArchGym are three-fold. First, all ML algorithms can equally find designs for a given target specification, but their performance is highly sensitive to hyperparameter selection. Therefore, finding optimal hyperparameters for each algorithm is akin to a lottery, and an exhaustive search reveals at least one set of parameters for each algorithm that achieves comparable performance (i.e., meets target design) to others.

Second, normalizing comparisons using sample efficiency is necessary, as it can be difficult to compare ML algorithms when each can achieve the best architectural solution given unlimited resources for hyperparameter optimization. Instead, by considering the constraints of sample efficiency, such as the number of samples that can effectively be collected from a given architectural simulator, a family of ML algorithms can be selected and compared effectively.

Third, ArchGym provides standard interfaces used by all the ML algorithms. The information passed through this interface can create a standardized dataset. This feature ensures that each experiment results in a usable artifact, irrespective of the type of ML algorithm. For instance, the diverse dataset collected from all ML agents for the same problem can be used to build a proxy model with better accuracy to overcome the sample efficiency problem of most slower architecture simulators.

6.1 Hyperparameter Lottery and Domain Specific Operators

Using ArchGym, we demonstrate that across different optimization objectives and DSE problems, *at least one* set of hyperparameters exists that results in the same performance as other ML algorithms. A poorly selected (random selection) hyperparameter for the ML algorithm or its baseline can lead to a misleading conclusion that a particular family of ML algorithms is better than another. For instance for some algorithms finding the key set of hyperparameters is easy. Note that we might still need a large sweep to find the optimal values for the key hyperparameters. A case in point is using reinforcement learning or supervised learning algorithms with enough infrastructure and research momentum to identify the important set of hyperparameters. On the other hand, for the algorithms that are fallen out of the limelight, finding the right set of algorithms requires exhaustive search or even luck to make it competitive as its baseline.

To demonstrate the existence of the hyperparameter lottery, we compare the performance of each ML agent for the same architecture optimization problem. ArchGym provides a standardization interface which makes all the agent solve the problem using the same environment, target objective. This allows for a fair apples-to-apples comparison in terms of each agent's ability to solve the task. To benchmark the agent's performance, we compare it across the following axis: (1) How the target objective affect the ML agent's performance? (2) How the complexity of the architecture system affects the ML Agent's performance? To that end, we compare against three objectives namely, power, latency, and joint objective of minimizing latency and power. Likewise for comparing against increasing complexity of the architecture system, we vary from component-level, IP-level, and SoC level. For the component level, we want to design a custom DRAM memory controller for different workload traces. For the IP-level, we aim to design a custom neural network accelerator for different neural network architectures. Lastly, for the SoC level, we aim to design a custom DSSoC for different target workloads.

Significant statistical variations. Fig. 4 shows the comparison of different ML agents (named as ACO, BO, GA, Random Walker, RL) for the architecture DSE problem of finding optimal memory controller parameters for four different memory traces namely cloud-1, cloud-2, streaming access, and random memory access. We evaluate the performance of the ML agents for three different objectives: low power, low latency, and multi-objective optimization of latency and power. As shown in Fig. 4, irrespective of the design objectives, we see a huge variance in the ML agents' performance depending upon the selected hyperparameter choice. In the worst case, there is up to 90% statistical spread (measured as the interquartile range) across different workloads and target objectives.

We observe the same trend across varying complexity of the architecture systems from component-level to SoC-level architecture exploration as shown in Fig. 5. For example, we use the streaming access workload for the DRAM memory controller. For DNN hardware accelerator design, we search for an Eyeriss-like [18] hardware accelerator for the ResNet50 model. For SoC design, we use FARSI to evaluate different SoCs for edge detection workload. Likewise for DNN mapping, we find the best mapper for ResNet18 model. Overall, on average, we perform more than 1.54 billion hardware simulations across 20 experimental setups (five for DRAMGym, TimeloopGym, FARSIGym, and MaestroGym respectively).

Effectiveness of domain specific operators. We take GAMMA [52], that uses MAESTRO [58] for simulation, as an example. GAMMA has introduced domain-specific operators, namely "Aging", "Growth", "reordering". We compare the performance of GAMMA with four variants of genetic algorithms: 'GA-V1' (GA in GAMMA), 'GA+RO' (GA with reordering only), 'GA+AG' (GA with aging only), and 'GA+GR' (GA with growing). In addition, we integrate MASTERO [58] into ArchGym for comparisons with 'GA Arch-Gym', without domain-specific operators. We perform an extensive hyperparameter sweep (~4000) experiments running for two days.

Fig. 6 summarizes the results for VGG16 and ResNet18. The results illustrate that different variants of GA are equally effective in identifying the favorable design point. Interestingly, GA in ArchGym, which does not have domain-specific operators, achieves better results than GAMMA. These results further validate that when evaluating different search algorithms, it is critical to properly tune both the algorithm and its baselines, before making any conclusions about the efficiency of algorithms.

Implications. The variation implies a number of implications. First, the choice of hyperparameter is critical not only for the ML agent of interest but also equally vital for other baselines when we compare

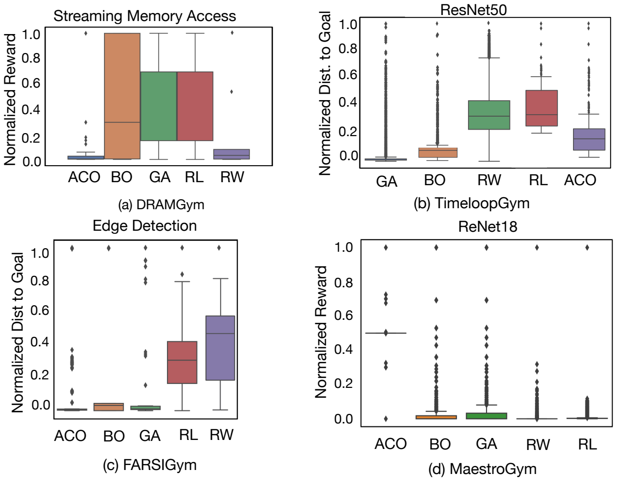


Figure 5: Hyperparameter lottery across different simulators and search algorithms for (a) DRAMGym, (b) TimeloopGym, (c) FARSIGym, and (d) MaestroGym. For TimeloopGym, FARSIGym, and MaestroGym achieving lower distance or reward is better.

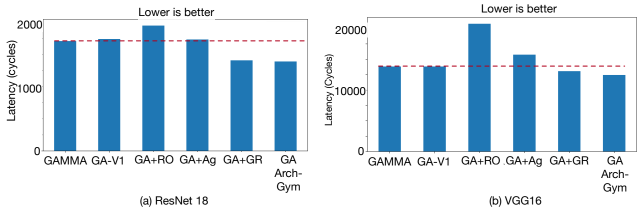


Figure 6: Comparision of latency of two ML models namely ResNet18 and VGG16 with GAMMA (with domain-specific operators) and vanilla genetic algorithm variants.

the performance of different ML algorithms for architecture design space exploration. Second, after an exhaustive hyperparameter search of 21,600 experiments for these five ML agents, we found at least one hyperparameter configuration is equally competitive to other ML agents. Though, in all likelihood, we may have missed a lot of hyperparameter combinations which makes our subset of hyperparameters akin to winning a lottery.

A call to action. The takeaway is that future ML-aided design requires us to report statistical distributions rather than report the state-of-the-art ML algorithm for a given architecture exploration problem. As we use these popular algorithms to tackle longstanding problems in architecture design space exploration, it is important to understand pitfalls. Otherwise, it is hard to operationalize the solutions in production and ensure industry adoption. Moreover, we must choose algorithms by considering domain challenges (e.g., scarcity in architecture datasets, the tradeoff between accuracy and speed with architecture simulators, etc.) rather than biasing towards any one algorithm approach since it is popular.

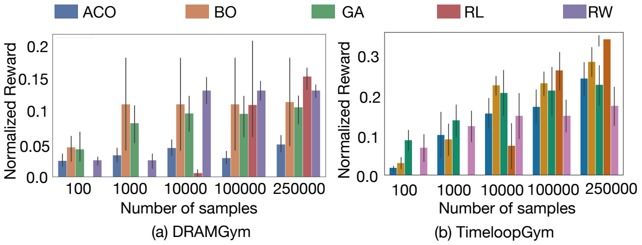


Figure 7: Mean normalized reward (target objective) of ACO, BO, GA, RL, and RW for DRAM memory controller design (DRAMGym) and ML accelerator design (TimeloopGym) in a constrained setting, limiting the number of samples accessed by an algorithm from the simulator.

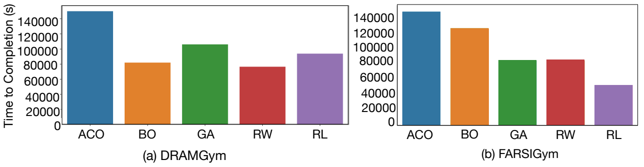


Figure 8: Comparison of ACO, BO, GA, RW, and RL in terms of time to completion for DRAMGym and FARSIGym.

6.2 Trade-off Between ML Methods

Though in Section 6.1, we demonstrated that given a free run (unlimited number of resources to search), it is possible to find at least one solution that is equally good as the others. However, as we consider the key challenges of the architecture domain, we observe certain trends that can be beneficial in selecting a particular ML algorithm for the architecture design space exploration. For instance, estimating the cost (e.g., power, latency, and area) for a given architecture parameter configuration can vary depending on the underlying architecture model, from analytical model-based (fast) to cycle-level (slow), and RTL simulation (extremely slow).

In such a scenario, how many samples we can effectively collect from the architecture model can be a useful constraint in determining different trade-offs in selecting a given ML algorithm for architecture design space exploration.

We use the number of samples from the architecture simulator as a normalization metric to compare the trade-offs between the search algorithms. We look at the number of samples we can query from simulator, in the range of 100, 1K, 100K, and 250K.

Fig. 7 compares different ML agents under sample efficiency constraints for the DRAM memory controller problem and DNN accelerator design. The x-axis denotes the number of samples we can access from the simulator. The y-axis denotes the mean normalized reward for each agent. We base our observation of the trade-offs on dividing the sample count into two regimes, namely the low sample count regime, and the high sample count regime.

Low sample count. In the low sample count regime (≤10000 samples), most ML agents perform decently well. Note that even simpler algorithms like Random walker (denoted as RW) are equally competitive to Bayesian optimization, genetic algorithm, and ant colony optimization. Finally, it is also worth noting that the performance of reinforcement learning is poor, as it is known that these algorithms are extremely sample inefficient.

Higher sample count. In the higher sample count regime (100000 samples), we observe that simpler algorithms (with exhaustive hyperparameter search) still remain competitive. However, we also see the performance of the reinforcement learning algorithm improve drastically compared to the lower sample count regime.

These results suggest that depending upon the speed of the architectural models collecting large samples can be prohibitively expensive; all popular algorithms except sample-inefficient algorithms like RL perform equally well. For an architecture model that trades speed vs. fidelity, where it is relatively easier to collect large data, we observe that emerging algorithms like reinforcement learning show increasingly better performance. However, it is important to note that other algorithms also remain equally competitive.

Time to Completion. Fig. 8 illustrates the time to completion of various agents in DRAMGym and FARSIGym environment respectively. However, this comparison is not a fair assessment as it disregards the fact that some agents, such as ACO and GA, are multi-agent algorithms with different levels of optimization. ACO, for example, takes longer to complete as it relies on sequential evaluation, whereas GA benefits from parallel evaluation. Additionally, the run time of RL is comparable to that of RW and BO, despite the fact that RL is accelerated through GPU implementation.

For these reasons, we use sample efficiency, as a better comparison point across different ML agents. Additionally, sample efficiency, unlike other metrics, such as the time to completion and the number of hardware resources required depends upon the optimization effort to parallelize and finetune the ML agent implementation. In such cases, the number of samples from the simulator is a fair comparison because it directly considers the key domain challenges, such as architecture fidelity, speed, and licensing cost, to get useful data from the simulator.

Implications. The bottleneck in ML-aided architecture design space exploration is not the ML algorithm but the sheer slowness in the high-fidelity architecture cost model. On the one hand, slower architecture cost models make applying new emerging learning-based algorithms to architecture design space exploration harder. On the other hand, a faster architecture cost model allows sample inefficient learning-based algorithms (e.g., RL) to shine.

A call to action. Given an increased momentum toward novel learning-based algorithms like reinforcement learning and offline-RL [59], we will likely see many different formulations and variants developed in the near future. Already, there are various learning frameworks [7, 24, 27, 39, 42, 61, 80] built over popular ML research infrastructures like Tensorflow [3] and Pytorch [79]. While novel learning algorithms formulations continue to use Atari-like games as a test bed, applying them to a real-world problem like architecture design exploration is challenging. Therefore, there is a need for open-source frameworks like ArchGym, that enables fair 'apples-to-apples' comparison of the efficacy of rapidly evolving learning algorithms. Finally, a community-standard approach, akin to traditional cycle-level simulators like gem5 etc., allows dataset aggregation and high-quality dataset creation, which can help create more data-driven architecture cost models (e.g., proxy ML models).

Table 4: Architectural parameters found by different search algorithms for finding a low-power DRAM memory controller (target goal: 1 Watt) design for a pointer chasing memory access pattern.

Parameter	Parameter Values				
	RL	**RW**	**BO**	**GA**	**ACO**
Page Policy	Open Adaptive	Open	Open	Open Adaptive	Open
Scheduler	Fifo	Fifo	FrFcFs	FrFcFs	FrFcFs
SchedulerBuffer	Shared	Shared	Shared	ReadWrite	Bankwise
Request Buffer Size	1	4	4	1	4
RespQueue	Reorder	Fifo	Reorder	Reorder	Fifo
Refresh Max Postponed	4	8	4	4	2
Refresh Max Pulledin	8	4	4	8	8
Arbiter	Reorder	Fifo	Reorder	Reorder	Fifo
Max Active Trans.	1	1	1	1	1

6.3 Analysis of Designed Hardware

Table 4 demonstrates the designed hardware for a DRAM memory controller across different agents. We use a memory trace with random address access (e.g., pointer chasing). The primay goal is to design a memory controller that achieves a power consumption of 1 Watt. As shown in Table 4, all the agents are able to find *at least* one design that satisfies the target power consumption. We observe that all agents keep the 'Max Active Trans.' buffer size minimal with value of one. Nonetheless, when the buffer sizes are different, for example 'Request Buffer' or 'Refresh Max Postponed', the agents reach to different 'Page Policy', 'Scheduler', and 'SchedulerBuffer' in order to achieve the same power target of 1 Watt.

7 DATASET GENERATION

Our results in Section 6.2 show that a faster architectural model can unlock novel applications of learning-based algorithms such as reinforcement learning [51] or data-driven offline reinforcement learning [89]. Indeed we notice the upward trend of using expensive learning-based algorithms [51, 64, 89, 100] for design space exploration that solely rely on analytical models [58] or proxy ML-based cost models [89], thus bypassing slow architectural simulators.

However, like all statistical models, these ML-based proxy models are imperfect and suffer high prediction errors for out-of-distribution data [38]. Thus, even though fast proxy models enable using learning-based algorithms (e.g., RL) for architecture design space exploration, the quality of designs generated from such architecture design space exploration needs more scrutiny. For instance, these proxy models continue to trade off accuracy for speed (see Section 1). Hence, in this section, we answer the question of *how to improve the accuracy of the proxy model while leveraging the gain in simulation speed.*

To that end, creating a unified interface using ArchGym for all ML agents also allows the creation of datasets that can be used to design better data-driven ML-based proxy architecture cost models to improve architecture simulator's speed. Using ArchGym datasets from DRAMGym explored across different ML agents, we construct an proxy cost model for predicting the latency, power, and energy. Our results show that two things are important to improve the accuracy of the proxy models: *Dataset size* and *Dataset diversity*. ArchGym methodology seamlessly provides a means to improve dataset size and diversity.

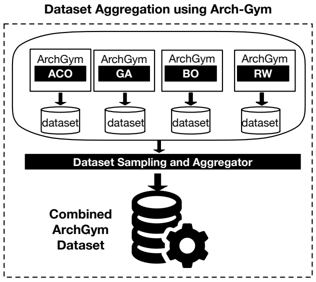
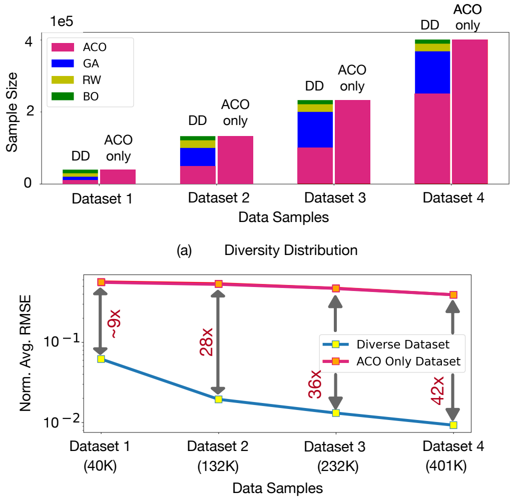


Figure 9: Dataset aggregation via ArchGym. Since all the ML agents use the same standardized interfaces, each experiment data can be leveraged to build a larger and diverse dataset.



(a) Diversity Distribution

(b) Dataset diversity and Model's RMSE Error

Figure 10: (a) Dataset characteristics and (b) its corresponding proxy model RMSE of the proxy model.

7.1 Dataset Construction

Fig. 9 shows the dataset aggregation setup using ArchGym. The information exchange between the agent and the architecture environment is logged in a standardized dataset format [77, 81] for each hyperparameter exploration study. The standardized dataset can be seamlessly merged (for size) or sampled by an ML agent type (for diversity) to construct a high-quality, large, and diverse dataset.

Using the setup shown in Fig. 9, we construct four datasets, namely 'Dataset 1', 'Dataset 2', 'Dataset 3', and 'Dataset 4'. We categorize these four datasets into two groups: 'Diverse dataset' (DD) and 'ACO-only Dataset.' As the name suggests, the data is sourced from multiple agents in the Diverse dataset. This data comes from numerous hyperparameter explorations of ACO, GA, RW, and BO. Fig. 10-a shows the exact distribution in the composition of different datasets in Diverse dataset category. In the ACO-only case, the entire dataset is constructed only from the ACO's agent. Ideally, we can use the data from any other agent as well. Also, to construct the datasets of specific sizes (for example, Dataset 1 size < Dataset 2 size), we use a random sampling utility in pandas to sample these two categories of datasets.

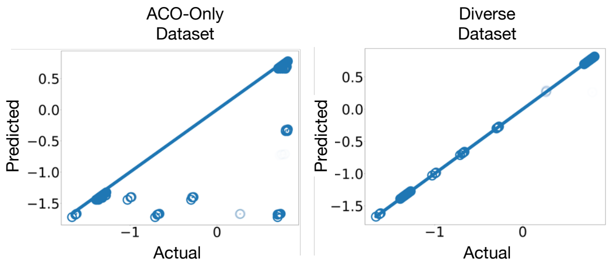


Figure 11: Comparison of actual vs predicted values for the power model between single source dataset and diverse dataset. For single source, we source all the data from one ML agent (ACO).

7.2 High-Fidelity Proxy Model Training

We use Random Forest [11] model to predict the latency, power, and energy for the data collected from the DRAMGym environment. Each target (e.g., latency) is predicted by a separate random forest regression model. The features fed into the random forest model are the DRAMGym architecture parameters (see Fig. 3 for the list of parameters). We conducted a random hyperparameter search for each model across every dataset size to obtain models that achieves the lowest root mean square error (RMSE).

7.3 Implications

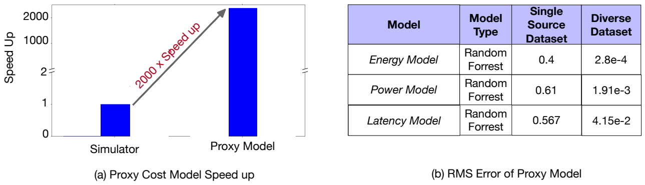
Our results show that constructing a high-fidelity proxy cost model requires not just the quantity of the data but also the diversity of the dataset. To that end, we analyze the performance of the proxy cost model as we increase the dataset size without diversity as well as increasing the dataset size with diversity.

Dataset size matters. Our results demonstrate that the dataset size plays an important role in the accuracy of the proxy model as shown in Fig. 10-b (the trend line denoted for ACO Only Dataset). While this is intuitive, it is important to note that we still need to rely on slow architecture simulators to collect large datasets, which hinders large-scale data collection.

ArchGym tackles this problem better since it inherently facilitates the aggregation of large data from all the ML agents. Although one needs to run large sweeps due to the hyperparameter lottery, we believe each exploration experiment can be a useful artifact. Whether or not each run results in a better design is inconsequential since all these exploration data can be aggregated seamlessly due to its standardization provided by ArchGym. Prior work has shown that invalid designs and other sources of sub-optimal designs can be beneficial for architecture design space exploration [89].

Dataset diversity matters. Our results demonstrate that dataset diversity also plays an important role in improving the accuracy of the proxy model as shown in Fig. 10-b (denoted by the trend line for Diverse Dataset). In fact, as the dataset size increases, the effect of sourcing data from diverse sources (in our case, different agents) is more pronounced. From aggregating data from DRAMGym, we observe that, on average, we can reduce the RMSE error by 42×.

Fig. 11 visualizes the RMSE error, comparing the values predicted by the proxy models (for energy, latency, and power) and the actual



(a) Proxy Cost Model Speed up

Model	Model Type	Single Source Dataset	Diverse Dataset
Energy Model	Random Forrest	0.4	2.8e-4
Power Model	Random Forrest	0.61	1.91e-3
Latency Model	Random Forrest	0.567	4.15e-2

(b) RMS Error of Proxy Model

Figure 12: (a) Speed-up of the ML-based proxy cost model with cycle accurate simulator as the baseline. (b) The RMSE error for proxy cost model for latency, power, and energy.

ground truth values obtained from the DRAMSys simulator. We observe a consistent trend where the actual and predicted values are less correlated when we use a dataset from a single source across different proxy models. A diverse dataset helps improve coverage of the design space, thus resulting in a higher correlation.

Narrowing the gap between speed and accuracy. Another key insight of using ArchGym for dataset aggregation is that we can bridge the gap between the speed and accuracy of architecture cost models. Using the dataset aggregated from the DRAMGym environment using ArchGym, we show that the accuracy of the proxy model is comparable to the ground truth simulator while also resulting in 2000× speed up as shown in Fig. 12. These high-accuracy yet speedy architecture cost models can be used to explore new and emerging learning-based algorithms such as reinforcement learning, offline-RL [59], and multi-agent reinforcement learning, which were limited by the slowness in architectural simulators.

While we demonstrate this using DRAMGym, we believe ArchGym natively provides the much-needed diversity in the dataset aggregation process for other slower architecture simulators. Furthermore, since all ML agents will have a different policy (see Fig. 2 in Section 4), the way they explore the design space is also different. Thus, the exploration dataset aggregated through ArchGym across different ML agents (whether run by an individual researcher or community-wide adoption followed by aggregation) helps create a diverse dataset. These key features, in turn, improve the accuracy of the proxy model.

8 DISCUSSION ON EXTENDING ARCHGYM

Integrating other proxy models. We demonstrate how ArchGym can aid in creation of standardized and diverse datasets, which can be easily aggregated to balance the trade-off between accuracy and speed. By utilizing an accurate and high-speed proxy model, we can augment conventional slower architectural simulators while retaining their original interfaces. This enables us to leverage machine learning algorithms, including data-driven offline learning methods [57] or offline reinforcement learning [59], within ArchGym. Since ArchGym interfaces capture complex data, such as compiled IR or XLA graphs, we can train other deep learning models, such as GNNs [55], that achieve high accuracy. Regardless of the proxy model type, all models can be encapsulated using the same interface.

Integrating other algorithms. ArchGym provides a unified interface not only for integrating hardware cost models, but also for integrating search algorithms or industry grade frameworks [32]. Since each search algorithm (or agent) can be represented as a combination of a policy and hyperparameters (see Section 4), any new search algorithm can be abstracted and integrated into ArchGym. Additionally, unified search spaces, such as hardware-aware NAS [60], require querying the hardware cost model (e.g., a simulator or real hardware) for state and receiving feedback based on optimization objectives. These problems can also be mapped into ArchGym (see Figure 1).

9 CONCLUSION

ArchGym is an open source gymnasium for ML-aided architecture design space exploration. Using our framework, we show the existence of the hyperparameter lottery. Moreover, ArchGym provides a standardized interface that can be extended and allows for fair comparison between different ML algorithms for a given architecture exploration problem. The standard interfaces for all ML algorithms further enable the creation of diverse datasets that can be used to explore novel, data-driven offline learning algorithms and build fast architectural cost models with high-fidelity.

ACKNOWLEDGMENTS

The authors would like to thank anonymous reviewers for their feedback on improving the manuscript's quality. Specifically, we would like to thank Sheng-Chun Kao for his help in understanding GAMMA codebase. We would like to thank Prof. Radhika Nagpal for helping us understand Ant-conlony optimization and Genetic algorithms. We would also like to thank Natasha Jaques, Shayegan Omidshafiei, Izzeddin Gur from Google Research for helpful discussions on reinforcement learning. We extend our gratitude towards Cliff Young, James Laudon, and extended Google Research, Brain Team for their feedback and comments. We also thank Douglas Eck and Hardik Sharma for feedback on the early draft of this work. This work is supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), via 2022-21102100013. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of ODNI, IARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

A ARTIFACT APPENDIX

A.1 Abstract

This section summarizes the artifact evaluation for this work. First, we provide the check-list for this artifact. Next, we describe the directory structure for the code. Finally, the installation, experiment workflow, and evaluation illustrate how to use the artifact to reproduce some of the key results.

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

- **Algorithm:** Random Walker, Ant-Colony optimization, Genetic Algorithm, Bayesian Optimization, Reinforcement Learning
- **Program:** Python, Docker, Colab
- **Data set:** ArchGym dataset for DRAMGym
- **OS environment:** Ubuntu 18.04
- **Hardware:** Intel Server CPUs

- **Execution:** Sole user.
- **Metrics:** Reward, Normalized Average RMSE, RMSE Error
- **Output:** Experiments produce outputs in console and generates log files. The log files will generate trajectory information.
- **How much disk space required (approximately)?:** 128 GB
- **How much time is needed to prepare workflow (approximately)?:** 24 hours
- **How much time is needed to complete experiments (approximately)?:** We ran a total of 21,000 experiments across three different enviornments in distributed cluster inside Google. The scheduling and parallelization code for cluster level hyperameter sweep is proprietary to Google. However, we provide a simple workflow with DRAMGym. This is the fastest (less than one week) and something that can run on local machine within a week and still reproduce some of the results. The results in Section VII runs on Google colab and should take less than a day to finish.
- **Publicly available?:** Yes, will be made public.
- **Workflow framework used?:** Docker, Google Colab
- **Archived (provide DOI)?: 10.5281/zenodo.7791715**

A.3 Description

A.3.1 How to access. The code required for artificat evaluation is hosted here: https://github.com/srivatsankrishnan/iscaae. We have also published the initial artifact on Zenodo. The Zenodo DOI URL is: https://doi.org/10.5281/zenodo.7791592

A.3.2 Hardware dependencies. To reproduce some of the results found in this paper we suggest using commensurate hardware. We also provide a simple to use colab to reporduce results in Section VII.

A.3.3 Software dependencies. We provide a dockerized environment to run the training experiments. The repository has a build script that will build the docker image with all the dependencies.

A.3.4 Data sets. ArchGym trajectories are available in the repository (https://doi.org/10.5281/zenodo.7791592). Upon downloading and unzipping the repository, the dataset should be under the 'archgym-trajectory' folder. The colab has a cell to upload this dataset to reproduce results in Section VII.

A.4 Installation

A.4.1 Installation for DRAMGym Training. For the ease of artifact evaluation, we provide sample bash scripts that automate launching and analyzing experiments. To install the necessary packages we provide a Docker build script (in iscaae/sims/DRAM/build.sh). Upon running this script, a docker image with all the software dependencies to run Ant-Colony optimization, Bayesian optimization, Genetic Algorithm, Reinforcement Learning, and Random Walker will be created.

Key steps include:

- Download the artifact from: https://doi.org/10.5281/zenodo.7791592
- This will download a zip file named 'isca-ae.zip'. Please extract it. It should have three sub-folders named: 'archgym-trajectory', 'iscaae', and 'colab_notebook'.
- To run the training code for DRAMGym, please first build the docker image. Please use sudo before the script if you have not setup the docker for sudoless workflow.
  `(cd isca-ae/iscaae/sims/DRAM/; ./build.sh)`
- Enter the docker image using the following command:

```
sudo docker run \
    -entrypoint /bin/bash \
    -it dram_arch_gym
```

A.4.2 Installation for Proxy Model. All packages necessary to reproduce the proxy models are preinstalled within Google Colab. Please upload the python notebook
`Reprod_Arch_Gym_Proxy_Models.ipynb`

Upload the notebook (`cd iscaae/colab_notebook`) into a Google colab workspace The dataset is obtained from training DRAMGym with exhaustive hyperparameter sweeps running over several weeks. To artifact evaluation, we provide the trajectories (`cd iscaae/archgymtrajectory`) as the starting point for training the proxy model. Please untar the tarball and load it into the colab.

A.5 Experiment workflow

A.5.1 Training Agents for DRAMGym. ArchGym provides five agents namely random walker, Ant colony optimization, genetic algorithm, reinforcement learning, and Bayesian optimization for architecture design space exploration.

Random Walker Agent. To train a random walker agent with DRAMGym, please run the following command.

`python launch_gcp.py -algo=random_walk`

This should start the training and generate log files named `random_walker_logs`.

Bayesian Optimization. To train a Bayesian optimizaion agent with DRAMGym, please run the following command.

`python launch_gcp.py -algo=bo`

This should start the training and generate log files named `bo_logs`.

Ant Colony Optimization. To train a Ant-Colony optimization agent with DRAMGym, please run the following command.

`python launch_gcp.py -algo=aco`

This should start the training and generate log files named `aco_logs`.

Reinforcement Learning. To train a reinforcement learning agent with DRAMGym, please run the following command.

`python launch_gcp.py -algo=rl` This should start the training and generate log files named
`Algo_ppo_....._rewardscale_false`.

Genetic Algorithm. To train a genetic algorithm agent with DRAMGym, please run the following command.

`python launch_gcp.py -algo=ga`

This should start the training and generate log files named `ga_logs`

A.6 Training Proxy Model

To reproduce the proxy model Figures 10, Figure 11, and Figure 12 found in section VII, please upload the `Reprod_Arch_Gym_Proxy_Models.ipynb` to Google colab. You will also need to upload the `proxy_model_data_subsettar.gz` file. However, the steps to upload the tar file are found within the `Reprod_Arch_Gym_Proxy_Models.ipynb` itself.

To ensure accurate reproduction of results, follow the steps outlined in the `.ipynb` script sequentially. The script is comprehensive, encompassing all necessary steps - from data loading and transformation to model training and figure generation. Please refrain from skipping any cells or executing them out of order. The script proceeds as follows:

(1) Importing all required dependencies.
(2) Prompting the user to upload the `.tar.gz` file.
(3) Loading and cleaning the data.
(4) Sampling the data into eight sub-datasets and applying appropriate transformations.
(5) Training energy, power, and latency models for each of the eight sub-datasets.
(6) Displaying the Mean Squared Error (MSE) for each model and saving each model.
(7) Loading the saved models and generating the three figures mentioned above.

A.7 Evaluation and expected results

Hyperparameter Lottery. The goal of hyperameter lottery is to show that all ML algorithms are equally probable find optimial solution. For DRAMGym, the goal is to obtain an optimal memory controller resource allocation that achieves a power target of 1 W for random memory address trace. The maximum possible reward (based on the target objective for power in Table III) is 346.02). A simple grep on the log files for each agent (after the completion of the experiments) should show that all the agents are able to find the best design (i.e., 1 Watts). On a normalized reward scale, this should corresponds to the Fig 4-(a) (Low Power) for 'random' memory trace.

Proxy Model Training. Reproducing the proxy models involves training several random forest models, which may take in total approximately 1-2 hours to complete. Once all models have been trained and saved, one can expect to reproduce Figures 10, Figure 11, and Figure 12 found in section VII,

A.8 Experiment customization

The ArchGym framework facilitates adding new agents and new environments, workloads, optimization objectives etc. The open source version will have examples on how to easily integrate custom architectural cost models and new agents.

A.9 Notes

Please be aware that that current workflow is intended to faciliate artifact evaluation of this paper on a local machine within one week. For that purpose, we have shown examples of how to reproduce the training scripts and results for DRAMGym for one memory trace. To enable large scale hyperparameter sweep studies for slower simulator/cost model such as Timeloop and FARSI would require distributed cluster and significant hardware resources.

Please be aware that supplementary steps, such as data exploration, hyperparameter tuning, and experimentation with alternative machine learning models, have been omitted from the `Reprod_Arch_Gym_Proxy_Models.ipynb` file. This decision was made to streamline the reproduction process for the figures presented in Section VII.

A.10 Methodology

Submission, reviewing and badging methodology:

- Artifact Review Badging
- Artifact Submission Guide
- Artifact Reviewing Guidelines

REFERENCES

[1] [n. d.]. Scikit-opt. https://scikit-opt.github.io/

[2] [n. d.]. Scikit-Optimize. https://scikit-optimize.github.io/

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vol. 16. Savannah, GA, USA, 265–283.

[4] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil J Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. 2022. Do As I Can, Not As I Say: Grounding Language in Robotic Affordances. *arXiv preprint arXiv:2204.01691* (2022).

[5] Dario Amodei, Girish Sastry Danny Hernandez, Jack Clark, Greg Brockman, and Ilya Sutskever. 2018. AI and Compute. https://openai.com/blog/ai-and-compute/

[6] Yehia Arafa, Abdel-Hameed A Badawy, Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. 2019. Ppt-gpu: Scalable gpu performance modeling. *IEEE Computer Architecture Letters* (2019), 55–58.

[7] Dominik Bauer, Timothy Patten, and Markus Vincze. 2021. Reagent: Point Cloud Registration Using Imitation and Reinforcement Learning. In *Conference on Computer Vision and Pattern Recognition (CVPR).*

[8] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research* 13, 10 (2012), 281–305. http://jmlr.org/papers/v13/bergstra12a.html

[9] Kshitij Bhardwaj, Marton Havasi, Yuan Yao, David M. Brooks, José Miguel Hernández Lobato, and Gu-Yeon Wei. 2019. Determining Optimal Coherency Interface for Many-Accelerator SoCs Using Bayesian Optimization. *IEEE Computer Architecture Letters (CAL)* (2019).

[10] Behzad Boroujerdian, Ying Jing, Devashree Tripathy, Amit Kumar, Lavanya Subramanian, Luke Yen, Vincent Lee, Vivek Venkatesan, Amit Jindal, Robert Shearer, et al. 2023. FARSI: An early-stage design space exploration framework to tame the domain-specific system-on-chip complexity. *ACM Transactions on Embedded Computing Systems* 22, 2 (2023), 1–35.

[11] Leo Breiman. 2001. Random Forests. *Machine learning* 45, 1 (2001), 5–32.

[12] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems (NeuRIPS* 33 (2020), 1877–1901.

[13] Jacob Buckman, Danijar Hafner, George Tucker, Eugene Brevdo, and Honglak Lee. 2018. Sample-Efficient Reinforcement Learning with Stochastic Ensemble Value Expansion. In *Conference on Neural Information Processing Systems (NeurIPS.*

[14] Edvinas Byla and Wei Pang. 2019. DeepSwarm: Optimising Convolutional Neural Networks using Swarm Intelligence. In *UK Workshop on Computational Intelligence.*

[15] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[16] Leandro Nunes de Castro. 2007. Fundamentals of Natural Computing (Chapman & Hall/Crc Computer and Information Sciences).

[17] Hongming Chen, Ola Engkvist, Yinhai Wang, Marcus Olivecrona, and Thomas Blaschke. 2018. The Rise of Deep Learning in Drug Discovery. *Drug discovery today* (2018).

[18] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *The 42nd Annual International Symposium on Computer Architecture (ISCA).*

[19] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar

Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling Language Modeling with Pathways.

[20] NVIDIA Corporation. 2022. NVIDIA H100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/h100/

[21] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, et al. 2022. Compilergym: Robust, performant compiler optimization environments for ai research. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 92–105.

[22] Miguel de Prado, Andrew Mundy, Rabia Saeed, Maurizo Denna, Nuria Pazos, and Luca Benini. 2020. Automated design space exploration for optimized deployment of dnn on arm cortex-a cpus. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40, 11 (2020), 2293–2305.

[23] Jonas Degrave, Federico Felici, Jonas Buchli, Michael Neunert, Brendan Tracey, Francesco Carpanese, Timo Ewalds, Roland Hafner, Abbas Abdolmaleki, Diego de las Casas, Craig Donner, Leslie Fritz, Cristian Galperti, Andrea Huber, James Keeling, Maria Tsimpoukelli, Jackie Kay, Antoine Merle, Jean-Marc Moret, Seb Noury, Federico Pesamosca, David Pfau, Olivier Sauter, Cristian Sommariva, Stefano Coda, Basil Duval, Ambrogio Fasoli, Pushmeet Kohli, Koray Kavukcuoglu, Demis Hassabis, and Martin Riedmiller. 2022. Magnetic Control of Tokamak Plasmas Through Deep Reinforcement Learning. *Nature* (2022).

[24] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. 2017. OpenAI Baselines.

[25] Lieven Eeckhout, Robert H. Bell Jr., Bastiaan Stougie, Koen De Bosschere, and Lizy K. John. 2004. Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies. In *Proceedings of the 31st Annual International Symposium on Computer Architecture* (München, Germany) *(ISCA '04)*. IEEE Computer Society, USA, 350.

[26] Ahmed T Elthakeb, Prannoy Pilligundla, Fatemehsadat Mireshghallah, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. 2020. ReLeQ: A Reinforcement Learning Approach for Deep Quantization of Neural Networks. *IEEE Micro* (2020).

[27] Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. 2019. Seed RL: Scalable and Efficient Deep-RL with Accelerated Central Inference. *arXiv preprint arXiv:1910.06591* (2019).

[28] Matthias Feurer and Frank Hutter. 2019. Hyperparameter optimization. *Automated machine learning: Methods, systems, challenges* (2019), 3–33.

[29] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. 2018. A configurable cloud-scale DNN processor for real-time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1–14.

[30] Yiheng Gao and Benjamin Carrion Schafer. 2021. Effective high-level synthesis design space exploration through a novel cost function formulation. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–5.

[31] Simon Goss, Serge Aron, Jean-Louis Deneubourg, and Jacques Marie Pasteels. 1989. Self-Organized Shortcuts in the Argentine Ant. *Naturwissenschaften* (1989).

[32] Sergio Guadarrama, Anoop Korattikara, Oscar Ramirez, Pablo Castro, Ethan Holly, Sam Fishman, Ke Wang, Ekaterina Gonina, Neal Wu, Efi Kokiopoulou, Luciano Sbaiz, Jamie Smith, Gábor Bartók, Jesse Berent, Chris Harris, Vincent Vanhoucke, and Eugene Brevdo. 2018. TF-Agents: A library for Reinforcement Learning in TensorFlow. https://github.com/tensorflow/agents. https://github.com/tensorflow/agents [Online; accessed 25-June-2019].

[33] Gagan Gupta, Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. 2017. Kickstarting semiconductor innovation with open source hardware. *Computer* 50, 6 (2017), 50–59.

[34] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*. PMLR, 1861–1870.

[35] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362.

[36] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning memory access patterns. In *International Conference on Machine Learning*. PMLR, 1919–1928.

[37] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W Fletcher. 2021. Mind mappings: enabling efficient algorithm-accelerator mapping space search. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 943–958.

[38] Dan Hendrycks and Kevin Gimpel. 2017. A Baseline for Detecting Misclassified and Out-of-Distribution Examples in Neural Networks. In *International Conference on Learning Representations*.

[39] Matthew W. Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Nikola Momchev, Danila Sinopalnikov, Piotr Stańczyk, Sabela Ramos, Anton Raichuk, Damien Vincent, Léonard Hussenot, Robert Dadashi, Gabriel Dulac-Arnold, Manu Orsini, Alexis Jacq, Johan Ferret, Nino Vieillard, Seyed Kamyar Seyed Ghasemipour, Sertan Girgin, Olivier Pietquin, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, Sarah Henderson, Abe Friesen, Ruba Haroun, Alex Novikov, Sergio Gómez Colmenarejo, Serkan Cabi, Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Andrew Cowie, Ziyu Wang, Bilal Piot, and Nando de Freitas. 2020. ACME: A Research Framework for Distributed Reinforcement Learning. *arXiv preprint arXiv:2006.00979* (2020).

[40] Ramsey Hourani, Ravi Jenkal, W Rhett Davis, and Winser Alexander. 2009. Automated Design Space Exploration for DSP Applications. *Journal of Signal Processing Systems* (2009).

[41] Qijing Huang, Charles Hong, John Wawrzynek, Mahesh Subedar, and Yakun Sophia Shao. 2022. Learning A Continuous and Reconstructible Latent Space for Hardware Accelerator Design. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 277–287.

[42] Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, and Jeff Braga. 2021. CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms. *arXiv preprint arXiv:2111.08819* (2021).

[43] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. 2008. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, USA, 39–50. https://doi.org/10.1109/ISCA.2008.21

[44] Tang Jie and Pieter Abbeel. 2010. On a connection between importance sampling and the likelihood ratio policy gradient. *Advances in Neural Information Processing Systems* 23 (2010).

[45] Hadi S. Jomaa, Josif Grabocka, and Lars Schmidt-Thieme. 2019. Hyp-RL : Hyperparameter Optimization by Reinforcement Learning. *arXiv preprint arXiv:1906.11527* (2019).

[46] Norman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff Young, Xiang Zhou, Zongwei Zhou, and David Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. arXiv:2304.01433 [cs.AR]

[47] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, and Jonathan Ross. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

[48] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. 2021. Highly Accurate Protein Structure Prediction with AlphaFold. *Nature* (2021).

[49] Matthias Jung, Christian Weis, and Norbert Wehn. 2015. DRAMSys: A Flexible DRAM Subsystem Design Space Exploration Framework. *IPSJ Transactions on System LSI Design Methodology* (2015).

[50] Eunsuk Kang, Ethan Jackson, and Wolfram Schulte. 2010. An Approach for Effective Design Space Exploration. In *Monterey Workshop*.

[51] Sheng-Chun Kao, Geonhwa Jeong, and Tushar Krishna. 2020. Confuciux: Autonomous hardware resource assignment for dnn accelerators using reinforcement learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 622–636.

[52] Sheng-Chun Kao and Tushar Krishna. 2020. Gamma: Automating the hw mapping of dnn models on accelerators via genetic algorithm. In *Proceedings of the 39th International Conference on Computer-Aided Design*. 1–9.

[53] Tejas S. Karkhanis and James E. Smith. 2004. A First-Order Superscalar Processor Model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture* (München, Germany) *(ISCA '04)*. IEEE Computer Society, USA, 338.

[54] Sam Kaufman, Phitchaya Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. 2021. A learned performance model for tensor processing units. *Proceedings of Machine Learning and Systems* 3 (2021), 387–400.

[55] Samuel Kaufman, Phitchaya Mangpo Phothilimthana, and Mike Burrows. 2019. Learned TPU cost model for XLA tensor programs. In *Proc. Workshop ML Syst. NeurIPS*. 1–6.

[56] Srivatsan Krishnan, Zishen Wan, Kshitij Bhardwaj, Paul Whatmough, Aleksandra Faust, Sabrina Neuman, Gu-Yeon Wei, David Brooks, and Vijay Janapa Reddi. 2022. Automatic Domain-Specific SoC Design for Autonomous Unmanned Aerial Vehicles. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 300–317.

[57] Aviral Kumar, Amir Yazdanbakhsh, Milad Hashemi, Kevin Swersky, and Sergey Levine. 2022. Data-Driven Offline Optimization For Architecting Hardware Accelerators. In *International Conference on Learning Representations (ICLR)*.

[58] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. 2020. Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings. *IEEE micro* 40, 3 (2020), 20–29.

[59] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. 2020. Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems. *CoRR* abs/2005.01643 (2020). arXiv:2005.01643 https://arxiv.org/abs/2005.01643

[60] Chaojian Li, Zhongzhi Yu, Yonggan Fu, Yongan Zhang, Yang Zhao, Haoran You, Qixuan Yu, Yue Wang, Cong Hao, and Yingyan Lin. 2021. {HW}-{NAS}-Bench: Hardware-Aware Neural Architecture Search Benchmark. In *International Conference on Learning Representations*.

[61] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2018. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*. PMLR, 3053–3062.

[62] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, and journal=arXiv preprint arXiv:1509.02971 Tassa, Yuval and Silver, David and Wierstra, Daan, year = 2015. [n. d.]. Continuous Control with Deep Reinforcement Learning. ([n. d.]).

[63] Ting-Ru Lin, Drew Penney, Massoud Pedram, and Lizhong Chen. 2019. Optimizing Routerless Network-on-Chip Designs: An Innovative Learning-based Framework. *arXiv preprint arXiv:1905.04423* (2019).

[64] Ting-Ru Lin, Drew Penney, Massoud Pedram, and Lizhong Chen. 2020. A deep reinforcement learning framework for architectural exploration: A routerless NoC case study. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 99–110.

[65] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jerónimo Castrillón, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc S. Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152* (2020).

[66] Wesley J Maddox, Maximilian Balandat, Andrew G Wilson, and Eytan Bakshy. 2021. Bayesian optimization with high-dimensional outputs. *Advances in neural information processing systems* 34 (2021), 19274–19287.

[67] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (2002), 50–58.

[68] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. 2018. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 522–531.

[69] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on machine learning*. PMLR, 4505–4515.

[70] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, et al. 2021. A Graph Placement Methodology for Fast Chip Design. *Nature* (2021).

[71] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *arXiv preprint arXiv:1312.5602* (2013).

[72] Jonas Močkus. 1975. On Bayesian Methods for Seeking the Extremum. In *Optimization techniques IFIP technical conference*. Springer.

[73] Vittoriano Muttillo, Paolo Giammatteo, Giuseppe Fiorilli, and Luigi Pomante. 2020. An OpenMP Parallel Genetic Algorithm for Design Space Exploration of Heterogeneous Multi-processor Embedded Systems. In *PARMA-DITAM*.

[74] Vu Nguyen. 2019. Bayesian optimization for accelerating hyper-parameter tuning. In *2019 IEEE second international conference on artificial intelligence and knowledge engineering (AIKE)*. IEEE, 302–305.

[75] Derek B Noonburg and John P Shen. 1994. Theoretical modeling of superscalar processor performance. In *Proceedings of the 27th annual international symposium on Microarchitecture*. 52–62.

[76] Mark Oskin, Frederic T. Chong, and Matthew Farrens. 2000. HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Designs. In *Proceedings of the 27th Annual International Symposium on Computer Architecture* (Vancouver, British Columbia, Canada) *(ISCA '00)*. Association for Computing Machinery, New York, NY, USA, 71–82. https://doi.org/10.1145/339647.339656

[77] David Paper and David Paper. 2021. TensorFlow Datasets. *State-of-the-Art Deep Learning Models in TensorFlow: Modern Machine Learning in the Google Colab Ecosystem* (2021).

[78] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. 2019. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 304–315.

[79] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[80] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. 2019. Stable Baselines3.

[81] Sabela Ramos, Sertan Girgin, Léonard Hussenot, Damien Vincent, Hanna Yakubovich, Daniel Toyama, Anita Gergely, Piotr Stanczyk, Raphael Marinier, Jeremiah Harmsen, et al. 2021. RLDS: an Ecosystem to Generate, Share and Use Datasets in Reinforcement Learning. *arXiv preprint arXiv:2111.02767* (2021).

[82] Ravishankar Rao, Mark H Oskin, and Frederic T Chong. 2002. Hlspower: Hybrid statistical modeling of the superscalar power-performance design space. In *High Performance Computing—HiPC 2002: 9th International Conference Bangalore, India, December 18–21, 2002 Proceedings 9*. Springer, 620–629.

[83] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[84] Brandon Reagen, José Miguel Hernández-Lobato, Robert Adolf, Michael Gelbart, Paul Whatmough, Gu-Yeon Wei, and David Brooks. 2017. A case for efficient accelerator design space exploration via bayesian optimization. In *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 1–6.

[85] Rajarshi Roy, Jonathan Raiman, Neel Kant, Ilyas Elkin, Robert Kirby, Michael Siu, Stuart Oberman, Saad Godil, and Bryan Catanzaro. 2021. Prefixrl: Optimization of parallel prefix circuits using deep reinforcement learning. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 853–858.

[86] Ananda Samajdar, Jan Moritz Joseph, Matthew Denton, and Tushar Krishna. 2021. AIRCHITECT: Learning Custom Architecture Design and Mapping Space. *arXiv preprint arXiv:2108.08295* (2021).

[87] Benjamin Carrion Schafer. 2017. Parallel high-level synthesis design space exploration for behavioral ips of exact latencies. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 22, 4 (2017), 1–20.

[88] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *CoRR* abs/1707.06347 (2017). arXiv:1707.06347 http://arxiv.org/abs/1707.06347

[89] Kiran Seshadri, Berkin Akin, James Laudon, Ravi Narayanaswami, and Amir Yazdanbakhsh. 2022. An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 79–91.

[90] Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn, and Pablo Villalobos. 2022. Compute Trends Across Three Eras of Machine Learning. *arXiv preprint arXiv:2202.05924* (2022).

[91] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th*

International Conference on Architectural Support for Programming Languages and Operating Systems (San Jose, California) *(ASPLOS X)*. Association for Computing Machinery, New York, NY, USA, 45–57. https://doi.org/10.1145/605397.605403

[92] Zhan Shi, Chirag Sakhuja, Milad Hashemi, Kevin Swersky, and Calvin Lin. 2020. Learned Hardware/Software Co-Design of Neural Accelerators. *arXiv preprint arXiv:2010.02075* (2020).

[93] Ghassan Shobaki, Vahl Scott Gordon, Paul McHugh, Theodore Dubois, and Austin Kerbow. 2022. Register-Pressure-Aware instruction scheduling using ant colony optimization. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 2 (2022), 1–23.

[94] Greg Snider. 2001. Spacewalker: Automated Design space Exploration for Embedded Computer Systems. *HP Labs Palo Alto HPL-2001-220* (2001).

[95] Lukas Steiner, Matthias Jung, Felipe S Prado, Kirill Bykov, and Norbert Wehn. 2020. DRAMSys4. 0: a fast and cycle-accurate systemC/TLM-based DRAM simulator. In *Embedded Computer Systems: Architectures, Modeling, and Simulation: 20th International Conference, SAMOS 2020, Samos, Greece, July 5–9, 2020, Proceedings 20*. Springer, 110–126.

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

[97] Ondřej Sỳkora, Phitchaya Mangpo Phothilimthana, Charith Mendis, and Amir Yazdanbakhsh. 2022. GRANITE: A Graph Neural Network Model for Basic Block Throughput Estimation. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 14–26.

[98] Synopsys. 2022. Deliver Better, Faster, Cheaper Semiconductors with DSO.ai. https://www.synopsys.com/implementation-and-signoff/ml-ai-design/dso-ai.html

[99] Thierry Tambe, En-Yu Yang, Glenn G Ko, Yuji Chai, Coleman Hooper, Marco Donato, Paul N Whatmough, Alexander M Rush, David Brooks, and Gu-Yeon Wei. 2021. 9.8 A 25mm2 SoC for IoT Devices with 18ms Noise-Robust Speech-to-Text Latency via Bayesian Speech Denoising and Attention-Based Sequence-to-Sequence DNN Speech Recognition in 16nm FinFET. In 2021 IEEE International Solid-State Circuits Conference (ISSCC), Vol. 64. 158–160.

[100] Hanrui Wang, Kuan Wang, Jiacheng Yang, Linxiao Shen, Nan Sun, Hae-Seung Lee, and Song Han. 2020. GCN-RL circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[101] Hanrui Wang, Jiacheng Yang, Hae-Seung Lee, and Song Han. 2018. Learning to design circuits. *arXiv preprint arXiv:1812.02734* (2018).

[102] Zirui Wang, Shangzhe Wu, Weidi Xie, Min Chen, and Victor Adrian Prisacariu. 2021. NeRF−−: Neural Radiance Fields without Known Camera Parameters. *arXiv preprint arXiv:2102.07064* (2021).

[103] Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, and Tony Nowatzki. 2021. UNIT: Unifying tensorized instruction compilation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 77–89.

[104] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S Emer. 2021. Sparseloop: An analytical, energy-focused design space exploration methodology for sparse tensor accelerators. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 232–234.

[105] Roland E Wunderlich, Thomas F Wenisch, Babak Falsafi, and James C Hoe. 2003. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th annual international symposium on Computer architecture*. 84–97.

[106] Feng Xia, Jiaying Liu, Hansong Nie, Yonghao Fu, Liangtian Wan, and Xiangjie Kong. 2019. Random walks: A review of algorithms and applications. *IEEE Transactions on Emerging Topics in Computational Intelligence* 4, 2 (2019), 95–107.

[107] Li Yang and Abdallah Shami. 2020. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing* 415 (2020), 295–316. https://doi.org/10.1016/j.neucom.2020.07.061

[108] Dan Zhang, Safeen Huda, Ebrahim Songhori, Kartik Prabhu, Quoc Le, Anna Goldie, and Azalia Mirhoseini. 2022. A full-stack search technique for domain optimized deep learning accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 27–42.

[109] Jun Zhang, Henry Shu-Hung Chung, Alan Wai-Lun Lo, and Tao Huang. 2008. Extended ant colony optimization algorithm for power electronic circuit design. *IEEE Transactions on power Electronics* 24, 1 (2008), 147–162.

[110] Matthew M Ziegler, Hung-Yi Liu, George Gristede, Bruce Owens, Ricardo Nigaglioni, and Luca P Carloni. 2016. A synthesis-parameter tuning system for autonomous design-space exploration. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1148–1151.