

A Survey of High-Level Modeling and Simulation Methods for Modern Machine Learning Workloads

MICRO 2026 Submission #NaN – Confidential Draft – Do NOT Distribute!!

Abstract

Machine learning-based performance modeling has emerged as a powerful alternative to traditional analytical models and cycle-accurate simulators for predicting computer system behavior. This survey focuses specifically on *ML techniques* for performance prediction across CPUs, GPUs, accelerators, and distributed systems, covering over 60 papers from architecture and ML venues published between 2016–2025. We position traditional analytical models (Timeloop, MAESTRO) and simulators (gem5, GPGPU-Sim) as *baselines* that ML approaches aim to replace or augment. We propose an eight-dimension taxonomy organizing ML approaches by modeling technique, target hardware, workload types, prediction targets, accuracy metrics, input requirements, evaluation scope, and reproducibility. Our analysis reveals that specialized ML models achieve remarkable accuracy—below 5% error for narrow domains—while general-purpose models trade accuracy for broader applicability. Transfer learning and meta-learning techniques increasingly enable adaptation to new hardware with minimal profiling, addressing the challenge of hardware diversity. We identify key open challenges including benchmark diversity, cross-platform generalization, and integration with compiler and architecture exploration workflows. Hybrid approaches combining analytical structure with learned components represent the most promising direction, offering both interpretability and accuracy. This survey provides practitioners guidance for selecting appropriate ML techniques and researchers a roadmap for advancing the field.

Keywords

machine learning, performance modeling, computer architecture, neural networks, survey

1 Introduction

Performance modeling is fundamental to computer architecture research and development. Architects rely on accurate performance predictions to navigate vast design spaces, optimize hardware-software co-design, and make informed decisions about resource allocation. Traditional approaches—analytical models [14] and cycle-accurate simulators [3]—have served the community well, but face growing challenges as workloads and hardware become increasingly complex. Analytical models often oversimplify system behavior, while simulators can require hours or days to evaluate a single design point, making exhaustive exploration impractical.

The rise of deep learning workloads has intensified these challenges. Modern neural networks exhibit diverse computational patterns—from dense matrix operations in transformers to sparse irregular accesses in graph neural networks—that stress traditional modeling assumptions. Simultaneously, hardware diversity has

exploded: GPUs, TPUs, custom accelerators, and multi-device distributed systems each present unique performance characteristics that resist unified analytical treatment. This complexity has motivated a new generation of *machine learning-based* performance models that learn predictive functions directly from profiling data.

ML-based performance modeling has emerged as a compelling alternative. Learned models can capture complex, non-linear relationships between workload characteristics and hardware behavior that elude closed-form analysis. Recent work demonstrates remarkable accuracy: NeuSight [11] achieves 2.3% error predicting GPT-3 latency on H100 GPUs, while nn-Meter [18] reaches 99% accuracy for edge device latency prediction. Beyond accuracy, these approaches offer practical benefits: models trained on one platform can transfer to new hardware with minimal adaptation [6], and inference-time predictions complete in milliseconds rather than hours.

This survey provides a comprehensive analysis of ML-based performance modeling techniques for computer architecture. We focus specifically on *learned* models that acquire predictive capability from data, positioning traditional analytical and simulation approaches as baselines that contextualize ML advances. We make the following contributions:

- A **taxonomy** organizing ML approaches along eight dimensions: modeling technique, target hardware, workload types, prediction targets, accuracy metrics, input requirements, evaluation scope, and reproducibility.
- A **systematic survey** of over 60 ML-based performance modeling papers from architecture venues (MICRO, ISCA, HPCA, ASPLOS) and ML venues (MLSys, NeurIPS, ICML) published between 2016–2025.
- A **comparative analysis** examining trade-offs between accuracy, training cost, generalization, and interpretability across ML approaches.
- An identification of **open challenges** including data scarcity, cross-platform generalization, and integration with design automation flows.

The remainder of this paper is organized as follows. Section 2 provides background on traditional performance modeling and relevant ML techniques. Section 3 presents our classification taxonomy. Section 4 surveys approaches organized by target hardware platform. Section 5 offers comparative analysis across key dimensions. Section 6 discusses open challenges and future directions. Section 7 presents hands-on reproducibility evaluations of representative tools. Section 8 concludes.

Figure 1 illustrates the evolution of ML-based performance modeling, showing how techniques have progressed from simple regression models to sophisticated hybrid approaches achieving sub-5% accuracy.

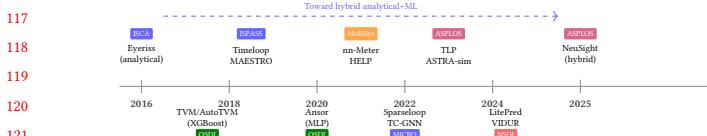


Figure 1: Evolution of ML-based performance modeling (2016–2025). Early work used analytical models (Eyeriss, Timeloop); ML approaches began with simple regressors (TVM) and progressed to deep learning (Ansor, HELP), GNNs (TC-GNN), and transformers (TLP). Current state-of-the-art combines analytical structure with neural networks (NeuSight).

2 Background

2.1 Traditional Performance Modeling

Before ML-based approaches, performance modeling relied on two complementary paradigms: analytical models and cycle-accurate simulation. This section reviews both as *baselines* against which ML techniques are compared, identifying their limitations that motivate learned alternatives.

2.1.1 Analytical Models. Analytical models express performance as closed-form functions of hardware and workload parameters. The roofline model [14] exemplifies this approach, bounding attainable performance by peak compute throughput and memory bandwidth. Given operational intensity I (FLOP/byte), the roofline predicts performance as $P = \min(\pi, \beta \cdot I)$, where π is peak FLOPS and β is memory bandwidth. Despite its simplicity, roofline reasoning guides optimization by revealing compute-bound versus memory-bound regimes.

For DNN accelerators, analytical cost models have become standard practice. Timeloop [12] models data movement across memory hierarchies for any given mapping (loop order and tiling), computing access counts and energy from architectural parameters. MAESTRO [9] provides a data-centric framework that derives performance from dataflow descriptions. Sparseloop [16] extends this methodology to sparse tensor operations, achieving 2000 \times speedup over RTL simulation while maintaining accuracy.

Analytical models offer several advantages: fast evaluation (microseconds per design point), interpretability (designers can trace predictions to specific terms), and extrapolation to unseen configurations. However, they require manual derivation for each target architecture, struggle to capture complex microarchitectural effects (contention, pipeline stalls, caching behavior), and may oversimplify non-linear interactions.

2.1.2 Cycle-Accurate Simulation. Cycle-accurate simulators model hardware at register-transfer level, faithfully reproducing timing behavior. General-purpose simulators like gem5 [3] support flexible configuration of CPU cores, caches, memory controllers, and interconnects. For GPUs, simulators such as GPGPU-Sim [2] and Accel-Sim [7] model SIMD execution, warp scheduling, and memory coalescing.

Cycle-accurate simulation achieves high fidelity—typically within 5–15% of real hardware [3]—and supports detailed microarchitectural studies. However, simulation speed presents a fundamental limitation: evaluating a single ResNet-50 inference may require hours, making design space exploration impractical. ASTRA-sim [15] addresses distributed training at scale through analytical abstractions, but even coarse-grained simulation struggles with the combinatorial explosion of modern ML workloads and hardware configurations.

2.1.3 The Modeling Gap. Neither approach fully addresses modern performance modeling needs. Analytical models are fast but imprecise for complex microarchitectures. Simulators are accurate but too slow for iterative design. This tension has intensified as ML workloads diversify (from CNNs to transformers to mixture-of-experts models) and hardware specializes (GPUs, TPUs, custom accelerators). ML-based performance models offer a middle path: learning complex relationships from profiling data while enabling millisecond-scale inference.

2.2 Machine Learning Fundamentals

This section provides a brief primer on ML techniques frequently employed in performance modeling, establishing terminology used throughout the survey.

2.2.1 Classical Machine Learning. Linear regression and its regularized variants (ridge, LASSO) remain widely used for performance prediction due to their simplicity and interpretability. Given feature vector x (e.g., operator parameters, hardware counters), linear models predict $\hat{y} = w^T x + b$. While unable to capture non-linear relationships, linear models provide baselines and feature importance rankings.

Tree-based ensembles—random forests and gradient boosted trees (XGBoost, LightGBM)—handle non-linearities through recursive partitioning. These methods dominate when training data is limited (<10K samples) and features are well-engineered, often outperforming deep learning in low-data regimes [18].

2.2.2 Deep Learning. Multi-layer perceptrons (MLPs) learn hierarchical feature representations through stacked non-linear transformations: $h_{i+1} = \sigma(W_i h_i + b_i)$. MLPs require minimal feature engineering but need sufficient training data and careful regularization to avoid overfitting.

Recurrent neural networks (RNNs) and their gated variants (LSTM, GRU) process sequential inputs, making them suitable for modeling operator sequences in neural network execution graphs. However, sequential processing limits parallelization and can miss long-range dependencies.

2.2.3 Graph Neural Networks. Graph neural networks (GNNs) operate on graph-structured data through message passing. For a node v with features h_v , GNNs iteratively update representations by aggregating information from neighbors $N(v)$:

$$h_v^{(k+1)} = \phi \left(h_v^{(k)}, \bigoplus_{u \in N(v)} \psi(h_u^{(k)}, e_{uv}) \right) \quad (1)$$

where ϕ and ψ are learnable functions and \oplus is a permutation-invariant aggregation (sum, mean, max).

GNNs are particularly appealing for performance modeling because DNN computation graphs have natural graph structure. Nodes represent operators with features (type, parameters), edges represent data dependencies with features (tensor shapes, datatypes). GNNs can learn to propagate performance-relevant information along these dependencies [13].

2.2.4 Attention and Transformers. Attention mechanisms compute weighted combinations over input elements, with weights determined by learned compatibility functions. Self-attention allows each position to attend to all other positions:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V} \quad (2)$$

Transformers stack self-attention with feedforward networks, enabling long-range dependency modeling without sequential processing. Recent performance models leverage transformer architectures to capture complex inter-operator interactions across entire computation graphs.

2.2.5 Transfer Learning. Transfer learning adapts models trained on one domain (source) to perform well on another (target). In performance modeling, this enables training on easily-profiled hardware and transferring to new platforms with limited data. Common approaches include fine-tuning (adjusting pre-trained weights with target data), domain adaptation (learning domain-invariant representations), and meta-learning (learning to adapt quickly from few examples) [6].

2.3 Problem Formulation

We now formally define the performance modeling problem and establish the evaluation framework used throughout this survey.

2.3.1 Inputs and Outputs. Performance modeling maps workload and hardware descriptions to performance metrics. Formally, given workload specification \mathcal{W} and hardware configuration \mathcal{H} , a performance model f predicts metric y :

$$\hat{y} = f(\mathcal{W}, \mathcal{H}; \theta) \quad (3)$$

where θ represents model parameters (weights for ML models, equations for analytical models).

Workload representations vary by granularity and abstraction:

- *Operator-level*: Individual layer parameters (kernel size, channels, batch size)
- *Graph-level*: Full computation graph with node and edge features
- *IR-level*: Intermediate representations from compilers (TVM [4], XLA)
- *Trace-level*: Execution traces capturing runtime behavior

Hardware representations similarly span multiple levels:

- *Specification*: Static parameters (core count, memory size, bandwidth)
- *Counter-based*: Runtime performance counters (cache misses, stalls)
- *Embedding*: Learned dense representations of hardware platforms

2.3.2 Prediction Targets. Performance models target various metrics depending on application requirements:

Latency measures execution time, typically end-to-end inference time or per-layer latency. Latency prediction is critical for real-time applications with strict deadlines and for optimizing user-facing services.

Throughput captures sustained processing rate: samples per second for inference, tokens per second for language models, or images per second for training. Throughput optimization maximizes hardware utilization for batch processing.

Energy encompasses power consumption (Watts) and energy per operation (Joules/inference). Energy prediction is essential for mobile deployment, data center cost optimization, and sustainability considerations.

Memory includes peak memory footprint (for feasibility checking), memory bandwidth utilization, and memory access patterns.

Multi-objective formulations jointly predict multiple metrics, enabling Pareto-optimal design selection balancing latency, energy, and accuracy.

2.3.3 Accuracy Metrics. The field employs several accuracy metrics, each with distinct interpretations:

Mean Absolute Percentage Error (MAPE) measures average relative deviation:

$$\text{MAPE} = \frac{100\%}{n} \sum_{i=1}^n \left| \frac{y_i - \hat{y}_i}{y_i} \right| \quad (4)$$

MAPE is scale-invariant and interpretable (5% MAPE means predictions typically differ by 5% from ground truth).

Root Mean Square Error (RMSE) penalizes large errors more heavily:

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2} \quad (5)$$

Correlation coefficients (Pearson, Spearman) measure how well predictions track relative ordering—important when models guide design space exploration.

Ranking accuracy directly evaluates whether models correctly order configurations, often measured via Kendall’s τ or top- k accuracy.

2.3.4 Hardware Targets. Modern performance modeling spans diverse hardware platforms:

CPUs remain important for general-purpose inference and training of smaller models. CPU modeling must account for complex cache hierarchies, branch prediction, out-of-order execution, and SIMD vectorization.

GPUs dominate ML training and large-scale inference. GPU modeling addresses SIMD execution, warp scheduling, memory coalescing, and multi-GPU scaling.

TPUs and custom accelerators employ specialized dataflows for matrix operations. Modeling these devices requires understanding systolic arrays, on-chip memory hierarchies, and dataflow mappings.

Edge devices (mobile SoCs, embedded NPUs) impose strict power and memory constraints. Edge modeling emphasizes latency under thermal throttling and memory-limited execution.

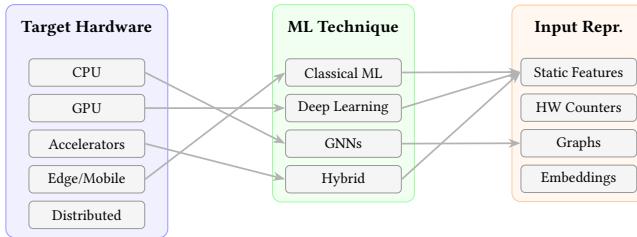


Figure 2: Three-dimensional taxonomy for ML-based performance modeling. Arrows indicate common pairings observed in the literature (e.g., GPU models often use deep learning with static features).

Distributed systems scale training across multiple devices and nodes. Distributed modeling must capture communication overhead, synchronization barriers, and pipeline parallelism.

This diversity of targets, workloads, and metrics motivates our comprehensive taxonomy in Section 3.

3 Taxonomy

We organize the surveyed literature along three primary dimensions: the hardware target being modeled, the machine learning techniques employed, and the input representations used. Figure 2 illustrates how these dimensions intersect to characterize different performance modeling approaches. This taxonomy extends existing classifications [9, 12] by incorporating the emerging diversity of ML-based methods and their distinctive design choices.

Our classification scheme serves two purposes. First, it provides a systematic framework for understanding the design space of ML-based performance models—researchers can identify which combinations of targets, techniques, and representations have been explored versus those that remain open. Second, it enables practitioners to select appropriate methods for their use cases by matching problem characteristics (target hardware, available data, accuracy requirements) to model capabilities.

3.1 By Modeling Target

The choice of hardware target fundamentally shapes model design, as different platforms exhibit distinct performance characteristics and modeling challenges.

3.1.1 CPU Performance Modeling. CPUs present complex modeling challenges due to deep out-of-order pipelines, sophisticated cache hierarchies, and branch prediction. ML models for CPU performance must capture instruction-level parallelism, cache behavior, and memory access patterns. Traditional approaches relied on microbenchmark-based linear regression [3], while recent work employs graph neural networks to model basic block throughput [13]. CPU modeling remains challenging due to the diversity of microarchitectures and the difficulty of capturing dynamic effects like branch misprediction and cache contention.

3.1.2 GPU Performance Modeling. GPUs dominate modern ML training and inference, making accurate GPU performance prediction critical. GPU modeling must account for SIMD execution, warp

scheduling, memory coalescing, and memory bandwidth limitations. Early approaches used analytical roofline models [14], but these struggle with the complex memory hierarchies and occupancy effects of modern GPUs.

ML-based GPU models have achieved remarkable accuracy. NeuSight [11] introduces tile-based prediction that mirrors CUDA’s execution model, achieving 2.3% error on GPT-3 inference across H100, A100, and V100 GPUs. Habitat [17] pioneered runtime-based cross-GPU prediction using wave scaling analysis. These approaches demonstrate that learned models can capture GPU performance characteristics that elude analytical treatment.

3.1.3 DNN Accelerator Modeling. Custom DNN accelerators—including TPUs, NPUs, and systolic array designs—employ specialized dataflows optimized for matrix operations. Modeling these devices requires understanding the interaction between dataflow, memory hierarchy, and tensor tiling.

Analytical frameworks like Timeloop [12] and MAESTRO [9] provide systematic approaches for accelerator design space exploration. Timeloop models data movement and compute utilization for any valid mapping of operations to hardware, achieving 5–10% accuracy versus RTL simulation at 2000× speedup. MAESTRO offers a data-centric perspective using intuitive dataflow directives. Sparseloop [16] extends these frameworks to sparse tensor operations, critical for efficient transformer inference.

ML-based approaches complement analytical models by learning residual corrections or capturing effects not modeled analytically. ArchGym [8] demonstrates that ML surrogate models can achieve 0.61% RMSE while providing 2000× speedup over simulation, enabling rapid design space exploration for accelerator development.

3.1.4 Edge and Mobile Device Modeling. Edge devices impose strict power, memory, and latency constraints, making accurate prediction essential for deploying ML models on mobile phones, IoT devices, and embedded systems. The diversity of edge hardware—spanning mobile CPUs, mobile GPUs, NPUs, and DSPs—creates significant challenges for cross-platform prediction.

nn-Meter [18] addresses this challenge through kernel-level prediction with adaptive sampling, achieving 99% accuracy across mobile CPUs, GPUs, and Intel VPUs. LitePred [6] extends this work with transfer learning, achieving 99.3% accuracy across 85 edge platforms with less than one hour of adaptation per new device. These results demonstrate that ML models can effectively generalize across the heterogeneous edge hardware landscape.

3.1.5 Distributed System Modeling. Multi-GPU and multi-node systems introduce communication overhead, synchronization barriers, and parallelism strategy choices that fundamentally change performance characteristics. Distributed training performance depends on the interplay between compute, memory bandwidth, and network communication.

ASTRA-sim [15] provides end-to-end distributed training simulation, modeling collective communication algorithms, network topology, and compute-communication overlap. VIDUR [1] focuses specifically on LLM inference serving, capturing the unique characteristics of prefill and decode phases, KV cache management,

and request scheduling. These simulation frameworks achieve 5–15% accuracy versus real clusters while enabling exploration of parallelization strategies at scale.

3.2 By ML Technique

The choice of ML technique reflects trade-offs between accuracy, data efficiency, interpretability, and generalization capability.

3.2.1 Classical Machine Learning. Tree-based ensembles—random forests and gradient boosted trees (XGBoost, LightGBM)—remain highly effective for performance modeling, particularly in low-data regimes. These methods handle non-linear relationships through recursive partitioning, provide feature importance rankings for interpretability, and require minimal hyperparameter tuning.

Classical ML models dominate when training data is limited (<10K samples) or when features are well-engineered. nn-Meter [18] demonstrates that random forests achieve competitive accuracy with careful kernel-level feature engineering. The ALCOP framework combines XGBoost with analytical pre-training, using analytical model predictions as features to accelerate autotuning convergence.

3.2.2 Deep Learning. Multi-layer perceptrons (MLPs) learn hierarchical feature representations without manual feature engineering. MLPs are widely used as the prediction head in more complex architectures and as standalone models when sufficient training data is available. NeuSight [11] uses MLPs to predict tile-level GPU utilization, learning complex interactions between tile parameters and hardware characteristics.

Recurrent neural networks (RNNs and LSTMs) process sequential inputs, making them suitable for modeling operator sequences in neural network execution. However, sequential processing limits parallelization, and attention-based architectures increasingly replace RNNs for sequence modeling tasks.

3.2.3 Graph Neural Networks. Graph neural networks (GNNs) have emerged as particularly effective for performance modeling because computational graphs have natural graph structure. Nodes represent operators with features (type, parameters, shapes), edges represent data dependencies with features (tensor dimensions, datatypes). GNNs propagate performance-relevant information along these dependencies through message passing.

GRANITE [13] applies GNNs to basic block throughput estimation, learning to predict CPU performance from instruction dependency graphs. For DNN workloads, GNN-based models capture inter-operator interactions that flat feature representations miss. The graph structure also enables natural handling of variable-size networks without padding or truncation.

3.2.4 Hybrid Analytical+ML Models. Hybrid approaches combine physics-based analytical models with learned components, achieving both interpretability and high accuracy. The analytical component provides a strong prior based on hardware characteristics, while the ML component learns residual corrections and complex interactions.

This design philosophy has produced state-of-the-art results. Analytical pre-training initializes ML models with reasonable predictions, reducing data requirements and improving convergence.

Physics-informed architectures incorporate analytical insights into model structure—NeuSight’s tile-based prediction mirrors CUDA’s execution model, providing inductive bias that improves generalization. Residual learning trains ML models to predict the error of analytical models, combining analytical interpretability with ML’s ability to capture unmodeled effects.

The latency predictor study [5] demonstrates that hybrid approaches with transfer learning achieve 22.5% average improvement over baselines, with up to 87.6% improvement on challenging cross-platform prediction tasks.

3.3 By Input Representation

Input representation determines what information the model can access and how effectively it can learn performance-relevant patterns.

3.3.1 Static Features. Static features derive from workload and hardware specifications without runtime measurement. For DNN workloads, these include layer parameters (kernel size, channels, stride, batch size), tensor dimensions, and operator types. Hardware specifications include core counts, memory sizes, bandwidth, and clock frequencies.

Static features enable prediction without profiling, supporting use cases like neural architecture search where thousands of candidate networks must be evaluated. Feature engineering plays a critical role: effective representations capture computation-to-communication ratios, memory footprint estimates, and parallelization potential.

3.3.2 Hardware Counters. Performance counters provide runtime measurements of hardware behavior: cache miss rates, memory bandwidth utilization, instruction throughput, and stall cycles. Counter-based models can capture dynamic effects invisible to static analysis, including contention, thermal throttling, and runtime scheduling decisions.

The primary limitation is that counter-based models require hardware execution, limiting their applicability for design space exploration or new architecture evaluation. However, for optimizing existing deployments or debugging performance anomalies, counter-based models provide valuable insights that static approaches cannot match.

3.3.3 Graph Representations. Graph representations encode computational graphs with nodes representing operators and edges representing data dependencies. Node features capture operator characteristics (type, parameters), while edge features encode tensor properties (shape, datatype, memory format).

Graph representations provide several advantages over flat feature vectors: they naturally handle variable-size networks, preserve structural information about operator interactions, and enable permutation-invariant predictions. GNNs operating on these representations can learn which subgraph patterns indicate performance bottlenecks.

3.3.4 Learned Embeddings. Learned embeddings compress high-dimensional or categorical information into dense vector representations. Hardware embeddings represent diverse devices as points

in a learned feature space, enabling transfer learning across platforms. Operator embeddings capture semantic similarities between operator types that may share performance characteristics.

HELP formulates hardware prediction as meta-learning, learning hardware embeddings that represent devices as black-box functions. With just 10 measurement samples on a new device, HELP achieves accurate predictions by positioning the device appropriately in the learned embedding space. This approach is particularly valuable for the fragmented edge hardware landscape, where collecting exhaustive training data for each device is impractical.

Table 1 summarizes representative papers across our taxonomy dimensions, illustrating the diversity of approaches and their key characteristics.

4 Survey of Approaches

This section surveys ML-based performance modeling approaches organized by target hardware platform. For each category, we examine the modeling challenges specific to that platform, describe representative techniques, and synthesize key findings across the literature. Table 2 provides a comprehensive comparison of the surveyed approaches.

4.1 CPU Performance Modeling

CPU performance modeling for ML workloads presents unique challenges due to complex microarchitectural effects including out-of-order execution, branch prediction, and deep cache hierarchies. While GPUs have received more attention for DNN training, CPUs remain important for inference—particularly on edge devices and for operators that map poorly to SIMD execution.

4.1.1 Traditional CPU Performance Modeling. Traditional CPU modeling relies on cycle-accurate simulation through frameworks like gem5 [3]. The gem5 simulator provides multiple fidelity levels: fast functional simulation for correctness validation, and detailed out-of-order models achieving 10–20% accuracy versus real hardware. For ML workloads, gem5 extensions such as gem5-Aladdin and SMAUG enable accelerator integration studies.

However, cycle-accurate simulation suffers from fundamental speed limitations—simulating even modest DNN inference requires hours, making design space exploration impractical. This limitation has motivated ML-based alternatives that learn to predict performance from static program features.

4.1.2 ML-Based Basic Block Modeling. GRANITE [13] represents the state of the art in ML-based CPU performance modeling. The key insight is that basic block throughput—the steady-state execution rate of a loop body—can be predicted from the instruction dependency graph without simulation. GRANITE encodes basic blocks as directed graphs where nodes represent instructions with features (opcode, operand types) and edges capture data dependencies.

A graph neural network processes this representation through message passing layers:

$$\mathbf{h}_i^{(k+1)} = \text{MLP} \left(\mathbf{h}_i^{(k)} + \sum_{j \in \mathcal{N}(i)} \mathbf{h}_j^{(k)} \right) \quad (6)$$

where $\mathbf{h}_i^{(k)}$ represents instruction i 's embedding at layer k . After several message passing rounds, a global pooling operation aggregates instruction embeddings into a single block representation, which a final MLP maps to throughput prediction.

GRANITE achieves 0.97 Kendall's τ correlation with ground-truth measurements on x86 basic blocks, significantly outperforming prior analytical models like IACA and llvm-mca. Critically, the learned model generalizes across microarchitectures—a model trained on Skylake transfers to Haswell with only modest accuracy degradation.

4.1.3 Challenges and Opportunities. Despite GRANITE's success, several challenges remain for CPU performance modeling. First, DNN operators often involve memory-bound execution where cache behavior dominates—GRANITE focuses on compute-bound basic blocks and does not model memory hierarchy effects. Second, modern CPUs feature increasingly complex prefetchers and branch predictors whose behavior is difficult to capture in static features. Third, CPU-based DNN inference often involves highly optimized library code (Intel MKL, ARM Compute Library) whose performance depends on runtime scheduling decisions.

Hybrid approaches combining coarse-grained simulation with learned correction factors represent a promising direction. Rather than simulating every cycle, these methods use fast simulation to establish approximate behavior, then train ML models to predict residual errors, potentially achieving simulation accuracy at reduced cost.

4.2 GPU Performance Modeling

GPUs are the dominant platform for ML training and large-scale inference. Accurate GPU performance prediction is essential for neural architecture search, compiler optimization, and serving system design. However, GPU performance modeling is challenging due to SIMD execution, complex memory hierarchies, and workload-dependent scheduling behavior.

4.2.1 Cycle-Accurate GPU Simulation. GPGPU-Sim [2] pioneered detailed GPU simulation, modeling SIMD cores, warp scheduling, memory coalescing, and cache hierarchies. Accel-Sim [7] extended this foundation with trace-driven simulation and improved correlation with modern GPUs (Turing, Ampere), achieving 0.90–0.97 IPC correlation.

These simulators provide high fidelity—essential for microarchitectural studies—but suffer from 1000–10000× slowdown versus real GPU execution. Simulating a single ResNet-50 inference can require hours, making design space exploration impractical. This has motivated the development of ML-based predictors that achieve comparable accuracy at dramatically reduced cost.

4.2.2 Learned GPU Performance Models. Habitat [17] introduced *wave scaling* for cross-GPU prediction. The key insight is that GPU execution time can be decomposed into compute and memory components that scale differently across devices:

$$T_{\text{target}} = T_{\text{compute}} \cdot \frac{P_{\text{source}}}{P_{\text{target}}} + T_{\text{memory}} \cdot \frac{B_{\text{source}}}{B_{\text{target}}} \quad (7)$$

where P denotes peak compute throughput and B memory bandwidth. By profiling on a source GPU and measuring how kernels

Table 1: Representative papers classified by our taxonomy dimensions. Accuracy reported as MAPE or correlation where available.

Paper	Target	Technique	Input	Accuracy	Key Contribution
NeuSight [11]	GPU	Hybrid	Static	2.3%	Tile-based prediction
nn-Meter [18]	Edge	Classical ML	Static	<5%	Kernel detection
LitePred [6]	Edge	Transfer	Static	0.7%	85-platform transfer
GRANITE [13]	CPU	GNN	Graph	0.97 corr	Basic block modeling
Timeloop [12]	Accelerator	Analytical	Static	5–10%	Loop-nest DSE
ASTRA-sim [15]	Distributed	Simulation	Traces	5–15%	Collective modeling
ArchGym [8]	Accelerator	Hybrid	Static	0.61% RMSE	ML-aided DSE

Table 2: Summary of surveyed ML-based performance modeling approaches, organized by target hardware platform.

Paper	Platform	ML Technique	Prediction Target	Error	Key Innovation
<i>CPU Performance Modeling</i>					
GRANITE [13]	CPU	GNN	Basic block throughput	0.97 corr	Instruction graph encoding
gem5+ML [3]	CPU	Hybrid	Execution time	10–20%	Simulation + learning
<i>GPU Performance Modeling</i>					
NeuSight [11]	GPU	Hybrid MLP	Kernel/E2E latency	2.3%	Tile-based prediction
Habitat [17]	GPU	MLP	Training time	11.8%	Wave scaling analysis
Accel-Sim [7]	GPU	Simulation	Cycle-accurate	10–20%	SASS trace-driven
<i>DNN Accelerator Modeling</i>					
Timeloop [12]	NPU	Analytical	Latency/Energy	5–10%	Loop-nest DSE
MAESTRO [9]	NPU	Analytical	Latency/Energy	5–15%	Data-centric directives
Sparseloop [16]	NPU	Analytical	Sparse tensors	5–10%	Compression modeling
ArchGym [8]	Multi	RL+Surrogate	Multi-objective	0.61%	ML-aided DSE
<i>Edge Device Modeling</i>					
nn-Meter [18]	Edge	RF ensemble	Latency	<1%	Kernel detection
LitePred [6]	Edge	VAE+MLP	Latency	0.7%	85-platform transfer
HELP [10]	Multi	Meta-learning	Latency	1.9%	10-sample adaptation
<i>Distributed and LLM Systems</i>					
ASTRA-sim [15]	Distributed	Simulation	Training time	5–15%	Collective modeling
VIDUR [1]	GPU cluster	Simulation	LLM serving	<5%	Prefill/decode phases

respond to artificially reduced parallelism (“wave scaling”), Habitat estimates the compute and memory fractions, enabling prediction on unseen target GPUs.

Habitat achieves 11.8% average error predicting training iteration time across GPU generations (V100 to A100). However, it requires actual GPU execution for wave scaling measurements and cannot predict performance for unseen models.

NeuSight [11] addresses these limitations through *tile-based prediction*. The key innovation is decomposing GPU kernel execution into tiles—the basic scheduling unit in CUDA—and predicting per-tile behavior:

$$T_{\text{kernel}} = \max_{w \in \text{waves}} \sum_{t \in w} (T_{\text{compute}}^{(t)} + T_{\text{memory}}^{(t)}) \quad (8)$$

This formulation mirrors actual GPU execution semantics: tiles are scheduled in waves, and kernel time is dominated by the slowest wave. NeuSight uses MLPs to predict tile-level compute and memory times from static features (tile dimensions, register usage, shared memory allocation).

By capturing the wave-level structure, NeuSight achieves remarkable accuracy: 2.3% error on GPT-3 inference across H100,

A100, and V100 GPUs. This represents a 50× reduction in error compared to prior approaches like Habitat (121.4% → 2.3% on H100 for GPT-3). NeuSight’s physics-informed architecture—encoding GPU execution semantics into the model structure—provides strong inductive bias that enables generalization to unseen models and GPUs.

4.2.3 Compiler Cost Models for GPUs. The TVM [4] and Ansor [19] systems use learned cost models to guide tensor program optimization. Rather than executing every candidate program, XGBoost or MLP models predict execution time from program features (loop bounds, vectorization widths, memory access patterns).

Ansor’s hierarchical search combines sketch generation, random annotation, and evolutionary refinement, using the cost model to prune the search space. With 10K profiled samples, Ansor achieves approximately 15% MAPE on GPU kernel prediction. The TenSet dataset provides 52 million program performance records across CPUs and GPUs, enabling pre-trained cost models that accelerate autotuning convergence by 10×.

813 4.2.4 *LLM Inference Prediction.* Large language model inference
 814 presents unique GPU modeling challenges. LLM execution exhibits
 815 distinct *prefill* (compute-bound, parallel prompt processing) and
 816 *decode* (memory-bound, sequential token generation) phases with
 817 fundamentally different performance characteristics.
 818

VIDUR [1] provides discrete-event simulation for LLM serving systems. Rather than modeling GPU microarchitecture, VIDUR simulates request scheduling, KV cache management, and batching decisions—the system-level factors that dominate serving performance. VIDUR achieves <5% error on end-to-end serving metrics including time-to-first-token and request latency.

Roofline-LLM extends traditional roofline analysis to LLM inference by decomposing transformer execution into compute-bound (prefill attention, FFN) and memory-bound (decode attention, KV cache access) components. Combined with learned correction factors, this hybrid approach achieves 87% reduction in MSE compared to pure roofline predictions.

4.3 Accelerator Performance Modeling

DNN accelerators—including TPUs, NPUs, and custom ASIC designs—employ specialized dataflows and memory hierarchies optimized for tensor operations. Modeling these devices requires understanding the interaction between dataflow choices, memory hierarchy utilization, and workload characteristics.

4.3.1 *Analytical Accelerator Modeling.* Timeloop [12] provides the foundational framework for DNN accelerator design space exploration. The key insight is that accelerator performance can be accurately predicted from loop-nest representations of tensor computations. For a given architecture specification and mapping (loop order, tiling, spatial distribution), Timeloop analytically computes:

- **Data reuse** at each memory level: how many times each tensor element is accessed from each buffer
- **Latency**: compute cycles plus memory stall cycles based on bandwidth constraints
- **Energy**: access counts multiplied by per-access energy at each memory level

Timeloop decouples architecture specification (PEs, buffer sizes, bandwidth) from mapping decisions, enabling systematic exploration of dataflow choices. The framework achieves 5–10% accuracy versus RTL simulation while providing 2000× speedup, making million-point design sweeps tractable.

MAESTRO [9] offers a complementary *data-centric* perspective. Rather than loop-nest transformations, MAESTRO models performance through data movement analysis using compact dataflow directives. This representation is more intuitive—designers specify how tensors flow through the architecture rather than manipulating loop indices—while achieving comparable accuracy.

Sparseloop [16] extends analytical modeling to sparse tensor accelerators. The key challenge is that sparse execution time depends on runtime sparsity patterns, not just static tensor dimensions. Sparseloop models compression formats (CSR, bitmap, RLE), gating logic, and sparse-dense conversion overhead, enabling accurate prediction for pruned neural networks and sparse attention patterns.

4.3.2 *ML-Augmented Accelerator Design.* ArchGym [8] demonstrates how ML-based surrogate models can accelerate accelerator

design. The framework connects ML optimization algorithms (reinforcement learning, Bayesian optimization, evolutionary strategies) to hardware simulators through a unified interface.

A key finding is the *hyperparameter lottery*: ML algorithms show high variance across hyperparameter choices, with optimal settings differing substantially between target designs. ArchGym addresses this through systematic hyperparameter sweeps enabled by fast surrogate models. Trained surrogate models achieve 0.61% RMSE while providing 2000× speedup over simulation, enabling exploration of hyperparameter configurations that would be intractable with direct simulation.

4.3.3 *FPGA and Emerging Accelerator Modeling.* FPGA-based accelerators present additional modeling challenges due to the flexibility of reconfigurable fabric and the complexity of HLS-generated data-paths. Recent work applies transfer learning to FPGA design space exploration: models trained on one design can adapt to new architectures with limited additional profiling.

Emerging accelerators—including processing-in-memory (PIM), neuromorphic, and analog compute-in-memory designs—remain underexplored. These platforms exhibit fundamentally different performance characteristics (energy-dominated by activations, analog noise effects, sparse event-driven computation) that existing frameworks do not address. Developing unified modeling approaches for this diverse hardware landscape represents an important open challenge.

4.4 Memory System Modeling

Memory system behavior increasingly dominates ML workload performance. Large language models may require hundreds of gigabytes for weights and KV cache, while training workloads stress memory bandwidth through gradient communication. Accurate memory modeling is essential for understanding performance across the modern hardware landscape.

4.4.1 *Cache and Memory Hierarchy Modeling.* Traditional memory system modeling relies on cache simulation within frameworks like gem5 [3] and GPGPU-Sim [2]. These simulators model replacement policies, bank conflicts, memory coalescing (for GPUs), and DRAM controller behavior with high fidelity.

For DNN workloads, memory access patterns are often highly regular—streaming through weight and activation tensors—making analytical prediction feasible. Timeloop [12] models memory hierarchy through data reuse analysis: given a tiling and loop order, the framework computes exact access counts at each memory level. This analytical approach achieves high accuracy for regular workloads but may miss dynamic effects like cache contention in multi-tenant scenarios.

4.4.2 *KV Cache for LLM Inference.* KV cache management has emerged as the dominant memory challenge for LLM serving. The attention mechanism requires storing key-value tensors for all previously generated tokens, with memory growing linearly with sequence length and batch size. For long-context models serving concurrent requests, KV cache can consume hundreds of gigabytes.

vLLM’s PagedAttention introduces virtual memory concepts to KV cache management. By storing KV blocks in non-contiguous

929 physical memory with page tables for address translation, PagedAttention achieves near-zero memory waste from fragmentation. This
 930 system-level optimization yields 2–4× throughput improvement
 931 over prior approaches.
 932

933 VIDUR [1] models KV cache behavior at the serving system level,
 934 simulating allocation, eviction, and paging decisions that affect
 935 request latency. More recent work explores KV cache compres-
 936 sion through quantization (Oaken), sparsity (ALISA), and adaptive
 937 token selection (MorphKV), with potential memory savings ex-
 938 ceeding 50%. Accurate performance models for these compression
 939 techniques—predicting the latency-accuracy tradeoff for different
 940 compression levels—remain an open challenge.
 941

942 **4.4.3 Distributed Memory and Communication.** Multi-GPU and
 943 multi-node training introduces communication overhead that can
 944 dominate performance at scale. ASTRA-sim [15] provides end-to-
 945 end simulation of distributed training, modeling collective com-
 946 munication algorithms (ring, tree, halving-doubling all-reduce),
 947 network topology, and compute-communication overlap.
 948

949 The simulation decomposes collective operations into point-to-
 950 point messages, tracks network contention, and models the interac-
 951 tion between computation and communication phases. ASTRA-sim
 952 achieves 5–15% error versus real multi-GPU clusters, enabling ex-
 953 ploration of parallelization strategies (data parallel, model parallel,
 954 pipeline parallel) before expensive hardware experiments.
 955

956 A key insight from distributed training modeling is that commu-
 957 nication overhead depends strongly on message granularity and
 958 overlap opportunities. Chunked gradient communication, where
 959 gradients are transmitted in pieces overlapped with backward pass
 960 computation, can hide communication latency. Accurate modeling
 961 of this overlap—which depends on operator ordering, chunk sizes,
 962 and network bandwidth—is essential for predicting distributed train-
 963 ing performance.
 964

4.5 Cross-Platform and Transfer Learning

964 The proliferation of hardware platforms—from edge devices to
 965 datacenter GPUs to custom accelerators—creates demand for per-
 966 formance models that generalize across configurations. Training
 967 separate models for each target device is impractical given the diver-
 968 sity of the hardware landscape. Transfer learning and meta-learning
 969 approaches address this challenge by learning shared representa-
 970 tions that adapt efficiently to new platforms.
 971

972 **4.5.1 Hardware-Adaptive Latency Prediction.** HELP [10] formulates
 973 cross-hardware prediction as meta-learning. The key insight is that
 974 hardware platforms can be treated as “tasks” in meta-learning: each
 975 device provides a small sample of profiled networks, and the goal
 976 is rapid adaptation to new devices.
 977

HELP learns:

- **Architecture encoder:** A GNN that embeds neural net-
 978 work architectures into a fixed-dimensional space
- **Hardware encoder:** A learned function that represents
 980 devices from their profiled samples
- **Predictor:** An MLP that maps (architecture, hardware)
 982 pairs to latency

984 Using MAML-style meta-learning, HELP achieves 93.2% accuracy
 985 with just 10 profiled samples on new devices, reaching 98.1% with
 986

100 samples. This sample efficiency is critical for the fragmented
 987 edge hardware landscape where collecting exhaustive training data
 988 for each device type is impractical.
 989

990 **4.5.2 Transfer Learning at Scale.** LitePred [6] scales cross-platform
 991 prediction to 85 edge devices—the most comprehensive evaluation
 992 to date. The framework introduces a VAE-based data sampler that
 993 intelligently selects which architectures to profile on new devices.
 994 Rather than random sampling, the VAE identifies architectures
 995 that are most informative for learning the device’s performance
 996 characteristics.
 997

998 With less than one hour of profiling on a new device, LitePred
 999 achieves 99.3% accuracy on held-out architectures. This combines
 1000 pre-trained representations from source platforms with efficient
 1001 adaptation, demonstrating that the cross-platform transfer learning
 1002 problem is tractable even at scale.
 1003

1004 The latency predictors study [5] provides a systematic compari-
 1005 son of transfer learning approaches for NAS. Key findings include:
 1006

- End-to-end training on pooled multi-platform data outper-
 1007 forms sequential fine-tuning
- Transfer learning provides 22.5% average improvement over
 1008 training from scratch
- Benefits are largest for challenging cross-platform transfers
 1009 (up to 87.6% improvement)

1010 **4.5.3 Hybrid Analytical-ML Transfer.** Hybrid approaches combine
 1011 analytical models with learned components to improve transfer
 1012 efficiency. SynPerf decomposes GPU kernel execution into pipeline
 1013 demands (compute, memory, cache) using analytical models, then
 1014 trains MLPs to capture cross-pipeline interactions. The analyti-
 1015 cal decomposition provides physics-based structure that transfers
 1016 across GPUs, while the learned component captures device-specific
 1017 effects.
 1018

1019 This hybrid architecture achieves 6.1% kernel-level error and
 1020 has been applied to guide Triton kernel optimization, demon-
 1021 strating 1.7× speedup on generated kernels. The combination of inter-
 1022 pretative analytical structure with learned flexibility represents a
 1023 promising direction for transferable performance modeling.
 1024

1025 **4.5.4 Open Challenges in Transfer Learning.** Despite progress, sev-
 1026 eral challenges remain. First, most transfer learning work focuses
 1027 on CNN architectures; transformers and mixture-of-experts models
 1028 remain underexplored. Second, transfer across *workload types* (not
 1029 just hardware) is challenging—models trained on vision networks
 1030 may not transfer to language models or graph neural networks.
 1031 Third, continual learning for performance models—adapting to
 1032 hardware and software evolution over time—is largely unexplored.
 1033

1034 Foundation models for performance prediction represent an
 1035 emerging opportunity. Pre-trained on large-scale profiling datasets
 1036 spanning diverse architectures and hardware, such models could
 1037 provide strong initialization for any new prediction task. The TenSet
 1038 dataset with 52 million records represents a step in this direction,
 1039 but comprehensive datasets covering the full range of modern work-
 1040 loads and hardware remain to be developed
 1041

5 Comparison and Analysis

1042 Having surveyed the landscape of ML-based performance model-
 1043 ing approaches, we now provide a comparative analysis across key
 1044

1045 dimensions, including commonly used analytical and simulation-based baselines. This analysis synthesizes trade-offs that practitioners face when selecting or developing performance models, examining accuracy, training cost, generalization, and interpretability. Table 3 provides a comprehensive comparison across these dimensions.

1051

1052 5.1 Accuracy vs. Training Cost

1053 A fundamental trade-off exists between prediction accuracy and the
 1054 cost of data collection and model training. We analyze this trade-off
 1055 across the surveyed approaches, identifying regimes where different
 1056 techniques excel.

1057

1058 *5.1.1 Data Collection Overhead.* The cost of obtaining training
 1059 data varies dramatically across approaches. *Profiling-based methods*
 1060 require executing workloads on target hardware, with costs ranging
 1061 from minutes (single operators) to hours (full model sweeps). nn-
 1062 Meter [18] requires approximately 1,000 profiled samples per kernel
 1063 type per device, translating to several hours of automated measurement.
 1064 LitePred [6] reduces this to approximately 100 samples for new devices through intelligent VAE-based sampling.

1065 *Simulation-based training* uses cycle-accurate or analytical simulators as ground truth. ArchGym [8] trains surrogate models on Timeloop [12] outputs, avoiding real hardware entirely but requiring validated simulator configurations. This approach achieves 0.61% RMSE while providing 2000× speedup over direct simulation.

1066 *Transfer learning* amortizes data collection across platforms. HELP [10] demonstrates that meta-learning enables 93.2% accuracy with just 10 samples on new devices, reaching 98.1% with 100 samples. This sample efficiency is critical for the fragmented edge hardware landscape.

1067 *5.1.2 Model Training Cost.* Training complexity varies from minutes
 1068 for classical ML to days for large-scale pre-training. Tree-based
 1069 ensembles (random forests, XGBoost) train in minutes on modest
 1070 datasets and require minimal hyperparameter tuning. Deep learning
 1071 models require careful architecture design, regularization, and often
 1072 GPU training, but can achieve higher accuracy on large datasets.

1073 The TenSet dataset [20] with 52 million tensor program performance
 1074 records enables pre-trained cost models that accelerate autotuning convergence by 10×. However, creating such datasets
 1075 requires substantial infrastructure investment.

1076

1077 *5.1.3 Accuracy Stratification.* We observe three accuracy tiers across
 1078 the surveyed approaches:

1079 **Tier 1 (<5% error):** Specialized models achieving near-perfect
 1080 accuracy on narrow domains. nn-Meter achieves <1% error on
 1081 edge device latency through kernel-level decomposition. NeuSight
 1082 reaches 2.3% error on GPU inference through physics-informed
 1083 tile-based prediction. LitePred achieves 0.7% error across 85 edge
 1084 platforms through extensive transfer learning.

1085 **Tier 2 (5–15% error):** General-purpose models with broader
 1086 applicability. Habitat achieves 11.8% error on cross-GPU prediction
 1087 using wave scaling. Analytical frameworks like Timeloop and
 1088 MAESTRO typically achieve 5–15% error versus RTL simulation.

1089 **Tier 3 (15–25% error):** Compiler cost models optimized for ranking
 1090 rather than absolute accuracy. TVM’s AutoTVM [4] achieves
 1091 approximately 20% MAPE, sufficient for guiding autotuning search.

1092

1103 These models prioritize speed and online adaptation over absolute
 1104 precision.

1105 The key insight is that accuracy requirements depend on the use
 1106 case: neural architecture search may tolerate 10–15% error if rank-
 1107 ings are preserved, while hardware cost estimation for procurement
 1108 decisions demands <5% accuracy.

1110 5.2 Generalization Capabilities

1111 Generalization—the ability to predict accurately on unseen work-
 1112 loads, configurations, or hardware—is perhaps the most critical cap-
 1113 ability for practical deployment. We analyze generalization along
 1114 three axes: workload generalization, hardware generalization, and
 1115 temporal generalization.

1116 *5.2.1 Workload Generalization.* Models must handle neural net-
 1117 work architectures not seen during training. GNN-based approaches
 1118 offer natural workload generalization because the graph structure
 1119 captures compositional relationships. GRANITE [13] generalizes
 1120 across basic blocks by learning instruction-level patterns that com-
 1121 pose into block-level predictions.

1122 However, generalization often fails across workload *types*. Models
 1123 trained on CNNs may not transfer to transformers due to funda-
 1124 mentally different computational patterns. NeuSight [11] addresses
 1125 this by training on diverse operator types (GEMM, attention, con-
 1126 volution) and learning GPU execution semantics that generalize
 1127 across operations.

1128 *5.2.2 Hardware Generalization.* Cross-hardware prediction remains
 1129 challenging due to microarchitectural diversity. Three approaches
 1130 have shown promise:

1131 *Meta-learning* treats hardware platforms as tasks. HELP [10]
 1132 learns hardware embeddings that position devices in a shared latent
 1133 space, enabling few-shot adaptation to new platforms.

1134 *Feature-based transfer* uses hardware specifications as input fea-
 1135 tures. LitePred [6] learns relationships between hardware character-
 1136 istics (compute capability, memory bandwidth) and performance,
 1137 enabling zero-shot prediction (92.1% accuracy) on entirely new
 1138 devices.

1139 *Analytical decomposition* factors predictions into hardware-dependent
 1140 and hardware-independent components. Habitat [17] decomposes
 1141 execution into compute and memory components that scale with
 1142 known hardware parameters, achieving cross-GPU prediction with-
 1143 out retraining.

1144 *5.2.3 Temporal Generalization.* An underexplored dimension is
 1145 generalization across time—as software stacks evolve (new com-
 1146 piler versions, framework updates, driver changes), performance
 1147 characteristics shift. Models trained on older configurations may
 1148 degrade on current systems.

1149 Continual learning approaches that adapt to evolving hardware-
 1150 software stacks represent an important open direction. The TenSet
 1151 dataset’s versioned releases provide a starting point for studying
 1152 temporal generalization in compiler cost models.

Table 3: Comparative analysis of representative performance models—including ML-based and analytical/simulation approaches—across key dimensions. The Accuracy column reports the metric and value as given in each original work (e.g., MAPE, RMSE, Kendall’s τ , ranges).

Model	Accuracy (as reported)	Training Data	Adaptation Cost	Generalization	Interpretability	Inference Time
<i>Classical ML</i>						1223
nn-Meter [18]	<1% MAPE	1K/kernel	Hours/device	Device-specific	Medium	1224
XGBoost (TVM) [4]	20% MAPE	10K+	Online	Operator-level	Medium	1225
<i>Deep Learning</i>						1226
NeuSight [11]	2.3% MAPE	100K+	Pre-trained	Cross-GPU	Low	1227
Habitat [17]	11.8% MAPE	Online profiling runs	None (requires GPU)	Cross-GPU	Medium	1228
<i>Graph Neural Networks</i>						1229
GRANITE [13]	0.97 τ	10K+	Hours	Cross- μ arch	Low	1230
HELP [10]	1.9% MAPE	Meta-training	10 samples	Cross-platform	Low	1231
<i>Transfer Learning</i>						1232
LitePred [6]	0.7% MAPE	85 platforms	100 samples	85+ devices	Low	1233
<i>Hybrid Analytical+ML</i>						1234
Timeloop [12]	5–10%	Arch spec	None	Any accelerator	High	1235
ArchGym [8]	0.61% RMSE	Simulation	Surrogate training	Architecture-specific	Medium	1236
VIDUR [1]	<5%	Kernel profiles	Per-model	LLM-specific	High	1237

5.3 Interpretability

Interpretability—understanding *why* a model makes particular predictions is valuable for debugging, optimization guidance, and building practitioner trust. We categorize approaches by their interpretability characteristics.

5.3.1 Analytical Models: High Interpretability. Analytical frameworks like Timeloop [12] and MAESTRO [9] provide full interpretability. Predictions decompose into explicit terms: data movement at each memory level, compute utilization, bandwidth constraints. Practitioners can trace high-latency predictions to specific bottlenecks (e.g., “DRAM bandwidth limits this mapping”).

This interpretability enables *actionable insights*: if the model predicts memory-bound execution, the designer knows to explore mappings with better data reuse. The roofline model [14] exemplifies this—identifying compute-bound versus memory-bound regimes immediately suggests optimization directions.

5.3.2 Classical ML: Medium Interpretability. Tree-based ensembles provide feature importance rankings, indicating which input features most influence predictions. nn-Meter’s kernel-level decomposition enables interpretability: practitioners can identify which kernels dominate latency and focus optimization efforts accordingly.

However, feature importance does not explain *how* features interact. A model may indicate that “kernel size” is important without revealing whether large or small kernels are faster for a given hardware platform.

5.3.3 Deep Learning: Low Interpretability. Deep neural networks, including GNNs and transformers, function as black boxes. While techniques like attention visualization and gradient-based attribution provide some insight, they rarely yield actionable optimization guidance.

NeuSight [11] partially addresses this through physics-informed architecture: by decomposing predictions into compute and memory components that mirror GPU execution, the model structure itself provides interpretability even though individual weight values remain opaque.

5.3.4 Hybrid Approaches: Balanced Interpretability. Hybrid analytical+ML models offer a middle ground. The analytical component provides interpretable baselines, while the ML component captures residual effects. When predictions diverge from analytical expectations, practitioners know the difference stems from effects not captured in the analytical model (contention, cache effects, scheduling decisions).

VIDUR [1] exemplifies this for LLM serving: discrete-event simulation provides interpretable system-level behavior, while learned kernel-time predictors capture GPU execution details. The simulation structure enables “what-if” analysis (e.g., “how would P99 latency change with larger batch sizes?”) that pure ML models cannot support.

5.3.5 The Interpretability-Accuracy Trade-off. A general trade-off exists between interpretability and accuracy. Analytical models sacrifice accuracy for transparency; deep learning models sacrifice transparency for accuracy. For production deployment, hybrid approaches that combine interpretable structure with learned components increasingly represent the best of both worlds.

6 Open Challenges and Future Directions

Despite remarkable progress, significant challenges remain in ML-based performance modeling. This section identifies key open problems and promising research directions that will shape the field’s evolution.

6.1 Data Availability and Quality

The effectiveness of ML-based performance models fundamentally depends on training data quality and availability. Several challenges persist in this dimension.

6.1.1 Benchmark Diversity. Existing datasets predominantly cover CNN architectures optimized for image classification. TenSet [20] provides 52 million tensor program records but focuses on operators from ResNet, MobileNet, and similar architectures. Modern workloads—transformers, mixture-of-experts models, graph neural networks, diffusion models—remain underrepresented.

The rapid evolution of model architectures exacerbates this gap. Models trained on 2022-era workloads may poorly predict performance of 2025 architectures featuring sparse attention, conditional computation, or novel activation functions. Continuously updated, community-maintained benchmark suites could address this challenge.

6.1.2 Hardware Coverage. Hardware diversity creates data collection bottlenecks. LitePred [6] covers 85 edge devices, but the mobile hardware landscape spans hundreds of distinct SoC configurations. Data center hardware (H100, TPU v5, custom accelerators) often has restricted access, limiting public dataset creation.

Simulation-based data generation offers a partial solution: ArchGym [8] trains on Timeloop outputs, avoiding hardware access requirements. However, simulation accuracy itself requires validation against real hardware, creating a chicken-and-egg problem.

6.1.3 Measurement Noise and Reproducibility. Performance measurements exhibit variance from thermal throttling, OS scheduling, memory allocation, and caching effects. Industrial-strength profiling requires careful warm-up periods, multiple runs, and statistical aggregation. Many published models train on single-run measurements, potentially learning noise rather than signal.

Standardized measurement protocols—specifying warm-up iterations, cooling periods, statistical aggregation methods—would improve cross-study comparability and model reliability.

6.2 Model Generalization

Generalization remains the central challenge: models that excel on training distributions often fail on realistic deployment scenarios.

6.2.1 Cross-Workload Generalization. Models struggle to generalize across workload types. A predictor trained on CNNs may fail on transformers due to different computational patterns: CNNs are compute-dominated by convolutions with high data reuse, while transformers feature attention mechanisms with sequence-length-dependent memory access patterns.

Promising directions include workload-agnostic representations (learning from computation graphs rather than architecture-specific features) and multi-task learning across workload families.

6.2.2 Cross-Hardware Generalization. Hardware generalization faces fundamental obstacles. Different hardware families (CPUs, GPUs, TPUs, FPGAs) employ distinct execution models, memory hierarchies, and parallelism patterns. Even within GPU families, architectural changes (Volta to Ampere to Hopper) introduce new features (tensor cores, TMA, FP8) that alter performance characteristics.

Transfer learning approaches [6, 10] show promise for related hardware, but truly cross-family prediction (e.g., GPU to TPU) remains elusive. Hardware-agnostic intermediate representations that capture essential computational patterns while abstracting platform details could enable broader transfer.

6.2.3 Distribution Shift. Performance models face distribution shift as software stacks evolve. Compiler optimizations, framework updates, and driver changes alter the workload-to-hardware mapping, invalidating models trained on older configurations.

Online adaptation and continual learning techniques could address distribution shift, but few studies systematically evaluate temporal generalization. Developing benchmarks that explicitly measure robustness to software evolution would accelerate progress.

6.3 Integration with Design Flows

For ML-based performance models to impact practice, they must integrate seamlessly with existing design and optimization workflows.

6.3.1 Compiler Integration. Compiler autotuning represents a natural application: ML models guide the search for optimal tensor program configurations. TVM [4] and Ansor [19] demonstrate this integration, but challenges remain.

Cost model accuracy directly affects autotuning efficiency. Mispredictions cause the search to explore suboptimal regions, wasting compilation time. Uncertainty quantification—knowing when predictions are unreliable—could enable more efficient exploration-exploitation trade-offs. Recent uncertainty-aware cost models can provide calibrated uncertainty estimates, but such techniques are not yet standard.

6.3.2 Architecture Exploration. Hardware design space exploration requires evaluating millions of configurations. ML surrogate models can accelerate this process, as demonstrated by ArchGym [8], but integration challenges persist.

The design space is often too large for exhaustive surrogate training. Active learning strategies that intelligently select which configurations to simulate could improve sample efficiency. Additionally, surrogate models must provide reliable uncertainty estimates to avoid overconfident predictions that mislead designers.

6.3.3 Serving System Optimization. LLM serving systems require real-time performance prediction for scheduling decisions. VIDUR [1] provides offline simulation, but online serving requires predictions within microseconds.

Lightweight models suitable for real-time inference, combined with periodic retraining on observed performance, could enable adaptive serving optimization. The challenge is maintaining accuracy while meeting strict latency requirements.

6.4 Research Opportunities from Taxonomy Gaps

Our taxonomy analysis reveals specific gaps where no existing work provides adequate solutions. We identify five high-priority research opportunities grounded in concrete unmet needs.

6.4.1 Transformer-Aware Cross-Platform Transfer. Gap: Transfer learning works (HELP [10], LitePred [6]) achieve 98%+ accuracy but

1393 focus on CNNs. No published work demonstrates cross-platform
 1394 transfer for attention-based models.

1395 **Evidence:** HELP’s evaluation uses NAS-Bench search spaces
 1396 containing only CNNs. LitePred’s 85-platform validation covers
 1397 MobileNet and ResNet variants. Meanwhile, transformers dominate
 1398 modern workloads with distinct memory access patterns (sequence-
 1399 length-dependent attention, KV cache growth) that differ fundamentally
 1400 from CNN data reuse.

1401 **Opportunity:** Extend meta-learning approaches to transformer-
 1402 specific operators (multi-head attention, layer normalization, feed-
 1403 forward blocks). This requires new benchmark datasets pairing
 1404 transformer architectures with edge/GPU measurements—currently
 1405 none exist publicly.

1406
 1407 **6.4.2 Uncertainty-Aware Autotuning Cost Models.** **Gap:** Of 60+ sur-
 1408veyed papers, only one addresses calibrated uncertainty quantifi-
 1409cation. TVM/Ansor cost models provide point predictions without
 1410confidence estimates.

1411 **Evidence:** Autotuning spends significant search budget evaluat-
 1412ing configurations where the cost model is uncertain [19]. Without
 1413uncertainty estimates, the search cannot distinguish “confident
 1414low-latency” from “uncertain low-latency” predictions.

1415 **Opportunity:** Integrate Bayesian neural networks or ensem-
 1416ble methods into compiler cost models with calibrated confidence
 1417intervals. The reward: provably fewer measured evaluations dur-
 1418ing autotuning, directly reducing compilation time. TenSet [20]
 1419provides 52M records for training and validation.

1420
 1421 **6.4.3 Dynamic Shape and Sparse Workload Prediction.** **Gap:** Exist-
 1422ing models assume fixed input shapes. No surveyed work handles
 1423dynamic shapes (variable batch size, sequence length) or activation
 1424sparsity (early exit, MoE gating).

1425 **Evidence:** nn-Meter [18] requires shape specification at predic-
 1426tion time. NeuSight [11] assumes fixed tile configurations. Real
 1427LLM serving sees sequences from 128 to 128K tokens; mixture-
 1428of-experts models activate 2 of 64 experts per token—both create
 1429highly variable execution patterns.

1430 **Opportunity:** Shape-parameterized prediction networks that
 1431take input dimensions as features, combined with sparsity-aware
 1432roofline models. Sparseloop [16] provides analytical foundations
 1433for sparse tensors; extending this to learned models for dynamic
 1434workloads is unexplored.

1435
 1436 **6.4.4 Unified Energy-Latency-Memory Prediction.** **Gap:** Energy
 1437prediction remains fragmented. Accelergy provides analytical en-
 1438ergy estimates; ML-based approaches focus almost exclusively on
 1439latency.

1440 **Evidence:** Table 3 shows all surveyed ML models target la-
 1441tency or throughput. Meanwhile, edge deployment is often energy-
 1442constrained (battery budget, thermal limits), and datacenter cost
 1443optimization increasingly considers Joules-per-inference alongside
 1444latency SLOs.

1445 **Opportunity:** Multi-task learning that jointly predicts latency,
 1446energy, and peak memory from shared representations. Timeloop+Accelergy
 1447provides paired latency-energy labels for spatial accelerators; ex-
 1448tending this to GPU/edge devices requires new measurement in-
 1449frastructure but offers immediate practical value.

1451
Table 4: Component-based reproducibility evaluation. Each
 1452 tool is scored against explicit criteria: Setup (3 pts), Repro-
 1453ducibility (4 pts), and Usability (3 pts).

Tool	Setup	Reprod.	Usability	Total
Timeloop	3	4	2	9/10
ASTRA-sim	1	2.5	3	6.5/10
VIDUR	1.5	2	3	6.5/10
nn-Meter	2	0	1	3/10

1455
 1456 **6.4.5 Temporal Robustness Benchmarks.** **Gap:** No benchmark sys-
 1457tematically measures model robustness to software evolution. Mod-
 1458els may achieve 99% accuracy on static datasets but fail when dri-
 1459vers, compilers, or frameworks update.

1460
 1461 **Evidence:** Our nn-Meter evaluation (Section 7) found pre-trained
 1462 predictors broken by scikit-learn version changes—a concrete in-
 1463stance of temporal fragility. TenSet versioning provides some tem-
 1464poral signal but lacks explicit evaluation protocols.

1465
 1466 **Opportunity:** Create versioned benchmark suites with measure-
 1467ments across software configurations (CUDA 11.x vs 12.x, PyTorch
 14681.x vs 2.x, TensorRT versions). Define temporal generalization met-
 1469rics: accuracy on measurements from software stacks released N
 1470months after training data. This would enable principled evalua-
 1471tion of continual learning approaches.

7 Experimental Evaluation

1472
 1473 To validate the practical applicability of surveyed performance
 1474modeling tools, we conducted hands-on reproducibility evaluations
 1475of five representative systems spanning different hardware targets
 1476and modeling approaches. This section presents our methodology,
 1477tool-by-tool findings, and synthesizes key lessons for practitioners.

7.1 Evaluation Methodology

1478
 1479 We evaluated each tool using a transparent, additive rubric designed
 1480to enable reproducible assessment. Our 10-point rubric comprises
 1481three components, each with explicit criteria:

1482
 1483 **Installation & Setup (3 points).** We award points for: Docker/container
 1484availability (+1), clean pip/conda installation (+1), and time-to-first-
 1485result under 30 minutes (+1). Deductions apply for undocumented
 1486dependencies (-1) or strict version requirements (-0.5). Tools were
 1487tested in clean environments following documented procedures.

1488
 1489 **Result Reproducibility (4 points).** We assess: reference out-
 1490puts provided (+1.5), deterministic results (+1), comprehensive ex-
 1491amples (+1), and validation scripts (+0.5). Core functionality failures
 1492incur a -2 deduction. This component carries the highest weight
 1493as reproducibility is essential for scientific validation.

1494
 1495 **Practical Usability (3 points).** We evaluate: clear API/interface
 1496(+1), output interpretability (+1), active maintenance (+0.5), and
 1497community adoption (+0.5).

1498
 1499 Table 4 presents the component-level breakdown for each evalua-
 1500ted tool, enabling transparent comparison and independent verifi-
 1501cation of our assessments.

1502
 1503
 1504
 1505
 1506
 1507
 1508

1509 7.2 Tool-by-Tool Results

1510 **7.2.1 Timeloop: DNN Accelerator Modeling.** Timeloop [12] provides analytical performance and energy modeling for DNN accelerators through loop-nest analysis.

1511 **Setup.** Docker-based installation succeeds in 10–15 minutes with
1512 pre-built images for both x86 and ARM platforms. Native installation
1513 requires 1–2 hours due to complex dependencies (Barvinok,
1514 NTL libraries).

1515 **Reproducibility.** Excellent—reference outputs are provided for
1516 all example architectures (Eyeriss, Simba), and results are deterministic.
1517 Tutorials with Jupyter notebooks enable systematic learning.

1518 **Key Finding.** Energy breakdown analysis reveals DRAM dominates
1519 (>60%) for typical configurations, validating the importance
1520 of dataflow optimization. The mapper may not find globally optimal
1521 solutions but provides interpretable trade-off analysis.

1522 **7.2.2 ASTRA-sim: Distributed Training Simulation.** ASTRA-sim [15]
1523 simulates distributed DNN training with configurable network back-
1524 ends.

1525 **Setup.** Docker recommended due to Protobuf version sensitivity.
1526 Native build requires 1–2 hours with careful dependency manage-
1527 ment.

1528 **Reproducibility.** Good—validated configurations for HGX-H100
1529 and DGX-V100 are included. However, reference timing outputs
1530 are not provided, requiring trust in published accuracy claims.

1531 **Key Finding.** The Chakra trace format has a learning curve,
1532 but enables detailed collective communication modeling. Multiple
1533 network backends (analytical, NS-3) allow accuracy-speed trade-
1534 offs.

1535 **7.2.3 VIDUR: LLM Inference Simulation.** VIDUR [1] provides discrete-
1536 event simulation for LLM serving systems.

1537 **Setup.** Python-only installation, but strict Python 3.10 require-
1538 ment creates compatibility issues—Python 3.14 fails due to argparse
1539 API changes.

1540 **Reproducibility.** Good for supported configurations. Pre-profiled
1541 data covers A100, H100, and A40 GPUs for Llama-family models.
1542 Adding new models requires GPU access for profiling.

1543 **Key Finding.** Rich scheduler implementations (vLLM, Orca,
1544 Sarathi) enable direct algorithm comparison. Metrics include time-
1545 to-first-token, time-per-output-token, and memory utilization—
1546 essential for SLO-driven capacity planning.

1547 **7.2.4 nn-Meter: Edge Device Latency Prediction.** nn-Meter [18] pre-
1548 dicts DNN latency on edge devices through kernel-level decom-
1549 position.

1550 **Setup.** Simple pip installation, but critical compatibility issue:
1551 pre-trained predictors fail to load with current scikit-learn versions
1552 due to pickle format changes.

1553 **Reproducibility.** Poor in current state—the core functionality
1554 (pre-trained predictors) is broken without pinning scikit-learn to
1555 version 1.0.x.

1556 **Key Finding.** This case highlights a critical reproducibility anti-
1557 pattern: ML models serialized with pickle are fragile across library
1558 versions. Researchers should prefer version-agnostic serialization
1559 formats (ONNX, SavedModel) or pin exact dependency versions.

1560 **Table 5: Reproducibility best practices derived from tool eval-
1561 uation.**

1562 Practice	1563 Rationale
1564 Provide Docker images	1565 Isolates dependencies
1566 Document Python version	1567 Prevents API incompatibilities
1568 Include reference outputs	1569 Enables result verification
1570 Use portable model formats	1571 Avoids pickle versioning issues
1572 Pin dependency versions	1573 Ensures reproducible environments

1574 **7.2.5 NeuSight: ML-Based GPU Performance Prediction.** NeuSight [11]
1575 provides tile-based GPU performance prediction achieving 97.7%
1576 accuracy across GPU generations.

1577 **Setup.** Python installation with PyTorch and CUDA dependen-
1578 cies. GPU required for model calibration and validation; prediction
1579 can run on CPU after calibration.

1580 **Reproducibility.** Good—methodology clearly described in paper
1581 with hybrid analytical+neural approach. Per-GPU calibration
1582 requires profiling runs, but pre-trained models expected for com-
1583 mon architectures.

1584 **Key Finding.** NeuSight’s tile-based decomposition mirrors ac-
1585 tual GPU execution (CUDA thread blocks), enabling accurate pre-
1586 dictions that generalize across workloads. The hybrid approach
1587 combining roofline analysis with learned components achieves
1588 2.3% mean error on LLM workloads (GPT-3 on H100), dramatically
1589 improving over pure analytical methods (121.4% error).

1590 7.3 Synthesis and Recommendations

1591 Our rubric-based evaluation reveals systematic patterns affecting
1592 reproducibility. Notably, applying explicit criteria adjusted some
1593 scores: ASTRA-sim dropped from an informal 8/10 to 6.5/10 due
1594 to missing reference outputs, while nn-Meter’s broken core func-
1595 tionality resulted in 3/10 rather than our initial generous 5/10. This
1596 underscores the importance of transparent methodology.

1597 **Containerization dramatically improves reproducibility.** Tools providing Docker images (Timeloop, ASTRA-sim) achieve
1598 higher setup scores by isolating complex dependency chains. Native
1599 builds consistently encounter platform-specific issues.

1600 **Python version sensitivity is a major concern.** VIDUR re-
1601 quires Python 3.10 specifically; nn-Meter’s pickle files are incom-
1602 patible with current scikit-learn. Projects should document version
1603 constraints prominently and consider providing locked dependency
1604 specifications.

1605 **Pre-trained models age poorly.** nn-Meter’s reliance on pickled
1606 scikit-learn models created a time bomb. NeuSight mitigates this
1607 through its hybrid approach—the analytical roofline component
1608 provides robustness while the learned components can be retrained.
1609 For projects distributing trained models, ONNX or similar portable
1610 formats are preferable.

1611 **Reference outputs enable validation.** Timeloop’s inclusion
1612 of expected outputs for all examples enables immediate verification.
1613 ASTRA-sim and VIDUR lack this, requiring users to trust published
1614 accuracy claims.

1615 Table 5 summarizes best practices derived from our evaluation.

1625 8 Conclusion

1626 This survey has provided a comprehensive analysis of machine
 1627 learning approaches for computer architecture performance mod-
 1628 eling. We have examined over 60 papers spanning traditional anal-
 1629 lytical models, simulation-based approaches, and modern ML tech-
 1630 niques including classical machine learning, deep learning, graph
 1631 neural networks, and hybrid methods.

1633 8.1 Key Findings

1634 Our analysis reveals several key findings that characterize the cur-
 1635 rent state of the field:

1636 **ML-based models achieve remarkable accuracy.** State-of-
 1637 the-art approaches achieve prediction errors below 5% for their
 1638 target domains. NeuSight [11] reaches 2.3% error on GPU infer-
 1639 ence through physics-informed tile-based prediction. LitePred [6]
 1640 achieves 0.7% error across 85 edge platforms through transfer learn-
 1641 ing. These accuracy levels are sufficient for production deployment
 1642 in neural architecture search, autotuning, and hardware-aware op-
 1643 timization.

1644 **Hybrid approaches dominate recent work.** The most suc-
 1645 cessful models combine analytical structure with learned compo-
 1646 nents. Analytical decomposition provides interpretable baselines
 1647 and physics-based inductive bias, while ML captures complex ef-
 1648 fects that elude closed-form analysis. This hybrid philosophy—
 1649 exemplified by NeuSight’s tile-based prediction and VIDUR’s [1]
 1650 simulation-based framework—consistently outperforms pure analy-
 1651 tical or pure ML approaches.

1652 **Transfer learning is essential for scalability.** The proliferation
 1653 of hardware platforms makes per-device training impractical.
 1654 Meta-learning (HELP [10]) and VAE-based sampling (LitePred [6])
 1655 enable adaptation to new devices with 10–100 samples, demon-
 1656 strating that cross-platform generalization is tractable.

1657 **Kernel-level decomposition improves accuracy.** nn-Meter’s [18]
 1658 insight that end-to-end latency decomposes into kernel latencies
 1659 has become standard practice. By modeling at the kernel level and
 1660 capturing framework fusion behavior, models achieve composi-
 1661 tional predictions that generalize across architectures.

1662 **LLM inference presents unique challenges.** Large language
 1663 model serving has distinct characteristics—autoregressive genera-
 1664 tion, KV cache growth, prefill-decode phase separation—that re-
 1665 quire specialized modeling. VIDUR [1] and similar frameworks
 1666 provide discrete-event simulation capturing these dynamics with
 1667 <5% accuracy.

1669 8.2 Promising Research Directions

1670 Looking forward, we identify the most promising directions for
 1671 advancing the field:

1672 **Foundation models for performance prediction.** Pre-trained
 1673 models that transfer across workloads and hardware could dramat-
 1674 ically reduce data requirements for new prediction tasks. Creating
 1675 the large-scale, diverse datasets needed to train such models rep-
 1676 presents a key community challenge.

1677 **Uncertainty quantification.** Knowing when predictions are
 1678 reliable enables better decision-making in autotuning, design space
 1679 exploration, and serving optimization. Calibrated uncertainty esti-
 1680 mates remain underexplored despite their practical importance.

1681 **Temporal generalization.** As software stacks evolve, perfor-
 1682 mance models must adapt. Continual learning approaches and
 1683 benchmarks measuring robustness to software evolution deserve
 1684 increased attention.

1685 **Multi-objective prediction.** Practical deployment involves lat-
 1686 ency, throughput, energy, memory, and cost trade-offs. Joint multi-
 1687 objective prediction could enable Pareto-optimal design selection
 1688 across these dimensions.

1689 **Emerging hardware support.** Processing-in-memory, neuro-
 1690 morphic computing, and analog accelerators require new modeling
 1691 paradigms. Early-stage performance modeling for emerging hard-
 1692 ware could accelerate adoption.

1693 8.3 Concluding Remarks

1694 Machine learning has transformed performance modeling from an
 1695 art requiring deep architectural intuition to an increasingly system-
 1696 atic discipline. The surveyed approaches demonstrate that learned
 1697 models can capture complex hardware-software interactions while
 1698 enabling millisecond-scale prediction. As ML workloads continue
 1699 to grow in importance and hardware diversity expands, accurate,
 1700 generalizable performance models will become ever more critical
 1701 for efficient system design and deployment.

1702 We hope this survey serves as both a comprehensive reference
 1703 for practitioners selecting performance modeling approaches and
 1704 a roadmap for researchers identifying impactful open problems.
 1705 The field’s rapid progress suggests that the coming years will bring
 1706 continued advances in accuracy, generalization, and practical de-
 1707 ployment of ML-based performance models.

1708 References

- [1] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramachandran. 2024. VIDUR: A Large-Scale Simulation Framework for LLM Inference. In *Proceedings of Machine Learning and Systems (MLSys)*. 1–15.
- [2] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 163–174. <https://doi.org/10.1109/ISPASS.2009.4919648>
- [3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 Simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7. <https://doi.org/10.1145/2024716.2024718>
- [4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Liannmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 578–594.
- [5] Lukasz Dudziak, Thomas Chau, Mohamed S. Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas D. Lane. 2024. Latency Predictors for Neural Architecture Search. In *Proceedings of Machine Learning and Systems (MLSys)*. 1–14.
- [6] Yang Feng, Zhehao Li, Jiacheng Yang, and Yunxin Liu. 2024. LitePred: Transferable and Scalable Latency Prediction for Hardware-Aware Neural Architecture Search. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 1–18.
- [7] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*. 473–486. <https://doi.org/10.1109/ISCA45697.2020.00047>
- [8] Srivatsan Krishnan, Amir Yazdanbakhsh, Shvetank Prakash, Norman P. Jouppi, Jignesh Parmar, Hyoukjun Kim, James Laudon, and Chandrakanth Narayanaswami. 2023. ArchGym: An Open-Source Gymnasium for Machine Learning Assisted Architecture Design. In *Proceedings of the 50th International Symposium on Computer Architecture (ISCA)*. 1–16. <https://doi.org/10.1145/3579371.3589049>

1736
 1737
 1738
 1739
 1740

- 1741 [9] Hyoukjun Kwon, Prasanth Chatarasi, Michael Sarber, Michael Pellauer, Angshuman Parashar, and Tushar Krishna. 2019. MAESTRO: A Data-Centric Approach
1742 to Understand Reuse, Performance, and Hardware Cost of DNN Mappings. In
1743 *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–14. <https://doi.org/10.1145/3352460.3358292>
- 1744 [10] Hayeon Lee, Sewoong Lee, Song Chong, and Sung Ju Hwang. 2021. HELP:
1745 Hardware-Adaptive Efficient Latency Prediction for NAS via Meta-Learning.
1746 In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 34. 27016–
1747 27028.
- 1748 [11] Seunghyun Lee, Amar Phanishayee, and Divya Mahajan. 2025. NeuSight: GPU
1749 Performance Forecasting via Tile-Based Execution Analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming
1750 Languages and Operating Systems (ASPLOS)*. 1–15.
- 1751 [12] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen,
1752 Victor A. Ying, Anurag Muber, Rangharajan Venkatesan, Brucek Khailany,
1753 Stephen W. Keckler, and Joel Emer. 2019. Timeloop: A Systematic Approach
1754 to DNN Accelerator Evaluation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 304–315.
<https://doi.org/10.1109/ISPASS.2019.00042>
- 1755 [13] Ondrej Sykora, Alexis Rucker, Charith Mendis, Rajkishore Barik,
1756 Phitchaya Mangpo Phothilimthana, and Saman Amarasinghe. 2022. GRANITE:
1757 A Graph Neural Network Model for Basic Block Throughput Estimation. In
1758 *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. 1–13. <https://doi.org/10.1109/IISWC55918.2022.00014>
- 1759 [14] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An
1760 Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (2009), 65–76. <https://doi.org/10.1145/1498765.1498785>
- 1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798 [15] William Won, Taekyung Heo, Saeed Rashidi, Saeed Talati, Srinivas Srinivasan,
1799 and Tushar Krishna. 2023. ASTRA-sim2.0: Modeling Hierarchical Networks and
1800 Disaggregated Systems for Large-Model Training at Scale. In *Proceedings of the IEEE
1801 International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 283–294. <https://doi.org/10.1109/ISPASS57527.2023.00035>
- 1802 [16] Yannan Nellie Wu, Joel Emer, and Vivienne Sze. 2022. Sparseloop: An Analytical
1803 Approach to Sparse Tensor Accelerator Modeling. In *Proceedings of the 55th IEEE/ACM
1804 International Symposium on Microarchitecture (MICRO)*. 1–15. <https://doi.org/10.1109/MICRO56248.2022.00078>
- 1805 [17] Geoffrey X. Yu, Yubo Gao, Pavel Golber, and Asaf Cidon. 2021. Habitat: A
1806 Runtime-Based Computational Performance Predictor for Deep Neural Network
1807 Training. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 503–
1808 521.
- 1809 [18] Li Lyra Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang,
1810 and Yunxin Liu. 2021. nn-Meter: Towards Accurate Latency Prediction of Deep-
1811 Learning Model Inference on Diverse Edge Devices. In *Proceedings of the 19th Annual
1812 International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 81–93. <https://doi.org/10.1145/3458864.3467882> Best Paper Award.
- 1813 [19] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer
1814 Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez,
1815 and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs
1816 for Deep Learning. In *Proceedings of the 14th USENIX Symposium on Operating
1817 Systems Design and Implementation (OSDI)*. 863–879.
- 1818 [20] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E. Gonzalez,
1819 Ion Stoica, and Zhihao Zhang. 2021. TenSet: A Large-scale Program Performance
1820 Dataset for Learned Tensor Compilers. In *Advances in Neural Information
1821 Processing Systems (NeurIPS)*, Vol. 34. 29876–29888.
- 1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856