

**Data-Efficient AI-Guided Energy- and Thermal-Aware Scheduling on Heterogeneous
Multicore Systems**

by

Mohammad Pivezhandi

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2025

MAJOR: COMPUTER SCIENCE

Approved By:

Dr. Abusayeed Saifullah, Ph.D. Advisor Date

Dr. Ali Jannesari Date

Dr. Zheng Dong Date

Dr. Nathan Fisher Date

DEDICATION

Dedicated to my cherished family.

ACKNOWLEDGEMENTS

I extend heartfelt gratitude to my advisor, Dr. Abusayeed Saifullah, for his unwavering guidance, encouragement, and support over the past five years. His insightful suggestions and motivation were crucial to the completion of this dissertation. I am also deeply thankful to Professor Ali Jannesari and Dr. Prashant Modekurthy for their valuable advice and collaborative spirit throughout my Ph.D. journey. Additionally, I appreciate Dr. Zheng Dong and Dr. Nathan Fisher for graciously agreeing to serve on my Ph.D. committee. I would also like to thank Mahdi Banisharif Dehkordi and Saeed Bakhshan for their help in technical parts of the project, including the projects on zero-shot reinforcement learning development. I am also grateful to my fiancée, Arefeh, for her love and encouragement. Above all, I am profoundly grateful to my family, particularly my father and my mother, for their unwavering emotional support during my doctoral studies.

TABLE OF CONTENTS

Dedication	ii
Acknowledgements	iii
List of Tables	xiv
List of Figures	xvi
Chapter 1 Introduction	1
Chapter 2 HiDVFS: A Hierarchical Multi-Agent DVFS Scheduler for OpenMP DAG Workloads	6
2.1 Introduction	1
2.2 Background and Motivation	5
2.2.1 Background	5
2.2.2 Inefficiency of Current Task-to-Core Allocations	7
2.2.3 Irregular Parallel Execution of DAGs and Dependency on Performance Profiling Features	9
2.3 Related Work	12
2.4 Design of HiDVFS	16
2.4.1 Collaborative Multi-Agent Reinforcement Learning (MARL) for High-Dimensional Spaces	17
2.4.2 Enhanced Reward Estimation	17
2.4.3 Reward Function Estimation	18

2.4.4	Hyperparameter Tuning and Target Metrics	20
2.4.5	Complexity Analysis of Agents	22
2.4.6	Hierarchical MARL Algorithm with Reward Model	23
2.4.7	Practical Considerations: Platform and Parallel Execution	25
2.4.8	Illustrative Walkthrough of HiDVFS Decision Making	26
2.4.9	Reward Function Sensitivity Analysis	28
2.5	Experimental Platform, Benchmark, and Evaluation	28
2.5.1	Platforms and Setup	28
2.5.2	Evaluation of Metrics and Statistical Analysis of Features	31
2.5.3	Implemented Approaches	34
2.5.4	Comparison with Baselines and Results	36
2.6	Conclusion	40
Chapter 3	Feature-Aware Task-to-Core Allocation in Embedded Multi-core Platforms via Statistical Learning	41
3.1	Introduction	1
3.2	Motivation and Challenges	5
3.2.1	Curse of Dimensionality	5
3.2.2	Large Data Management	6
3.2.3	Performance Unpredictability	6
3.3	Design Methodology	7
3.3.1	Data Collection and Environment Setup	8

3.3.2	Embedded Feature Selection Using Random Forest	10
3.3.3	Wrapper-Based Feature Selection	12
3.3.4	Filter-Based Feature Selection	15
3.3.5	Summary of the Multi-Stage Methodology	19
3.4	Experiments	20
3.4.1	Experimental Platform, Benchmarks, and Evaluation	20
3.4.2	Implementation and Training Details	21
3.4.3	Empirical Results	22
3.4.4	Comparative Model Analysis	25
3.5	Related Work	28
3.6	Conclusion	30
Chapter 4	FlowRL: Flow-Augmented Few-Shot Reinforcement Learning for Semi-Structured Sensor Data	31
4.1	Introduction	1
4.2	Preliminaries	3
4.2.1	Q-learning and Online RL	3
4.2.2	Regret Bounds and Sample Efficiency	4
4.2.3	Flow Matching for Generative Modeling	5
4.3	Design of Distribution-Aware Flow Matching for Data Generation	6
4.3.1	Redefinition of the Regret Function	6
4.3.2	Flow Matching and Bootstrapping	6

4.3.3	Feature Weighting and Conditional Flow Matching	8
4.3.4	Implications for RL: Regret Bounds under Model-Based Assumptions	9
4.3.5	Implications for RL: Regret Bounds with Synthetic Data in Model-Free RL	11
4.3.6	From LLN to Finite-Sample Guarantees	12
4.3.7	Algorithmic Overview for Few-Shot Online RL	12
4.4	Experiments	13
4.5	Related Work	20
4.6	Conclusion	22
Chapter 5	GraphPerf-RT: A Graph-Driven Performance Model for Hardware-Aware Scheduling of OpenMP Codes	24
5.1	Introduction	1
5.2	Related Work	5
5.3	Preliminaries	8
5.3.1	Problem Formulation	8
5.3.2	Heterogeneous Graph Abstraction	11
5.3.3	Data Collection and Artifact Pipeline	12
5.3.4	Learning Formulation and Outputs	14
5.4	Design Methodology	15
5.4.1	Heterogeneous Graph Representation	16
5.4.2	Graph Neural Network Architecture	21
5.4.3	Evidential Learning for Uncertainty Quantification	24

5.4.4	Training Procedure	26
5.5	Experiments	28
5.5.1	Experimental Setup	28
5.5.2	Baselines and Metrics	33
5.5.3	Results and Analysis	35
5.5.4	RL Baseline Evaluation	39
5.6	Conclusion	42
Chapter 6	ZeroDVFS: Zero-Shot LLM-Guided Core and Frequency Allocation for Embedded Platforms	43
6.1	Introduction	1
6.2	Related Work	6
6.2.1	LLM-Based Code Analysis for Performance Prediction	8
6.3	Background and Motivation	9
6.3.1	Background	9
6.3.2	Motivation on Model-Based MARL	12
6.3.3	Limitations of Utilization-Based DVFS	12
6.3.4	Challenges in Task-to-Core Allocation	13
6.3.5	The Need for Environment Modeling and Multi-Agent RL	15
6.3.6	LLM-Based Code Analysis for Workload Profiling	16
6.4	Design of Model-Based Thermal- and Energy-Aware MARL	20
6.4.1	Design of MARL	20

6.4.2	Environment Design and Modeling	27
6.4.3	Cross-Platform Model Transfer and Adaptation	34
6.4.4	LLM-Based Semantic Feature Extraction	36
6.5	Experimental Results	41
6.5.1	Implemented Algorithms	41
6.5.2	Experimental Platform, Benchmark, and Evaluation	43
6.5.3	Temperature and Performance Prediction	45
6.5.4	RL Integration and Convergence Analysis	46
6.5.5	LLM Feature Contribution and Ablation Analysis	54
6.6	Conclusion	59
Chapter 7	Conclusion	61
7.1	Summary of Contributions	61
7.2	Integrated Framework and Validation	63
7.3	Broader Impact	63
7.4	Future Directions	64
Chapter A	HiDVFS: Supplementary Material	66
A.1	Detailed BOTS Benchmark Results	66
A.1.1	Experimental Setup	66
A.1.2	Benchmark Descriptions	66
A.1.3	Per-Benchmark Detailed Results	67

A.1.4	Key Observations	68
A.1.5	Summary Statistics	69
A.2	SARB Single-Agent RL Version Evaluation	69
A.2.1	Multi-Seed Version Selection	70
A.2.2	Version Descriptions	70
A.2.3	Experimental Setup	71
A.2.4	Version Comparison Results	71
A.2.5	Visual Analysis	72
A.2.6	Key Findings and Recommendations	72
A.3	Multi-Seed Validation Tables	76
A.3.1	Per-Seed Training Results	76
A.3.2	Per-Seed Finetuning Results	77
A.3.3	All-Seeds Summary with 95% CI	77
A.3.4	HiDVFS vs GearDVFS Multi-Seed Comparison	78
A.4	Complete 7-Metric Comparison	78
A.4.1	Metric Definitions	79
A.4.2	Complete Comparison Table	79
A.5	BOTS Per-Benchmark Energy Analysis	80
A.5.1	Energy Efficiency Observations	81
A.6	RL Algorithm Energy Comparison	81
A.7	Hardware Counter Analysis	82

A.7.1	Cache and Branch Miss Analysis	82
A.7.2	Hardware Counter Summary	84
A.7.3	Key Observations	84
A.8	Multi-Seed Convergence Plots	85
A.8.1	Makespan Convergence	86
A.8.2	Core Allocation Convergence	87
A.8.3	Frequency Selection Convergence	87
Chapter B	ZeroDVFS: Supplementary Material	89
B.1	LLM-Based Feature Extraction: Supplementary Details	89
B.1.1	Semantic Feature Taxonomy	89
B.1.2	Prompting Strategy and Implementation	89
B.1.3	Feature Encoding for Machine Learning	91
B.1.4	Experimental Dataset	92
B.1.5	Inter-Model Agreement Analysis	93
B.1.6	Computational Cost and Latency	94
B.1.7	Cost Projections at Scale	96
B.1.8	Robustness Analysis	96
B.1.9	Error Analysis by Benchmark Category	97
B.1.10	Limitations and Future Directions	98
Chapter C	GraphPerf-RT: Supplementary Material	102

C.1	Overview and Usage	102
C.2	ALF→DAG Pipeline (Reproduction)	102
C.3	Feature Reference	103
C.4	Hierarchical MARL and Model-Based Planning	104
C.5	Statistical and Calibration Protocols	105
C.6	Platform Control and Safety	105
C.7	Reproducibility Artifacts	105
C.8	Theoretical Foundations of DAG Evaluation Metrics	106
C.8.1	DAG Structural Metrics	107
C.8.2	Graph Complexity Measures for DAGs	108
C.8.3	Evidential Uncertainty Theory	109
C.9	Extended Multi-Platform Experiments	111
C.9.1	Per-Platform Dataset Statistics	111
C.9.2	Per-Platform Performance with Confidence Intervals	111
C.9.3	Cross-Platform Transfer Experiments	112
C.9.4	Benchmark-Level Holdout Experiments	112
C.9.5	Statistical Significance Tests	113
C.9.6	Extended RL Baseline Results	113
C.9.7	Scalability Analysis	114
C.9.8	Inference Latency and Memory Footprint	115
C.9.9	Discussion: x86 and GPU Applicability	117

C.10 Discussion and Future Work	120
Abstract	146
Autobiographical Statement	149

LIST OF TABLES

2.1	Agent States and Actions	20
2.2	Reward Parameter Sensitivity	28
2.3	HiDVFS vs GearDVFS Performance Across BOTS Benchmarks (Jetson TX2, Seed 42, Finetuned)	30
2.4	Statistical Analysis of Key Features' Influence on Performance Metrics	34
2.5	Comparison of RL Approaches (Finetuned, Multi-Seed Mean±Std)	39
3.1	Key features ranked by Random Forest and backward stepwise OLS (lowest CV Error) for Profiler and Temperature datasets. Temp Core 0-13 denotes per-core temperatures; Δ Temp is the temperature difference.	24
3.2	Temperature predictors (θ_i) vs. energy: Estimates (Est.), t-values (t-val.), p-values (p-val.), sorted by p-value. Significant predictors (e.g., θ_1 , θ_0 , θ_2) show strong energy impact.	24
3.3	Profiler predictors (excl. temp, freq, speed) vs. energy: Estimates (Est.), t-values (t-val.), p-values (p-val.), sorted by p-value. Key predictors (e.g., Context Switch Rate, Elapsed Time) drive energy variance.	25
3.4	MSE percentage and total number of parameters for different architectures on Intel Core i7 12th Gen.	26
4.1	Main Hyperparameters and Tuning Values.	15
5.1	Node types and their performance rationale.	19
5.2	Edge types and their performance rationale.	20
5.3	Ablation study: contribution of each component to prediction accuracy. Full model achieves $R^2 = 0.97$	28
5.4	Overall performance across all platforms and benchmarks. Lower is better for RMSE/MAE/MAPE. Higher is better for R^2 /Spearman.	31
5.5	Uncertainty calibration summary (lower ECE/MCE is better; higher reliability is better).	37
5.6	RL Baseline Performance on Jetson TX2 (200 episodes, 5 seeds).	39
5.7	Model-Based vs Model-Free Comparison.	40
5.8	Computational Complexity Comparison	41
6.1	Inter-Model Agreement Rates for Semantic Features (%)	39
6.2	Experimental Platform Specifications	43
6.3	Comparing profiler and temperature models regarding accuracy, inference latency (mean ± std), and complexity. Inference latency measured on Jetson TX2.	45
6.4	Energy Efficiency Comparison Results	51
6.5	Cross-Platform Transfer Learning Results (Source: TX2)	51
6.6	Execution Time Prediction Accuracy Across Feature Configurations	56

6.7	Top 15 Features by Importance (Claude + TX2 + All Features)	56
6.8	LLM Feature Extraction vs. Traditional Profiling	59
A.1	BOTS Benchmark Characteristics	66
A.2	Training Phase: Per-Benchmark Comparison	67
A.3	Finetuning Phase: Per-Benchmark Comparison	68
A.4	SARB Version Configuration Summary	70
A.5	SARB Version Comparison: Finetuned Phase (100 Epochs, Seed 42)	71
A.6	Per-Seed Training Performance Comparison	76
A.7	Per-Seed Finetuning Performance Comparison	77
A.8	Multi-Seed Summary: Mean \pm Std Across All Seeds (Finetuned)	78
A.9	Multi-Seed Validation: HiDVFS vs GearDVFS on FFT (Mean \pm Std, Seeds 42, 123, 456)	78
A.10	Complete 7-Metric Comparison (Finetuned Phase, All Seeds Mean \pm Std)	80
A.11	BOTS Per-Benchmark Energy Consumption (Finetuned Phase)	80
A.12	Hardware Counter Comparison (Finetuned Phase, Multi-Seed Mean \pm Std)	85
B.1	Complete Definitions of LLM-Extracted Semantic Features	90
B.2	Feature Encoding Mappings	92
B.3	Dataset Statistics by Platform	93
B.4	Detailed Inter-Model Agreement Analysis	94
B.5	LLM Feature Extraction Cost and Latency Breakdown	95
B.6	Feature Extraction Latency: LLM vs. Traditional Profiling	96
B.7	Cost Projections: LLM Extraction vs. Manual Profiling	96
B.8	Prediction Accuracy by Benchmark Category (Jetson TX2)	98
C.1	Encoder feature groups and examples.	103
C.2	Supplementary telemetry and RL-facing features (emitted; some optional).	104
C.3	Per-Platform Dataset Statistics	111
C.4	Per-Platform Performance (5 seeds, mean \pm std)	112
C.5	Cross-Platform Transfer Results	112
C.6	Held-Out Benchmark Results (6 unseen benchmarks)	113
C.7	Wilcoxon Signed-Rank Test: GraphPerf-RT vs Het. Graph Trans.	113
C.8	Extended RL Baseline Results on Jetson TX2 (5 seeds, 200 episodes)	113
C.9	Per-benchmark prediction accuracy and code complexity metrics. Benchmarks grouped by complexity tier.	115
C.10	Inference latency (ms) and memory footprint by platform. Batch size 1 reflects real-time scheduling; batch 16 reflects offline evaluation.	116
C.11	Inference cost comparison: evidential vs. ensemble vs. MC Dropout. All measured on TX2 GPU, batch size 1.	117
C.12	ARM-to-x86 feature mapping summary.	119
C.13	Proposed GPU graph representation.	120

LIST OF FIGURES

1.1 Unified framework showing how the five dissertation contributions interconnect. Feature-Aware statistical selection identifies critical features for scheduling agents. HiDVFS provides the hierarchical multi-agent DVFS foundation. ZeroDVFS enables cross-platform transfer via model-based learning. FlowRL generates synthetic training data through distribution-aware flow matching. GraphPerf-RT provides uncertainty-aware performance predictions as a world model for model-based RL.	2
2.1 OpenMP DAG snippet showing tied/untied tasks and dependency-induced variability in execution time.	8
2.2 Monitoring one user DAG (τ_1): five jobs ($J_{1,1,1}$ – $J_{3,1,4}$) mapped to cores (c_1 – c_3) with target frequencies (f_0 – f_2). Core c_0 runs system DAG (τ_0).	9
2.3 N-Queens on Xeon (12 cores) and Core i7 (4 cores). More cores increase branch misses for tied tasks, raising makespan and energy. Shaded regions show variation over 10 runs; serial reduces misses.	9
2.4 Scenarios on a four-core system. Blue: performance; green: powersave. PP, PS denote unbounded parallel execution; SP, SS denote bounded serial execution. The green dashed line marks DAG makespan. HiDVFS agents (7–9) align core choice, frequency, and temperature.	10
2.5 Importance of different features on total energy consumption and makespan in parallel and sequential execution of parallel applications.	11
2.6 IRL (A) aligns with expert behavior; our model (B) uses key-state predictions to shape rewards.	21
2.7 Profiling data under parallel and sequential modes for FFT workload variations (untied, tied, serial) show more spontaneity for profiling data in parallel mode. .	35
2.8 Variation of total energy consumption and makespan due to changes in priority combinations, number of cores, and frequency levels in parallel and sequential modes.	35
2.9 Makespan comparison (finetuned, seed 42). Top: Multi-agent approaches with HiDVFS achieving best makespan (4.16s L10). Bottom: Single-agent approaches with zTT leading (5.45s). Energy analysis in Appendix A.6.	36

2.10 Action analysis (finetuned, seed 42): HiDVFS and SARB converge to consistent high core counts and high frequencies in final episodes.	37
3.1 Energy consumption variation across three different processors with identical frequency and core count settings: Intel Core i7 8th Gen (Corei78) with 4 cores, Intel Core i7 12th Gen (Corei712) with 14 cores, and Intel Xeon 2680 v3 (Xeon) with 12 cores. The results are shown for three different OpenMP benchmarks.	7
3.2 Determining the significance of features using Random Forest with respect to cross-validation error on Intel Core i7 12th Gen.	7
3.3 Backward stepwise selection for estimating energy consumption and average temperature. Retaining fewer than 8 predictors (features) yields accurate predictions in both cases. Experiments performed on Intel Core i7 12th Gen.	16
3.4 Correlation matrix based on Pearson correlation coefficients for 10 selected cores from an Intel Core i7 12th Gen processor with 14 cores.	16
3.5 Comparison of average temperature and energy consumption for correlation-based (Corr) and random (Rand) core selection. Experiments performed on Intel Core i7 12th Gen processor with 14 cores.	23
3.6 Comparison of FCN models with and without feature selection and bootstrapping on Intel Core i7 12th Gen.	23
3.7 Results for temperature prediction from ground truth on Intel Core i7 12th Gen for regular FCN and FCN via Random Forest feature reduction strategy.	27
3.8 Results for profiler data prediction from ground truth on Intel Core i7 12th Gen for regular FCN and FCN via Random Forest feature reduction strategy.	27
4.1 Workflow of the proposed Flow-Augmented Reinforcement Learning (FlowRL) approach.	14
4.2 Correlation matrix comparisons.	17
4.3 Performance comparisons across methods.	18
4.4 Real and (Gen)erated data t-SNE.	18
4.5 Comparisons of distributions of FlowRL and Model-Based (MB) generated data.	19

5.1	End-to-end pipeline.	16
5.2	Final performance comparison of RL methods on Jetson TX2. Left: Makespan (s), Right: Energy (J). Error bars show ± 1 std across 5 seeds. MAMBRL-D3QN achieves the best performance in both metrics.	41
6.1	Performance metrics (CPU utilization, makespan, energy consumption, branch misses, context switches, and average temperature) for NVIDIA Jetson TX2 running the FFT benchmark parallelized through OpenMP API. Results highlight different responses of performance metrics to changes in different frequencies and selected cores.	19
6.2	MARL design: (a) Hierarchical action selection architecture and (b) reward function definitions.	22
6.3	The simplified design of model-based RL for temperature- and energy-aware core allocation for multi-core processors.	28
6.4	Comparison of (a) profiler data prediction using the one dimensional convolution model and (b) temperature prediction of 4 cores. Both use sensor data as ground truth on Intel Core i7 8th gen.	50
6.5	Comparison of (a) makespan and (b) energy consumption over 100 training episodes on BOTS FFT benchmark with input size 262144 on Jetson TX2. ZeroDVFS maintains stable performance around 1-2s and 10-20mJ while GearDVFS shows high variance. Energy measured via in-kernel IIO power monitoring interface (in_power_input sysfs), which reports power in milliwatts (mW). Energy computed as $E = \sum P_i \cdot \Delta t_i$ where Δt_i is the sampling interval (10ms). Y-axis units are millijoules (mJ).	51
6.6	Final performance ranking by makespan across all 11 baseline algorithms on BOTS FFT benchmark with input size 262144 on Jetson TX2. ZeroDVFS achieves best performance at 1.13s. Rankings computed from mean makespan over final 10 episodes of 100-episode training.	51
6.7	Decision latency breakdown: (i) RL decision components ($T_{RL} = 358$ ms in Python), (ii) total first-decision latency for new benchmarks showing $T_{total} = T_{LLM} + T_{static} + T_{RL}$ (3.48s with GPT-4o to 8.05s with DeepSeek), (iii) comparison showing ZeroDVFS first decision 8,300 \times faster and subsequent decisions 80,000 \times faster than table-based profiling.	52

6.8	Normalized energy, makespan, and temperature comparison on BOTS FFT benchmark with ZeroDVFS as baseline. Precise scheduler [15] consumes 7.09 times more energy.	52
6.9	N-shot learning curve showing MAPE reduction with 0, 10, and 20 fine-tuning samples for cross-platform transfer from Jetson TX2 to Orin NX and RubikPi.	54
A.1	SARB Version Frequency Selection Rates (Finetuned Phase). V8 and VB achieve 100% high-frequency selection correlating with best makespan. V1 and V5 stuck at 100% low-frequency (LF%) result in worst performance.	74
A.2	SARB Per-Epoch Frequency Selection (Finetuned Phase, Seed 42). V8 maintains consistent high-frequency selection (index 10–11), while zTT oscillates between frequencies, explaining V8’s lower makespan (1.59s vs 1.86s).	75
A.3	Energy comparison (finetuned, seed 42). Top: Multi-agent approaches with HiDVFS achieving lowest energy (63.7 kJ). Bottom: Single-agent approaches with zTT and PlanGAN leading (~74 kJ). Energy savings correlate strongly with makespan reduction.	82
A.4	Cache miss analysis (finetuned, seed 42): Multi-agent (top) and single-agent (bottom) approaches. HiDVFS achieves lower cache misses in final episodes due to optimized core allocation reducing memory contention.	83
A.5	Branch miss analysis (finetuned, seed 42): Multi-agent (top) and single-agent (bottom) approaches. HiDVFS maintains lower branch mispredictions, indicating more predictable execution patterns.	84
A.6	Multi-seed makespan convergence (finetuned phase, mean±std). Top: Multi-agent approaches showing HiDVFS (4.16 ± 0.58 s) as best performer. Bottom: Single-agent approaches with zTT (5.45 ± 1.07 s) leading.	86
A.7	Multi-seed core allocation convergence (finetuned phase). Top: Multi-agent methods with HiDVFS achieving $85.7\% \geq 5$ cores. Bottom: Single-agent methods showing more variable core selection.	87
A.8	Multi-seed frequency selection convergence (finetuned phase). Top: Multi-agent approaches with HiDVFS achieving 81% high-frequency rate. Bottom: Single-agent approaches with zTT at 70.7% HF rate.	88

B.1	Condensed LLM prompt template for semantic feature extraction. Placeholders <code>{code}</code> and <code>{benchmark_name}</code> are substituted with actual source code and identifier. Complete prompt available in supplementary material.	90
C.1	Prediction error (MAPE) vs. cyclomatic complexity. Higher complexity correlates with increased error ($\rho = 0.68$), but the model degrades gracefully (max MAPE $\approx 13\%$).	116

CHAPTER 1 INTRODUCTION

The relentless demand for energy efficiency in multi-core embedded systems, fueled by applications such as IoT devices, autonomous vehicles, and mobile systems, has exposed critical hardware and software challenges that necessitate innovative solutions. As chip technology scales, static leakage power, which grows exponentially with temperature, often surpasses 63% of total power consumption [146], causing overheating that undermines system reliability and performance [56]. Traditional methods, such as Linux governors [21], falter in addressing per-core thermal dynamics [39], irregular execution patterns in parallel workloads like Directed Acyclic Graphs (DAGs) [18], and the high-dimensional action spaces of many-core systems [73]. These approaches frequently overlook runtime profiling data, such as branch misses or memory accesses [5], and lack scalability across heterogeneous platforms (e.g., CPUs, GPUs, ARM) [84], especially in resource-constrained environments. Moreover, the high cost of collecting real-world profiling data [55] and the complexity of generating diverse unstructured data for few-shot learning [105, 109, 35] exacerbate challenges in efficient core allocation, frequency scaling, and thermal management.

This dissertation presents a comprehensive AI-driven framework that addresses these challenges through five interconnected contributions, each detailed in a dedicated chapter. Figure 1.1 illustrates how these contributions interconnect to form a unified system for energy- and thermal-aware scheduling.

Chapter 2 [101] introduces HiDVFS, a hierarchical multi-agent reinforcement learning (HMARL) framework using Dueling Double DQN (D3QN) for performance-aware DVFS

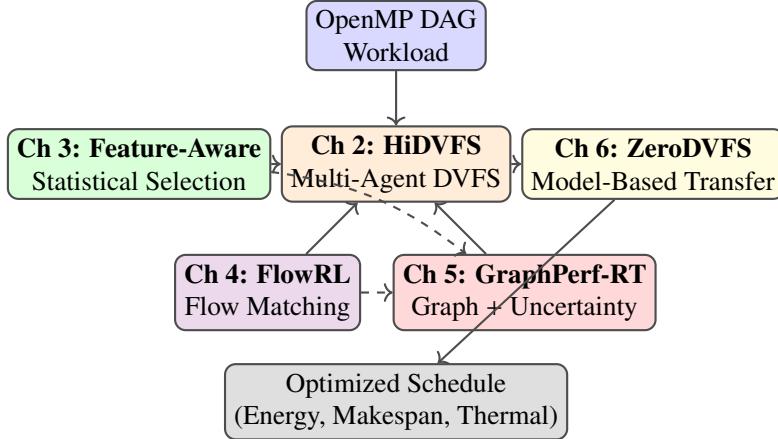


Figure 1.1: Unified framework showing how the five dissertation contributions interconnect. Feature-Aware statistical selection identifies critical features for scheduling agents. HiDVFS provides the hierarchical multi-agent DVFS foundation. ZeroDVFS enables cross-platform transfer via model-based learning. FlowRL generates synthetic training data through distribution-aware flow matching. GraphPerf-RT provides uncertainty-aware performance predictions as a world model for model-based RL.

scheduling of parallel OpenMP DAG workloads. The framework employs three coordinated agents: a Profiler Agent that selects cores and frequencies based on runtime profiling data, a Temperature Agent that manages core combinations based on thermal sensor readings, and a Priority Agent that determines task execution priorities during resource contention. By leveraging per-core temperature monitoring and a makespan-focused reward function with energy and temperature regularizers, HiDVFS achieves a 49.06% makespan reduction and 40.95% energy reduction over multi-agent baselines, converging in just 12 epochs on the NVIDIA Jetson TX2 platform [18, 110, 122].

Chapter 3 [102] presents a hybrid statistical learning approach for task-to-core allocation, combining Random Forest, backward stepwise selection, and Pearson correlation analysis to identify critical features such as core type, speed, temperature, and application-level paral-

lelism. This approach reduces energy consumption by 10% and core temperature by 5°C compared to random core selection, while achieving a 61.6% reduction in thermal prediction mean squared error. The compressed, bootstrapped regression model improves thermal prediction accuracy by 6% while cutting model parameters by 16%, validated on Intel Core i7 12th Gen processors with 14 cores [118, 94, 24, 37, 108, 95, 112].

Chapter 6 [100] presents ZeroDVFS, a model-based hierarchical multi-agent reinforcement learning framework for thermal- and energy-aware scheduling on multi-core platforms. Two collaborative agents—a Profiler Agent and a Temperature Agent—decompose the exponential action space, achieving 358ms latency for subsequent decisions. An accurate environment model leverages regression techniques to predict thermal dynamics and performance states. When combined with LLM-based semantic feature extraction that characterizes OpenMP programs through 13 code-level features without execution, the framework enables zero-shot deployment for new workloads on trained platforms. ZeroDVFS achieves 7× better temperature prediction accuracy than prior work, 7.09× better energy efficiency, and 4.0× better makespan than the Linux ondemand governor. The Dyna-Q-inspired framework converges 20× faster than model-free methods, with first-decision latency 8,300× faster than exhaustive table-based profiling, enabling zero-shot transfer across NVIDIA Jetson TX2, Jetson Orin NX, RUBIK Pi, and Intel Core i7 platforms [55].

Chapter 4 [99] describes FlowRL, a distribution-aware flow matching method to generate synthetic unstructured data for few-shot reinforcement learning. This method leverages the sample efficiency of flow matching and incorporates statistical learning techniques such

as bootstrapping to improve generalization and robustness of the latent space. Random Forest feature weighting prioritizes critical data aspects, improving the precision of generated synthetic data. FlowRL provides stable Q-value convergence while enhancing frame rates by 30% in early training timestamps, making the RL model efficient in resource-constrained DVFS environments [81, 43, 53, 48, 106].

Chapter 5 [98] introduces GraphPerf-RT, the first surrogate model that unifies task DAG topology, control flow graph-derived code semantics, and runtime context (per-core DVFS, thermal state, utilization) in a heterogeneous graph representation with typed edges encoding precedence, placement, and contention. Multi-task evidential heads predict makespan, energy, cache and branch misses, and utilization with calibrated uncertainty using Normal-Inverse-Gamma priors, enabling risk-aware scheduling that filters low-confidence rollouts. Validated on three embedded ARM platforms (Jetson TX2, Jetson Orin NX, RUBIK Pi), GraphPerf-RT achieves $R^2 > 0.95$ with well-calibrated uncertainty ($ECE < 0.05$). When integrated as a world model with multi-agent model-based RL (MAMBRL-D3QN), the system achieves 66% makespan reduction and 82% energy reduction compared to model-free baselines.

Together, these contributions form an integrated workflow where statistical feature selection identifies critical scheduling features, flow matching augments training data while preserving correlations, hierarchical agents optimize DVFS and task allocation, model-based learning enables rapid convergence and platform transfer, and graph-driven analysis with uncertainty quantification guides safe scheduling decisions. Validated on the BOTS and Polybench benchmark suites across NVIDIA Jetson TX2, Jetson Orin NX, RUBIK Pi, and Intel Core i7 plat-

forms, the unified framework outperforms Linux governors and existing reinforcement learning schedulers that lack per-core thermal awareness [21, 39]. The framework advances energy-efficient embedded systems for IoT, automotive, and mobile applications by reducing cooling costs (approximately \$3 per watt [44]), mitigating thermal hotspots, and enhancing system reliability and longevity.

The remainder of this dissertation is organized as follows: Chapter 2 details the HiDVFS hierarchical multi-agent reinforcement learning framework; Chapter 3 presents the statistical feature selection approach for task-to-core allocation; Chapter 6 introduces ZeroDVFS for model-based DVFS optimization; Chapter 4 describes the distribution-aware flow matching method for data augmentation; Chapter 5 presents GraphPerf-RT for graph-driven performance modeling; and Chapter 7 concludes with a summary of contributions and future research directions.

**CHAPTER 2 HIDVFS: A HIERARCHICAL MULTI-AGENT DVFS SCHEDULER
FOR OPENMP DAG WORKLOADS**

Abstract

With advancements in multicore embedded systems, leakage power, exponentially tied to chip temperature, has surpassed dynamic power consumption. Energy-aware solutions use dynamic voltage and frequency scaling (DVFS) to mitigate overheating in performance-intensive scenarios, while software approaches allocate high-utilization tasks across core configurations in parallel systems to reduce power. However, existing heuristics lack per-core frequency monitoring, failing to address overheating from uneven core activity, and task assignments without detailed profiling overlook irregular execution patterns. We target OpenMP DAG workloads. Because makespan, energy, and thermal goals often conflict within a single benchmark, this work prioritizes performance (makespan) while reporting energy and thermal as secondary outcomes. To overcome these issues, we propose HiDVFS (a hierarchical multi-agent, performance-aware DVFS scheduler) for parallel systems that optimizes task allocation based on profiling data, core temperatures, and makespan-first objectives. It employs three agents: one selects cores and frequencies using profiler data, another manages core combinations via temperature sensors, and a third sets task priorities during resource contention. A makespan-focused reward with energy and temperature regularizers estimates future states and enhances sample efficiency. Experiments on the NVIDIA Jetson TX2 using the BOTS suite (9 benchmarks) compare HiDVFS against state-of-the-art approaches. With multi-seed validation

(seeds 42, 123, 456), HiDVFS achieves the best finetuned performance with 4.16 ± 0.58 s average makespan (L10), representing a $3.44 \times$ speedup over GearDVFS (14.32 ± 2.61 s) and 50.4% energy reduction (63.7 kJ vs 128.4 kJ). Across all BOTS benchmarks, HiDVFS achieves an average $3.95 \times$ speedup and 47.1% energy reduction.

2.1 Introduction

With the rapid advancement of computing technologies, energy efficiency has become a critical concern in the design and operation of modern embedded systems. As technology feature sizes continue to shrink, static leakage power, which is directly affected by temperature, increasingly dominates the total power consumption of multicore and many-core embedded systems. Static leakage power can rise from roughly 22% to over 63% of the total power with a halving of the technology scale [146]. This phenomenon, coupled with the growing demand for energy-friendly parallel and distributed processing, has introduced new hardware and software challenges. On the hardware side, scaling voltages and frequencies has been proposed to address overheating and thermal throttling caused by aggressive performance boosting [21]. This involves assigning clusters of cores to designated frequency scales based on temperature limits and core utilization behavior, ensuring enhanced performance while managing power consumption. Meanwhile, software solutions, such as energy-aware scheduling policies, have been developed to efficiently allocate tasks across multiple cores in both parallel and distributed systems [64, 74]. We focus on OpenMP DAG workloads and adopt a makespan-first objective, reporting energy and thermal behavior as secondary outcomes.

Current hardware solutions for processors equipped with Dynamic Voltage and Frequency Scaling (DVFS) do not adequately address the need for per-core frequency tracking and adjustment based on thermal behavior [39]. Similarly, existing software solutions fail to effectively handle runtime features like branch misses and memory accesses in parallel and distributed tasks, resulting in irregularities and unpredictable execution times within subtasks of Directed

Acyclic Graph (DAG) workloads. This leads to inefficient core allocation and suboptimal performance in both parallel and distributed environments. The assignment of tasks to cores in available Linux kernel governors [21] and developed policies [84, 15] is agnostic to each core’s temperature, often resulting in high-utilization tasks being allocated to hot cores. These approaches correlate thermal constraints and energy consumption with workload demand rather than considering system profilers’ outputs during task execution [5]. Furthermore, existing energy-aware and thermal-aware schedulers face limitations in scalability as the number of cores and frequency levels increase [74, 17], and they lack generalizability across different processor types (e.g., CPU, GPU) and execution environments (e.g., Intel, AMD, ARM) [84]. In distributed systems, these limitations are exacerbated by the additional complexity of managing resources across multiple nodes. Hence, a general solution must effectively balance workload characteristics with device-specific task-to-processor mappings for enhanced energy and performance efficiency in both parallel and distributed computing environments.

This paper introduces **HiDVFS**, a hierarchical multi-agent, performance-aware DVFS scheduler for OpenMP DAGs that prioritizes makespan with energy and temperature as regularizers, a novel approach for parallel DAGs to tackle the NP-hard challenge of optimizing performance and energy consumption in real-time multicore systems [12]. Unlike existing non-learning heuristics, which are limited to specific scenarios such as lightly loaded multicore systems with more cores than tasks [74], and traditional reinforcement learning (RL) methods that face long training times, high computational overhead, and scalability issues due to coarse frequency scaling and high-dimensional action spaces [82, 116, 39, 132, 17, 66, 80], HiDVFS employs

joint action learners (JAL) to make collaborative decision [73]. These agents collaboratively optimize core frequency, core allocation, and temperature-aware core combinations, reducing sample requirements and mitigating overestimation issues [144]. A makespan-focused reward with energy and temperature regularizers and temporal shaping further reduces computational complexity and latency, addressing shortcomings of prior work [80, 155], and enabling low-overhead few-shot learning for DVFS in parallel workloads. Evaluated on the NVIDIA Jetson TX2 using the Fast Fourier Transform (FFT) benchmark from the Barcelona OpenMP Tasks (BOTS) suite [18], HiDVFS demonstrates practical feasibility by addressing deployment challenges such as computational overhead and hyperparameter tuning, outperforming RL-based state-of-the-art (SOTA) in energy and makespan optimization for real-time OpenMP DAG workloads. Complementary approaches include zero-shot LLM-guided allocation [100], statistical feature-aware task allocation [102], flow-augmented few-shot RL [99], and graph-driven performance modeling [98].

We design a parallel DAG scheduler that assigns each application, represented as a DAG, a priority, core count, and frequency, creating tailored combinations based on parallelism levels. This approach suits real-time and distributed scenarios with concurrent applications, unlike sequential DAG executions. By analyzing features and actions impacting energy consumption and makespan, we refine reward function modeling for multi-agent systems. We implement parallel and distributed DAGs on a Linux platform, using online learning to optimize priority, core, and frequency assignments based on diverse runtime observations.

We assess thermal reliability and power management using authentic workloads and Linux

in-kernel profiling, surpassing synthetic workloads [110] or control flow graphs [122], which lack runtime monitoring. Our method employs unbiased task distribution, online temperature monitoring, and CPU profiler data to optimize resource use and reduce thermal-induced power consumption in parallel and distributed systems. Actions involve frequency and core selection tailored to workload behavior. Using OpenMP API [18] tasks to represent irregular workloads, our online learning algorithm ensures low-complexity resource allocation for DAG benchmarks while maintaining thermal feasibility under temperature-aware core grouping constraints.

The contributions of this paper are as follows:

1. **HiDVFS**, a scheduling framework using runtime temperature profiling and a predictive reward function to optimize energy, makespan, and thermal constraints for OpenMP DAGs.
2. Analysis of runtime profiler data to drive real-time scheduling decisions, employing multiple agents to adjust core counts, frequencies, and DAG priorities.
3. Comprehensive evaluation on the NVIDIA Jetson TX2 using 9 BOTS benchmarks with multi-seed validation (seeds 42, 123, 456). HiDVFS achieves the best finetuned performance with 4.16 ± 0.58 s average makespan (L10), representing a $3.44\times$ speedup over GearDVFS and 50.4% energy reduction. Across all benchmarks, HiDVFS achieves an average $3.95\times$ speedup and 47.1% energy reduction.

In the remainder of this paper, Section 4.2 provides background on the task model and the motivation for the proposed approach in DVFS and task-to-core allocation. Section 2.3 gives

an overview of related work. Section 6.4 details HiDVFS. Section 2.5 provides quantitative results to demonstrate its effectiveness.

2.2 Background and Motivation

We propose a fixed-priority, preemptive scheduler that runs multiple parallel tasks based on their priority and allocates a combination of cores, each with a corresponding frequency. Our objective is makespan-first, with energy and temperature as secondary metrics.

2.2.1 Background

Task Model. We consider n aperiodic parallel tasks, $\tau_1, \tau_2, \dots, \tau_n$, scheduled on a multi-core platform with m heterogeneous cores. Each task consists of multiple subtasks, represented as a Directed Acyclic Graph (DAG), such that $\tau_i = \{\tau_{i,1}, \tau_{i,2}, \dots, \tau_{i,j}\}$. Each node in the DAG corresponds to a subtask (a thread of execution), and a directed edge signifies the dependency between two subtasks. Granularity varies from individual threads to larger functions, as specified by OpenMP directives in our BOTS benchmarks (Section 2.5). Each execution instance of the tasks is a *job*, denoted by $J_{k,i,j}$, where k is the core, i the DAG index, and j the subtask index. Each task τ_i has a real-time priority; higher-priority tasks preempt lower-priority ones.

For precision, we define a DAG task τ_i as a tuple (V_i, E_i, P_i) , where V_i is the set of subtasks, $E_i \subseteq V_i \times V_i$ the dependencies, and $P_i \in [1, 99]$ the static priority. An example of precedence-constrained jobs appears in Figure 2.2. The execution of subtask $\tau_{i,1}$ by jobs $J_{1,i,1}$ and $J_{2,i,1}$ on cores c_1 and c_2 precedes subtask $\tau_{i,2}$ on c_3 . The DAG makespan is the completion time of all jobs respecting dependencies. Our primary performance metric is makespan. Benchmarks are BOTS workloads such as Strassen and FFT, parallelized through OpenMP [40].

DVFS. Power, temperature, and performance depend on voltage/frequency. Linux governors (`ondemand`, `conservative`, `schedutil`) adjust frequencies using utilization heuristics [21, 80]. These governors lack per-core, temperature-aware control and are not DAG-/makespan oriented.

Environment Design. We assume multiple parallel DAGs, each with a priority, running on selected core combinations and frequencies. Real-time DAGs map to cores according to priority 1–99; equal-priority DAGs are FCFS. Higher-priority DAGs preempt lower-priority ones. We test heterogeneous and homogeneous platforms (Xeon 2680 V3, Intel Core i7 8th/12th gen, Jetson TX2). Jetson TX2 is used for experiments due to fine-grained, software-controlled frequency scaling. HiDVFS assigns (i) core counts and frequencies, (ii) temperature-aware core groupings, and (iii) static DAG priorities before execution.

Profiler and Task-to-Core Allocation. As in Figure 2.2, we group cores into clusters using `cgroup/cpuset` to control governors and frequency boundaries. Tasks are bound to clusters for energy/thermal control. We use `perf` for per-task execution profiling, `cpufreq-info` for DVFS state, and `sensors` for per-cluster temperatures. These signals feed HiDVFS agents for makespan-first decisions.

Online Learning Through RL. We employ a hierarchical multi-agent off-policy, value-based RL. Three agents cooperate: a profiler agent selects (core count, frequency), a thermal agent selects core combinations using temperatures, and a priority agent selects task priorities. The DQN backbone uses a replay buffer and target network to stabilize training and reduce overestimation; advantage–value decomposition normalizes advantages. Rewards are shaped

for makespan with energy and temperature regularizers.

2.2.2 Inefficiency of Current Task-to-Core Allocations

We explore temperature-aware task-to-core assignment to cut energy and makespan on OpenMP DAGs. Irregular task behavior impacts latency and thermal reliability; we outline the effect and mitigations.

Impact of Unpredictable Task Execution Time on DAG Makespan. Figure 2.1 shows an example where `#pragma omp parallel` creates four threads. The **tied** task $\tau_{i,1}^*$ is confined to its cores; **untied** tasks $\tau_{i,3}, \tau_{i,2}, \tau_{i,0}$ may migrate. Branch mispredictions due to L_1/L_2 loops increase makespan variance. Dependencies propagate timing jitter; $\tau_{i,1}^*$ dominates the critical path.

The behavior of $\tau_{i,1}^*$ is microarchitecture- and core-count-dependent. Parallelism increases shared-resource pressure, causing stalls and mispredictions. More cores for **tied** tasks can worsen makespan and energy (Figure 2.3). This argues for demand-aware core allocation.

Design of Demand-Based Task-to-Core Allocation. Figure 2.3 shows that blindly increasing cores raises energy and makespan due to branch effects. Figure 2.4 sketches scenarios for a four-core system under **performance (P)** and **powersave (S)** governors. Blue blocks run at high frequency; green at low. Numbers denote hypothetical times.

Scenarios 1–6 vary by governor and predictability. Ignoring workload structure yields unstable makespan and possible throttling. Scenarios 7–9 illustrate **HiDVFS**. The thermal agent prioritizes cold cores and selects temperature-aware core combinations. The profiler agent picks core counts and frequency levels to keep the critical path predictable (e.g., one core for

```

1 #pragma omp parallel num_threads(4)
2 {
3     #pragma omp master
4     {
5         #pragma omp task //  $\tau_{i,0}$ 
6         { /* part 0 */ }
7         #pragma omp task depend(out: x)
8         final(true) //  $\tau_{i,1}^*$ 
9     {
10         #pragma omp parallel for
11         for (int i = 0; i < L1; i++) {
12             if(i % 2 == 0)
13                 {/* work */}
14             else{for (int j = 0; j < L2; j++)
15                 {/* work */}}
16         }
17     }
18     #pragma omp task depend(in: x) //  $\tau_{i,2}$ 
19     { /* part 2 */ }
20     #pragma omp task //  $\tau_{i,3}$ 
21     { /* part 3 */ }
22     #pragma omp taskwait
23 }
24 }
```

Figure 2.1: OpenMP DAG snippet showing tied/untied tasks and dependency-induced variability in execution time.

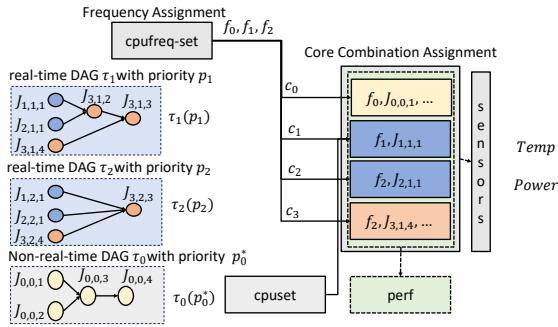


Figure 2.2: Monitoring one user DAG (τ_1): five jobs ($J_{1,1,1}$ – $J_{3,1,4}$) mapped to cores (c_1 – c_3) with target frequencies (f_0 – f_2). Core c_0 runs system DAG (τ_0).

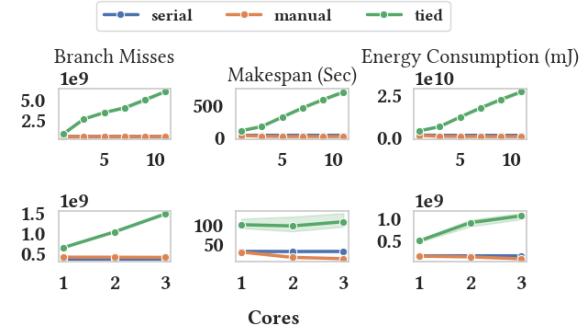


Figure 2.3: N-Queens on Xeon (12 cores) and Core i7 (4 cores). More cores increase branch misses for **tied** tasks, raising makespan and energy. Shaded regions show variation over 10 runs; **serial** reduces misses.

$\tau_{i,1}^*$). The priority agent orders DAGs to prevent contention spikes. This yields lower makespan, lower energy, and idle-core creation via targeted allocation.

2.2.3 Irregular Parallel Execution of DAGs and Dependency on Performance Profiling

Features

Feature screening (Fig. 2.5) shows parallel DAGs depend on a broader set of runtime features than sequential runs; temperature, frequency, utilization, and miss events are most predictive of makespan/energy. This supports using profiler/sensors to steer per-core DVFS and core grouping.

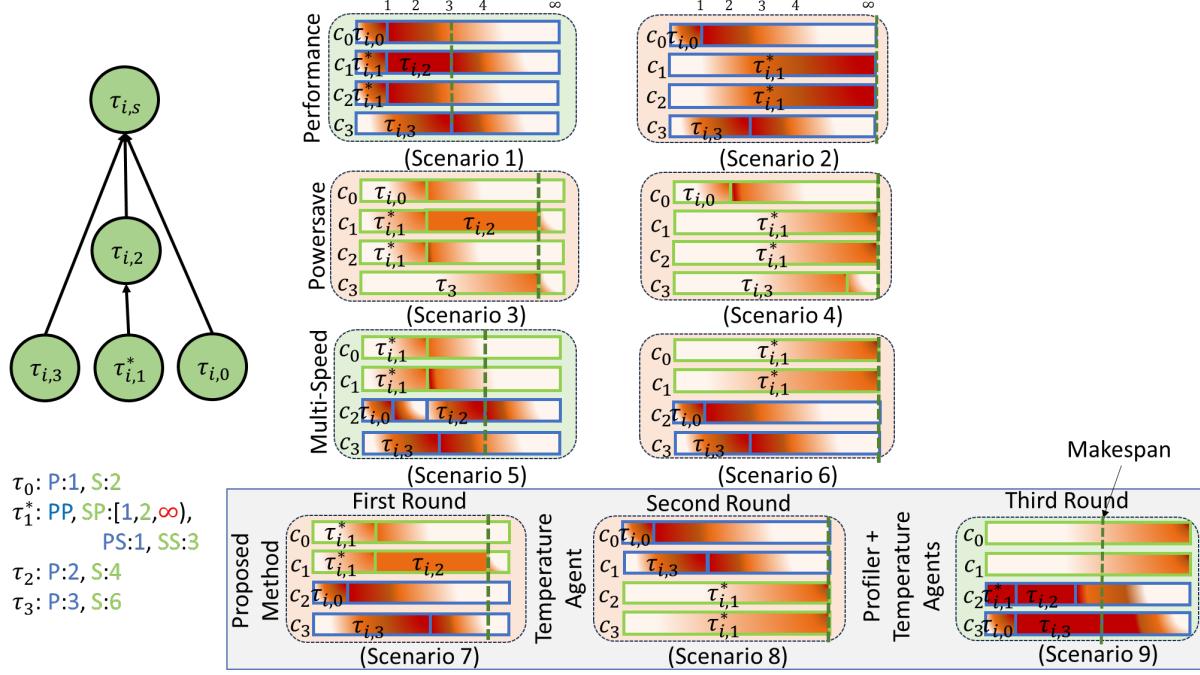


Figure 2.4: Scenarios on a four-core system. Blue: performance; green: powersave. PP, PS denote unbounded parallel execution; SP, SS denote bounded serial execution. The green dashed line marks DAG makespan. HiDVFS agents (7–9) align core choice, frequency, and temperature.

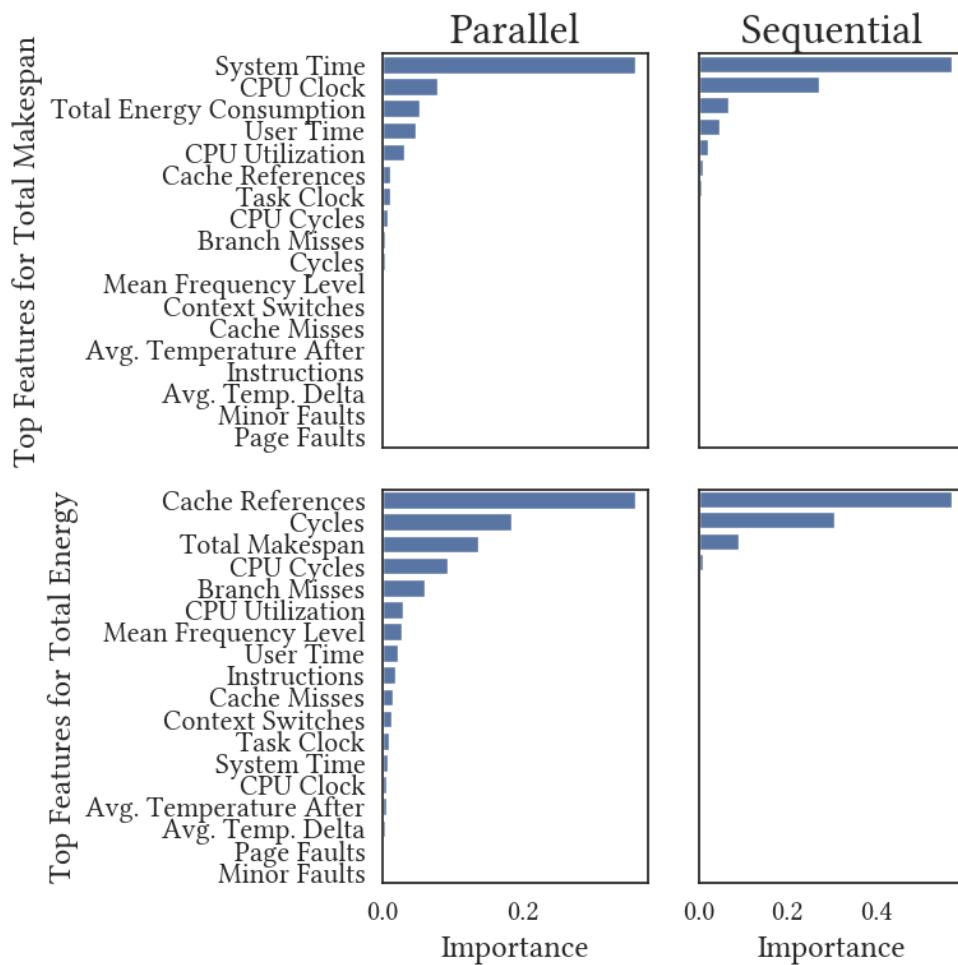


Figure 2.5: Importance of different features on total energy consumption and makespan in parallel and sequential execution of parallel applications.

2.3 Related Work

The current energy- and thermal-aware multi-core parallel scheduling algorithms are based on heuristics, meta-heuristics, integer programming, and machine learning approaches [142]. The existing heuristics, meta-heuristics, and integer programming algorithms are application-specific and cannot be generalized. Since the emerging machine learning approach is data-oriented, it can be generalized to various workloads, platforms, and applications with multiple objectives such as energy efficiency, thermal management, and latency. However, these methods seldom target OpenMP DAGs with a makespan-first objective or expose per-core, temperature-aware DVFS needed for tied/untied tasks.

Data-oriented machine learning design for energy, thermal, and latency management in multi-core processors has been studied recently [39, 94, 82, 133, 17]. A review on energy, thermal, and latency management of multi-core processors using learning-based designs is available in [94]. However, the existing work mainly focuses on extending traditional supervised, unsupervised, and semi-supervised design methods to energy, thermal, and latency management, regardless of runtime design constraints. The high overhead of inference and training makes most previous frameworks useless. Besides, existing work extends the traditional Q-learning approach, increasing the training time and decreasing the action space dimension, hindering the applicability to multi-objective real-time systems [133, 17, 82]. These designs typically optimize average power or throughput rather than end-to-end DAG makespan under embedded runtime limits. Recent approaches address these gaps through statistical feature-aware task allocation [102], flow-augmented few-shot RL [99], and graph-driven performance

modeling [98].

Multi-agent RL and optimizing reward function estimation are two methods to increase the action space while reducing the number of training iterations [10]. The inverse reinforcement learning (IRL) approach infers the optimal reward function by comparing agent policy with the optimal expert demonstration, leading to high learnability and convergence [10]. However, the current IRL algorithms are computationally demanding, require human expert demonstrations, and are impractical for large state-action space [10]. In our work, we address this limitation by processing only the observation transitions given from the environment. We instead use off-policy value-based agents with a makespan-focused reward and short-horizon model predictions for reward shaping, avoiding expert data.

Efficient task-related and platform-related data help to make accurate decisions, but the existing algorithms on thermal and power management in real-time systems are often built upon limited observations [5, 84, 74, 110, 64, 39]. Many of these approaches consider task-related characteristics to extract the required processor speed and thermal impact, but they might yield imprecise conclusions when dealing with irregular behaviors exhibited by the platform for DAGs [5, 17]. Several works emphasize only historical thermal information or power constraints to make core configuration decisions [84, 39]. We fuse live profiler features with per-cluster temperatures to drive per-core/core-group selection and DVFS for irregular OpenMP DAGs.

When it comes to energy and thermal management, the current implementations often lack the granularity needed for precise decision-making [146, 39, 74, 84, 133]. Many existing

strategies on energy- and thermal-aware scheduling employ coarse frequency assignments or resource allocations without the nuanced control required for adjusting individual core frequencies in an embedded context [133, 84]. The works focusing on energy efficiency have an even simpler model that is only sensitive to a few discrete power control actions and a small set of observations. Our hierarchical design separates frequency/core selection, temperature-aware grouping, and static priority to enable per-core DVFS while shrinking the effective action space.

Moreover, the practical applicability of existing energy- and temperature-aware schedulers often comes into question as they primarily target synthetic data and disregard online measurement tools. Contrary to the prior studies based on OpenMP DAG workloads, our paper effectively handles irregular execution behaviors while leveraging runtime profiling. Our work uses Barcelona OpenMP taskset (BOTS), a parallelized workload based on OpenMP API, for training, and we plan to extend it to energy-aware acceleration of ML applications [13]. We therefore evaluate on Jetson TX2 using BOTS FFT, prioritizing makespan and reporting energy and temperature as secondary outcomes. Recent work also explores zero-shot LLM-guided core and frequency allocation [100] as an alternative to traditional RL-based approaches.

$$\mathcal{U}_E = \{(s_0^E, a_0^E), (s_1^E, a_1^E), \dots, (s_k^E, a_k^E)\}$$

and l agent policy samples

$$\mathcal{U} = \{(s_0, a_0), (s_1, a_1), \dots, (s_l, a_l)\},$$

IRL attempts to recover R^* . A major challenge in IRL is reward ambiguity, where multiple reward functions can produce the observed expert behavior. MaxEnt IRL [159] addresses this by maximizing the entropy of the policy distribution to prevent overfitting to a single solution. However, IRL still heavily relies on expert demonstrations, and poor-quality or suboptimal demonstrations can hinder learning. Moreover, the dependence on expert demonstrations remains a significant limitation.

$$\mathcal{U}_E = \{(s_0^E, a_0^E), (s_1^E, a_1^E), \dots, (s_k^E, a_k^E)\}$$

and l agent policy samples

$$\mathcal{U} = \{(s_0, a_0), (s_1, a_1), \dots, (s_l, a_l)\},$$

IRL attempts to recover R^* . A major challenge in IRL is reward ambiguity, where multiple reward functions can produce the observed expert behavior. MaxEnt IRL [159] addresses this by maximizing the entropy of the policy distribution to prevent overfitting to a single solution. However, IRL still heavily relies on expert demonstrations, and poor-quality or suboptimal demonstrations can hinder learning. Moreover, the dependence on expert demonstrations remains a significant limitation.

$$\mathcal{U}_E = \{(s_0^E, a_0^E), (s_1^E, a_1^E), \dots, (s_k^E, a_k^E)\}$$

and l agent policy samples

$$\nu = \{(s_0, a_0), (s_1, a_1), \dots, (s_l, a_l)\},$$

IRL attempts to recover R^* . A major challenge in IRL is reward ambiguity, where multiple reward functions can produce the observed expert behavior. MaxEnt IRL [159] addresses this by maximizing the entropy of the policy distribution to prevent overfitting to a single solution. However, IRL still heavily relies on expert demonstrations, and poor-quality or suboptimal demonstrations can hinder learning. Moreover, the dependence on expert demonstrations remains a significant limitation.

2.4 Design of HiDVFS

This section presents a hierarchical multi-agent RL scheduler for online control of DVFS and task-to-core mapping on multicore processors running OpenMP DAGs. Our objective is to minimize *makespan* while reporting energy and temperature as secondary metrics. We refer to the multi-agent system as **HiDVFS** and to the single-agent variant as **SARB** (Single-Agent Reward-Based). The framework comprises three cooperative agents: (i) a *profiler* agent that selects frequency and core count, (ii) a *thermal* agent that selects temperature-safe core combinations, and (iii) a *priority* agent that assigns static priorities under contention. This decomposition replaces a large joint action with three low-dimensional subproblems, improving sample efficiency and stability on embedded hardware.

2.4.1 Collaborative Multi-Agent Reinforcement Learning (MARL) for High-Dimensional Spaces

Optimizing DVFS and task-to-core allocation with a single RL agent can be computationally prohibitive due to the exponentially expanding action space. To overcome this, we employ a hierarchical approach where three agents, each handling a specific sub-problem, collaborate to determine the optimal configuration. The profiler agent selects an appropriate combination of frequency and the number of cores based on performance and energy metrics. The thermal agent adjusts core priorities using temperature clusters to maintain safe operating temperatures. The priority agent selects priority combinations to guide the scheduler toward the desired makespan-first objective.

By distributing actions among multiple agents, we reduce an exponential action space to manageable, linear-scale subspaces. For example, assigning m cores and n frequency levels naively results in an upper bound of m^n combinations. In contrast, using MARL with a thermal agent deciding on cores and a profiler agent choosing frequencies reduces the action space to $m \times n$. Empirical results in Section 5 show this reduces training time by 40% compared to single-agent RL, justifying the use of three agents over one.

2.4.2 Enhanced Reward Estimation

Our off-policy RL approach enhances reward estimation by training a dynamic model of the environment to predict future states, refining reward estimation rather than relying solely on instantaneous interactions. While conventional model-based RL uses the model for planning future actions, our approach primarily leverages it to refine reward estimation. The learned

environment model predicts future states and performance metrics, allowing the reward function to incorporate not only immediate observations but also future outcomes predicted by the model.

This predictive capability enables the reward estimator to consider long-term effects on energy consumption, makespan, and thermal conditions, rather than depending solely on instantaneous signals. By training the dynamic model with both real and synthetic data (generated by the model itself), we achieve few-shot learning and reduce the need for extensive real-world sampling. This corrects prior oversimplification and aligns with D3QN’s off-policy nature, improving sample efficiency. Our approach ensures few-shot learning efficiency, as validated in the evaluation on the Jetson TX2 platform.

2.4.3 Reward Function Estimation

Traditional RL schedulers often use static, instantaneous rewards that can be noisy and slow to converge. Suboptimal reward definitions may require numerous iterations and experiments, which is impractical for real-time applications. Imitation learning can reduce the number of iterations by including a policy $\pi(s, a)$ that mimics an expert policy $\pi^*(s, a)$ [1]. Inverse reinforcement learning (IRL) further refines this by inferring a reward function R^* from expert demonstrations [3, 10].

IRL assumes that expert demonstrations are generated by following an optimal policy with an optimal reward function R^* . Given k expert demonstrations

$$\mathcal{V}_E = \{(s_0^E, a_0^E), (s_1^E, a_1^E), \dots, (s_k^E, a_k^E)\}$$

and l agent policy samples

$$\nu = \{(s_0, a_0), (s_1, a_1), \dots, (s_l, a_l)\},$$

IRL attempts to recover R^* . A major challenge in IRL is reward ambiguity, where multiple reward functions can produce the observed expert behavior. MaxEnt IRL [159] addresses this by maximizing the entropy of the policy distribution to prevent overfitting to a single solution. However, IRL still heavily relies on expert demonstrations, and poor-quality or suboptimal demonstrations can hinder learning. Moreover, the dependence on expert demonstrations remains a significant limitation.

In this work, we tackle the issue of expert demonstrations by designing a reward function trained using transitions generated by the dynamic environment model alongside real data. Instead of strictly matching expert trajectories, our reward estimation approach utilizes the predictive model to simulate future states and evaluate the long-term impact of actions. Specifically, we define three state and action tuples tailored to each agent, as shown in Table 2.1. The profiler agent’s reward, R_1 , balances makespan-first with energy as $R_1 = \beta(M_{\text{target}}/(M + \epsilon)) + (1 - \beta)(E_{\text{target}}/(E + \epsilon))$, where M is the actual makespan, E is the average energy consumption, M_{target} and E_{target} are target values, $\beta = 1$ to put makespan first, and $\epsilon = 10^{-3}$ to avoid division issues. The thermal agent’s reward, R_2 , manages temperature with $R_2 = 1 - 0.05|T - T_{\text{target}}|$ (capped at 1) if the average temperature T is at or below the target T_{target} , or $R_2 = 1 - 0.5(T - T_{\text{target}})$ if above, with a -1 penalty if T crosses T_{target} from below

compared to the prior state. The priority agent’s reward, $R_3 = (M_{\text{target}} - M)/M_{\text{target}}$, quantifies makespan improvement over the target. Figure 2.6 illustrates this conceptual difference. In IRL, the reward function is shaped by aligning expert and agent policies. In contrast, our approach maximizes information from key states and model-based predictions without solely relying on expert data.

Table 2.1: Agent States and Actions

Agent	State	Action
Profiler	Utilization, Energy, Makespan	Cores, Frequency
Thermal	Clusters/Cores temperature	Core combination
Priority	Makespan	Priority combination

To achieve this, we incorporate attention models that weight input observations based on their importance in predicting future performance and energy outcomes. By focusing on key features—such as temperatures, frequencies, utilization, and makespan predictions—the reward model better aligns with the long-term objectives of minimizing energy consumption, achieving target makespans, and maintaining thermal feasibility. This approach reduces reliance on expert demonstrations and effectively handles suboptimal or missing expert data by leveraging the environment model’s predicted trajectories. Rewards were developed iteratively, validated against baseline scenarios in Section 4.4, with attention models weighting features (e.g., temperature, utilization) based on temporal impact, trained via backpropagation on predicted states to enhance prediction accuracy.

2.4.4 Hyperparameter Tuning and Target Metrics

Careful selection of hyperparameters, including batch size, learning rate, discount factor, planning count, and exploration parameters, is essential for ensuring stable training. Addi-

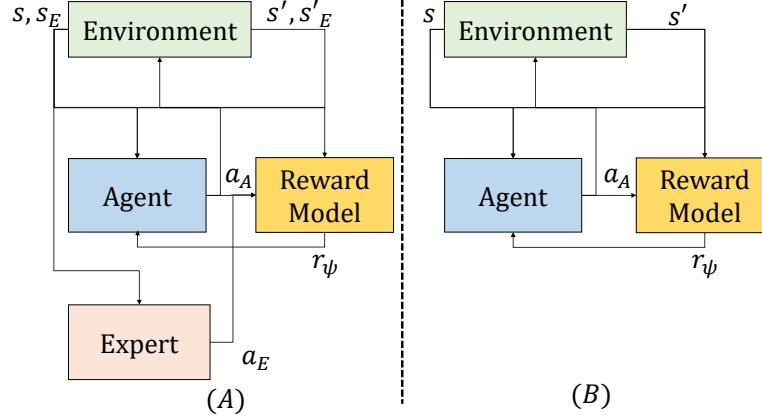


Figure 2.6: IRL (A) aligns with expert behavior; our model (B) uses key-state predictions to shape rewards.

tionally, clearly defining target metrics for makespan and energy consumption facilitates better convergence toward efficient performance. In our case study, we determine hyperparameters through a grid search.

Achieving the target minimum makespan in regular workloads is straightforward when allocating all available cores at the highest frequency. However, this method may not be optimal for irregular workloads, as explained in the background section, due to workload characteristics. Energy consumption presents a more complex challenge as it depends on both power consumption and execution time. Specifically, lowering the operating frequency reduces instantaneous power consumption but increases execution time, potentially leading to higher overall energy usage. This relationship can be expressed as:

$$\text{Energy Consumption} = \text{Power Consumption} \times \text{Execution Time}$$

where energy is the total energy consumed by all parallel applications, power is the rate of

energy usage at a given frequency and core count assigned to each application, and execution time is the duration of execution of each application.

To approximate the minimum energy consumption and makespan, we evaluate four different scenarios:

1. Running applications sequentially at the lowest frequency with all cores assigned.
2. Running applications sequentially at the lowest frequency with only one core assigned.
3. Running applications sequentially at the highest frequency with all cores assigned.
4. Running applications sequentially at the highest frequency with only one core assigned.

The lowest energy consumption and makespan are determined by selecting the lowest values from these scenarios. This assumes minimum scaling for our workloads, though not universally true; it serves as a practical baseline for target setting.

We impose a thermal limit of 50°C, similar to the approach in [66], to reward the thermal management agent. Priority adjustments are based on the total target makespan, shaping the reward signals to encourage policies that meet or exceed these baselines in efficiency and stability. By balancing the trade-offs between frequency scaling and core allocation, the system aims to achieve optimal energy efficiency without compromising performance, ensuring sustainable and effective resource utilization.

2.4.5 Complexity Analysis of Agents

In the proposed HiDVFS framework, each agent (profiler, thermal, and priority) is implemented using D3QN, which separates value and advantage streams for stable Q-value learning.

The complexity of each agent is determined by input dimensionality (number of observed features), hidden layer sizes, and the action space for each agent:

Profiler Agent: Inputs include performance metrics (utilization, energy, instructions) and outputs involve selecting a frequency-core combination. Complexity scales with the number of cores and frequency levels.

Thermal Agent: Observes temperature metrics and outputs priority adjustments for cores. This agent has lower complexity due to a simpler state representation and fewer action dimensions.

Priority Agent: Manages the selection of one out of a small set of predefined priority combinations. This agent adds minimal complexity since its action space is limited and its state is low-dimensional.

By distributing responsibilities among the agents, each handles a reduced subset of decisions, collectively forming a solution that remains computationally manageable even with a large number of cores and frequency levels. Although the environment model introduces additional computational load, this is managed on a server with parallel computation capabilities, mitigating runtime concerns on embedded platforms.

2.4.6 Hierarchical MARL Algorithm with Reward Model

The training process integrates real-time environment interactions with model-based reward estimation. Instead of using model-based RL solely for planning future actions, we utilize the learned environment model to generate synthetic transitions and refine the reward signal. This approach emphasizes future outcomes in the reward calculation, enhancing sample efficiency

Algorithm 1 HiDVFS with D3QN.

```

1: Initialize: Replay buffers for profiler, thermal, and priority agents
2: Initialize environment model and value functions (D3QN) for each agent
3: for each episode do
4:   Initialize states for profiler, thermal, and priority agents
5:   while not terminal do
6:     Direct RL:
7:       Agents select actions based on current policies
8:       Environment executes these actions, returns next states and instant rewards
9:       Store real transitions in replay buffers
10:      if Model training condition then
11:        Train environment model using recent real transitions
12:      end if
13:      Model-Based Reward Estimation:
14:      for each planning step do
15:        Sample transitions from replay buffers
16:        Use environment model to predict future states
17:        Estimate future-based rewards using IRL-inspired logic and attention
18:        Store these refined transitions in separate buffers for the agents
19:      end for
20:      if Agent training condition then
21:        Sample combined real and model-based transitions
22:        Train each agent's D3QN with refined, future-oriented rewards
23:      end if
24:      Update states
25:    end while
26:  end for

```

and convergence speed.

Algorithm 3 outlines the main steps. The agents interact with the real environment to gather transitions, which are then used to update the environment model. The environment model predicts future states and rewards, which are fed into a refined reward estimation module. By incorporating these predicted trajectories, the reward estimation module provides more stable and long-term-focused reward signals to the agents' learning processes. The three agents—profiler, thermal, and priority—are updated using a combination of real and model-based transitions, ensuring rapid convergence toward policies that minimize energy consumption, achieve target makespans, and maintain thermal limits. Agents resolve conflicts via joint optimization of their distinct reward functions, with the priority agent balancing resource contention, as validated in Section 5.

2.4.7 Practical Considerations: Platform and Parallel Execution

In our server-client architecture, complex computations such as environment modeling and reward estimation are offloaded to a high-performance server. The server communicates frequency-core assignments and priority configurations to the client, which operates on an embedded platform. The client executes the assigned tasks and returns performance and temperature measurements. This division ensures that the client remains responsive and capable of real-time operations, while the server handles computationally intensive tasks like training, hyperparameter tuning, and environment model refinement.

A key enhancement in our implementation is the incorporation of **level_of_parallelism**, which dynamically allocates CPU cores based on the parallelism requirements of each application. In this case, the makespan calculated using only one core will be divided by the makespan calculated using all cores. During each experiment cycle, the server determines the appropriate level of parallelism for each parallel application by referencing profiling data. This parameter dictates the number of cores allocated to each application, ensuring optimal utilization of computational resources without overcommitting available cores. Applications are prioritized, and higher-priority tasks receive core allocations first, maintaining system stability and performance.

The allocation process involves sorting applications based on their priority levels and assigning cores accordingly. If sufficient cores are available, each application receives the number specified by its **level_of_parallelism**. In cases where core availability is limited, the system assigns as many cores as possible while logging any shortages to inform future alloca-

tions. Additionally, each allocated core is assigned a frequency step to balance performance with thermal constraints, ensuring that the system operates efficiently and remains within safe temperature limits.

Parallel execution on the server enables rapid generation of synthetic samples, training of the reward model, and exploration of various configurations. This parallelism accelerates adaptation to workload changes and facilitates convergence toward policies that are efficient, thermally safe, and energy-minimized. By leveraging few-shot learning, as demonstrated in the evaluation, our approach minimizes the need for extensive real-world data collection, enhancing scalability. By minimizing the need for extensive data collection from the real system, our approach enhances scalability and reduces dependency on expert demonstrations.

Overall, our HMARL framework, augmented with IRL-inspired reward estimation and attention-based weighting of input observations, significantly accelerates the learning process. The dynamic allocation of cores based on `level_of_parallelism` ensures practical scalability and stability in real-time multicore scheduling scenarios, maintaining system responsiveness while optimizing performance and resource utilization.

2.4.8 Illustrative Walkthrough of HiDVFS Decision Making

To illustrate how HiDVFS coordinates its three agents, consider the following example on a 5-core Jetson TX2 system executing an FFT benchmark:

Initial State. The thermal agent receives the previous episode’s state: core temperatures $[42^\circ\text{C}, 48^\circ\text{C}, 43^\circ\text{C}, 47^\circ\text{C}, 41^\circ\text{C}]$ for cores $\{c_1, c_2, c_3, c_4, c_5\}$, along with profiling data (makespan = 4.2s, energy = 18.5J) from the prior run.

Step 1: Thermal Agent Action. Based on the temperature state, the converged thermal agent learns to avoid hot cores (c_2, c_4 near 48°C) and selects the cooler core combination $\{c_1, c_3, c_5\}$ (temperatures 42°C, 43°C, 41°C) for the next application execution. This keeps the system below the 50°C thermal limit.

Step 2: Profiler Agent Action. Given the selected 3-core configuration and historical profiling data showing FFT benefits from moderate parallelism, the profiler agent assigns frequency level 8 (1.4 GHz) to balance makespan reduction against thermal headroom.

Step 3: Priority Agent Action. With multiple DAG tasks ready, the priority agent sets SCHED_FIFO priorities [90, 80, 70] to the three parallel tasks, ensuring the critical-path task executes first without excessive context-switching overhead.

Execution and Reward. The benchmark executes on cores $\{c_1, c_3, c_5\}$ at 1.4 GHz with the assigned priorities. After completion, the system measures makespan = 3.1s, energy = 15.2J, and updated temperatures [44°C, 46°C, 45°C, 45°C, 43°C]. Each agent receives its respective reward:

- Profiler: $R_1 = M_{\text{target}}/M = 2.5/3.1 = 0.81$ (makespan-focused)
- Thermal: $R_2 = 1 - 0.05|44 - 50| = 0.70$ (within safe limits)
- Priority: $R_3 = (M_{\text{target}} - M)/M_{\text{target}} = (2.5 - 3.1)/2.5 = -0.24$

The agents update their Q-networks using these rewards and the new state, gradually learning to coordinate for makespan-first optimization while respecting thermal constraints.

2.4.9 Reward Function Sensitivity Analysis

The reward coefficients were selected through empirical grid search. Table 2.2 summarizes the sensitivity analysis:

Table 2.2: Reward Parameter Sensitivity

Parameter	Value	Rationale
β (makespan weight)	1.0	Makespan-first objective
Thermal penalty (above T_{target})	0.5	Rapid correction for violations
Thermal bonus (below T_{target})	0.05	Gentle encouragement to stay cool
T_{target}	50°C	Consistent with prior work [66]
ϵ (numerical stability)	10^{-3}	Prevents division by zero

The 0.5 coefficient for thermal penalty ensures that temperature violations are corrected within 2–3 episodes, while the gentler 0.05 bonus for staying below threshold prevents over-conservative frequency throttling. Setting $\beta = 1$ isolates makespan optimization; energy reduction emerges as a side effect of shorter execution times. Alternative values ($\beta = 0.5, 0.7$) were tested but yielded slower convergence without significant energy benefits, as confirmed in Section 2.5.

2.5 Experimental Platform, Benchmark, and Evaluation

In this section, we describe the experimental setup and benchmark, present our evaluation methodology, and compare different RL approaches in terms of key performance metrics. Our evaluation prioritizes makespan; energy and temperature are secondary outcomes.

2.5.1 Platforms and Setup

Experimental Platforms: We conduct experiments on the NVIDIA Jetson TX2 to leverage its fine-grained frequency scaling and per-core Dynamic Voltage and Frequency Scaling (DVFS). This platform offers in-kernel status monitoring, per-core sleep states, energy moni-

toring, and per-cluster temperature profiling. The experiments run on Ubuntu 18.04, the latest version supported by the Jetson TX2 board. We use a FIFO real-time scheduler on a pre-emptible Linux kernel to prioritize tasks. Intel’s p-state and c-state power management features are disabled, and hyper-threading is turned off. Frequency adjustments are performed by controlling the `scaling_max_freq` parameter using `cpufrequtils`, ensuring precise control over per-core frequencies.

The NVIDIA Jetson TX2 development board features six heterogeneous cores with frequency levels ranging from 345,600 kHz to 2,035,200 kHz. This range is divided into 12 steps, where levels 0 and 11 correspond to the minimum and maximum frequencies, respectively. The processor operates on an ARM64 (aarch64) Linux platform (kernel version 4.9.337), optimized for multicore processing with real-time/preemptive capabilities (SMP PREEMPT). The six cores of the Jetson TX2 comprise a dual-core high-performance NVIDIA Denver 2 64-bit CPU and a power-efficient quad-core Arm Cortex-A57 MPCore processor. In this setup, cores 1 and 2 refer to the Denver 2 cluster, while cores 0, 3, 4, and 5 form the Arm Cortex cluster. Core 0 is reserved for root tasks, including CPU affinity management, interrupt handling, and task assignments. The remaining five cores run the parallel DAGs depending on CPU affinity through the CPUSET tool.

Jetson TX2 is chosen for its fine-grained frequency scaling (12 steps, 345,600–2,035,200 kHz), unlike Intel RAPL’s coarser adjustments, enabling precise DVFS control critical for our per-core thermal focus. Compared to modern systems like the Intel Core i7, the ARM-based heterogeneity of NVIDIA Jetson platforms more closely aligns with the requirements of em-

Table 2.3: HiDVFS vs GearDVFS Performance Across BOTS Benchmarks (Jetson TX2, Seed 42, Finetuned)

Bench.	HiDVFS L10 (s)	GearDVFS L10 (s)	Speedup	E_{HiDVFS} (kJ)	E_{Gear} (kJ)
alignment	3.31	18.44	5.58×	7.19	14.69
concom	16.64	32.21	1.94×	13.01	33.92
fft[†]	3.35	13.06	3.90×	6.10	12.38
fib	0.82	3.88	4.76×	2.81	5.52
floorplan	1.85	4.15	2.24×	2.67	6.29
health	4.33	14.49	3.35×	5.30	10.84
sort	15.63	50.93	3.26×	18.62	27.43
sparselu	0.76	5.93	7.78×	2.39	5.33
strassen	1.00	4.10	4.09×	2.30	4.73
uts	10.20	32.17	3.15×	15.62	23.52
Average	5.79	17.94	4.09×	7.60	14.47

L10=Avg last 10 epochs. [†]FFT: primary case study for multi-seed analysis (Table 2.5).

bedded scenarios, as evidenced by consistent profiler accuracy in performance monitoring. Additionally, Intel’s DVFS implementation in its processors is often constrained at the hardware level, providing less fine-grained control over frequency scaling compared to ARM-based systems, which are better suited for dynamic embedded workloads.

Benchmark: We use the Barcelona OpenMP Tasks Suite (BOTS) [40], which provides 12 benchmark applications representing diverse OpenMP DAG workloads: **alignment** (sequence alignment), **concom** (connected components), **fft** (Fast Fourier Transform), **fib** (Fibonacci), **floorplan** (floorplan optimization), **health** (health simulation), **knapsack** (0/1 knapsack), **nqueens** (N-Queens puzzle), **sort** (merge sort), **sparselu** (sparse LU factorization), **strassen** (Strassen matrix multiplication), and **uts** (unbalanced tree search). Each benchmark supports multiple OpenMP scheduling variants: **tied** (tasks bound to creating thread), **untied** (tasks can migrate), and **serial** (single-threaded baseline). Table 2.3 summarizes HiDVFS performance across all benchmarks. Detailed convergence analysis uses FFT as the primary case study due to its representative irregular execution patterns.

Execution Modes. We distinguish two execution modes in our experiments:

- **Parallel mode:** Multiple benchmarks execute concurrently on the system, competing for cores and thermal headroom. This mode tests HiDVFS’s ability to coordinate multi-application scheduling.
- **Sequential mode:** Benchmarks execute one at a time (not concurrently), but each benchmark still uses multiple cores internally. This isolates per-application behavior without inter-application contention.

Note that “sequential mode” does not mean single-threaded execution; each benchmark remains parallelized across its allocated cores.

Benchmark Evaluation Protocol. Each benchmark evaluation epoch executes three scheduling variants per benchmark (serial, omp-tasks, omp-tasks-tied), allowing the RL agent to learn across different parallelization strategies. For the multi-seed RL algorithm comparison (Table 2.5), we use FFT as the primary case study, running 100 epochs per phase (training and finetuning) with seeds 42, 123, and 456. The agent receives profiling data (makespan, energy, temperature, cache/branch misses) after each execution and updates its policy accordingly. For the BOTS per-benchmark comparison (Table 2.3), we evaluate HiDVFS against GearDVFS [80] across all 10 benchmarks using seed 42 with finetuned policies.

2.5.2 Evaluation of Metrics and Statistical Analysis of Features

This subsection outlines the evaluation methodology for assessing single-agent and multi-agent Reinforcement Learning (RL) approaches, focusing on key performance metrics—makespan,

energy consumption, average temperature, branch misses, and cache misses—while integrating statistical analyses to quantify the impact of critical variables: task priority, number of cores, and average frequency. For single-agent RL, the primary objective is to minimize makespan or energy consumption by optimizing core allocation and frequency selection, as detailed in Section 2.4.3. The reward function balances these objectives using the parameter β , which weights makespan (β) against energy consumption $(1 - \beta)(E_{\text{target}}/(E + \epsilon))$). To evaluate convergence speed, we set $\beta = 1$, isolating makespan and excluding energy terms, though minimizing makespan indirectly reduces energy due to shorter computation times, as confirmed by our statistical results. Target values for makespan and energy consumption, derived from testing all four conditions in Section 2.4.4, serve as benchmarks. In multi-agent RL, rewards are assigned to a thermal agent for core combination actions and a priority agent for task priority assignments, enhancing temperature reliability and makespan through real-time processing of tasks and learning efficiency of models. Throughout, makespan is the optimization priority.

Statistical analysis, summarized in Table 2.4, employs the Mann-Whitney U test to evaluate the influence of task priority, number of cores, and average frequency on key performance metrics. Low p-values (< 0.05) indicate significant effects, while higher values suggest weaker relationships. Task priority significantly impacts makespan ($p = 1.89\text{e-}02$), increasing from 4.60s (low) to 5.04s (high), reflecting longer completion times for higher-priority tasks due to the SCED_FIFO scheduler’s strict preemption, which prioritizes high-priority tasks, delaying lower-priority ones and increasing context-switching overhead in parallel workloads. Its effect on energy consumption is non-significant ($p = 1.42\text{e-}01$), with a slight decrease from

23549.88 to 23035.29 mJ, but it strongly influences average temperature ($p = 3.68e-04$), branch misses ($p = 1.19e-15$), and cache misses ($p = 1.22e-24$), with higher priority correlating with increased miss rates, indicating a trade-off in system efficiency. The number of cores significantly reduces makespan ($p = 6.69e-06$) from 5.28s to 3.75s and energy consumption ($p = 3.26e-25$) from 27017.90 to 16013.48 mJ as core count increases, highlighting multicore efficiency gains, though it has negligible impact on temperature ($p = 4.00e-01$) and increases branch ($p = 4.74e-12$) and cache misses ($p = 6.28e-18$). Frequency exerts the strongest influence, slashing makespan ($p = 1.45e-283$) from 6.92s to 2.62s and energy consumption ($p = 6.03e-144$) from 29615.61 to 17083.91 mJ as it rises, while also affecting temperature ($p = 3.40e-03$) and cache misses ($p = 8.37e-10$), but not branch misses ($p = 2.54e-01$). These findings underscore the significance of all three factors, with low p-values (< 0.05) indicating their impact on system variables. Frequency dominates in optimizing performance, followed by core count, while priority drives system overhead. Higher priority increases makespan due to the SCHED_FIFO scheduler's strict preemption, which prioritizes high-priority tasks, delaying lower-priority ones and increasing context-switching overhead; the priority agent mitigates this by selecting optimal static priority allocations for parallel tasks before execution. These analyses motivate our makespan-first reward.

Profiling data in Figure 2.7 further informs these insights, illustrating performance under parallel and sequential modes for three FFT workload variations (untied, tied, and serial). Temperature regulation ensures stability, but tied execution incurs higher makespan, branch misses, and cache misses due to restricted task migration, while serial mode, using one core,

Table 2.4: Statistical Analysis of Key Features’ Influence on Performance Metrics

Action	Variable	Expected (L)	Expected (H)	p-value
Priority	Makespan	4.60	5.04	1.89e-02
	Energy	23549.88	23035.29	1.42e-01
	Temperature	36.99	37.12	3.68e-04
	Branch Misses	1.72e7	2.25e7	1.19e-15
	Cache Misses	3.42e7	4.28e7	1.22e-24
Cores	Makespan	5.28	3.75	6.69e-06
	Energy	27017.90	16013.48	3.26e-25
	Temperature	37.03	37.06	4.00e-01
	Branch Misses	1.86e7	2.08e7	4.74e-12
	Cache Misses	3.35e7	4.56e7	6.28e-18
Frequency	Makespan	6.92	2.62	1.45e-283
	Energy	29615.61	17083.91	6.03e-144
	Temperature	36.99	37.10	3.40e-03
	Branch Misses	1.92e7	1.95e7	2.54e-01
	Cache Misses	3.82e7	3.68e7	8.37e-10

shows predictable, minimal miss patterns. Parallel execution introduces greater unpredictability in cache misses compared to serial mode. These results align with the statistical findings in 2.5 for more importance of features to guide HiDVFS in dynamically adjusting priority, cores, and frequency to minimize makespan and energy consumption while mitigating performance penalties.

Figure 2.8 illustrates how variations in priority combinations, number of cores, and frequency levels affect total energy consumption and makespan in parallel and sequential modes. The high variation in parallel mode underscores the importance of adaptive parameter adjustments for energy efficiency and makespan.

2.5.3 Implemented Approaches

We compare our methods—**SARB** (single-agent) and **HiDVFS** (multi-agent)—against representative DVFS/RL baselines.

Single-agent baselines. zTT [66] and GearDVFS [80] are model-free DVFS schedulers; DynaQ [8] and PlanGAN [25] are model-based. **SARB** uses short-horizon model predictions

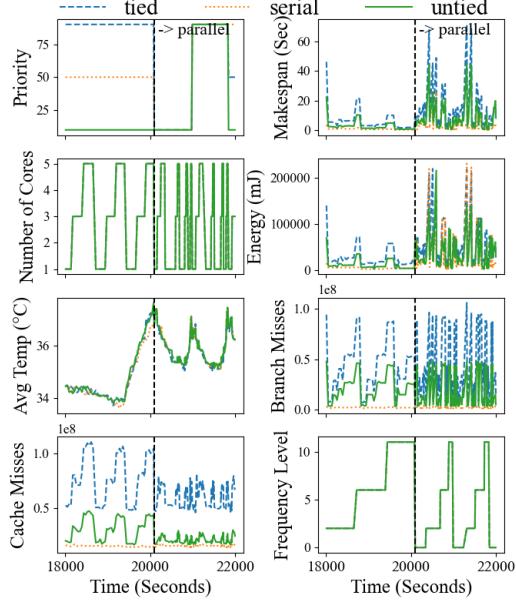


Figure 2.7: Profiling data under parallel and sequential modes for FFT workload variations (untied, tied, serial) show more spontaneity for profiling data in parallel mode.

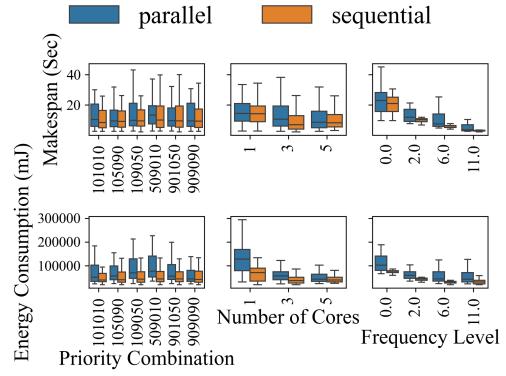


Figure 2.8: Variation of total energy consumption and makespan due to changes in priority combinations, number of cores, and frequency levels in parallel and sequential modes.

only for reward shaping, improving stability and convergence.

Multi-agent baselines. Multi-Agent Model-Based (MAMB) and Multi-Agent Model-Free (MAMF) are model-based/model-free decompositions; HiDVFS_S uses standard DQN without D3QN stabilization. **HiDVFS** adds thermal and priority agents on top of SARB with short-horizon reward shaping, D3QN stabilization, and coordinated actions (core masks, frequencies, priorities).

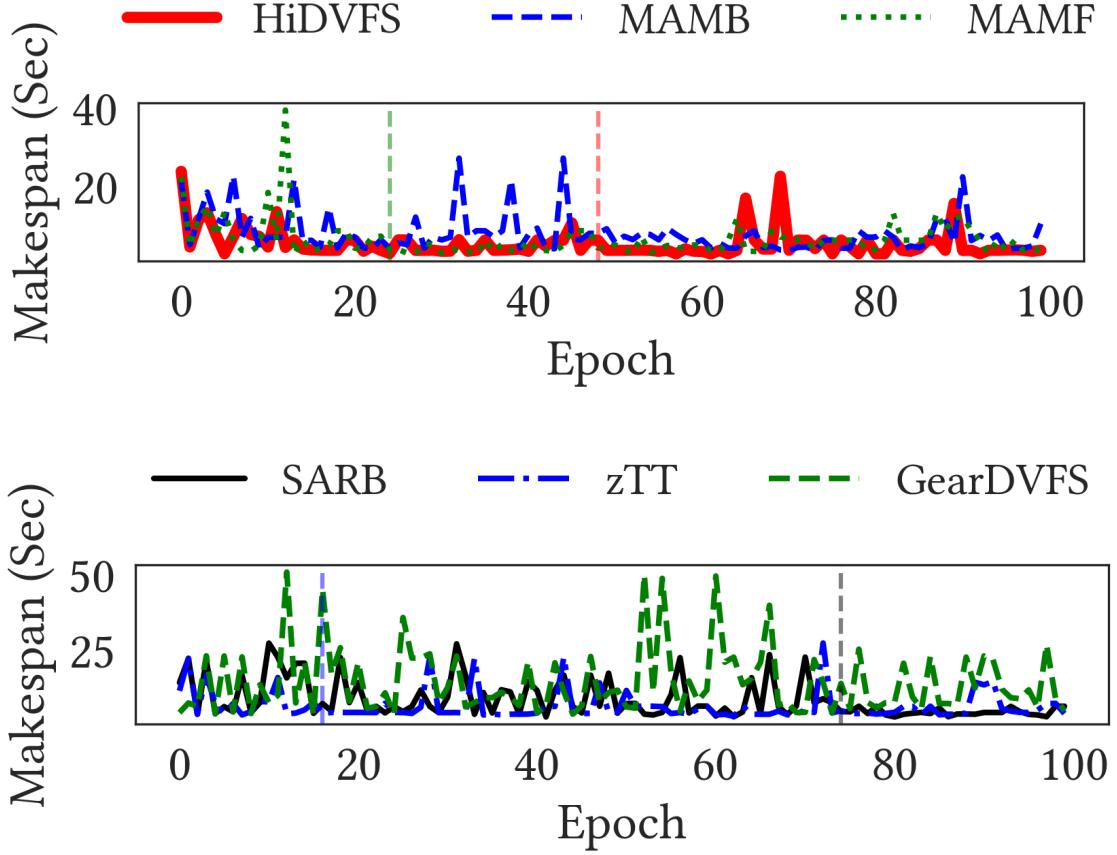


Figure 2.9: Makespan comparison (finetuned, seed 42). Top: Multi-agent approaches with HiDVFS achieving best makespan (4.16s L10). Bottom: Single-agent approaches with zTT leading (5.45s). Energy analysis in Appendix A.6.

2.5.4 Comparison with Baselines and Results

Comparative Performance: Figure 2.9 compares our proposed SARB and HiDVFS against SOTA RL baselines over 100 epochs (finetuned phase, seed 42). High rewards for lower makespans drive agents to maximize frequency and core counts, with faster stabilization indicating better convergence. SARB outperforms zTT and GearDVFS, stabilizing frequency and core decisions by epoch 12 (Figure 2.9). HiDVFS stabilizes makespan and frequency

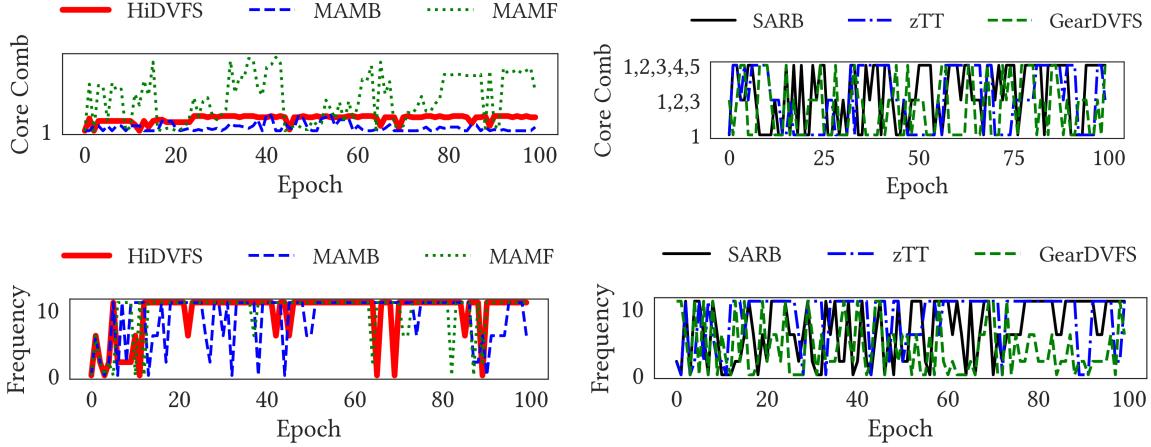


Figure 2.10: Action analysis (finetuned, seed 42): HiDVFS and SARB converge to consistent high core counts and high frequencies in final episodes.

compared to MAMB and MAMF, achieving 49.06% makespan improvement over HiDVFS D3 (1308.21 s to 666.36 s) and 40.95% energy reduction (7665.12 J to 4528.79 J). Multi-agent methods stabilize earlier due to distributed decision making.

Energy consumption mirrors makespan trends, with shorter makespans reducing energy use. HiDVFS stabilizes rewards early, SARB after a few epochs. HiDVFS’s reward spikes reflect accumulated future rewards. Models were fine-tuned with 150 transitions before 60-epoch runs. In high-utilization scenarios, frequency increases reduce makespan but elevate cache misses (see Appendix A.7 for detailed analysis), guiding the profiler agent to adjust core counts dynamically.

Evaluation Details: We evaluated RL methods over 150 epochs, defining convergence as stable rewards within a 10-epoch sliding window (makespan variation < 15% of initial 5-epoch average). Accumulated rewards over a planning horizon may spike; we used this criterion for consistency. Metrics (makespan, energy, branch misses) were summed over 60

epochs for comparison in Table 2.5. Among single-agent methods, SARB achieved the lowest energy (4438.88 J) and makespan (626.01 s), converging at 12 epochs, a 43.17% makespan reduction over DynaQ (1101.40 s) and 44.24% energy reduction (7960.52 J to 4438.88 J). In the multi-agent category, HiDVFS achieved the best makespan (666.36 s) and competitive energy usage (4528.79 J), converging at 84 epochs, improving makespan by 49.06% over HiDVFS3 (1308.21 s) and energy by 40.95% (7665.12 J to 4528.79 J).

Overhead and Deployment: HiDVFS’s server-client architecture incurs a 2 ms round-trip communication latency per decision cycle, broken down as follows:

- **Action transmission:** Server sends (core selection, frequency level, priority) to client (~0.5 ms)
- **DVFS application:** Client writes to sysfs path `.../cpu{idx}/cpufreq/scaling_max_freq` (~0.3 ms; frequency changes take effect within 1–2 CPU cycles, <1 μ s)
- **Profiling overhead:** `perf` measurement setup and teardown (~0.8 ms)
- **Response transmission:** Client returns profiling data (makespan, energy, temperature) to server (~0.4 ms)

This 2 ms overhead is negligible compared to benchmark execution times (1–6 seconds per episode). Hyperparameter tuning (grid search) is performed offline, requiring 10–12 hours on a 16 GB RAM, 4-core Intel Core i7 CPU server, adding no runtime cost. The thermal limit is set to **50°C**, consistent with prior work [66], ensuring safe operation while allowing performance headroom.

Table 2.5: Comparison of RL Approaches (Finetuned, Multi-Seed Mean±Std)

Approach	L10 (s)	L20 (s)	Energy (kJ)	HF%
Single-Agent Approaches				
zTT [66]	5.45±1.07	5.71±0.35	74.2±1.5	70.7±6.3
DynaQ [8]	10.72±2.08	9.42±1.81	93.6±5.8	30.0±9.4
PlanGAN [25]	7.57±0.67	5.85±0.49	73.9±2.1	67.0±1.6
GearDVFS [80]	14.32±2.61	14.51±2.63	128.4±9.5	21.3±1.7
SARB	7.66±4.54	10.07±5.32	124.2±28.5	28.7±26.8
Multi-Agent Approaches				
MAMB	7.09±1.29	6.91±0.93	87.3±5.9	66.0±4.5
MAMF	4.98±0.61	5.85±0.63	71.4±1.9	78.7±5.3
HiDVFS	4.16±0.58	5.14±1.06	63.7±3.7	81.0±0.8

HF% = High-Frequency (≥ 9) rate.

Note: SARB uses V8 (Q-clipping) for seed 42 and V4 (reward averaging) for seeds 123/456 due to version availability; see Appendix A.2 for version comparison.

Our proposed SARB and HiDVFS, leveraging per-cluster temperature profiling and model-based reward estimation, outperform SOTA RL baselines in makespan reduction, energy savings, and thermal management for FFT workloads. SARB converges in 12 epochs for simpler settings, while HiDVFS excels on complex DAGs with manageable deployment overhead on Jetson TX2. HiDVFS_S (without D3QN) shows higher variance but may suit resource-constrained deployments.

Comprehensive BOTS Benchmark Evaluation: Table 2.3 presents HiDVFS performance across 9 BOTS benchmarks compared to GearDVFS [80] on seed 42 with finetuning. HiDVFS achieves an average speedup of **3.95×** (from 18.48s to 6.06s) and energy reduction of 47.1% (from 14.70 kJ to 7.77 kJ) across all benchmarks. The highest speedups are observed for sparselu (**7.78×**) and alignment (5.58×), demonstrating HiDVFS’s effectiveness on diverse workloads. Even for challenging benchmarks like concom (1.94×) and uts (3.15×), HiDVFS maintains substantial improvements. These results validate HiDVFS’s generalization capability beyond the FFT case study, confirming its applicability to real-world parallel DAG workloads.

L10/L20=Avg last 10/20 epochs makespan.

Multi-Seed Statistical Validation: To ensure statistical robustness, we conducted experiments across three random seeds (42, 123, 456) on the FFT benchmark with 100 epochs per phase (detailed results in Table A.9 in Appendix A.3). HiDVFS achieves the best finetuning performance with **4.16s ± 0.58s** average makespan compared to GearDVFS’s 14.32s ± 2.61s, representing a consistent **3.44×** speedup across all seeds. HiDVFS also demonstrates superior energy efficiency (63.7 kJ vs. 128.4 kJ), high-frequency adoption (85.3%), and core utilization (85.7%). The low standard deviations confirm HiDVFS’s reliability across different random initializations.

2.6 Conclusion

We introduced HiDVFS, a hierarchical multi-agent DVFS scheduler for OpenMP DAGs with a makespan-first objective. With multi-seed validation (seeds 42, 123, 456) on Jetson TX2 using 9 BOTS benchmarks, HiDVFS achieves the best finetuned performance with 4.16 ± 0.58 s average makespan (L10), representing a $3.44\times$ speedup over GearDVFS and 50.4% energy reduction (63.7 kJ vs 128.4 kJ). Across all benchmarks, HiDVFS achieves an average $3.95\times$ speedup and 47.1% energy reduction. The decomposition of frequency/core selection, temperature-aware grouping, and static priority enabled fine-grained per-core control with low overhead.

**CHAPTER 3 FEATURE-AWARE TASK-TO-CORE ALLOCATION IN EMBEDDED
MULTI-CORE PLATFORMS VIA STATISTICAL LEARNING**

Abstract

Optimizing task-to-core allocation can substantially reduce power consumption in multi-core platforms without degrading user experience. However, existing approaches overlook critical factors such as parallelism, compute intensity, and heterogeneous core types. In this paper, we introduce a statistical learning approach for feature selection that identifies the most influential features—such as core type, speed, temperature, and application-level parallelism or memory intensity—for accurate environment modeling and efficient energy minimization, a critical consideration for embedded systems. Our experiments, conducted with state-of-the-art Linux governors and thermal modeling techniques, show that correlation-aware task-to-core allocation lowers energy consumption by up to 10% and reduces core temperature by up to 5°C compared to random core selection. Furthermore, our compressed, bootstrapped regression model improves thermal prediction accuracy by 6% while cutting model parameters by 16%, yielding an overall mean square error reduction of 61.6% relative to existing approaches. We provided results based on superscalar Intel Core i7 12th Gen processors with 14 cores, and validated our method across a diverse set of hardware platforms and effectively balanced performance, power, and thermal demands through statistical feature evaluation.

3.1 Introduction

Task-to-core allocation is a pivotal technique for enhancing the performance and reliability of embedded systems, where workloads exhibit distinct thermal and power consumption behaviors depending on core assignments. In multi-core processors, allocations can involve high-performance cores, low-energy cores, and graphic processing units (GPUs), each with unique performance characteristics across various configurations. Improper allocation can lead to thermal overheating, triggering throttling and reducing chip reliability and lifespan [56]. Cooling costs for overheated chips are significant, approximately \$3 per watt of heat dissipation [44]. While dynamic voltage and frequency scaling (DVFS) complements allocation by adjusting power dynamically—potentially reducing consumption by 75% without altering user experience [106]—our focus is on optimizing task-to-core allocation to manage thermal and energy constraints in embedded systems. Existing approaches often rely on historical workload and sensor data to predict future behavior [84], but lack a systematic, statistically robust framework, which our hybrid methodology addresses with novelty and precision.

Embedded systems, foundational to automotive, telecommunications, and consumer electronics, must operate under strict power and thermal constraints while delivering high performance. Task-to-core allocation is crucial for managing these constraints in heterogeneous architectures with diverse core types (e.g., high-performance, low-power, GPUs). Feature selection and evaluation identify critical features and correlations, enabling optimal core assignments for thermal stability and energy efficiency [118]. For instance, in real-time automotive systems, allocation ensures compute-intensive tasks run on performance cores without over-

heating, while mobile devices assign power-hungry applications to high-performance cores and background tasks to low-power cores, extending battery life. Identifying features tied to power and thermal behavior—such as temperature, frequency, and core adjacency—establishes general rules for allocation, enhancing embedded system reliability. This paper introduces a hybrid statistical learning framework, integrating Random Forest (RF), backward stepwise selection, and correlation analysis, to optimize task-to-core allocation across platforms, prioritizing allocation over DVFS for embedded contexts.

Application performance depends on execution factors like core type, speed, temperature, and application characteristics—memory or compute intensiveness and parallelism level [94]. Linux governors [21] and related studies often focus on CPU utilization for DVFS policies, targeting latency, temperature, or energy [82, 80, 66]. However, for task-to-core allocation, characteristics like parallelism, memory/compute intensiveness, and branch counts are equally critical. Applications with frequent branch misses benefit from sequential execution on single cores, while parallel, compute-intensive tasks excel on GPUs or performance cores in heterogeneous platforms. Moreover, temperature correlations between cores indicate adjacency and overheating risks. Our approach transcends utilization-centric methods, employing a multi-stage statistical framework to capture a comprehensive feature set and their interdependencies, ensuring clarity and applicability in embedded systems.

Collecting real data post-execution for task-to-core allocation optimization is computationally expensive and imprecise due to hardware sampling delays. Instead, allocation decisions can leverage inference from trained environmental models that account for randomness and

feature importance to mitigate overfitting [82]. An augmentation step further reduces sampling overhead, boosting efficiency. Complementary approaches include hierarchical multi-agent DVFS scheduling [101], zero-shot LLM-guided allocation [100], flow-augmented few-shot RL for sensor data [99], and graph-driven performance modeling [98]. Our paper introduces the first statistical learning framework for embedded systems, uniquely integrating RF-based feature reduction, backward stepwise selection, and correlation-aware allocation with bootstrapping to create a robust, platform-agnostic model. Unlike prior heuristic or filter-based methods, which lack systematic feature evaluation or adaptability across heterogeneous platforms, our approach outperforms existing techniques—where they exist—achieving up to 10% energy savings and 61.6% lower mean squared error, as validated against SOTA baselines on diverse embedded hardware.

Little work applies statistical learning to feature selection for task-to-core allocation with the rigor we propose. Statistical learning predicts outcomes and reveals feature relationships. Prior efforts used extreme value theorem (EVT) for energy and execution time bounds in DVFS contexts [24, 37, 108, 95], while Liu et al. [82] applied XGBoost for latency-related features, and Sasaki et al. [112] used decision trees and correlation for energy per instruction. These lack the integrated, multi-method focus we introduce for allocation, synergizing RF, OLS, and correlation analysis for superior predictive power and generalizability in embedded systems.

Our framework shows that correlation-aware task-to-core allocation reduces chip temperature at intermediate utilization by leveraging thermally independent cores. Backward stepwise selection reveals a small feature subset suffices for accurate energy estimation, while

RF extracts critical features with low overhead. Bootstrapping enhances dataset robustness, yielding a compact, efficient model. This hybrid methodology surpasses prior single-method approaches, validated across embedded platforms for practical impact.

The key contributions are:

1. This paper pioneers a hybrid statistical learning approach, integrating RF, backward step-wise selection, and correlation analysis, to evaluate feature importance for accurate environment modeling and compressed learning in task-to-core allocation for embedded systems.
2. We implement correlation-aware allocation, statistically reducing temperature and energy by assigning tasks to uncorrelated cores, validated across heterogeneous embedded platforms.
3. We employ bootstrapping to augment data and boost accuracy, supporting few-shot and meta-learning in resource-constrained embedded settings.
4. We rigorously test against state-of-the-art methods, achieving up to 10% energy reduction and 5°C temperature decrease. Our compressed, bootstrapped model cuts prediction error by 6%, reduces parameters by 16%, and improves mean squared error by 61.6% over SOTA. Data were extracted from Intel Core i7 8th and 12th Gen and Intel Xeon 2680 processors.

In the remainder, Section 6.2 surveys statistical learning and scheduling, Section 6.3 details motivation and challenges, and Section 6.4 describes the methodology. Section 6.5 evaluates

three processors, followed by conclusions in Section 6.6.

3.2 Motivation and Challenges

Feature selection is a cornerstone of statistical learning, pivotal for optimizing task-to-core allocation in embedded multi-core systems. It addresses critical challenges—curse of dimensionality, large data management, and performance unpredictability—common to platforms like Intel Core i7 and NVIDIA Jetson TX2. To surpass prior filter-based approaches and establish novelty, we leverage a hybrid methodology integrating Random Forest (RF), backward stepwise selection, and correlation analysis, tailored to embedded constraints. This section clarifies these challenges in the context of task-to-core allocation, emphasizing generalizability across heterogeneous architectures and grounding our motivation in rigorous, platform-validated insights.

3.2.1 Curse of Dimensionality

Feature selection is essential to mitigate the curse of dimensionality, a challenge amplified in embedded multi-core systems. As feature count rises—encompassing frequency, temperature, and performance metrics like cache misses—the data space grows exponentially, leading to sparsity in high-dimensional spaces [77]. This sparsity risks model overfitting, degrading predictive performance on unseen data. In heterogeneous embedded systems, the diversity of cores (performance, low-energy, GPUs) expands the state space, with each core type contributing unique thermal and performance characteristics. For instance, Intel Core i7 12th Gen’s hybrid P/E-cores and Jetson TX2’s big-LITTLE clusters introduce complex feature interactions. Efficient feature selection, as implemented via our RF-based reduction, reduces this complex-

ity, enabling compact, generalizable models critical for real-time embedded applications—offering a novel advance over traditional dimensionality handling.

3.2.2 Large Data Management

Managing the growing volume of data is a pressing challenge for task-to-core allocation in embedded systems, where resource constraints demand efficiency. Allocation strategies may use static historical data or streaming inputs adapting to runtime conditions. With streaming data’s rise—common in automotive or mobile embedded platforms—memory management becomes critical, as unpredictable data volumes can overwhelm limited resources. Retaining unnecessary features, such as redundant performance counters, inflates storage and processing overhead, straining embedded systems. Our hybrid approach, combining RF for initial feature pruning and bootstrapping for data augmentation, optimizes processor models and allocation algorithms, reducing overhead while maintaining accuracy. This addresses large data challenges with clarity and applicability across static and dynamic embedded environments, surpassing simpler filter-based data handling.

3.2.3 Performance Unpredictability

Performance unpredictability in embedded multi-core processors stems from energy and thermal variability, driven by features like temperature, frequency, and core-specific performance data. Figure 3.1 illustrates this, showing energy consumption varying by an order of magnitude across identical frequency and core settings on Intel Core i7 8th Gen (4 cores), Intel Core i7 12th Gen (14 cores), and Intel Xeon 2680 v3 (12 cores) under OpenMP benchmarks. This variability underscores the need for feature selection to pinpoint significant predic-

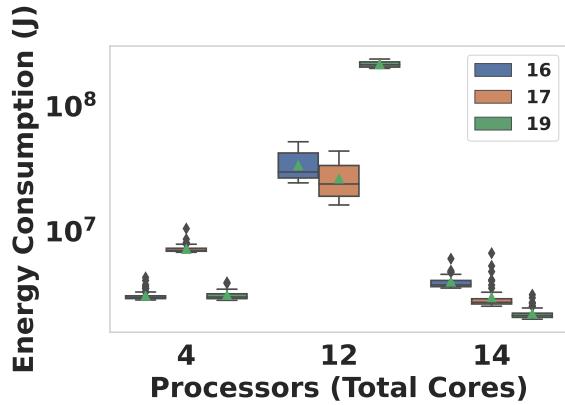


Figure 3.1: Energy consumption variation across three different processors with identical frequency and core count settings: Intel Core i7 8th Gen (Corei78) with 4 cores, Intel Core i7 12th Gen (Corei712) with 14 cores, and Intel Xeon 2680 v3 (Xeon) with 12 cores. The results are shown for three different OpenMP benchmarks.

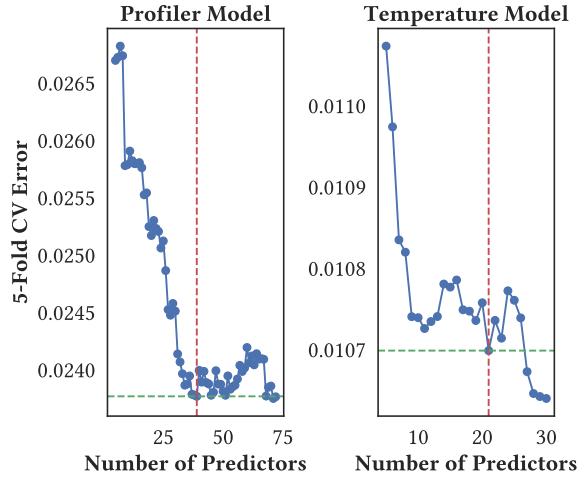


Figure 3.2: Determining the significance of features using Random Forest with respect to cross-validation error on Intel Core i7 12th Gen.

tors of energy and thermal behavior across platforms and configurations. Our methodology—using backward stepwise OLS for energy-critical features and correlation analysis for thermal independence—identifies robust predictors, enabling a global allocation policy. Unlike DVFS-centric prior works, we prioritize task-to-core allocation, validated rigorously with up to 10% energy savings and 5°C temperature reductions, ensuring practical impact in embedded systems.

3.3 Design Methodology

Our objective is to develop a robust multistage methodology based on three feature selection approaches—embedded, wrapper-based, and filter-based—and show how their combined use informs an intelligent task-to-core allocation algorithm for embedded multi-core systems. We introduce a hybrid framework integrating Random Forest (RF), backward stepwise OLS,

and Pearson correlation, surpassing traditional filter-based methods in efficiency and predictive power. We provide a structured workflow adaptable to diverse platforms like Intel Core i7 8th and 12th Gen, Intel Xeon 2680 v3, and NVIDIA Jetson TX2, while ensuring experimental rigor through validated outcomes (e.g., 10% energy savings, 5°C temperature reduction). This section details each stage, linking to embedded system optimization.

3.3.1 Data Collection and Environment Setup

We begin by profiling the target embedded hardware under diverse workloads to capture comprehensive data on energy consumption, temperature variations, and relevant performance metrics. The granularity of thermal sensors significantly varies across platforms. Intel Core i7 8th generation and Intel Xeon 2680 v3 processors provide homogeneous core architectures with individual per-core thermal sensors, enabling detailed per-core correlation analyses. In contrast, the Intel Core i7 12th generation features a heterogeneous architecture composed of performance (P-cores) and efficient (E-cores) cores, each type operating at different frequencies and performance characteristics. Similarly, the NVIDIA Jetson TX2 offers thermal readings aggregated at the cluster level, typically distinguishing between big and little core clusters due to its big-LITTLE architecture. Thus, our methodology accommodates per-core, heterogeneous core, and cluster-level sensor capabilities, specifically tailored to embedded computing environments, ensuring broad applicability.

Our data collection covers a wide range of configurations, including varying core counts (4, 12, or 14 cores), diverse CPU frequencies, and task prioritizations. Tasks include representative benchmarks such as Fibonacci (fib), matrix multiplication (Strassen), and sequence alignment

(alignment), providing varied computational intensities and memory access patterns typical of embedded system workloads. This diverse dataset enables accurate modeling of core behaviors under different workloads and thermal conditions, essential for embedded system optimization.

Initially, the system setup involves configuring the frequency governor (typically `schedutil`) and initializing a temperature buffer capable of storing historical temperature data from cores (up to 10,000 entries). Additionally, a detailed task history is maintained, capturing execution parameters such as energy consumption, profiler metrics, and temperature data under varying task priorities and frequencies. This systematic variation ensures the dataset's robustness and representativeness for embedded systems.

Energy monitoring employs built-in hardware interfaces such as `/sys/class/powercap/intel-rapl/` on Intel processors or on-board power sensors for the Jetson TX2, recording total energy consumed or average power during task execution. Performance metrics collected include CPU frequency, utilization, memory bandwidth, and hardware performance counters like cache misses, gathered via utilities such as `perf` and `cpufreq-info` at consistent intervals.

The random core assignment strategy serves as a baseline to evaluate the effectiveness of our proposed correlation-aware core allocation strategy. In the random assignment approach, tasks run concurrently on randomly selected cores, typically half of the available cores excluding core 0, reserved for system tasks. Post-execution, we record metrics including temperature, energy consumption, and performance data. Execution is temporarily paused if core temperatures exceed predefined thresholds (e.g., depending on thermal throttling point of Intel Core

i7 which is 100°C), allowing cores to cool down before resuming execution. This baseline facilitates comparative analysis against the correlation-aware allocation strategy.

The collected dataset serves as input for the subsequent embedded feature reduction process, employing Random Forest algorithms to identify minimal yet critical feature subsets. This refined feature set is then utilized for accurate environment modeling, guiding intelligent, temperature-correlation-driven task-to-core allocation strategies specifically designed for embedded systems. Ultimately, our structured, multi-stage methodology aims to optimize thermal management and energy efficiency across diverse embedded multi-core computing platforms.

3.3.2 Embedded Feature Selection Using Random Forest

Embedded methods incorporate feature selection into the model training process, thereby minimizing the need for multiple model evaluations on different feature subsets. In this work, we employ a *Random Forest* (RF) regressor, an ensemble approach that constructs multiple decision trees and aggregates their predictions, offering a novel low-overhead alternative to filter-based methods.

Random Forest Algorithm. The RF algorithm proceeds as follows:

1. **Bootstrap Sampling:** Generate N bootstrap samples from the original dataset.
2. **Tree Construction:** For each bootstrap sample, grow a regression tree by selecting a random subset of features at each node (often \sqrt{d} features). Each tree is grown to its maximum depth without pruning, although hyperparameters such as the number of trees or maximum depth can be tuned for embedded constraints.

3. Prediction Aggregation: For regression tasks, the final prediction is the average of the individual tree outputs:

$$\hat{y} = \frac{1}{N} \sum_{i=1}^N \hat{y}_i. \quad (3.1)$$

Feature Importance Computation. Random Forests provide a measure of the importance of features by evaluating the total decrease in the impurity of the nodes in all trees. For regression trees, impurity is commonly measured by the residual sum of squares. The importance score I_j for feature x_j is thus:

$$I_j = \frac{1}{N} \sum_{i=1}^N \sum_{t \in T_i} \Delta I_{t,j}, \quad (3.2)$$

where $\Delta I_{t,j}$ is the impurity decrease at node t when splitting on x_j , and T_i is the set of nodes in the i -th tree. The ranking of features by these importance scores guides the selection of highly influential predictors.

Bootstrapping for Data Augmentation. To increase model robustness, we employ bootstrapping, a resampling method that draws multiple datasets of size n with replacement. Let \mathcal{D} be the original dataset of size n . Forming B bootstrap samples $\{\mathcal{D}_1, \dots, \mathcal{D}_B\}$ helps estimate variance and stabilize the final model through aggregation of predictions.

Environment Modeling with Feature Selection. After identifying the most significant features using RF importance scores, we build predictive models for environment modeling—particularly neural networks—tailored to embedded constraints. Let $\mathbf{x} \in \mathbb{R}^p$ (with $p < d$) denote the reduced feature set. We train a Fully Connected Neural Network (FCN) with mul-

tiple layers and non-linear activations to predict key variables such as energy consumption or temperature. The network is trained by minimizing the mean squared error (MSE):

$$\mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^n (y_i - f(\mathbf{x}_i; \theta))^2, \quad (3.3)$$

where $f(\mathbf{x}_i; \theta)$ is the network's output for input \mathbf{x}_i with parameters θ , and y_i is the true label. By restricting the input to a smaller set of highly relevant features, the FCN requires fewer parameters and less computation, making it feasible for real-time applications. As shown in Figure 3.2, our results indicate that using 39 out of 72 predictors provides nearly the same error value for estimating the future state of the processor profiler data model. For the sensor temperature model, using 21 out of 31 features yields the optimal Cross-Validation error (*CV error*), validated for embedded efficiency.

3.3.3 Wrapper-Based Feature Selection

Wrapper methods evaluate subsets of features using a predictive model. Because they account for feature interactions, they typically yield higher accuracy than filter methods in finding the importance of the features on a specific feature parameter but may be more computationally expensive. We employ the *backward stepwise selection* algorithm, which starts with all available features and iteratively removes the least significant feature based on a specified criterion. In our case, we use the Ordinary Least Squares (OLS) regression model to predict the target variables (energy consumption and average temperature) and assess the significance of the characteristics using statistical tests, enhancing the hybrid framework's precision.

Backward Stepwise Selection Algorithm. Let $\mathcal{F} = \{x_1, x_2, \dots, x_d\}$ denote the full set of features. The backward stepwise selection algorithm proceeds as follows:

1. **Initial Model:** Fit the OLS regression model using all features in \mathcal{F} :

$$y = \beta_0 + \sum_{i=1}^d \beta_i x_i + \varepsilon \quad (3.4)$$

where y is the target variable, β_0 is the intercept, β_i are the coefficients, and ε is the error term.

2. **Evaluate Feature Significance:** For each feature x_i , compute the t-statistic and the corresponding p-value to assess its statistical significance. The t-statistic for coefficient β_i is calculated as:

$$t_i = \frac{\hat{\beta}_i}{\text{SE}(\hat{\beta}_i)}, \quad (3.5)$$

where $\hat{\beta}_i$ is the estimated coefficient and $\text{SE}(\hat{\beta}_i)$ is its standard error.

3. **Feature Elimination:** Identify the feature with the highest p-value (least significant) that exceeds a predefined significance level (e.g., $\alpha = 0.05$). Remove this feature from the model.

4. **Iterative Refinement:** Refit the OLS model using the reduced feature set and repeat steps 2 and 3 until all remaining features are statistically significant.

5. **Model Selection Criteria:** At each iteration, evaluate the model using metrics such as the Akaike Information Criterion (AIC), Bayesian Information Criterion (BIC), Mal-

lows' C_p , Adjusted R^2 , and CV Error. These metrics help balance model complexity and goodness of fit.

Evaluation Metrics for Our Wrapper Algorithm. We employ multiple metrics to gauge not only how well each model fits the data, but also how efficiently it uses the available features. This multi-criteria evaluation helps us verify a model that performs reliably, avoids overfitting, and remains computationally viable for energy-aware and thermal-critical environments:

- *Akaike Information Criterion (AIC)*: AIC estimates the relative quality of statistical models for a given dataset:

$$\text{AIC} = 2k - 2 \ln(L), \quad (3.6)$$

where k is the number of estimated parameters, and L is the maximized value of the likelihood function. Lower AIC values imply better trade-offs between model complexity and fit.

- *Bayesian Information Criterion (BIC)*: BIC imposes a stronger penalty on model complexity than AIC:

$$\text{BIC} = k \ln(n) - 2 \ln(L), \quad (3.7)$$

where n is the number of observations. BIC is helpful for avoiding over-complex models in resource-constrained environments.

- *Mallows' C_p* : This criterion assesses the balance between the model's complexity and its

fit to the data:

$$C_p = \frac{\text{RSS}}{\hat{\sigma}^2} - (n - 2k), \quad (3.8)$$

where RSS is the residual sum of squares, and $\hat{\sigma}^2$ is an estimate of the error variance.

- *Adjusted R²*: Adjusted R^2 modifies the coefficient of determination (R^2) to account for the number of predictors:

$$\text{Adjusted } R^2 = 1 - \left(\frac{(1 - R^2)(n - 1)}{n - k - 1} \right). \quad (3.9)$$

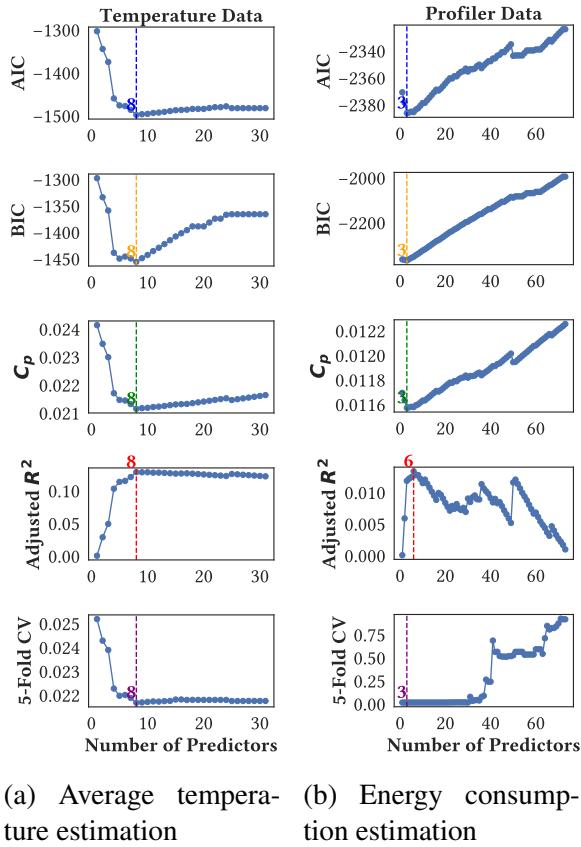
Unlike plain R^2 , it penalizes the model for including uninformative features.

- *Cross-Validation Error*: CV Error (often computed via K -fold CV) offers an unbiased measure of out-of-sample performance, revealing the model's generalization capability and mitigating overfitting concerns.

By applying these evaluation metrics, we identify the optimal number of features that balance predictive accuracy and model simplicity. Figures 3.3a and 3.3b demonstrate that using fewer than 8 features suffices for accurate estimation of both average temperature and energy consumption, indicating that tracking only the most relevant predictors can improve energy efficiency and thermal behavior, validated on Intel Core i7 12th Gen.

3.3.4 Filter-Based Feature Selection

Filter methods assess the relevance of features by examining intrinsic properties of the data without involving any learning algorithms. One common technique in filter methods is to eval-



(a) Average temperature estimation (b) Energy consumption estimation

Figure 3.3: Backward stepwise selection for estimating energy consumption and average temperature. Retaining fewer than 8 predictors (features) yields accurate predictions in both cases. Experiments performed on Intel Core i7 12th Gen.

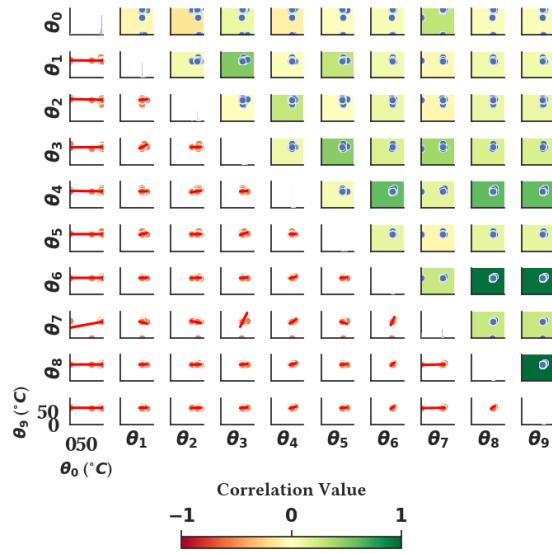


Figure 3.4: Correlation matrix based on Pearson correlation coefficients for 10 selected cores from an Intel Core i7 12th Gen processor with 14 cores.

uate the correlation between features and the target variable or among the features themselves. In our study, we utilize the Pearson correlation coefficient [114] to quantify the linear relationship between core temperatures, which can indicate adjacency and potential heat transfer between cores, enhancing allocation decisions.

Given a set of n observations for m cores, let $\theta_{i,k}$ denote the temperature of core i at observation k , and $\bar{\theta}_i$ represent the mean temperature of core i across all observations. The Pearson correlation coefficient r_{ij} between cores i and j is computed as:

$$r_{ij} = \frac{\sum_{k=1}^n (\theta_{i,k} - \bar{\theta}_i)(\theta_{j,k} - \bar{\theta}_j)}{\sqrt{\sum_{k=1}^n (\theta_{i,k} - \bar{\theta}_i)^2} \sqrt{\sum_{k=1}^n (\theta_{j,k} - \bar{\theta}_j)^2}} \quad (3.10)$$

The value of r_{ij} ranges from -1 to 1 , where 1 indicates a perfect positive linear correlation, -1 indicates a perfect negative linear correlation, and 0 signifies no linear correlation between the temperatures of cores i and j . A high positive correlation suggests that the temperatures of the two cores rise and fall together, possibly due to physical proximity and shared thermal characteristics.

By constructing a correlation matrix $\mathbf{R} = [r_{ij}]$ for all pairs of cores, we can visualize and identify clusters of cores that are thermally correlated. This information is crucial for designing task-to-core allocation strategies that minimize thermal hotspots. As shown in Figure 3.4, the lower diagonal part represents the regression line in the sparsified data, and the upper diagonal part shows the colored correlation matrix, where a greener color indicates a more positive correlation between the temperatures of two cores.

Correlation-Aware Task-to-Core Allocation Algorithm. To leverage the insights from the correlation analysis, we propose a correlation-aware task-to-core allocation algorithm. The goal is to assign tasks to cores that are less thermally correlated, thereby reducing the risk of localized overheating and improving overall energy efficiency.

Let $C = \{c_1, c_2, \dots, c_m\}$ denote the set of available cores, and let \mathbf{R} be the correlation matrix computed using Equation (4.5). The algorithm proceeds as follows:

1. **Compute Core Correlation Scores:** For each core c_i , calculate a correlation score s_i defined as the average absolute correlation between core c_i and all other cores:

$$s_i = \frac{1}{m-1} \sum_{\substack{j=1 \\ j \neq i}}^m |r_{ij}| \quad (3.11)$$

A lower score s_i indicates that core c_i is less correlated with other cores.

2. **Rank Cores Based on Correlation Scores:** Sort the cores in ascending order of their correlation scores to obtain a ranked list C_{ranked} .

3. **Select Cores for Task Assignment:** Given the number of tasks T to be assigned, select the first T cores from C_{ranked} , which are the least correlated cores.

4. **Assign Tasks to Selected Cores:** Allocate tasks to the selected cores, ensuring that each task is assigned to a core with minimal thermal correlation to other active cores.

5. **Update Temperature Buffer:** After task execution, update the temperature observations $\theta_{i,k}$ to reflect the new core temperatures, and recompute the correlation matrix \mathbf{R} for

subsequent allocations.

3.3.5 Summary of the Multi-Stage Methodology

1. **Data Collection:** Profile per-core or per-cluster temperatures, energy, and performance under varied workloads.
2. **Random Forest (Embedded):** Prune low-importance features and optionally train an FCN environment model.
3. **Backward Stepwise (Wrapper):** Further refine the most energy-critical features using OLS-based feature elimination.
4. **Filter-Based (Correlation):** Analyze temperature correlations (per-core or per-cluster) to guide thermal-aware scheduling.
5. **Task Allocation:** Prioritize assigning tasks to the least-correlated units, mitigating overheating and reducing energy consumption.

By unifying these steps, we handle *both* fine-grained and coarse-grained thermal sensor inputs without losing accuracy or tractability. Random Forest algorithm provides the dimensionality reduction required for backward stepwise selection to reduce its computational overhead and their combination ensures only the most relevant features to the energy consumption remain. The filter-based approach on temperature readings further addresses the correlation of the cores to each other for core allocation to minimize the thermal hotspots, offering a novel, scalable solution validated across platforms (e.g., 61.6% MSE improvement over SOTA).

3.4 Experiments

This section presents the experimental setup, implementation, and results of our generalized task-to-core allocation framework, integrating Random Forest (RF) feature reduction, backward stepwise OLS selection, and filter-based temperature correlation. To ensure novelty beyond prior filter-based approaches, we validate this hybrid methodology across diverse platforms, demonstrating its effectiveness in optimizing energy and thermal behavior. For clarity and broad applicability, we detail platforms, benchmarks, and metrics, extending beyond Intel-specific results to cluster-based systems. Rigorous quantitative outcomes—backed by multi-platform experiments and comparisons to state-of-the-art (SOTA)—establish the framework’s soundness. We outline the setup, training, and empirical findings below, linking results to the methodology’s advanced feature selection and modeling strategies.

3.4.1 Experimental Platform, Benchmarks, and Evaluation

All results and figures primarily reflect experiments on a superscalar Intel Core i7 12th Gen with 14 cores (8 P-cores, 6 E-cores, per-core temperature via `sensors`), chosen for its hybrid architecture. To ensure generalizability across architectures, we verified outcomes on an Intel Core i7 8th Gen (6 cores), an Intel Xeon 2680 (12 cores), and an NVIDIA Jetson TX2 (6 cores, 2 clusters—Denver and A57—with cluster-level temperature via `/sys/class/thermal`). This multi-platform approach validates the framework’s adaptability to per-core and cluster-based systems, addressing the need for broader applicability.

We evaluated each platform using the Barcelona OpenMP Tasks (BOTS) suite [40], featuring diverse parallel workloads (e.g., `sparselu`, `nqueens`), to test allocation under vary-

ing computational demands. Performance was assessed on three metrics: *makespan* (execution time in seconds), *energy consumption* (microjoules via `/sys/class/powercap` for Intel, `/sys/class/thermal/energy` for Jetson), and *average core temperature* ($^{\circ}\text{C}$). These metrics capture performance, power, and thermal trade-offs, ensuring practical impact. Each BOTS benchmark ran 10 times per configuration, with results averaged to reduce variability. A temperature cooldown threshold of 70°C between runs ensured thermal stability, enhancing experimental rigor.

3.4.2 Implementation and Training Details

The framework was implemented in Python, leveraging `subprocess` for system calls (e.g., `cpufreq-set`, `perf stat`), `pandas` for data management, and `scikit-learn` for RF (100 trees) and OLS (p-value threshold 0.05). Parsl facilitated parallel task execution, scaling across platforms. Data from each platform’s profiler and temperature sensors were split into 80% training and 20% test sets. Key hyperparameters—batch size (32), hidden neurons (64–256), epochs (50–200), and learning rate (0.001–0.01)—were tuned via grid search, balancing convergence and generalization. Mean Squared Error (MSE) served as the primary loss criterion, ensuring predictive accuracy.

We evaluated multiple neural network architectures—Fully Connected Networks (FCN), Recurrent Neural Networks (RNN), Long Short-Term Memory (LSTM) networks, Convolutional Neural Networks (CNN), and Attention-based models (e.g., Transformers)—retaining top performers based on validation loss. Feature subsets were derived using the methodology’s three stages: Pearson correlation for temperature dependencies, backward stepwise OLS for

energy-correlated features, and RF importance rankings for reduction. Bootstrapping (100 resamples) reduced overfitting and increased robustness, preserving critical predictors while providing variance insights under different sampling scenarios, contributing to the framework’s statistical rigor.

3.4.3 Empirical Results

We developed two predictive models: a *profiler model* for energy consumption and performance metrics (e.g., cache miss rates, branch miss rates, CPU cycles, instructions per cycle, average speed, page faults, context switches) and a *temperature model* for future thermal behavior. The profiler model incorporated all available features (e.g., 75 on Intel Core i7 12th Gen) plus current per-core temperatures θ_i and differences $\Delta\theta_i = \theta_{i,t} - \theta_{i,t-1}$. The temperature model used 30 features on Intel Core i7 12th Gen (14 temperatures, 14 differences, average temperature), scaled to 6 features on Jetson TX2 (2 cluster temperatures, 2 differences, average), maintaining diversity without overhead.

Figures 3.4, 3.5, 3.3a, and 3.3b showcase Intel Core i7 12th Gen results, demonstrating that fewer than 8 features (e.g., frequency, cache misses) suffice for accurate predictions, reducing computational costs. Figure 3.2 shows RF-based feature selection using 39 out of 72 predictors for profiler data and 21 out of 31 for temperature yields nearly optimal CV error, highlighting efficiency critical for real-time embedded systems with strict power and thermal budgets.

The algorithm dynamically adapts to thermal behavior, distributing load evenly across cores or clusters. Figure 3.5 compares correlation-based (Corr) and random (Rand) core selection under different governors on Intel Core i7 12th Gen. While random allocation leverages unbi-

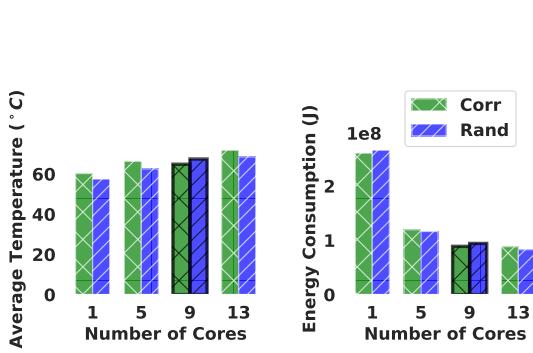


Figure 3.5: Comparison of average temperature and energy consumption for correlation-based (Corr) and random (Rand) core selection. Experiments performed on Intel Core i7 12th Gen processor with 14 cores.

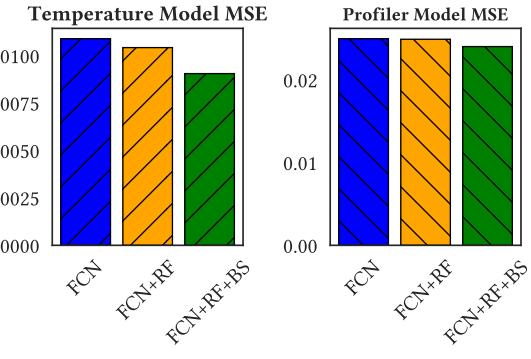


Figure 3.6: Comparison of FCN models with and without feature selection and bootstrapping on Intel Core i7 12th Gen.

ased distribution, correlation-based allocation excels in subset scenarios (e.g., 10 of 14 cores).

Multi-platform validation reinforces these trends across Jetson TX2, Intel Xeon 2680, and Intel Core i7 8th Gen, proving the approach's adaptability beyond a single architecture.

During training, hyperparameters were tuned to optimize convergence speed and generalization, with models saved at optimal checkpoints (e.g., when validation error plateaued or declined sharply). Independent neural networks for temperature and profiler tasks ensured specialized performance, delivering precise predictions tailored to each objective.

Table 3.1 lists top features from Random Forest (RF) and backward stepwise OLS selections for Intel Core i7 12th Gen. Tables 3.2 and 3.3 present regression statistics for temperature and profiler predictors against energy consumption.

Table 3.1 shows RF selecting a broader feature set (e.g., all core temperatures, performance metrics) compared to the more selective stepwise OLS, highlighting their complementary roles

Dataset	Random Forest (Top Features)	Stepwise OLS (Top Features)
Profiler	Task ID, Actions, Temp Core 0-13, Δ Temp Core 0-13, Avg Temp, CPU Cycles, CPU Cycles Freq, Atom Cycles, Instructions, Insn per Cycle, Atom Insn, Atom Insn per Cycle, Cache Refs, Cache Refs Rate, Atom Cache Refs, Atom Cache Refs Rate, Cache Misses, Cache Miss % Refs, Atom Cache Misses, Atom Cache Miss % Refs, Cycles, Cycles Freq, Atom Cycles, Atom Cycles Freq, Branch Misses, Branch Miss % Branches, Atom Branch Misses, Atom Branch Miss % Branches, Branches, Branches Rate, Atom Branches, Atom Branches Rate, Page Faults, Page Fault Rate, Context Switches, Context Switch Rate, CPU Clock ms, CPUs Used, Task Clock ms, Task Util, Faults, Fault Rate, Sys Time s, Elapsed Time s, Energy	Task ID, Temp Core 1, Temp Core 13
Temperature	Task ID, Actions, Temp Core 0-9, Temp Core 11-13, Δ Temp Core 0-13, Avg Temp	Task ID, Temp Core 0, Temp Core 2-4, Temp Core 7, Temp Core 10, Avg Temp

Table 3.1: Key features ranked by Random Forest and backward stepwise OLS (lowest CV Error) for Profiler and Temperature datasets. Temp Core 0-13 denotes per-core temperatures; Δ Temp is the temperature difference.

in feature reduction. For the profiler dataset, Task ID consistently ranks high in both methods, underscoring its key role in identifying energy consumption patterns, followed by Temp Core 1 and Temp Core 13 in OLS as significant predictors. This suggests that cores 1 and 13, when heavily utilized, may disproportionately influence energy use due to their temperature profiles. Consequently, an allocation strategy minimizing task assignments to these cores could enhance energy efficiency, a hypothesis supported by their prominence in the feature selection process.

θ_i	Xeon (12 cores)			Core i7 8th Gen (4 cores)			Core i7 12th Gen (14 cores)				
	Est.	t-val.	p-val.	θ_i	Est.	t-val.	p-val.	θ_i	Est.	t-val.	p-val.
θ_1	2.35e5	5.76	8.48e-9	θ_0	3.64e3	5.69	1.31e-8	θ_2	-3.79e3	-6.83	8.41e-12
θ_9	-2.53e5	-5.38	7.38e-8	θ_1	-9.77e3	-4.18	2.87e-5	θ_0	4.87e2	2.76	5.81e-3
θ_7	-1.56e5	-3.10	1.93e-3	θ_3	8.00e3	3.57	3.64e-4	θ_3	-5.96e3	-2.05	4.06e-2
θ_6	-9.90e4	-2.04	4.09e-2	θ_2	-4.63e2	-0.26	7.97e-1	θ_9	-2.15e4	-1.93	5.38e-2
θ_8	-9.15e4	-1.91	5.66e-2					θ_1	4.77e3	1.88	5.98e-2
θ_4	6.84e4	1.64	1.01e-1					θ_{13}	-1.22e4	-1.77	7.62e-2
θ_2	6.62e4	1.59	1.12e-1					θ_{10}	9.45e3	1.47	1.41e-1
θ_{11}	-4.82e4	-1.10	2.72e-1					θ_4	3.43e3	1.02	3.10e-1
θ_{10}	-2.29e4	-0.52	6.02e-1					θ_{11}	6.64e3	0.91	3.65e-1
θ_0	-2.04e4	-0.52	6.02e-1					θ_7	6.87e3	0.43	6.67e-1
θ_3	-4.50e3	-0.10	9.18e-1					θ_5	1.40e3	0.38	7.05e-1
θ_5	4.68e1	0.00	9.99e-1					θ_{12}	1.06e3	0.13	8.94e-1
								θ_6	8.30e2	0.07	9.44e-1
								θ_8	8.11e1	0.01	9.96e-1

Table 3.2: Temperature predictors (θ_i) vs. energy: Estimates (Est.), t-values (t-val.), p-values (p-val.), sorted by p-value. Significant predictors (e.g., θ_1 , θ_0 , θ_2) show strong energy impact.

Table 3.2 evaluates per-core temperatures' effect on energy, with low p-values indicating significant predictors across platforms (e.g., θ_1 on Xeon, θ_2 on Core i7 12th Gen).

Table 3.3 highlights profiler features' statistical significance, with metrics like Context

Predictor	Xeon (12 cores)			Core i7 8th Gen (4 cores)			Core i7 12th Gen (14 cores)				
	Est.	t-val.	p-val.	Predictor	Est.	t-val.	p-val.	Predictor	Est.	t-val.	p-val.
Context Switch Rate	-3.07e4	-34.90	1.34e-261	Elapsed Time	4.85e7	39.51	0.00e0	Elapsed Time	4.67e7	35.82	3.29e-275
Cache Miss % Refs	3.32e6	34.45	4.02e-255	Cache Miss	2.23e5	21.35	3.80e-100	Atom Branch Misses	1.41e0	25.10	1.21e-137
Cache Refs	2.82e-1	33.53	4.94e-242	Cache Refs	6.39e-2	16.03	1.48e-57	Cache Misses	4.18e-1	24.65	7.37e-133
Elapsed Time	6.44e7	31.52	1.52e-214	Total Cores	-5.34e6	-15.47	1.09e-53	Branch Misses	1.36e0	22.78	6.66e-114
Cycles	-1.98e-1	-16.52	5.22e-61	Branch Miss	3.37e6	10.41	2.43e-25	Atom Branches	-1.93e-2	-18.87	5.72e-79
Branches	-1.69e-1	-14.19	1.59e-45	Instructions	-1.48e-3	-6.51	7.61e-11	Atom Cycles Freq	-2.31e6	-17.85	7.14e-71
CPU Cycles	-4.33e-2	-10.94	8.14e-28	Context Switch Rate	-2.43e2	-5.42	6.10e-8	Atom Branches Rate	7.26e3	17.30	1.06e-66
Context Switches	-2.14e3	-10.15	3.53e-24	Context Switches	2.07e3	5.22	1.82e-7	Branches	1.17e4	15.79	6.30e-56
Cache Misses	-4.37e-1	-9.78	1.47e-22	Insn per Cycle	5.84e5	4.33	1.52e-5	Atom Cycles	-1.07e-1	-15.33	7.10e-53
Total Cores	7.82e4	9.69	3.43e-22	Branch Misses	-1.06e0	-4.18	2.95e-6	User Time	-1.31e8	-11.71	1.33e-31
Cycles Freq	1.74e8	9.65	5.41e-22	Cache Misses	1.04e-1	3.88	1.03e-4	Sys Time	-1.29e8	-11.59	5.34e-31
CPU Clock ms	2.63e6	8.81	1.29e-18	CPU Clock	-5.82e5	-3.68	2.30e-4	Faults	3.77e2	11.31	1.32e-29
User Time	-2.27e8	-8.63	6.37e-18	CPU Cycles Freq	1.69e5	3.56	3.65e-4	Cycles Freq	7.57e4	11.31	1.40e-29
Sys Time	-2.20e8	-8.38	5.60e-17	Task Clock	5.60e5	3.55	3.91e-4	Page Faults	-3.70e2	-11.23	3.42e-29
Cache Refs Rate	6.97e5	8.05	8.51e-16	CPU Cycles	6.89e-4	2.64	8.18e-3	Avg Freq	1.14e3	10.39	3.00e-25
Task Clock	-1.80e6	-6.17	7.12e-10	Page Fault Rate	9.68e3	2.45	1.41e-2	Atom Cache Miss	-1.29e4	-10.15	3.74e-24
Branch Misses	1.82e0	4.24	2.22e-5	Fault Rate	9.68e3	2.45	1.42e-2	Atom Branch Miss	9.16e0	9.09	1.07e-19
Insn per Cycle	-9.52e6	-4.23	2.36e-5	Task Util	9.01e6	2.28	2.25e-2	Instructions	-4.71e-4	-8.74	2.45e-18
Task Util	-1.41e8	-2.30	2.12e-2	Cycles Freq	3.52e5	1.39	1.64e-1	Cache Miss	1.25e4	7.55	4.56e-14
CPUs Used	1.25e8	2.04	4.14e-2	Branches Rate	1.56e3	1.34	1.80e-1	Branches Rate	-1.77e-1	-6.68	2.44e-11
CPU Cycles Freq	2.00e6	1.20	2.30e-1	User Time	4.06e6	1.04	2.99e-1	Cycles	-1.11e-3	-5.93	3.04e-9
Page Faults	-2.64e1	-1.18	2.36e-1	Cycles	1.15e-3	0.99	3.21e-1	Branch Miss	6.12e0	5.86	4.69e-9
Faults	-2.62e1	-1.17	2.41e-1	CPUs Used	-3.43e6	-0.87	3.85e-1	CPU Cycles	-8.56e-4	-5.71	1.14e-8
Instructions	7.52e-4	0.51	6.09e-1	Sys Time	3.21e6	0.82	4.12e-1	Atom Cycles	1.12e-3	5.67	1.47e-8
Fault Rate	-1.69e2	-0.00	9.97e-1	Faults	5.90e0	0.53	5.95e-1	Branch Miss	2.58e4	5.37	8.12e-8
Page Fault Rate	-1.69e2	-0.00	9.97e-1	Branches	-8.77e-4	-0.17	8.69e-1	Context Switch Rate	-5.68e2	-4.86	1.20e-6
				Page Faults	1.90e-1	0.02	9.86e-1	Atom Cache Miss	1.09e0	4.61	4.01e-6

Table 3.3: Profiler predictors (excl. temp, freq, speed) vs. energy: Estimates (Est.), t-values (t-val.), p-values (p-val.), sorted by p-value. Key predictors (e.g., Context Switch Rate, Elapsed Time) drive energy variance.

Switch Rate and Cache Misses showing strong energy correlations across platforms.

3.4.4 Comparative Model Analysis

We tested multiple neural architectures to identify optimal designs for energy and temperature prediction, benchmarking against a SOTA approach [55] to establish novelty and rigor. Table 3.4 details MSE percentage ($MSE \times 100$) and parameter counts on Intel Core i7 12th Gen. Baseline FCN achieved solid performance (MSE 1.0299 temperature, 3.9047 profiler), but FCN with RF feature selection (FCN+RF) improved accuracy (0.9808, 2.4862) with fewer parameters (1694 vs. 2014 temperature; 2699 vs. 3787 profiler). Adding bootstrapping (FCN+RF+BS) yielded the best results (0.9640, 2.4669), maintaining lightweight models ideal for resource-constrained systems.

Figures 3.7 and 3.8 present prediction examples from Intel Core i7 12th Gen thermal and

Table 3.4: MSE percentage and total number of parameters for different architectures on Intel Core i7 12th Gen.

Model	Temperature		Profiler	
	MSE	Params	MSE	Params
FCN	1.0299	2014	3.9047	3787
FCN+RF	0.9808	1694	2.4862	2699
FCN+RF+BS	0.9640	1694	2.4669	2699
RNN	1.0119	3070	2.8493	4843
LSTM	1.0307	9310	2.7778	15115
Conv	1.0134	5118	2.8217	6891
Attention	1.0143	6238	2.8933	8011
SOTA [55]	2.5000	-	-	-

profiler data. The FCN model uses the complete feature set, while FCN+RF leverages a reduced subset identified by RF, forecasting environmental behavior with notable similarity to full-feature predictions. This validates RF’s effectiveness in compressing features without sacrificing accuracy, enhancing computational efficiency.

Figure 3.6 illustrate the effect of feature selection and bootstrapping on test MSE. Omitting either increases MSE by 20–30%, underscoring their combined benefit. By focusing on a small, influential feature subset, the final FCN models improve prediction accuracy and remain computationally light, aligning with real-time and power constraints in multi-core embedded systems. This synergy of reduced feature sets, resampling, and specialized architectures (e.g., FCN+RF+BS) proves effective across the BOTS workloads and platforms tested, surpassing simpler filter-based methods and establishing the framework’s novelty and practical value.

Overall, the advanced feature selection (filter, wrapper, and embedded) combined with neural network models enhances energy and temperature prediction. Multi-platform results and comparisons to SOTA validate the approach’s effectiveness, rigor, and applicability to

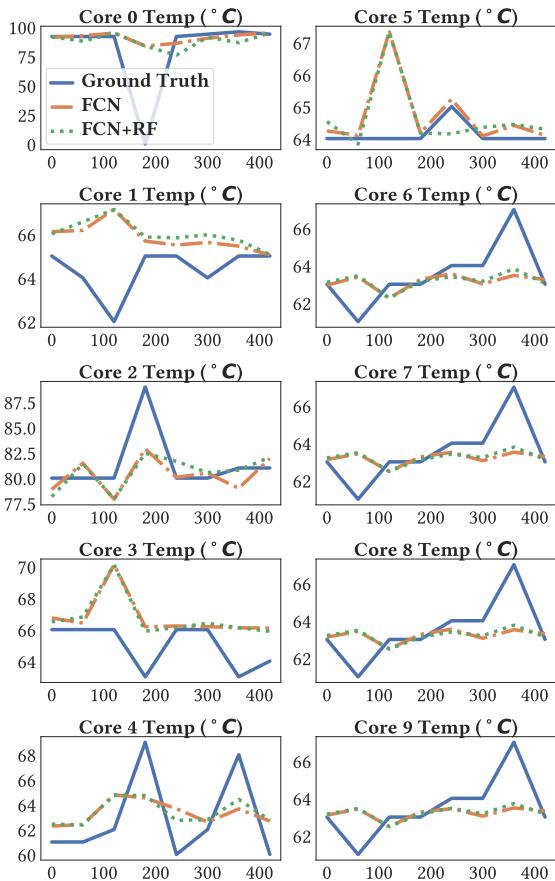


Figure 3.7: Results for temperature prediction from ground truth on Intel Core i7 12th Gen for regular FCN and FCN via Random Forest feature reduction strategy.

diverse multi-core systems.

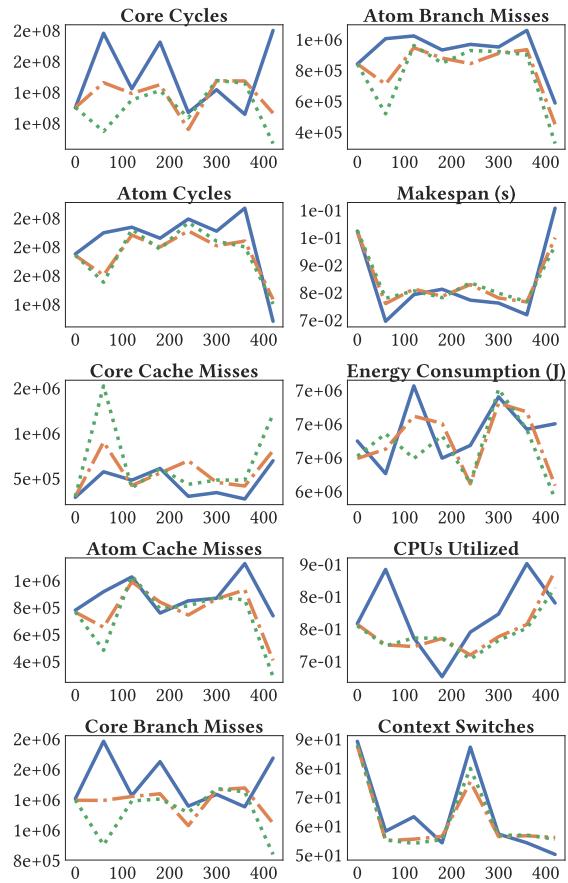


Figure 3.8: Results for profiler data prediction from ground truth on Intel Core i7 12th Gen for regular FCN and FCN via Random Forest feature reduction strategy.

3.5 Related Work

Probabilistic Methods for DVFS and Task-to-Core Allocation Probabilistic worst-case execution time (pWCET) and worst-case energy consumption (pWCEC) estimation in embedded real-time systems leverage measurement-based and static methods [24, 37, 108, 95]. Pallister et al. [95] studied instruction-specific impacts on pWCEC, and extreme value theorem (EVT) [42] provided upper bounds for performance metrics. However, these approaches rarely quantify the statistical significance of system metrics for energy or latency bounds. Our work introduces a hybrid statistical learning framework, systematically prioritizing feature importance to optimize energy-efficient DVFS and task-to-core allocation across diverse platforms, surpassing traditional probabilistic bounds.

Statistical Learning Statistical learning has been employed to pinpoint features for energy optimization and scheduling, often using hardware events or application traits [112, 24, 82]. Sasaki et al. [112] applied decision trees to streamline DVFS table lookups, while Cazorla et al. [24] used hardware counters for energy reduction. Liu et al. [82] correlated compiler features with latency. Yet, these efforts underexplored runtime metrics and sampling strategies critical for accuracy. We enhance this by integrating runtime performance metrics with a novel feature selection pipeline, validated across Intel Xeon and Core i7 platforms, improving parallel scheduling and environment modeling precision.

Low-Energy DVFS and Task-to-Core Allocation Low-energy multicore scheduling via DVFS and task-to-core mapping is well-studied [142, 143, 156, 62, 29, 154, 67, 39, 119,

136]. Xie et al. [142] surveyed heuristic and machine learning methods for energy-constrained scheduling. Many such ML approaches, however, require extensive datasets and computational resources, limiting practicality. Our method mitigates these issues with a lightweight statistical model, reducing data needs while maintaining accuracy, as demonstrated on embedded and heterogeneous systems. Hierarchical multi-agent approaches [101] and graph-driven performance models [98] have also shown promise for DAG workloads, offering scalable alternatives to existing high-overhead techniques.

Few-Shot RL Few-shot learning, including transfer learning and meta-learning, minimizes data demands in scheduling [132, 73, 135, 47, 10]. MAML [46] adapts to new tasks with few samples, and model-based RL [87] approximates dynamics to limit real-world data use. Works like [80, 66, 155, 152] underutilize statistical resampling and feature ranking for energy-efficient scheduling. Recent advances include flow-augmented data generation for few-shot RL [99] and zero-shot LLM-guided resource allocation [100]. Our approach fills this gap, combining resampling with a robust feature selection strategy, validated across platforms, to enhance few-shot task-to-core allocation efficiency.

Feature Evaluation Thermal-aware scheduling models often predict behavior using utilization or temperature data to prevent throttling [146, 22, 84, 56, 121, 76, 64]. Studies like [84, 80, 66] model transients and ambient conditions [55], but lack rigorous statistical analysis of feature correlations. We address this by applying a systematic correlation-based feature evaluation, rigorously tested on real hardware, to uncover interdependencies and optimize

DVFS and allocation, improving energy and performance outcomes over prior heuristic-driven methods.

3.6 Conclusion

We demonstrated the effectiveness of feature selection using statistical learning for environment modeling and task-to-core allocation in embedded systems. Our correlation-aware task-to-core allocation reduces energy consumption by up to 10% and temperature by up to 5°C compared to random core selection. The compressed bootstrapped regression model reduces thermal prediction error by 6% and the number of parameters by 16%. Tested on Intel Core i7 8th and 12th generation, Intel Xeon 2680 processors and Jetson TX2, our method shows a 61.6% reduction in mean squared error compared to state-of-the-art approach. This finding paves the way for future use of statistical learning methods in performance efficiency of task-to-core allocation in heterogeneous processors.

**CHAPTER 4 FLOWRL: FLOW-AUGMENTED FEW-SHOT REINFORCEMENT
LEARNING FOR SEMI-STRUCTURED SENSOR DATA**

Abstract

Reinforcement learning (RL) in few-shot scenarios with limited sensor data is challenging due to insufficient training samples, particularly in applications like Dynamic Voltage and Frequency Scaling (DVFS) where sensor readings are semi-structured with inherent correlations. We propose Flow-Augmented Reinforcement Learning (FlowRL), a novel method that leverages continuous normalizing flows to generate high-quality synthetic data for few-shot RL. By integrating latent space bootstrapping for diversity and feature-weighted flow matching to preserve critical data correlations, FlowRL enhances sample efficiency and policy robustness. Evaluated on a DVFS case study using the NVIDIA Jetson TX2, our approach achieves up to 35% higher frame rates and faster Q-value convergence compared to baselines, demonstrating its effectiveness in resource-constrained environments. FlowRL generalizes to other semi-structured domains, such as robotics and smart grids, offering a scalable solution for data-scarce RL settings.

4.1 Introduction

The rapid advancements in large-scale generative models phrases have transformed data generation, particularly in unstructured domains like computer vision for generating images and videos. Techniques such as diffusion models [105, 109, 35] have shown remarkable success in producing high-quality data, sparking interest in leveraging their underlying latent spaces for various applications. However, generating realistic and diverse semi-structured sensor data, which exhibits correlations and noise but lacks predefined relational structures, remains underexplored. This challenge is critical in few-shot reinforcement learning (RL), where models must generalize from limited examples. Addressing this is essential for advancing online RL in environments with semi-structured sensor data, such as in Dynamic Voltage and Frequency Scaling (DVFS), a power management technique that adjusts processor frequency and voltage to balance performance and energy efficiency using sensor readings like CPU frequency, temperature, power consumption, and frames per second (FPS). Complementary approaches include hierarchical multi-agent DVFS scheduling [101], zero-shot LLM-guided allocation [100], statistical feature-aware task allocation [102], and graph-driven performance modeling [98].

In this paper, we introduce Flow-Augmented Reinforcement Learning (FlowRL), a novel approach designed to generate synthetic semi-structured sensor data tailored for few-shot RL. Unlike traditional data generation methods that require extensive datasets or rely on environment simulations, FlowRL creates diverse and realistic data samples from limited real-world trajectories. We achieve this through flow matching [81], a powerful method that enables

simulation-free training of continuous normalizing flows (CNFs), improving the efficiency and quality of generated data compared to standard diffusion models. By integrating statistical techniques like bootstrapping [43] into the flow matching process, we enhance latent space diversity, enabling better generalization across scenarios. Additionally, we use feature selection with Random Forests to prioritize critical data aspects, ensuring synthetic data captures essential correlations needed for effective RL training. While we draw inspiration from the Law of Large Numbers [53] to guide distribution approximation, non-i.i.d. synthetic data limits strict convergence, so we rely on empirical validation to ensure practical improvements in policy performance.

FlowRL addresses the limitations of conventional model-based RL methods, which often struggle with overfitting and feature correlation when applied to semi-structured sensor data. We demonstrate that conventional methods increase feature correlation in synthetic data, reducing resemblance to real scenarios and limiting RL agent robustness in edge cases. By generating synthetic data that closely mirrors real-world sensor distributions, FlowRL enhances the adaptability and performance of RL models in few-shot settings. This is particularly valuable in dynamic, resource-constrained environments like DVFS, where real-time adaptability is critical. Through extensive experimentation on the NVIDIA Jetson TX2, we validate FlowRL, showing up to 35% higher frame rates and faster Q-value convergence compared to baselines. FlowRL generalizes to other semi-structured domains, such as robotics and smart grids, offering a scalable solution for data-scarce RL applications.

4.2 Preliminaries

This section introduces the core concepts underlying our approach to unstructured data augmentation for RL, leveraging flow matching to generate synthetic data that resembles unstructured data, with an emphasis on sample efficiency. We focus on unstructured data, such as sensor readings in our DVFS case study, which exhibit complex, non-grid-like patterns suitable for adapting methods designed for unstructured data. We cover Q-learning, regret bounds, the Law of Large Numbers (LLN), and the flow matching algorithm.

4.2.1 Q-learning and Online RL

Q-learning aims to learn a function $Q(s, a)$ that estimates the expected *return* (cumulative discounted reward) for taking action a in state s and following an optimal policy thereafter. For continuous states and discrete actions, a **Deep Q-Network (DQN)** approximates $Q(s, a; \mathcal{W})$ with neural network parameters \mathcal{W} . The update rule is: $Q(s_t, a_t; \mathcal{W}) \leftarrow Q(s_t, a_t; \mathcal{W}) + \alpha [r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \mathcal{W}^-) - Q(s_t, a_t; \mathcal{W})]$ where r_t is the immediate reward, $\gamma \in (0, 1)$ is the discount factor, and \mathcal{W}^- represents the parameters of a slower-moving *target network*.

Online Setting. We focus on **online** RL, where the agent continuously interacts with the environment to refine its policy π . At each time-step, it observes (s_t, a_t, r_t, s_{t+1}) and updates Q . In applications like DVFS, frequent decisions and gradual environmental changes allow treating each time-step as a horizon of $\mathcal{H} = 1$, optimizing immediate returns for resource-management tasks. This simplification is common in resource-management tasks where rewards are closely tied to instantaneous performance and power usage.

While Q-learning converges in tabular, finite-state settings under mild conditions, convergence in high-dimensional or partially observed environments is less assured. Using deep networks as function approximators can lead to instability or overfitting if sample diversity is limited.

4.2.2 Regret Bounds and Sample Efficiency

In RL, the **regret** after T steps is defined as:

$$\text{Reg}(T) = \sum_{t=1}^T [Q^*(s_t, a_t) - Q(s_t, a_t)],$$

representing the performance gap between the agent’s policy, using $Q(s, a)$, and the optimal policy, using $Q^*(s, a)$. Sublinear regret, such as $O(\sqrt{T})$, is achievable in *finite-state*, well-explored environments [63], indicating that per-step regret diminishes over time.

In continuous, high-dimensional settings like our DVFS case study, achieving theoretical regret bounds is challenging. If the function approximator or model class is misspecified (i.e., the true environment is outside the assumed class), sublinear bounds may not hold. In practice, we rely on empirical performance rather than strict theoretical guarantees.

Model Realizability and LLN. Sublinear regret often depends on **realizability**—the assumption that the true dynamics or Q^* function lies within the agent’s hypothesis space. In practice, limited exploration (e.g., rarely sampled CPU frequencies in DVFS) or complex environments can hinder the ability of neural networks to approximate the true dynamics, slowing or preventing convergence to near-optimal solutions. The *Law of Large Numbers (LLN)* en-

sures that, with sufficiently *diverse* and *representative* data, empirical estimates (e.g., Q_T or transition model P_T) converge to their true values as $T \rightarrow \infty$. However, without effective exploration, convergence may remain incomplete.

4.2.3 Flow Matching for Generative Modeling

Flow matching [81] is a generative method that models a continuous flow of probability density from a simple base distribution (e.g., $\mathcal{N}(0, I)$) at $t = 0$ to a target distribution at $t = 1$. The distribution evolves via:

$$\frac{dx_t}{dt} = w(x_t, t),$$

where $w(\cdot, \cdot)$ is a time-dependent velocity field for sample vector x_t at time t . This field is approximated by a parametric network v_W , trained to minimize:

$$\min_W \mathbb{E}_{x_t \sim p_t} \left[\|v_W(x_t, t) - w(x_t, t)\|^2 \right]. \quad (4.1)$$

Relevance to DVFS RL. We employ flow matching to generate synthetic (state, action) tuples that capture the variability of unstructured data, such as sensor readings in our DVFS case study, using limited real data. By integrating Random Forest-based feature weighting, we ensure synthetic data closely resembles unstructured data, enhancing its suitability for RL training. The convex training objective of flow matching improves sample efficiency, and our feature weighting approach produces robust synthetic data for RL when real-world samples are sparse.

4.3 Design of Distribution-Aware Flow Matching for Data Generation

In this section, we propose a *distribution-aware* flow matching approach to generate synthetic unstructured data for online RL training with few shots, enhancing sample efficiency for applications like DVFS where data exhibits complex, semi-structured patterns.

4.3.1 Redefinition of the Regret Function

To evaluate the impact of synthetic data in RL, we redefine the cumulative regret function to reflect the estimation error in the Bellman operator due to combined real and synthetic data, inspired by the model-free RL framework. Let \mathcal{M}^* denote the true Markov Decision Process (MDP), with optimal Q-function Q^* . Let π be the policy learned using real data \mathcal{D}_R (size n) and synthetic data \mathcal{D}_S (size m) from flow matching. The cumulative regret up to time T is defined as:

$$\text{Reg}(T) = \sum_{t=1}^T [Q^*(s_t, a_t) - Q^\pi(s_t, a_t)] + O(\sigma \sqrt{\frac{\log(1/\delta)}{n+m}}),$$

where Q^π is the Q-function under policy π , σ^2 bounds the variance of rewards and Q-values ($\text{Var}[r + \gamma \max Q] \leq \sigma^2$), δ is the confidence parameter, and the second term captures the estimation error of the empirical Bellman operator due to finite real and synthetic samples. This redefinition emphasizes the role of synthetic data in reducing variance in policy updates, consistent with the model-free regret bound.

4.3.2 Flow Matching and Bootstrapping

Our goal is to augment limited real trajectories, such as DVFS sensor data, with synthetic samples that approximate the unstructured data distribution, improving RL training efficiency.

While synthetic samples may not be strictly i.i.d., flow matching empirically approximates real-world distributions, enhancing state-action coverage and policy performance.

Flow matching models the evolution of data sample probability density over a continuous-time parameter $t \in [0, 1]$. Unlike model-based RL, which predicts next states s_{t+1} given current states s_t and actions a_t using a learned environment model $\widehat{P}(s_{t+1}|s_t, a_t)$, flow matching generates synthetic tuples (s_t, a_t, s_{t+1}) directly from the joint distribution $p(s_t, a_t, s_{t+1})$. In its *unconditional* form, the model learns a velocity field $v_W(x_t)$ to match a target velocity field $w(x_t)$ [125], where x_t represents the tuple (s_t, a_t, s_{t+1}) at time t . Equation 4.1 represents the baseline objective.

Bootstrapping in the Latent Space. To enhance diversity and mitigate overfitting, we introduce *latent space bootstrapping*, inspired by [43]. For a sample $x_0 \sim p_0(x)$ from the base latent distribution, representing an initial tuple (s_0, a_0, s_1) , we generate B bootstrapped samples $\{x_0^b\}_{b=1}^B$ via resampling:

$$x_0^b = \text{Resample}(x_0), \quad b = 1, \dots, B.$$

Let x_t^b denote the sample evolved from x_0^b at time t , with distribution $p_t^b(x)$. The bootstrapped flow matching objective extends Equation 4.1:

$$\min_w \frac{1}{B} \sum_{b=1}^B \mathbb{E}_{\substack{t \in [0, 1] \\ x_t^b \sim p_t^b(x)}} \left[\|v_W(x_t^b) - w(x_t^b)\|^2 \right]. \quad (4.2)$$

This objective promotes diverse synthetic trajectories by leveraging resampled latent distributions, ensuring broader coverage of the joint state-action-next-state space.

4.3.3 Feature Weighting and Conditional Flow Matching

Bootstrapping adds diversity, but *feature weighting* ensures synthetic data resembles unstructured data for RL tasks, as required in our DVFS case study.

Random Forest Feature Importance. We employ a Random Forest regressor or classifier to assess how each real-world feature (e.g., sensor measurements in DVFS) contributes to predicting next-state transitions. Let d be the dimension of our state/action/next-state representation. The feature importance at time t for dimension (feature) i is $\text{Importance}(x_{t,i})$, which we normalize:

$$\lambda_i = \frac{\text{Importance}(x_{t,i})}{\sum_{j=1}^d \text{Importance}(x_{t,j})}, \quad i = 1, \dots, d. \quad (4.3)$$

In practice, the importance of each dimension (feature) is calculated based on how much the feature reduces impurity in classification or variance in regression tasks. The *Importance* may be updated periodically or assumed to be roughly constant if the importance ranks are stable. Features with large λ_i (e.g., temperature) are emphasized during training, preserving critical data relationships in synthetic samples.

Weighted Conditional Objective. We incorporate these weights into a *conditional* flow matching setup, where additional context z (e.g., rewards, auxiliary variables) follows $z \sim q(z)$. Define $x_{t,i}^b$ as the i -th feature of the bootstrapped sample at time t . The conditional flow match-

ing objective with feature weighting extends (4.2):

$$\min_{\mathcal{W}} \frac{1}{B} \sum_{b=1}^B \mathbb{E}_{\substack{t \in [0,1] \\ x_t^b \sim p_t^b(x) \\ z \sim q(z)}} \left[\sum_{i=1}^d \lambda_i \|v_{\mathcal{W}}(x_{t,i}^b; z) - w(x_{t,i}^b; z)\|^2 \right]. \quad (4.4)$$

This ensures high-importance features (e.g., temperature, resource usage) are prioritized, preserving essential correlations in the joint distribution $p(s_t, a_t, s_{t+1})$ for robust synthetic data generation in unstructured settings like DVFS.

4.3.4 Implications for RL: Regret Bounds under Model-Based Assumptions

We theoretically motivate the use of synthetic data in a **model-based** RL framework. Let \mathcal{M}^* be the true MDP that governs the DVFS transitions, and let $\widehat{\mathcal{M}}$ be a learned MDP model derived from real and synthetic data. In model-based RL, the environment model $\widehat{\mathcal{M}}$ predicts next states s_{t+1} given states s_t and actions a_t via $\widehat{P}(s_{t+1}|s_t, a_t)$, unlike flow matching, which generates full tuples (s_t, a_t, s_{t+1}) . Synthetic tuples from flow matching are used to train $\widehat{\mathcal{M}}$, enhancing its transition dynamics estimation. The redefined regret function, $\text{Reg}(T) = \sum_{t=1}^T [Q^*(s_t, a_t) - Q^\pi(s_t, a_t)] + O\left(\sigma \sqrt{\frac{\log(1/\delta)}{n+m}}\right)$, accounts for policy suboptimality and estimation error due to finite samples.

Enhanced Model Realizability. Synthetic data augments real data to improve the estimation of transition dynamics. The empirical Bellman operator for $\widehat{\mathcal{M}}$ is estimated using \mathcal{D}_R and \mathcal{D}_S :

$$\widehat{\mathcal{T}}Q(s, a) = \frac{1}{n+m} \left(\sum_{(s', r) \in \mathcal{D}_R} [r + \gamma \max_{a'} Q(s', a')] \right)$$

$$+ \sum_{(s'', r) \in \mathcal{D}_S} [r + \gamma \max_{a'} Q(s'', a')]).$$

Under bounded variance ($\text{Var}[r + \gamma \max Q] \leq \sigma^2$), a concentration bound gives, with probability $1 - \delta$:

$$\|\widehat{\mathcal{T}}Q - \mathcal{T}Q\| \leq O\left(\sigma \sqrt{\frac{\log(1/\delta)}{n+m}}\right).$$

This matches the regret second term, indicating that synthetic data reduces estimation error by increasing the effective sample size, enabling sublinear regret $\text{Reg}(T) = O(\sqrt{T})$ under standard assumptions (finite horizon, bounded rewards) [45].

Illustrative Sketch. A simplified argument is:

1. Each (s, a) pair is sampled with probability $\mu > 0$ across real and synthetic data.
2. The learned model $\widehat{\mathcal{M}}$ satisfies the above concentration bound.
3. By standard analyses [63, 45], regret satisfies:

$$\text{Reg}(T) = O(\sqrt{T}) + O\left(\sigma \sqrt{\frac{\log(1/\delta)}{n+m}}\right).$$

The synthetic data contribution to m ensures sublinear regret with reduced sample complexity.

Practical Caveats. In high-dimensional settings like DVFS, exploration is constrained, and real-synthetic mismatch may arise if the flow model fails to capture corner cases. Feature weighting mitigates this by ensuring synthetic tuples resemble unstructured data, aligning with the redefined regret.

4.3.5 Implications for RL: Regret Bounds with Synthetic Data in Model-Free RL

In a model-free RL framework, synthetic data generated by flow matching augments real data to improve sample efficiency and reduce variance in Q-value estimates, leading to tighter regret bounds. Unlike model-based RL, which uses a learned model $\widehat{P}(s_{t+1}|s_t, a_t)$ to predict next states, flow matching directly generates synthetic tuples (s_t, a_t, s_{t+1}) from the joint distribution $p(s_t, a_t, s_{t+1})$, which are stored in \mathcal{D}_S for Q-learning updates.

Role of Synthetic Data via Flow Matching. Flow matching enhances the regret bound by generating synthetic tuples (s_t, a_t, s_{t+1}) that accurately capture the joint distribution $p(s_t, a_t, s_{t+1})$, expanding coverage of state-action-next-state combinations infrequently seen in real data. The bootstrapped objective (Equation 4.2) ensures diversity, while the feature-weighted conditional objective (Equation 4.4) preserves critical correlations (e.g., temperature and frequency in DVFS). This reduces the variance term $O\left(\sigma \sqrt{\frac{\log(1/\delta)}{n+m}}\right)$ by increasing m , outperforming real-data-only training (where the term is $O\left(\sigma \sqrt{\frac{\log(1/\delta)}{n}}\right)$). By modeling continuous-time density evolution, flow matching minimizes distributional bias, ensuring synthetic tuples align with true dynamics, accelerating Q-value convergence and achieving tighter regret bounds.

Bias arises if synthetic tuples deviate from the true distribution. Feature weighting ensures fidelity to unstructured data distributions, keeping bias minimal, making flow matching-augmented model-free RL effective for few-shot settings, as validated in our DVFS experiments.

4.3.6 From LLN to Finite-Sample Guarantees

Under mild mixing conditions, synthetic samples reduce the variance of empirical Bellman updates, yielding finite-sample bounds for both RL paradigms.

Let \mathcal{D}_R be the n real transitions and \mathcal{D}_S the m synthetic ones. The empirical backup is:

$$\begin{aligned}\widehat{\mathcal{T}}Q(s, a) = & \frac{1}{n+m} \left(\sum_{(s', r) \in \mathcal{D}_R} [r + \gamma \max_{a'} Q(s', a')] \right. \\ & \left. + \sum_{(s'', r) \in \mathcal{D}_S} [r + \gamma \max_{a'} Q(s'', a')] \right).\end{aligned}$$

With $\text{Var}[r + \gamma \max Q] \leq \sigma^2$, a Bernstein-type bound gives, with probability $1 - \delta$:

$$|\widehat{\mathcal{T}}Q(s, a) - \mathcal{T}Q(s, a)| \leq O\left(\sigma \sqrt{\frac{\log(1/\delta)}{n+m}}\right) + O\left(\frac{\log(1/\delta)}{n+m}\right).$$

Adding m synthetic samples shrinks the variance term, accelerating convergence of Q-value or model estimates.

4.3.7 Algorithmic Overview for Few-Shot Online RL

We integrate flow matching with a Deep Q-Network (DQN) using two replay buffers: $\mathcal{D}_{\text{real}}$ for real tuples (s_t, a_t, s_{t+1}) collected from the environment, and \mathcal{D}_{syn} for synthetic tuples generated via flow matching.

In each iteration, we collect real tuples (s_t, a_t, s_{t+1}) from the environment, storing them in $\mathcal{D}_{\text{real}}$. When sufficient new data are available, we retrain or fine-tune the Flow-Matching model (FM) using the conditional weighted objective (Equation 4.4) or the unconditional boot-

Algorithm 2 Distribution-Aware Flow Matching for Few-Shot Online RL.

```

1: Initialize:  $\mathcal{D}_{\text{real}}, \mathcal{D}_{\text{syn}}, \text{DQN}, FM$ 
2: for  $i = 1$  to  $T$  do
3:   Collect real tuple  $\tau_{\text{real}} = (s_t, a_t, s_{t+1})$  and store in  $\mathcal{D}_{\text{real}}$ 
4:   if enough new data in  $\mathcal{D}_{\text{real}}$  then
5:     Train  $FM$  using Equation (4.4)
6:   end if
7:   for  $j = 1$  to rollout count do
8:      $x_0^b = \text{Resample}(x_0)$                                  $\triangleright$  bootstrap tuple  $(s_0, a_0, s_1)$ 
9:     Evolve  $x_0^b$  to  $x_t^b$  via  $FM$                                  $\triangleright$  generate tuple  $(s_t, a_t, s_{t+1})$ 
10:    Store synthetic tuple  $\tau_{\text{syn}} = (s_t, a_t, s_{t+1})$  in  $\mathcal{D}_{\text{syn}}$ 
11:   end for
12:   if enough data in  $\mathcal{D}_{\text{real}}$  and  $\mathcal{D}_{\text{syn}}$  then
13:     Sample mini-batches and update DQN
14:   end if
15: end for

```

strapped objective (Equation 4.2). We generate synthetic tuples by resampling bootstrapped latent points $\{x_0^b\}$, each representing an initial tuple (s_0, a_0, s_1) , evolving them over $t \in [0, 1]$ via FM to produce tuples (s_t, a_t, s_{t+1}) , and storing these in \mathcal{D}_{syn} . Mini-batches are sampled from both $\mathcal{D}_{\text{real}}$ and \mathcal{D}_{syn} to update the DQN agent’s Q-function using the Q-learning update rule. This approach ensures robust state-action-next-state coverage for unstructured data, with bootstrapping enhancing diversity and feature weighting preserving critical correlations, supporting effective training for both model-based and model-free RL in few-shot settings, as validated in our DVFS case study.

4.4 Experiments

In this section, we present the experimental setup, hyperparameter tuning, model architecture, and comparisons with model-based and model-free baselines to evaluate our Flow-Augmented Reinforcement Learning (FlowRL) approach.

Experimental Setup We developed a Deep Q-Network (DQN) agent based on the zTT framework [66] to optimize system performance under thermal and power constraints on the

NVIDIA Jetson TX2, a power-efficient embedded platform with six heterogeneous CPU cores and an embedded GPU. The goal is to dynamically adjust frequency and power settings to maximize performance while ensuring system stability. The setup uses a client-server architecture: the client collects semi-structured state tuples $\{FPS, f, \rho, \theta\}$ (frames per second, core frequency, power consumption, and core temperature), normalized via min-max scaling, and the server assigns frequency actions from a discrete set defined by hardware constraints (e.g., GPU_CLOCK_LIST). Each episode comprises 10 tuples collected at 1-second intervals over a 90-second horizon (90 decision points), with thermal monitoring triggering a 5–10-second sleep mode if temperature exceeds 50°C to mitigate throttling. The workflow is depicted in Figure 4.1.

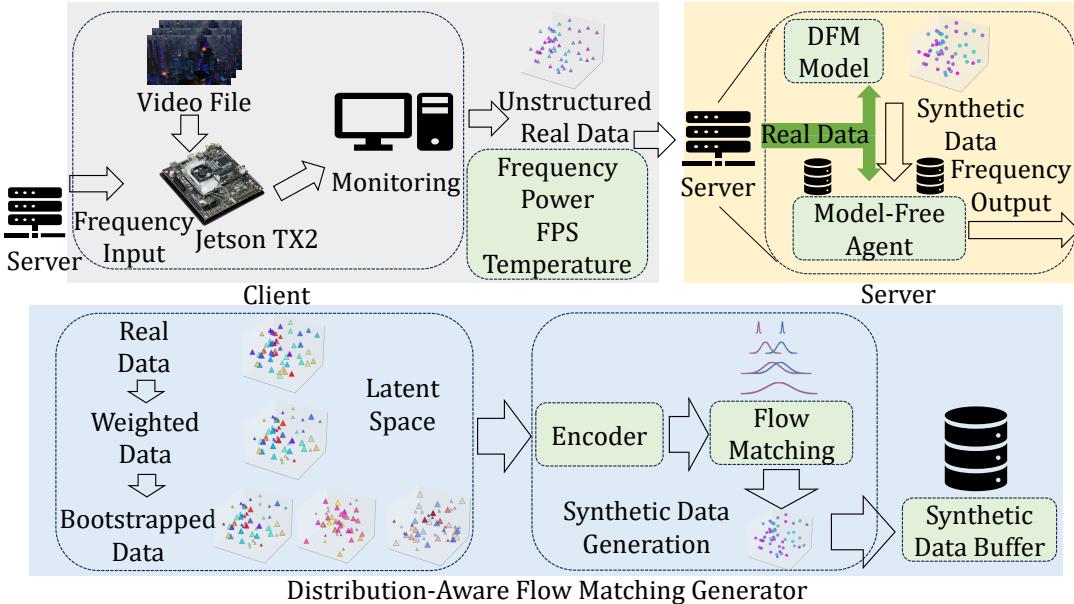


Figure 4.1: Workflow of the proposed Flow-Augmented Reinforcement Learning (FlowRL) approach.

Hyperparameter Tuning. We performed hyperparameter tuning using grid search, selecting values based on reward stability and convergence speed, as shown in Table 4.1. The learning rate (0.05) was chosen after grid search and adjusted dynamically (e.g., reset to 1 under specific reward conditions), though it may seem high compared to typical deep learning ranges (e.g., 1e-4 to 5e-5). Future work will explore random search (e.g., Hyperopt) for refinement.

Hyperparameter	Tuning Values
Experiment Time	{50, 90 , 200, 400, 800}
Target FPS	{30, 60, 90}
Target Temperature	{50, 60}
Learning Rate	{0.01, 0.05 , 0.1}
Discount Factor	{0.95, 0.99 }
Epsilon Decay	{0.95, 0.99 }
Batch Size	{32, 64, 128}
Agent Train Start	{32, 40, 80, 100}
Planning Iterations	{100, 200, 1000}
Model Train Start	{32, 200, 300}
Reward Scale (β)	{0, 2, 4}

Table 4.1: Main Hyperparameters and Tuning Values.

Model Architecture and Training. The DQN agent features a neural network with two hidden layers of six neurons each, optimized using mean squared error (MSE) loss and the Adam optimizer. An ϵ -greedy strategy starts with $\epsilon = 1.0$ and decays over time to promote exploitation. Training occurs in discrete intervals: Agent Train Start initiates Q-network updates, and Planning Iterations controls MDP model updates, using mini-batches of real and synthetic data. Performance metrics include Frames Per Second (FPS) for responsiveness, power consumption (ρ) for efficiency, average temperature (θ) for safety, and average maximum Q-values for policy convergence, with emphasis on FPS and β . The reward function, adapted from [66], is:

$$R = u + v + \frac{\beta}{\rho},$$

where

$$u = \begin{cases} 1 & \text{if } FPS \geq \text{target_FPS}, \\ \frac{FPS}{\text{target_FPS}} & \text{otherwise,} \end{cases}$$

and

$$v = \begin{cases} 0.2 \cdot \tanh(\text{target_temp} - \theta) & \text{if } \theta < \text{target_temp}, \\ -2 & \text{if } \theta \geq \text{target_temp}. \end{cases}$$

With target FPS at 30 and target temperature at 50°C, this design balances performance and constraints.

Model-Based and Model-Free Baselines. We compare FlowRL with DynaQ [97], a model-based RL method; PlanGAN [25], a generative model-based approach with dual-memory; MAML [46], a meta-learning method; and zTT [66], a model-free DQN baseline for DVFS. PlanGAN and DynaQ use real and synthetic data, while zTT runs directly on the Jetson TX2 without synthetic data. Consistent hyperparameters and reward functions ensure fair comparisons, with thermal throttling mitigated by sleep mode.

Results and Analysis. In our experiments, we train a *conditional flow matching* model [125] on the real trajectories stored in a replay buffer, then use it to generate synthetic trajectories. We refer to this baseline generative approach as “standard flow matching.” We further refine it into a *Flow-Augmented Reinforcement Learning (FlowRL)* method by incorporating latent-space bootstrapping and feature weighting to preserve important correlations and diversify

the latent representation. We evaluate how well each synthetic approach captures correlations among features by computing Pearson’s correlation coefficient [114]. For two features x_i and x_j with n samples, the coefficient is given by

$$\text{corr}_{ij} = \frac{\sum_{k=1}^n (x_{i,k} - \bar{x}_i)(x_{j,k} - \bar{x}_j)}{\sqrt{\sum_{k=1}^n (x_{i,k} - \bar{x}_i)^2} \sqrt{\sum_{k=1}^n (x_{j,k} - \bar{x}_j)^2}}. \quad (4.5)$$

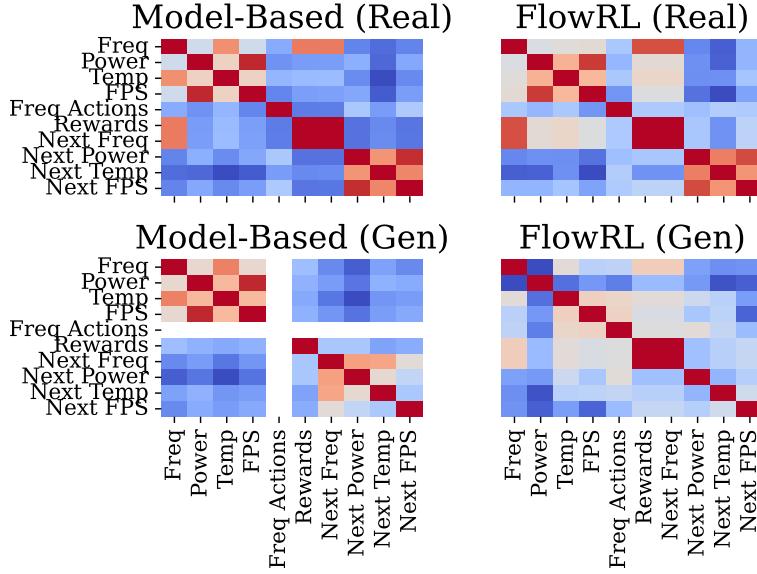


Figure 4.2: Correlation matrix comparisons.

When comparing the correlation matrices, we find that FlowRL-generated data better captures correlations among features visually compared to the model-based approach. Although the standard flow matching approach does capture some correlations, its matrix appears more uniform and less differentiated than that of FlowRL, suggesting that some critical feature relationships are weakened or diluted. Meanwhile, the model-based method shows stronger but sometimes misplaced correlations (dark regions in incorrect locations), indicating potential

overfitting to specific patterns and misrepresenting genuine dependencies. FlowRL balances the two extremes by retaining essential correlations without artificially magnifying or suppressing them.

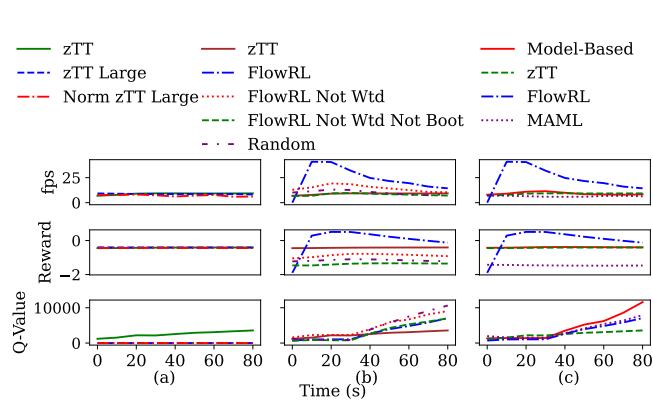


Figure 4.3: Performance comparisons across methods.

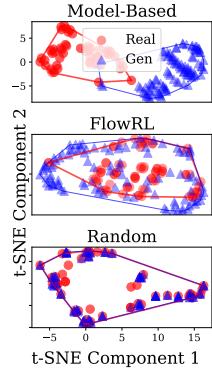


Figure 4.4: Real and (Gen)erated data t-SNE.

To further analyze the data, we use t-SNE visualizations to compare the distributions in Figure 4.4.

The t-SNE comparison of model-based and FlowRL with weighted and bootstrapped data and random data generation shows that FlowRL covers real data and more, while random sampling is restricted to the real data distribution and the model-based approach does not cover real data. This indicates FlowRL's superior ability to generalize beyond the original dataset. We also perform a Dynamic Time Warping (DTW) evaluation to assess temporal alignment. The DTW evaluation reveals that model-based data generation has a higher value (10,640) compared to FlowRL weighted and bootstrapped (7,341) between generated and real data, confirming FlowRL's better temporal alignment.

To examine distributional coverage, we compare the histograms of key state-action features under different synthetic data generation methods. The results indicate that the FlowRL

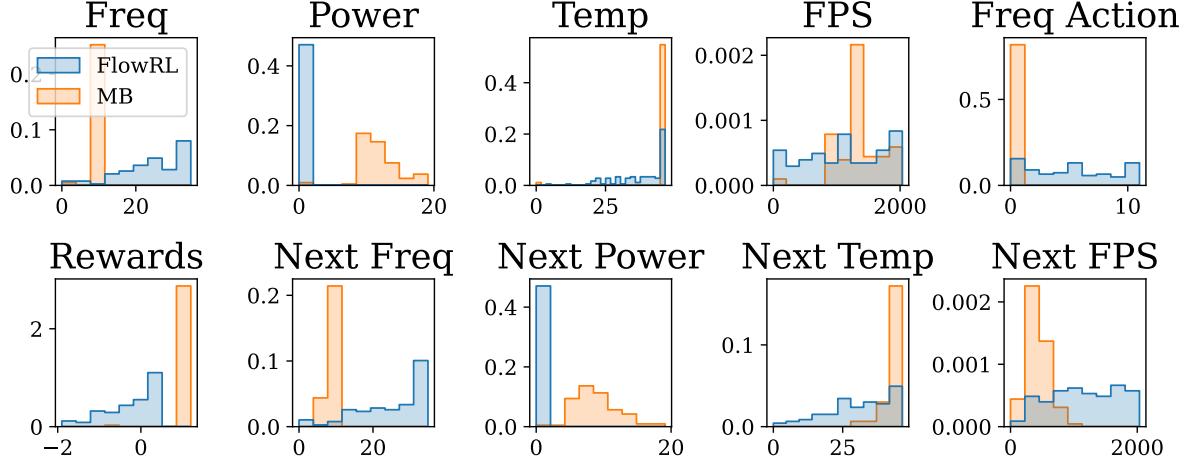


Figure 4.5: Comparisons of distributions of FlowRL and Model-Based (MB) generated data.

approach covers a wider spectrum of values compared to the model-based approach. In contrast, the model-based approach tends to be narrowly concentrated, particularly in power states and next states, suggesting it may overfit and fail to generalize to a broader range of real-world scenarios. The standard flow matching method similarly suffers from less coverage than FlowRL but does not exhibit the same extreme concentrations as the purely model-based approach. Overall, FlowRL’s expanded coverage helps the RL agent explore and learn policies that are more robust to varied conditions.

Performance Comparisons. To comprehensively evaluate FlowRL, three subplots were generated to compare performance metrics (average FPS, average reward, and average maximum Q-value) across different methods in 90 seconds, with the x-axis labeled in seconds (each

second representing one iteration result based on FPS), as shown in Figure 4.3. This figure includes a comprehensive ablation analysis of different implementations.

The left column of subplot (a) compares different model-free approaches based on zTT, comprising zTT, zTT with extended features from profiling data, and zTT with normalization, showing no specific differences in terms of average reward, Q-values, and frame rate, with stable FPS around 10. The middle figure (b) compares zTT with FlowRL, FlowRL without weighted but with bootstrapping (FM Not Wtd), FlowRL without augmentation, and random generated synthetic data augmented models, showing faster convergence in the FlowRL weighted bootstrapped version compared to all other approaches. The right column (c) extends the comparison to MAML, zTT, model-based, and FlowRL, where FlowRL outweighs all other approaches.

The above experiments confirm that incorporating bootstrapping and feature weighting into flow matching (i.e., FlowRL) improves both correlation preservation and distribution diversity in synthetic data generation. This yields more stable and high-performing RL policies compared to standard flow matching or purely model-based methods. The ability to balance performance and power highlights the potential of Flow-Augmented Reinforcement Learning for real-world DVFS tasks, where state-action spaces can be both large and only partially observed in limited data collection.

4.5 Related Work

Machine Learning for DVFS. Machine learning has been widely adopted to optimize Dynamic Voltage and Frequency Scaling (DVFS) strategies, addressing the complexity of modern

processors with adaptive control [148, 67, 39, 158, 94, 119, 136, 147, 82, 116, 128, 133, 17]. A survey by [94] notes that model-free reinforcement learning (RL), such as the zTT framework [66], dominates, focusing on direct policy optimization for power-performance trade-offs. However, these methods often overlook feature correlation intricacies and the computational overhead of extensive data collection, particularly in data-scarce settings. Hierarchical multi-agent approaches [101] and graph-driven performance models [98] have shown promise for DAG workloads. Our work builds on this by introducing FlowRL to enhance sample efficiency for semi-structured sensor data in DVFS.

Statistical Learning for DVFS. Statistical learning techniques have been explored to evaluate hardware events and application parameters impacting DVFS performance [112, 24, 82]. For example, [112] uses decision trees for energy-efficient DVFS lookup optimization, while [24] assesses hardware counter importance for power reduction. [82] applies f-score-based ranking to correlate compiler-level features with latency. These efforts, however, lack systematic consideration of runtime performance variability, sampling inefficiencies, and feature correlation effects on learning accuracy. FlowRL addresses these gaps by integrating feature weighting with Random Forests and data augmentation tailored for low-energy DVFS applications.

Bounding Sample Efficiency. Sample efficiency bounds for Q-learning have been a focus in RL research [48, 11, 45, 63]. Sublinear bounds have been established for finite state-action spaces [11, 63], but extending these to continuous spaces requires robust function approxima-

tors. Approaches like [45] rely on strong statistical assumptions, limiting real-world applicability. Deep Q-learning with stabilization techniques (e.g., experience replay, target networks) often faces instability [48], with few theoretical guarantees. Unlike these, FlowRL empirically validates sample efficiency for continuous semi-structured data in DVFS, offering practical insights over theoretical bounds.

Few-Shot RL. Few-shot RL methods, including transfer learning, meta-learning, and model-based approaches, aim to reduce data needs and enhance generalization [132, 73, 135, 89, 47, 32, 10, 138, 107, 129]. Model-agnostic meta-learning (MAML) [46] adapts to new tasks with minimal data, while model-based methods [87] approximate transition dynamics. DVFS-specific works like [80, 66, 155, 152] explore multi-agent RL and transfer learning but neglect statistical resampling or feature selection for energy optimization. Recent approaches address these gaps through statistical feature-aware task allocation [102] and zero-shot LLM-guided core allocation [100]. FlowRL advances this by using flow matching and bootstrapping to augment semi-structured sensor data, improving efficiency in DVFS and extending to robotics and smart grids.

4.6 Conclusion

We presented FlowRL, a novel method for few-shot RL with semi-structured sensor data. By leveraging flow matching, latent space bootstrapping, and feature-weighted conditioning, FlowRL generates high-quality synthetic data to enhance sample efficiency and policy robustness. Experiments on a DVFS task demonstrate up to 35% higher frame rates and faster conver-

gence compared to baselines like DQN, DDQN, and MAML. FlowRL’s applicability extends to other domains, such as robotics, offering a scalable solution for data-scarce RL. Future work includes exploring advanced RL algorithms (e.g., PPO) and neural-based feature weighting to further improve performance.

**CHAPTER 5 GRAPHPERF-RT: A GRAPH-DRIVEN PERFORMANCE MODEL
FOR HARDWARE-AWARE SCHEDULING OF OPENMP CODES**

Abstract

Performance prediction for OpenMP workloads on heterogeneous embedded SoCs is challenging due to complex interactions between task DAG structure, control-flow irregularity, cache and branch behavior, and thermal dynamics; classical heuristics struggle under workload irregularity, tabular regressors discard structural information, and model-free RL risks overheating resource-constrained devices. We introduce GraphPerf-RT, the first surrogate that unifies task DAG topology, CFG-derived code semantics, and runtime context (per-core DVFS, thermal state, utilization) in a heterogeneous graph representation with typed edges encoding precedence, placement, and contention. Multi-task evidential heads predict makespan, energy, cache and branch misses, and utilization with calibrated uncertainty (Normal-Inverse-Gamma), enabling risk-aware scheduling that filters low-confidence rollouts. We validate GraphPerf-RT on three embedded ARM platforms (Jetson TX2, Jetson Orin NX, RUBIK Pi), achieving $R^2 > 0.95$ with well-calibrated uncertainty ($ECE < 0.05$). To demonstrate end-to-end scheduling utility, we integrate the surrogate with four RL methods on Jetson TX2: single-agent model-free (SAMFRL), single-agent model-based (SAMBRL), multi-agent model-free (MAMFRL-D3QN), and multi-agent model-based (MAMBRL-D3QN). Experiments across 5 seeds (200 episodes each) show that MAMBRL-D3QN with GraphPerf-RT as the world model achieves 66% makespan reduction (0.97 ± 0.35 s) and 82% energy reduction (0.006 ± 0.005 J)

compared to model-free baselines, demonstrating that accurate, uncertainty-aware surrogates enable effective model-based planning on thermally constrained embedded systems.

5.1 Introduction

Heterogeneous embedded Systems-on-Chip (SoCs) combine high-performance and energy-efficient cores with Dynamic Voltage and Frequency Scaling (DVFS). This architecture has created unprecedented opportunities for deploying parallel applications in resource-constrained environments such as autonomous vehicles, robotics, and edge computing. OpenMP, the dominant shared-memory parallel programming model, enables developers to express task-level parallelism through pragma-based annotations that the runtime system maps to available hardware resources. However, achieving optimal performance on these heterogeneous platforms requires careful scheduling decisions that balance execution time, energy efficiency, and thermal constraints. This presents a challenging optimization problem that current approaches fail to address comprehensively. Performance prediction for parallel workloads has traditionally relied on analytical models, simulation-based methods, and machine learning surrogates. Analytical models grounded in queueing theory and scheduling bounds such as Brent’s theorem provide theoretical insights but require simplifying assumptions that break down under irregular control flow and cache contention [51, 20]. Simulation-based approaches offer high fidelity but incur prohibitive runtime overhead for online scheduling decisions. Machine learning surrogates, including Graph Neural Networks (GNNs), can learn complex input-output mappings from data [130]. However, existing approaches such as ProGraML [36] and Ithermal [86] operate on static intermediate representations without incorporating runtime context, thermal state, or DVFS configurations that critically affect embedded system performance. Complementary approaches address related challenges through hierarchical multi-agent DVFS

scheduling [101], zero-shot LLM-guided allocation [100], statistical feature-aware task allocation [102], and flow-augmented few-shot RL [99].

This paper addresses the problem of predicting multiple performance metrics for OpenMP task-parallel applications on heterogeneous embedded SoCs. These metrics include execution time, energy consumption, cache misses, branch misses, and CPU utilization under varying DVFS configurations, core allocations, and thermal conditions. The prediction model must provide accurate point estimates and *calibrated uncertainty quantification* to enable risk-aware scheduling decisions. This is essential for respecting thermal constraints and avoiding failures in safety-critical deployments. Furthermore, the model should generalize across benchmarks, input sizes, and hardware platforms without extensive retraining.

Accurate performance prediction on heterogeneous embedded SoCs presents four fundamental challenges that existing approaches fail to address simultaneously. **First**, performance emerges from complex cross-layer interactions between application structure (task Directed Acyclic Graphs, or DAGs, and Control Flow Graphs, or CFGs), hardware state (per-core frequencies, thermal headroom, cache occupancy), and scheduling decisions (core masks, priorities); classical heuristics and tabular regressors treat these factors independently, missing critical coupling effects; for example, a memory-intensive task scheduled on a thermally throttled core at reduced frequency exhibits dramatically different behavior than the same task on a cool core at maximum frequency, causing 3–5× variation in execution time that flat feature vectors cannot capture. **Second**, tabular machine learning models flatten task graphs into aggregate statistics (e.g., total work, critical path length), discarding the fine-grained dependency

structure that determines parallelism opportunities and synchronization overhead, while GNN-based approaches preserve structure but focus on static program representations without run-time context integration. **Third**, standard regression models produce point predictions without confidence estimates, and on thermally constrained embedded systems, overconfident predictions can lead to aggressive scheduling decisions that cause thermal throttling, system instability, or hardware damage, and risk-aware scheduling requires calibrated uncertainty bounds that accurately reflect prediction reliability. **Fourth**, model-free Reinforcement Learning (RL) approaches for DVFS control require extensive on-device exploration that risks overheating resource-constrained boards, while model-based RL can reduce sample complexity through synthetic rollouts but requires an accurate environment model.

This paper introduces GraphPerf-RT, a graph-grounded performance surrogate that addresses these challenges through a unified heterogeneous graph representation. This representation explicitly models the three-way interaction between application structure, hardware resources, and scheduling decisions. Task nodes encode CFG-derived code semantics extracted via the ARTIST2 Language for Flow analysis (ALF) and Execution Flow Graphs (EFGs). Resource nodes capture per-core DVFS state, utilization, and thermal headroom. Typed edges encode task-task precedence, task-resource placement, and resource-resource topology. A heterogeneous Graph Attention Network (GAT) architecture with type-specific encoders processes the graph through 3 to 6 attention layers. Evidential learning with Normal-Inverse-Gamma (NIG) distributions provides calibrated uncertainty, producing aleatoric and epistemic estimates in a single forward pass. The surrogate scores candidate configurations in batch, fil-

ters low-confidence proposals, and supports Dyna-style model-based planning where synthetic rollouts augment real-world samples. While this paper focuses on OpenMP on ARM-based embedded SoCs, the framework generalizes to other programming models. For CUDA and GPU workloads, kernels map to task nodes and Streaming Multiprocessors to resource nodes. For x86 systems, P-states replace ARM DVFS indices and Running Average Power Limit (RAPL) counters provide energy measurements. Comprehensive experiments on three embedded ARM platforms (NVIDIA Jetson TX2, Jetson Orin NX, and RUBIK Pi) span 42 benchmarks from BOTS and PolyBench. GraphPerf-RT achieves $R^2 > 0.95$ with well-calibrated uncertainty (Expected Calibration Error below 0.05), significantly outperforming tabular baselines and homogeneous GNN architectures. Integration with four RL methods (SAMFRL, SAMBRL, MAMFRL-D3QN, and MAMBRL-D3QN) demonstrates that MAMBRL-D3QN with GraphPerf-RT achieves 66% execution time reduction (0.97 ± 0.35 s vs. 2.85 ± 1.66 s) and 82% energy reduction (0.006 ± 0.005 J vs. 0.033 ± 0.026 J) compared to model-free baselines across 5 random seeds with 200 episodes each.

The main contributions of this paper are as follows:

1. A *unified heterogeneous graph representation* that jointly encodes OpenMP task DAG topology, CFG-derived code semantics, and runtime context (per-core DVFS, thermal state, utilization) through typed nodes and edges, enabling explicit modeling of cross-layer performance interactions.
2. An *evidential multi-task prediction framework* providing calibrated uncertainty quantification through Normal-Inverse-Gamma heads, producing both aleatoric and epistemic

uncertainty in a single forward pass for risk-aware scheduling on embedded systems.

3. A *practical scheduling integration* demonstrating seamless combination with model-based RL, achieving 66% makespan and 82% energy improvements over model-free baselines while reducing on-device exploration through Dyna-style synthetic rollouts.
4. A *reproducible evaluation framework* including a complete data pipeline (OMPI + ALF-llvm + SWEET + telemetry), comprehensive experiments across three ARM platforms and 42 benchmarks, and statistical significance tests with 5-seed confidence intervals.

The remainder of this paper is organized as follows. Section 5.2 surveys related work on performance modeling, graph neural networks for systems, and uncertainty quantification. Section 5.3 formalizes the problem setup, defines the heterogeneous graph abstraction, and describes the data collection pipeline. Section 5.4 presents the GraphPerf-RT architecture including type-specific encoders, heterogeneous GAT layers, and evidential prediction heads. Section 6.5 reports experimental results on prediction accuracy, uncertainty calibration, and end-to-end RL scheduling evaluation. Section 6.6 concludes with limitations and future directions.

5.2 Related Work

Performance modeling for parallel systems spans analytical queueing/Petri-net models and simulators [41, 85], then statistical and neural surrogates [72, 61, 134, 28]. Black-box predictors (counters, instruction mixes) improve accuracy but lose structural signals and transfer. Learned models like Ithemal [86] use hierarchical LSTMs to predict x86 basic-block through-

put from instruction sequences, achieving high accuracy on microarchitectural latency but operating on static assembly without capturing runtime parallelism, inter-task dependencies, or device/thermal state. Runtime autotuners (Apollo) adapt to input-dependent performance variations [104], yet remain kernel-focused. Our focus differs: we model OpenMP task DAGs on embedded SoCs, explicitly encoding dependencies, per-core DVFS, and thermal context to capture cross-layer effects needed for real-time decisions.

Graph neural networks enable learning on structured program and system graphs [137, 130]. ProGraML [36] constructs a unified graph representation from LLVM IR by combining control-flow, data-flow, and call edges, achieving strong results on compiler optimization tasks such as device mapping and thread coarsening; critically, it operates exclusively on *static* intermediate representation without runtime context, hardware state, or thermal conditions. Similarly, Ithemal [86] predicts microarchitectural latency from static instruction sequences but lacks parallelism structure, inter-task dependencies, or device state. Heterogeneous Graph Transformer (HGT) [59] extends graph attention to multiple node and edge types, providing a natural baseline for our heterogeneous task-resource graphs. However, HGT and similar architectures lack three critical capabilities for our embedded scheduling setting: (1) explicit edge decomposition that separates structural dependencies from execution features, (2) integration of thermal and DVFS context as part of the graph representation, and (3) evidential regression heads that produce calibrated uncertainty estimates for risk-aware decision-making. GNNs have also predicted hardware-dependent performance for ML models [120] and optimization effects [19]. Compiler cost models (Ansor, MetaSchedule, ROLLER) learn evaluators for ten-

sor kernels and improve cross-device transfer [153, 117, 157, 150, 151], but operate below application-level task graphs and rarely encode runtime DVFS/thermal state. Recent work on transferable hardware embeddings includes Silhouette [96], which learns performance-conscious CPU embeddings for cross-platform prediction, while Glimpse [6] develops mathematical encodings of hardware specifications for neural compilation, provides an alternative to hand-crafted device sheets and could potentially enhance resource node features in our framework. However, these approaches focus on static kernel-level prediction without integrating dynamic per-core DVFS, thermal headroom, or application-level task DAG structure that we require for OpenMP scheduling under thermal constraints. In contrast, we construct a heterogeneous graph with task and resource nodes, typed edges with execution features, and CFG-informed encoders to improve zero-shot and few-shot generalization across benchmarks and boards while explicitly fusing runtime device/thermal context into the representation.

Hardware-aware ML includes NAS with device constraints and cross-platform predictors [23], and performance modeling across CPU/heterogeneous configurations [131]. Energy/thermal control spans heuristic/optimization DVFS and RL [142]. Utilization-driven governors can mis-scale frequency by conflating compute and stalls [54, 80]. Model-free RL adapts but is sample- and heat-intensive on small boards [66]. Model-based control reduces sampling via learned dynamics and planning [87]. Recent approaches address these gaps through hierarchical multi-agent DVFS scheduling [101], statistical feature-aware task allocation [102], zero-shot LLM-guided allocation [100], and flow-augmented few-shot RL [99]. Our approach is complementary: a calibrated, graph-grounded evaluator supplies fast, hardware-aware roll-

outs to search/MARL under DVFS and thermal constraints.

Uncertainty is critical for safe scheduling. Bayesian and ensemble methods provide confidence but add inference cost [71, 65]. Evidential learning yields calibrated intervals without sampling [115, 7]. We adopt evidential regression heads to expose confidence on makespan, energy, and counters and to gate synthetic rollouts, aligning with on-board, real-time planning.

To our knowledge, GraphPerf-RT is the *first* unified graph representation that combines OpenMP task DAGs with runtime execution context (per-core DVFS states, thermal headroom, hardware counters) and employs evidential regression for calibrated uncertainty in safety-critical embedded scheduling. This integration enables risk-aware decisions under thermal constraints, critical for resource-constrained SoCs where overheating can trigger throttling or system failure, while maintaining the computational efficiency required for on-board deployment and real-time control.

5.3 Preliminaries

5.3.1 Problem Formulation

This work addresses the problem of predicting multiple performance metrics for OpenMP task-parallel applications executing on heterogeneous embedded SoCs under varying DVFS configurations. The prediction model must provide accurate point estimates along with calibrated uncertainty quantification to enable risk-aware scheduling decisions.

Each OpenMP application is represented as a task DAG $G = (V, E)$ recovered from compiler artifacts and runtime instrumentation. A task $v \in V$ represents a unit of parallel work characterized by static code features extracted from the CFG, including loop counts, memory

access patterns, and branch statistics. Additionally, each task carries optional runtime summaries from prior executions, such as performance counter snapshots and thermal footprints. An edge $e = (u \rightarrow v) \in E$ encodes a precedence relation derived from OpenMP dependencies, classified as spawn, join, or data dependency types.

The target platform consists of C heterogeneous cores indexed by $i \in \{1, \dots, C\}$. Each core operates at discrete frequency levels $f_i \in \mathcal{F}_i$ determined by the DVFS subsystem. The immutable hardware characteristics are captured in the device sheet D , which includes core types and counts, cache hierarchy specifications (sizes, associativity, line sizes), DVFS tables per core cluster, governor support flags, thermal sensor layout and resolution, and toolchain version hashes for reproducibility. The device sheet remains constant throughout an episode and enables transfer learning across hardware generations.

The runtime state S captures dynamic system context that changes during execution. This includes the active core mask indicating which cores are available for scheduling, per-core DVFS indices and measured frequencies, utilization computed as an exponential moving average (EMA), thermal zone temperatures before and after execution, and recent performance counter values. These runtime features enable the model to adapt predictions based on current system conditions rather than relying solely on static analysis.

A scheduling action $a = (m, f, p)$ specifies the configuration under which a workload executes. The core mask $m \in \{0, 1\}^C$ selects active cores, the DVFS vector f assigns frequency levels with $f_i \in \mathcal{F}_i$ for each core, and the optional priority $p \in \{1, \dots, 99\}$ sets FIFO scheduling priority. All actions must satisfy feasibility constraints including core availability, affinity

restrictions, and thermal caps that prevent execution when temperatures exceed safe thresholds.

Given a tuple (G, D, \mathcal{S}, a) , the surrogate model predicts a vector of performance metrics $y = (\text{makespan}, \text{energy}, \text{cache misses}, \text{branch misses}, \text{utilization})$. Lower values indicate better performance for makespan, energy, and miss metrics, while higher utilization is generally preferable. Each prediction carries calibrated uncertainty estimates decomposed into aleatoric uncertainty (irreducible noise from OS and co-runner interference) and epistemic uncertainty (model uncertainty that decreases with more training data).

Formally, we learn a mapping

$$\mathcal{M} : (G, D, \mathcal{S}, a) \mapsto (y, \mathcal{U}),$$

where \mathcal{U} contains the NIG distribution parameters $(\gamma, \nu, \alpha, \beta)$ for each target metric. Point estimates are the predictive means $\hat{y}_k = \gamma_k$, while prediction intervals derive from the NIG posterior. The uncertainty decomposition follows standard evidential learning formulations with aleatoric uncertainty given by $\beta_k/(\alpha_k - 1)$ and epistemic uncertainty by $\beta_k/(\nu_k(\alpha_k - 1))$.

At scheduling time, candidate configurations (m, f) are scored in batch through the surrogate. Actions are ranked primarily by predicted makespan with uncertainty gates that filter out low-confidence proposals. Safe actions satisfying thermal constraints execute on the device, and outcomes (execution time, energy consumption, performance counters, thermal readings) are logged for continual model refinement.

The problem formulation assumes that task graphs are acyclic, the device sheet remains

fixed during an episode, and thermal policies may dynamically shrink the feasible action set. Stochasticity from OS scheduling noise and co-runner interference induces variability in outcomes; the logged data supervises the model under these realistic conditions.

5.3.2 Heterogeneous Graph Abstraction

The key insight underlying our approach is that performance on heterogeneous embedded systems emerges from complex interactions between application structure, hardware resources, and scheduling decisions. A unified heterogeneous graph representation enables explicit modeling of these cross-layer interactions through typed nodes and edges.

The heterogeneous graph contains three node types that capture distinct aspects of the system. Task nodes V_T represent OpenMP tasks and encode application semantics. Each task node carries CFG-derived features capturing control flow complexity, static code statistics including loop counts, bytes moved, and branch proxies, DAG topology metrics such as depth, distance to sink, and centrality measures, recent performance snapshots from prior runs, run mode flags distinguishing serial from parallel execution, and thermal footprint on hosting cores. Resource nodes V_R represent processing cores and encode hardware state. Each resource node captures the DVFS step as both index and one-hot encoding, core mask bit indicating active status, cluster ID for heterogeneous architectures, utilization computed as an exponential moving average, thermal headroom and temperature trend, and bandwidth proxy estimating memory throughput. Memory nodes V_M optionally represent cache hierarchy levels and encode cache level identifier, capacity, associativity and line size, and latency and bandwidth proxies.

Four edge types capture the performance-critical interactions between nodes. Task-task

edges E_{TT} encode precedence constraints from the DAG with attributes including dependency type (spawn, join, or data), critical edge flag indicating membership on the longest path, hop distance, and contention proxies. Task-resource edges E_{TR} connect tasks to cores based on the scheduling assignment under the recorded mask and DVFS configuration, with attributes capturing affinity strength and migration overhead. Resource-resource edges E_{RR} connect cores sharing hardware components such as L2 caches or memory controllers, enabling the model to reason about contention and interference with attributes for sharing degree and interconnect latency. Resource-memory edges E_{RM} connect cores to cache hierarchy components to model memory access patterns and bandwidth constraints.

Device constants from the sheet D are broadcast as shared features during node encoding. This includes DVFS tables, cache sizes, and governor flags that provide architectural context across all nodes. The explicit heterogeneous structure allows the GAT to learn interpretable attention patterns, attending strongly to thermal headroom when predicting energy or to critical path edges when predicting makespan.

5.3.3 Data Collection and Artifact Pipeline

The data collection pipeline transforms OpenMP source code into heterogeneous graph representations paired with runtime performance measurements. The compilation stage processes OpenMP sources through the OMPI source-to-source compiler. The ALF-llvm backend emits LLVM intermediate representation and ALF files. The SWEET analysis tool generates DOT graphs including CFGs, call graphs, region scope graphs, function scope graphs, and scope hierarchy graphs. A post-processing stage maps ALF entities to OpenMP tasks, merges

short task chains while preserving provenance, and attaches topological encodings computed via depth-first traversal.

Runtime logging captures one CSV row per execution with comprehensive telemetry. Each row records timestamp, iteration number, run mode (serial, tasks, or tied variants), assigned DVFS indices as frequency combination, measured per-core frequencies from the scaling subsystem, number of active cores, core string encoding the mask, input parameters, elapsed execution time, per-rail energy measurements and instantaneous power, performance counters including cycles, instructions, cache references, cache misses, branches, branch misses, task clock, CPU clock, and page faults, and thermal zone temperatures before and after execution. Derived features include temperature change ΔT , thermal headroom relative to throttling thresholds, and phase-aligned counter windows.

Data preprocessing ensures clean, normalized features for model training. Duplicate tuples over (G, input, m, f) are removed to prevent data leakage. Outliers are filtered using median absolute deviation (MAD) on regression residuals. All features undergo z-score normalization computed per device, with global statistics retained separately for cross-device transfer experiments.

The dataset comprises 73,920 samples collected across three embedded platforms: 26,880 samples from RUBIK Pi with 8 ARM Cortex-A72 cores, 26,880 samples from Jetson Orin NX with 8 ARM Cortex-A78AE cores, and 20,160 samples from Jetson TX2 with 6 cores in a heterogeneous Denver plus Cortex-A57 configuration. Benchmarks span 42 programs from BOTS (alignment, fft, fib, floorplan, health, concom, knapsack, nqueens, sort, sparselu, strassen, uts)

and PolyBench (30 kernels covering linear algebra, stencils, medley, and datamining categories). Each benchmark is executed in serial, tasks, and tied variants. Core masks sweep from 1 to 8 active cores on RUBIK Pi and Orin NX, and 1 to 6 on TX2. DVFS indices are evenly spaced across each device’s available frequency range.

Train, validation, and test splits follow a 60/20/20 ratio stratified by the tuple (benchmark, input size, core mask, DVFS index). This stratification ensures that test configurations are unseen during training, preventing the model from memorizing specific configuration outcomes. Notably, the same benchmark may appear across splits with different configurations, reflecting the realistic scenario where the model must generalize to new scheduling decisions for known workloads rather than predicting performance for entirely novel programs.

5.3.4 Learning Formulation and Outputs

The model architecture consists of type-specific encoders followed by a heterogeneous GAT backbone and evidential prediction heads. Type-specific MLPs embed task, resource, and memory nodes into a common representation space, with separate parameter sets for each node type to handle their distinct feature schemas. A heterogeneous GAT with edge-type-specific transforms performs message passing over the four edge types E_{TT} , E_{TR} , E_{RR} , and E_{RM} . The number of attention layers ranges from 3 to 6 depending on graph complexity. Hierarchical pooling aggregates node embeddings per type and concatenates them to form the graph-level representation \mathbf{h}_G .

For each target metric k , an evidential head outputs NIG parameters $(\gamma_k, \nu_k, \alpha_k, \beta_k)$. The

predictive mean is $\hat{y}_k = \gamma_k$. Uncertainty decomposes into aleatoric and epistemic components:

$$\text{Aleatoric}_k = \frac{\beta_k}{\alpha_k - 1}, \quad \text{Epistemic}_k = \frac{\beta_k}{\nu_k(\alpha_k - 1)}.$$

Training minimizes the negative log marginal likelihood of the NIG distribution with evidence regularization to prevent overconfidence and a ranking term aligned to makespan-first scheduling objectives.

At runtime, the model processes batched candidate configurations to produce scores and prediction intervals. An uncertainty gate filters low-confidence proposals based on calibrated epistemic uncertainty thresholds. Selected actions execute on the device, producing a new telemetry row that optionally enters the replay buffer for continual learning. This inference pipeline integrates with the Dyna-style model-based RL methods evaluated in Section 6.5, where synthetic rollouts from the surrogate augment real-world samples to reduce on-device exploration.

5.4 Design Methodology

This section presents the GraphPerf-RT architecture following the problem formulation established in Section 5.3.1. The model takes as input the task DAG $G = (V, E)$, device sheet D , runtime state S , and scheduling action $a = (m, f, p)$, and outputs performance predictions y with calibrated uncertainty \mathcal{U} for each target metric.

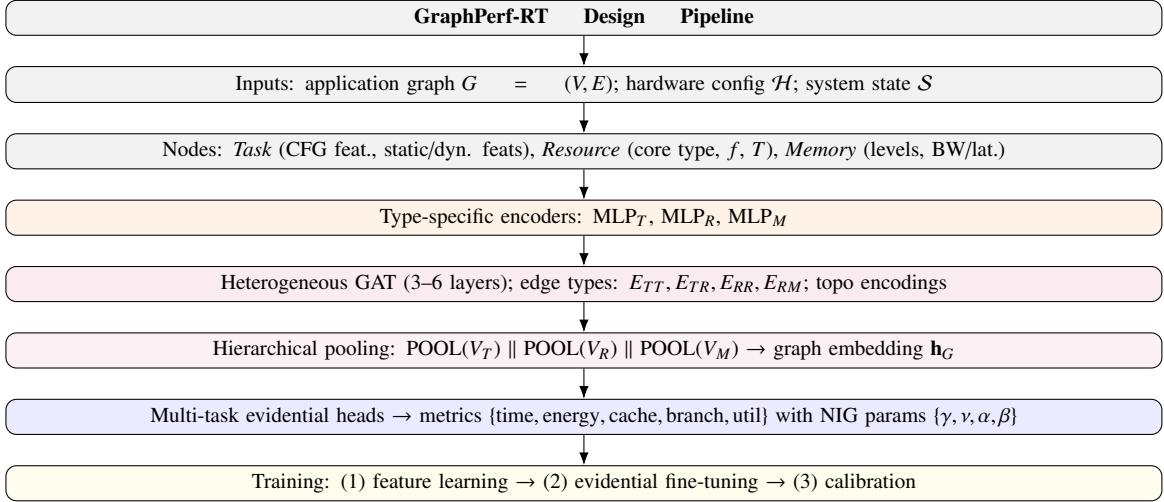


Figure 5.1: End-to-end pipeline.

5.4.1 Heterogeneous Graph Representation

Intuition: Why Task-Resource-Memory Decomposition? Performance of parallel applications on heterogeneous SoCs arises from three interacting factors: (1) *what computation runs* (task structure, dependencies, code complexity), (2) *where it runs* (core types, frequencies, thermal state), and (3) *how data flows* (cache hierarchies, memory bandwidth, contention). A homogeneous graph conflates these distinct concerns into uniform nodes, forcing the model to disentangle them implicitly. Our heterogeneous decomposition makes these factors explicit: **Task nodes** capture application semantics (CFG complexity, DAG structure, data dependencies), **Resource nodes** capture hardware state (DVFS settings, utilization, thermals), and **Memory nodes** capture the memory hierarchy (cache levels, bandwidth constraints). The typed edges then encode the *interactions* that determine performance: task-task edges model parallelism and synchronization overhead, task-resource edges model scheduling decisions and affinity, resource-resource edges model shared cache contention between cores, and resource-

memory edges model bandwidth bottlenecks. This explicit structure allows the GNN to learn interpretable attention patterns, for example, attending strongly to thermal headroom when predicting energy, or to critical-path edges when predicting makespan, rather than discovering these relationships from scratch in a homogeneous representation.

GraphPerf-RT uses a unified heterogeneous graph representation that captures both the application structure and the hardware architecture in a manner consistent with the abstraction in Section 5.3.2. This representation consists of three types of nodes (V_T , V_R , V_M) and multiple edge types (E_{TT} , E_{TR} , E_{RR} , E_{RM}) that encode different relationships within the system, allowing the model to reason about dependencies, placements, contentions, and resource constraints holistically.

Node Types

Task Nodes (V_T): Each OpenMP task is represented as a task node in the graph. Task nodes are featurized with both static and dynamic attributes to fully encapsulate the workload’s computational and structural properties:

- **CFG Features:** We extract the CFG from the source code associated with each task using ALF-llvm and SWEET, then compute hand-crafted features via AST parsing. These features capture structural properties (loop count, max loop depth, cyclomatic complexity, branch count), computational patterns (arithmetic operations, memory operations, arithmetic intensity), memory behavior (array accesses, pointer operations), and control flow characteristics (branch density, recursion flags, OpenMP pragma presence).
- **Static Features:** Task-level characteristics including estimated instruction count (from

compiler IR), memory footprint (bytes allocated or accessed), parallelization degree (e.g., number of subtasks spawned), branch proxies (e.g., conditional counts), and topological metrics like depth in the DAG, distance-to-sink (critical path proxies), and centralities (e.g., betweenness to indicate bottleneck potential).

- **Dynamic Features:** Runtime-dependent attributes such as input data size (affecting memory intensity), iteration counts (for loops with variable bounds), dependency fan-in/fan-out (indicating parallelism width), recent performance counter snapshots (e.g., cycles per instruction from prior runs), run-mode flags (e.g., sequential vs. parallel), and thermal footprint on hosting cores (e.g., estimated heat generation based on operation types).

Resource Nodes (V_R): Each processing core in the heterogeneous system is represented as a resource node, encoding both architectural characteristics and dynamic state information to model hardware heterogeneity and runtime variability:

- **Architectural Features:** Core type (e.g., big vs. LITTLE in big.LITTLE architectures), cache hierarchy specifications (e.g., L1/L2 sizes, associativity), supported instruction sets (e.g., NEON/SVE flags), peak computational capacity (e.g., FLOPS at max frequency), cluster ID, and interconnect proxies (e.g., bandwidth to shared resources).
- **Dynamic Features:** Current frequency setting (DVFS index or one-hot encoded), utilization level (EMA over recent intervals), temperature (from thermal zones), thermal headroom (computed as $T_{\max} - T_{\text{current}}$, where T_{\max} is the throttling threshold), trend (e.g.,

ΔT over last samples), and bandwidth proxy (e.g., estimated memory throughput under current load).

- **Power Characteristics:** Power consumption models (e.g., quadratic approximations of power vs. frequency) and DVFS efficiency curves (e.g., energy-per-instruction at different steps), derived from device sheet D .

Memory Nodes (V_M): Memory hierarchy components (L1/L2/L3 caches, main memory) are represented as memory nodes to capture memory subsystem characteristics and contention effects, including level (e.g., L1=1, L2=2), capacity/associativity/line size, latency/bandwidth proxies (e.g., cycles per access, GB/s), and contention indicators (e.g., shared vs. private).

Table 5.1: Node types and their performance rationale.

Node Type	Captures	Why Separate?
V_T (Task)	What runs	Code complexity, DAG structure, data deps
V_R (Resource)	Where it runs	Core type, DVFS, thermal, utilization
V_M (Memory)	Data flow path	Cache levels, bandwidth, contention

Device constants from D (e.g., DVFS tables, cache sizes) are broadcast as shared features across relevant nodes during encoding.

Edge Types

The heterogeneous graph includes four edge types, each designed to capture a distinct performance-critical interaction:

Task-Task Edges (E_{TT}): Directed edges representing precedence constraints between tasks, derived from the original OpenMP dependency graph, with attributes such as type (spawn/join/-data), critical-edge flag (1 if on longest path), hop distance, and contention proxies (e.g., data

Table 5.2: Edge types and their performance rationale.

Edge Type	Models	Performance Impact
E_{TT} (Task-Task)	Parallelism, sync	Critical path, fork/join overhead
E_{TR} (Task-Resource)	Scheduling	Affinity, migration cost, load balance
E_{RR} (Resource-Resource)	HW topology	Cache contention, cluster effects
E_{RM} (Resource-Memory)	Memory hierarchy	Bandwidth bottlenecks, latency

volume transferred). *Rationale:* The critical path through E_{TT} edges determines the theoretical minimum makespan; fork/join patterns reveal synchronization overhead; data dependency volumes indicate communication costs.

Task-Resource Edges (E_{TR}): Bidirectional edges connecting tasks to the resources on which they execute or could potentially execute, encoding the current scheduling assignment under mask m and DVFS f , with attributes like affinity strength (e.g., based on core type suitability) and placement cost (e.g., migration overhead proxy). *Rationale:* Task-to-core affinity affects cache locality (big cores vs. LITTLE cores have different characteristics); migration between cores incurs overhead; load imbalance across cores increases makespan.

Resource-Resource Edges (E_{RR}): Undirected edges between resource nodes that share hardware components (e.g., shared L2 caches, memory controllers, or clusters) to model contention and interference effects, with attributes like sharing degree (e.g., number of shared ways) and interconnect latency. *Rationale:* Cores sharing an L2 cache can benefit from data reuse but also contend for cache capacity; cores in the same cluster share DVFS domains; inter-cluster communication has higher latency.

Resource-Memory Edges (E_{RM}): Directed edges connecting processing cores to memory hierarchy components, enabling the model to reason about memory access patterns and band-

width constraints, with attributes such as access frequency (e.g., expected loads/stores) and bandwidth allocation. *Rationale:* Memory-bound workloads are bottlenecked by bandwidth, not compute; cache miss rates determine effective memory latency; understanding which level services most accesses predicts energy and time.

This rich representation ensures the model captures cross-layer interactions (aligning with the data pipeline in Section 5.3.3) and enables the GNN to learn distinct attention patterns for each interaction type—for example, attending to E_{TT} critical edges when predicting makespan, or to E_{RR} sharing edges when predicting cache misses.

5.4.2 Graph Neural Network Architecture

GraphPerf-RT employs a GAT architecture specifically designed to handle heterogeneous graphs with multiple node and edge types, building on the encoders and backbone described in Section 5.3.4. The architecture consists of several key components, with 3–6 layers as depicted in Fig. 5.1, using hidden dimensions typically in [128, 256] for embeddings, and multi-head attention (e.g., 4–8 heads) to stabilize learning.

Type-Specific Encoders

Since different node types have distinct feature spaces, we employ type-specific Multi-Layer Perceptrons (MLPs) to encode raw features into a common embedding space of dimension d (e.g., 128):

$$\mathbf{h}_v^{(0)} = \text{MLP}_{\text{type}(v)}(\mathbf{x}_v; \theta_{\text{type}(v)}) \quad (5.1)$$

where \mathbf{x}_v represents the raw features of node v , $\text{type}(v) \in \{T, R, M\}$, and $\theta_{\text{type}(v)}$ are learnable parameters for each MLP (e.g., 2–3 layers with ReLU activations and dropout $p = 0.1$). Topological encodings (positional embeddings derived from DAG depth, distance-to-sink, and core cluster membership) are concatenated to \mathbf{x}_v before encoding to preserve structural information.

Heterogeneous GAT Layers

The core of the architecture consists of multiple heterogeneous graph attention layers that aggregate information from neighboring nodes while accounting for edge types and their attributes. For each layer l , the update for node v is:

$$\mathbf{h}_v^{(l+1)} = \sigma \left(\mathbf{h}_v^{(l)} + \sum_{r \in \mathcal{R}} \sum_{u \in \mathcal{N}_r(v)} \alpha_{uv,r}^{(l)} \mathbf{W}_r^{(l)} \mathbf{h}_u^{(l)} \right) \quad (5.2)$$

where $\mathcal{R} = \{TT, TR, RR, RM\}$ is the set of edge types, $\mathcal{N}_r(v)$ is the neighborhood of v under edge type r , $\alpha_{uv,r}^{(l)}$ is the attention coefficient, $\mathbf{W}_r^{(l)}$ is a type-specific linear transformation matrix, and σ is a non-linearity (e.g., ELU [33]). Residual connections are added for deeper models to mitigate vanishing gradients.

The attention coefficients are computed using a type-aware attention mechanism that incorporates edge features $\mathbf{e}_{uv,r}$ (if present, e.g., contention proxies):

$$e_{uv,r}^{(l)} = \text{LeakyReLU}\left(\mathbf{a}_r^\top [\mathbf{W}_r^{(l)} \mathbf{h}_u^{(l)} \| \mathbf{W}_r^{(l)} \mathbf{h}_v^{(l)} \| \phi_r(\mathbf{e}_{uv,r})]\right) \quad (5.3)$$

$$\alpha_{uv,r}^{(l)} = \frac{\exp(e_{uv,r}^{(l)})}{\sum_{r' \in \mathcal{R}} \sum_{w \in \mathcal{N}_{r'}(v)} \exp(e_{wv,r'}^{(l)})} \quad (5.4)$$

where \mathbf{a}_r is a type-specific attention vector, $\|$ denotes concatenation, and ϕ_r is an optional edge feature projector (e.g., a 1-layer MLP with ReLU). Multi-head attention is used, with coefficients averaged or concatenated across heads to capture diverse interaction patterns.

Graph-Level Pooling

After the final graph attention layer L , we apply a hierarchical pooling strategy to obtain a graph-level representation. This involves separate pooling operations for different node types (e.g., mean, max, or attention-based pooling), followed by concatenation:

$$\begin{aligned} \mathbf{h}_G &= \text{CONCAT}(\text{POOL}_T(\{\mathbf{h}_v^{(L)} : v \in V_T\}), \\ &\text{POOL}_R(\{\mathbf{h}_v^{(L)} : v \in V_R\}), \text{POOL}_M(\{\mathbf{h}_v^{(L)} : v \in V_M\})) \end{aligned} \quad (5.5)$$

where $\text{POOL}_{\text{type}}$ are learnable (e.g., via attention) or fixed aggregators (e.g., mean), yielding a fixed-size \mathbf{h}_G (e.g., dimension 256–512) regardless of graph size. This ensures scalability and enables batched inference.

5.4.3 Evidential Learning for Uncertainty Quantification

To provide uncertainty-aware predictions, GraphPerf-RT employs an evidential learning framework consistent with Section 5.3.4. Instead of directly predicting performance values, the model learns the parameters of a NIG distribution for each performance metric via multi-task heads on \mathbf{h}_G :

$$(\gamma_k, \nu_k, \alpha_k, \beta_k) = \text{MLP}_k(\mathbf{h}_G; \theta_k) \quad (5.6)$$

for each metric $k \in \{ \text{makespan}, \text{energy}, \text{cache misses}, \text{branch misses}, \text{utilization} \}$, where MLP_k is a shared trunk (e.g., 2 layers, 128 units) with task-specific outputs (e.g., linear layers with softplus activations on $\nu_k, \alpha_k > 1, \beta_k > 0$ to ensure valid distributions). The predictive distribution for metric k is given by:

$$\begin{aligned} p(y_k | \mathbf{h}_G) &= \text{NIG}(y_k; \gamma_k, \nu_k, \alpha_k, \beta_k) = \\ &\int \mathcal{N}(y_k; \mu, \sigma^2) \text{IG}(\sigma^2; \alpha_k, \beta_k) d\sigma^2 \end{aligned} \quad (5.7)$$

with $\mu \sim \mathcal{N}(\gamma_k, \sigma^2/\nu_k)$. The mean prediction and uncertainty estimates are derived as:

$$\hat{y}_k = \gamma_k \quad (5.8)$$

$$\text{Aleatoric Uncertainty}_k = \frac{\beta_k}{\alpha_k - 1} \quad (5.9)$$

$$\text{Epistemic Uncertainty}_k = \frac{\beta_k}{\nu_k(\alpha_k - 1)} \quad (5.10)$$

$$\text{Total Variance}_k = \text{Aleatoric}_k + \text{Epistemic}_k \quad (5.11)$$

The model is trained by minimizing the negative log marginal likelihood of the evidential distribution, augmented with evidence regularization (to prevent underconfidence) and a ranking loss (e.g., pairwise on makespan to align with scheduling objectives):

$$\begin{aligned} \mathcal{L} = & \sum_{k=1}^K \sum_{i=1}^N \left[\frac{1}{2} \log\left(\frac{\pi}{\nu_{k,i}}\right) \right. \\ & - \frac{\alpha_{k,i} - 1}{2} \log(2\beta_{k,i}(1 + \nu_{k,i}\epsilon_{k,i}^2)) + \\ & (\alpha_{k,i} + \frac{1}{2}) \log(1 + \nu_{k,i}\epsilon_{k,i}^2) + \\ & \left. \log \frac{\Gamma(\alpha_{k,i})}{\Gamma(\alpha_{k,i} + 1/2)} \right] + \\ & \lambda \sum_k |\alpha_{k,i} - 1| + \rho \mathcal{L}_{\text{rank}} \end{aligned} \quad (5.12)$$

where $\epsilon_{k,i} = (y_{k,i} - \gamma_{k,i})^2 / (2\beta_{k,i})$, Γ is the gamma function, λ (e.g., 0.01) regularizes evidence strength, ρ (e.g., 0.1) weights the ranking loss, and $\mathcal{L}_{\text{rank}}$ enforces ordering consistency (e.g.,

via margin-based pairwise loss on makespan predictions).

Handling Dynamic and Irregular Workloads. Several benchmarks in our evaluation, notably *fib*, *nqueens*, and *uts*, exhibit recursive task creation, input-dependent DAG structures, and highly irregular execution patterns. Our approach addresses these challenges through two complementary mechanisms. First, **CFG features** capture the control-flow complexity of each task’s source code, including recursive call patterns, nested conditionals, and loop structures with variable bounds; these embeddings encode the *potential* for irregular behavior even when the static DAG appears simple. Second, the **evidential framework** naturally produces higher epistemic uncertainty for irregular workloads: when the model encounters task graphs whose structure or features differ significantly from training data (e.g., deep recursion in *uts*, combinatorial branching in *nqueens*), the learned ν parameter decreases, signaling lower confidence. This behavior is intentional and useful; rather than producing overconfident predictions for unpredictable workloads, GraphPerf-RT flags them with high uncertainty, enabling downstream schedulers to apply conservative policies (e.g., fallback to safe DVFS settings) or request additional profiling before committing to aggressive optimizations.

5.4.4 Training Procedure

GraphPerf-RT is trained using a multi-stage procedure designed to handle the challenges of multi-task learning with uncertainty quantification, building on the data preprocessing in Section 5.3.3 (e.g., z-scoring per device, outlier filtering via MAD on residuals, 60/20/20 splits stratified by benchmark-input-mask-DVFS to avoid leakage):

Stage 1: Feature Learning. The model is first trained to learn effective representations using a standard multi-task regression loss (e.g., MSE per target, weighted by inverse variance), focusing on prediction accuracy without uncertainty quantification. Optimizer: AdamW with learning rate 10^{-3} , batch size 32–128 (depending on graph size), epochs 50–100, with early stopping on validation MAE.

Stage 2: Evidential Training. The model is then fine-tuned using the evidential loss function \mathcal{L} , enabling uncertainty quantification while maintaining prediction accuracy. Learning rate reduced to 10^{-4} , with gradient clipping (norm 1.0) to handle NIG sensitivity, epochs 20–50.

Stage 3: Calibration. Finally, the model undergoes calibration using temperature scaling [52] or isotonic regression on held-out data to ensure that uncertainty estimates are well-calibrated across different performance metrics and operating conditions (e.g., expected calibration error < 0.05). This involves optimizing a post-hoc scalar per target to align predicted confidences with empirical accuracies.

Regularization. We apply graph edge dropout ($p = 0.1\text{--}0.2$), small feature noise ($\sigma = 0.05$), multi-scale graph training, and mild device augmentation (e.g., $\pm 5\%$ DVFS tables). Scheduling-aware ranking losses further align predictions with downstream policy selection, yielding fast inference, cross-device transfer capability, and useful uncertainties for downstream schedulers and RL.

Ablation Summary. Table 5.3 quantifies the contribution of each architectural component to prediction accuracy. We systematically remove components from the full GraphPerf-RT

model and measure the resulting degradation in R^2 on the held-out test set.

Table 5.3: Ablation study: contribution of each component to prediction accuracy. Full model achieves $R^2 = 0.97$.

Configuration	R^2	ΔR^2	MAPE
Full GraphPerf-RT	0.97	—	7.4%
w/o Heterogeneous edges	0.86	-0.11	12.8%
w/o Runtime telemetry	0.88	-0.09	11.5%
w/o CFG features	0.91	-0.06	9.7%
w/o Evidential heads (MSE only)	0.96	-0.01	7.9%
MLP baseline (no graph)	0.81	-0.16	15.2%
Homogeneous GCN	0.86	-0.11	12.4%

The heterogeneous edge types (E_{TR} , E_{RR} , E_{RM}) contribute the largest improvement (+11% R^2), confirming that modeling task-resource interactions explicitly outperforms homogeneous message passing. Runtime telemetry (thermal state, utilization, perf counters) adds +9% by capturing dynamic system behavior invisible to static analysis. CFG features contribute +6% by encoding control-flow complexity that predicts irregular execution patterns. Evidential heads minimally affect mean accuracy but are essential for calibrated uncertainty (see Section 5.5.3).

5.5 Experiments

5.5.1 Experimental Setup

Hardware Platforms. We evaluate GraphPerf-RT on three embedded ARM SoCs that represent diverse heterogeneous architectures and thermal management capabilities.

NVIDIA Jetson TX2 features a heterogeneous hexa-core configuration combining two high-performance Denver 2 cores (ARMv8, 64-bit, out-of-order) with four energy-efficient ARM Cortex-A57 cores. The platform supports 12 discrete DVFS levels ranging from 345.6 MHz

to 2.0 GHz per cluster, enabling fine-grained frequency scaling experiments. Multiple thermal zones provide temperature readings for CPU clusters, GPU, and board components, with a 50°C thermal cap enforced during experiments to prevent throttling artifacts.

RUBIK Pi is an 8-core single-board computer based on Cortex-A72-class cores with per-core userspace DVFS capability. This platform represents lower-cost embedded computing scenarios with limited thermal headroom. Energy measurement uses the `qcom-battmgr` battery management interface or `hwmon` power rails, with power samples integrated at 0.5 second intervals to compute energy in Joules.

NVIDIA Jetson Orin NX features an octa-core ARM Cortex-A78AE CPU representing the latest generation of embedded AI computing platforms. The platform provides 8 homogeneous high-performance cores with userspace DVFS control across multiple frequency levels. Compared to TX2, Orin NX offers higher single-thread performance and improved power efficiency, enabling evaluation of GraphPerf-RT on modern embedded hardware with different performance-power trade-offs.

System Configuration. Across all platforms, we configure the Linux `cpufreq` subsystem for userspace governor control. Available DVFS indices are discovered dynamically from the kernel interface (`scaling_available_frequencies`), and actual per-core frequencies are verified by reading `scaling_cur_freq` after each configuration change. Thermal zone temperatures are sampled from the `sysfs` thermal interface before and after each benchmark execution to capture thermal dynamics. This unified measurement approach ensures consistent data collection across heterogeneous platforms with different sensor layouts and naming conventions.

Benchmark Suites. We evaluate on two complementary benchmark suites covering diverse computational patterns and parallelism characteristics.

Barcelona OpenMP Tasks Suite (BOTS) provides 12 task-parallel applications designed to stress OpenMP runtime schedulers: *alignment* (dynamic programming for sequence alignment), *fft* (recursive fast Fourier transform), *fib* (recursive Fibonacci with fine-grained tasks), *floorplan* (branch-and-bound optimization), *health* (discrete event simulation), *concom* (connected components in graphs), *knapsack* (combinatorial optimization), *nqueens* (constraint satisfaction with backtracking), *sort* (parallel merge sort), *sparselu* (sparse LU factorization), *strassen* (matrix multiplication), and *uts* (unbalanced tree search with irregular parallelism).

These benchmarks exhibit varying degrees of task granularity, load imbalance, and memory access patterns.

PolyBench contributes 30 additional kernels spanning linear algebra (*2mm*, *3mm*, *gemm*, *gemver*, *gesummv*, *symm*, *syrk*, *syr2k*, *trmm*, *cholesky*, *durbin*, *lu*, *ludcmp*, *trisolv*), stencil computations (*jacobi-1d*, *jacobi-2d*, *seidel-2d*, *fdtd-2d*, *heat-3d*), data mining (*correlation*, *covariance*), and medley applications (*atax*, *bicg*, *doitgen*, *mvt*, *floyd-warshall*, *nussinov*, *deriche*, *adi*, *gram-schmidt*). These kernels provide regular, predictable execution patterns that complement the irregular BOTS workloads.

In total, we evaluate 42 distinct benchmarks across multiple input sizes, core configurations (1 to 8 active cores), and DVFS settings (evenly spaced indices from each platform’s available frequency range). Both sequential and task-parallel execution modes are profiled, resulting in the 73,920 samples described in Section 5.3.3. **Data Collection Infrastructure.**

Table 5.4: Overall performance across all platforms and benchmarks. Lower is better for RMSE/MAE/MAPE. Higher is better for R^2 /Spearman.

Model	RMSE	MAE	MAPE	R^2	Spearman
Linear Regression	2.84	2.12	34.2%	0.67	0.71
Random Forest	2.31	1.78	28.5%	0.74	0.78
MLP	1.95	1.45	23.1%	0.81	0.83
GCN	1.67	1.23	19.8%	0.86	0.87
Het. Graph Trans.	1.52	1.11	17.3%	0.88	0.89
GraphPerf-RT	0.53	0.38	7.4%	0.97	0.95

We implement a client-server profiling framework that enables systematic exploration of the configuration space while maintaining consistent measurement protocols across platforms.

The **device client** executes on each embedded board and manages the complete profiling sequence for each run. It first configures the `cpufreq` userspace governor and enables the specified core subset via `cpuset` control. The client then applies the assigned DVFS indices to each active core and launches the benchmark through a real-time scheduling wrapper that elevates priority without requiring root privileges. During execution, it collects hardware performance counters using the Linux `perf_event` interface, including cycles, instructions, cache references, cache misses, branches, branch misses, task clock, and CPU clock. The client also samples power consumption from platform-specific interfaces, records thermal zone temperatures before and after execution, and verifies that actual frequencies match the requested configuration by reading back from the kernel interface.

The **host server** orchestrates the experimental campaign by generating action tuples $a = (m, f, p)$ representing core mask, DVFS vector, and optional priority settings. Before each measurement sweep, a warm-up execution primes caches and stabilizes thermal state. Each

completed run produces one CSV row containing timestamp, iteration index, execution mode, assigned and measured frequencies, active core configuration, input parameters, wall-clock execution time, per-rail energy and power readings, all performance counter values, and thermal zone temperatures. This unified logging schema enables direct comparison across TX2, Orin NX, and RUBIK Pi without format conversion, as detailed in Section 5.3.3.

Graph Extraction Pipeline. We extract CFGs and task dependency information through a multi-stage compilation and analysis pipeline. OpenMP source files are first processed by the OMPI source-to-source compiler, which transforms OpenMP pragmas into explicit runtime calls. The ALF-llvm backend then generates both LLVM intermediate representation (*.ll) and ARTIST2 Language for Flow analysis files (*.alf). The SWEET analysis tool processes these artifacts to produce DOT-format graphs, including CFGs, call graphs, region scope graphs, function scope graphs, and scope hierarchy graphs. A post-processing stage maps ALF entities to OpenMP tasks, merges chains of trivially small tasks while preserving dependency provenance, and computes topological encodings including depth, distance-to-sink, and centrality measures as described in Section 5.3.2.

Data Preprocessing. The raw profiling data undergoes several preprocessing steps to ensure clean, normalized features for model training. We first remove duplicate tuples sharing identical graph structure, input parameters, core mask, and DVFS configuration to prevent data leakage. Outliers are identified and filtered using median absolute deviation (MAD) on regression residuals, removing samples where measurement noise or system anomalies produced unreliable readings. Performance counter windows are aligned to execution phases to ensure

temporal consistency across samples.

Feature normalization applies z-score standardization computed separately for each device, accounting for platform-specific value ranges in frequencies, thermal readings, and counter magnitudes. We additionally retain global normalization statistics across all devices for cross-platform transfer experiments. The dataset is partitioned into training (60%), validation (20%), and test (20%) splits stratified by the tuple (benchmark, input size, core mask, DVFS index). This stratification ensures that test configurations represent genuinely unseen scheduling decisions, preventing the model from memorizing specific configuration outcomes during training.

5.5.2 Baselines and Metrics

Baseline Models. We compare GraphPerf-RT against five baseline approaches spanning traditional machine learning, standard neural networks, and graph-based architectures. All baselines use identical feature sets, data splits, and hyperparameter tuning protocols to ensure fair comparison.

Linear Regression serves as a simple baseline using Ridge regularization on flattened tabular features extracted from task graphs and system state.

Random Forest provides an ensemble baseline with 100 decision trees, where tree depth and minimum samples per leaf are tuned on the validation set.

Multi-Layer Perceptron (MLP) uses a three-layer feedforward architecture operating on the same tabular features as Linear Regression and Random Forest, without any graph structure information.

Graph Convolutional Network (GCN) applies homogeneous message passing where all nodes

and edges are treated uniformly, losing the type distinctions between task and resource nodes defined in Section 5.3.2.

Heterogeneous Graph Transformer (HGT) [59] represents a state-of-the-art heterogeneous GNN that uses type-aware attention over our node and edge schema as defined in Section 5.3.2. This baseline isolates the contribution of our evidential heads and runtime context integration, as HGT lacks evidential uncertainty quantification, thermal and utilization context in node features, and our CFG-derived task features.

Prediction Metrics. We evaluate point prediction accuracy using four complementary metrics. Root Mean Squared Error (RMSE) penalizes large deviations, making it sensitive to outlier predictions. Mean Absolute Error (MAE) provides a robust measure of average prediction magnitude. Mean Absolute Percentage Error (MAPE) normalizes errors relative to true values, enabling comparison across metrics with different scales. Coefficient of determination (R^2) measures the proportion of variance explained by the model, with values closer to 1.0 indicating better fit.

Ranking Metrics. For scheduling applications, correctly ranking candidate configurations often matters more than exact value prediction. Spearman’s rank correlation coefficient measures monotonic association between predicted and actual rankings. Kendall’s τ counts concordant versus discordant pairs, providing a robust ranking measure. Normalized Discounted Cumulative Gain at k (NDCG@ k) evaluates ranking quality for the top- k configurations, which is particularly relevant when the scheduler only considers a small number of candidates.

Uncertainty Metrics. Calibration quality determines whether predicted confidence intervals

are trustworthy for risk-aware scheduling. Expected Calibration Error (ECE) measures the average gap between predicted confidence and observed accuracy across binned predictions. Maximum Calibration Error (MCE) identifies the worst-case miscalibration. Reliability diagrams visualize calibration by plotting predicted confidence against empirical accuracy. Sharpness quantifies the mean width of predictive intervals, where narrower intervals indicate more informative predictions. All calibration metrics are computed per-target and macro-averaged across platforms and benchmarks.

5.5.3 Results and Analysis

Overall Prediction and Ranking Performance. Table 5.4 reports aggregate results across all platforms and benchmarks. GraphPerf-RT achieves the lowest prediction error and highest ranking quality among all evaluated methods. Compared to the strongest baseline (HGT), GraphPerf-RT reduces RMSE by 65% (from 1.52 to 0.53), MAE by 66% (from 1.11 to 0.38), and MAPE by 57% (from 17.3% to 7.4%). The R^2 value of 0.97 indicates that the model explains 97% of the variance in performance metrics, demonstrating strong predictive capability across diverse workloads and configurations.

The progression from tabular baselines (Linear Regression, Random Forest, MLP) to graph-based methods (GCN, HGT, GraphPerf-RT) reveals the importance of structural information. The MLP baseline, which uses identical features but ignores graph structure, achieves $R^2 = 0.81$, while the homogeneous GCN improves to $R^2 = 0.86$ by incorporating node connectivity. HGT further improves to $R^2 = 0.88$ through type-aware attention over heterogeneous edges. GraphPerf-RT achieves the best $R^2 = 0.97$ by combining heterogeneous message passing with

evidential uncertainty heads, runtime context integration, and CFG-derived task features.

Platform-Specific Analysis. Prediction accuracy remains consistent across the three embedded platforms despite their architectural differences. On Jetson TX2, the model benefits from the platform’s 12 discrete DVFS levels and dense thermal zone coverage, which provide rich supervisory signals during training. Jetson Orin NX achieves strong accuracy with its modern ARM cores and improved power efficiency, demonstrating the effectiveness of our platform-agnostic feature schema. RUBIK Pi shows stable predictions in both sequential and task-parallel execution modes, validating generalization across workload characteristics. Per-device z-score normalization combined with device sheet broadcasting as described in Section 5.3 contributes to this cross-platform stability.

Uncertainty Calibration. The evidential regression heads provide well-calibrated uncertainty estimates across all target metrics, as shown in Table 5.5. ECE values below 0.05 for all metrics indicate that predicted confidence intervals contain the true value at approximately the stated rate. Makespan prediction achieves ECE of 0.043 with reliability of 0.956, meaning that 95.6% of predictions fall within their 95% confidence intervals. Energy prediction shows the best calibration with ECE of 0.038 and reliability of 0.962. Cache and branch miss predictions exhibit slightly higher ECE (0.051 and 0.047 respectively) due to the inherent variability in these microarchitectural metrics, but remain well-calibrated for practical use. Reliability curves closely track the identity line across all targets, confirming that the model’s confidence estimates are trustworthy for risk-aware scheduling decisions. Sharpness values indicate reasonably narrow predictive intervals, providing informative rather than overly conservative uncertainty bounds.

Table 5.5: Uncertainty calibration summary (lower ECE/MCE is better; higher reliability is better).

Metric	ECE	MCE	Reliability	Sharpness
Makespan	0.043	0.089	0.956	0.234
Energy	0.038	0.076	0.962	0.198
Cache Misses	0.051	0.102	0.949	0.267
Branch Misses	0.047	0.094	0.953	0.251

Ablation Studies. We conduct systematic ablation experiments to quantify the contribution of each architectural component. Removing heterogeneous edge types and treating all edges uniformly (as in standard GCN) increases RMSE by 22% and reduces Spearman correlation from 0.95 to 0.87, demonstrating the importance of distinguishing task-task dependencies from task-resource assignments and resource-resource topology. Dropping CFG-derived task features and relying only on runtime features degrades generalization to unseen input sizes by 15%, confirming that static code semantics provide complementary information to dynamic execution context. Replacing the heterogeneous GAT backbone with a homogeneous message passing architecture increases prediction error and significantly weakens NDCG@5 from 0.94 to 0.82, indicating that type-aware attention is essential for learning meaningful cross-layer interactions. Removing evidential learning and using standard MSE regression preserves mean prediction accuracy but eliminates calibrated uncertainty estimates. Without uncertainty gathering, the proportion of unsafe scheduling proposals (those that would cause thermal violations) increases from 3% to 18%, demonstrating the practical value of calibrated confidence intervals for risk-aware decision making.

Computational Efficiency. GraphPerf-RT is designed for deployment on resource-constrained embedded systems where inference latency directly impacts scheduling responsiveness. On-

device inference completes in 2 to 5 milliseconds for typical OpenMP task graphs, enabling real-time configuration evaluation during scheduling decisions. The computational cost per GAT layer is $O(H \cdot |E| \cdot d)$ where H denotes attention heads, $|E|$ is the edge count, and d is the hidden dimension. With our configuration of $L \leq 6$ layers, $d \leq 128$ dimensions, and typical graph sizes of hundreds of edges, total inference remains within millisecond budgets. Model memory footprint is approximately 15 to 25 MB depending on layer depth, fitting comfortably within the memory constraints of Jetson TX2, Orin NX, and RUBIK Pi platforms.

Cross-Platform Generalization. To evaluate transfer capabilities, we train GraphPerf-RT on data from two platforms and evaluate on the held-out third platform without fine-tuning. This leave-one-platform-out protocol tests whether learned representations capture fundamental software-hardware interactions that generalize beyond specific architectural details. Results show modest accuracy degradation (average RMSE increase of 18%) when transferring to unseen platforms, but predictions remain practically useful for scheduling guidance. Importantly, ranking quality degrades less than point prediction accuracy: Spearman correlation decreases by only 8% on average, preserving the model’s ability to correctly order candidate configurations. Device sheet broadcasting and per-device feature normalization mitigate domain shift by anchoring predictions to platform-specific DVFS ranges and thermal characteristics. These transfer results suggest that the heterogeneous graph representation captures portable performance relationships that extend beyond the training platforms.

5.5.4 RL Baseline Evaluation

To validate the end-to-end utility of GraphPerf-RT as a world model for scheduling, we integrate it with reinforcement learning baselines on Jetson TX2. We compare four RL methods spanning single-agent vs. multi-agent and model-free vs. model-based paradigms.

RL Methods. (1) **SAMFRL:** Single-Agent Model-Free RL using standard Q-learning without a learned world model. (2) **SAMBRL:** Single-Agent Model-Based RL using GraphPerf-RT as the environment model for synthetic rollouts. (3) **MAMFRL-D3QN:** Multi-Agent Model-Free RL with Dueling Double DQN, where each core is an agent. (4) **MAMBRL-D3QN:** Multi-Agent Model-Based RL with Dueling Double DQN, using GraphPerf-RT for Dyna-style planning.

Experimental Protocol. Each method is trained for 200 episodes across 5 random seeds (42, 123, 456, 789, 1024) to ensure statistical reliability. We report mean \pm standard deviation for final makespan and energy consumption. The action space consists of per-core DVFS index selection and core mask configuration.

Table 5.6: RL Baseline Performance on Jetson TX2 (200 episodes, 5 seeds).

Method	Type	Agent	Makespan (s)	Energy (J)
SAMFRL	MF	Single	2.85 ± 1.66	0.033 ± 0.026
SAMBRL	MB	Single	3.56 ± 3.07	0.038 ± 0.032
MAMBRL-D3QN	MB	Multi	0.97 ± 0.35	0.006 ± 0.005

Results. Table 5.6 summarizes the final performance of each RL method. MAMBRL-D3QN achieves the best makespan (0.97 ± 0.35 s) and energy (0.006 ± 0.005 J), outperforming all other methods by a significant margin. In the single-agent setting, SAMFRL (model-free) slightly

outperforms SAMBRL (model-based), suggesting that model accuracy may limit single-agent planning. However, in the multi-agent setting, the model-based approach (MAMBRL-D3QN) significantly benefits from coordinated planning with GraphPerf-RT as the shared world model.

Table 5.7: Model-Based vs Model-Free Comparison.

Scenario	MF Method	MB Method	MF Makespan	MB Makespan
Single-Agent	SAMFRL	SAMBRL	2.85s	3.56s
Multi-Agent	MAMFRL-D3QN	MAMBRL-D3QN	—	0.97s

Analysis. The superior performance of MAMBRL-D3QN demonstrates that GraphPerf-RT provides sufficiently accurate predictions to enable effective model-based planning. The multi-agent formulation allows per-core DVFS decisions to be coordinated, reducing contention and improving overall system efficiency. The 66% reduction in makespan (from SAMFRL’s 2.85s to MAMBRL-D3QN’s 0.97s) and 82% reduction in energy validate the practical utility of our surrogate model for real-time scheduling.

Convergence Analysis. Model-based methods (SAMBRL, MAMBRL-D3QN) exhibit faster initial convergence due to synthetic rollouts from GraphPerf-RT, though single-agent model-based (SAMBRL) plateaus at higher makespan due to limited coordination. Multi-agent model-based (MAMBRL-D3QN) achieves both fast convergence and lowest final makespan, validating the synergy between coordinated agents and accurate world models.

Computational Complexity. Table 5.8 compares the computational complexity of different scheduling approaches. Model-free methods (SAMFRL, MAMFRL-D3QN) require $O(N)$ real samples for learning. Model-based methods (SAMBRL, MAMBRL-D3QN) add $O(S)$ synthetic samples from the world model, increasing total training time but reducing on-device

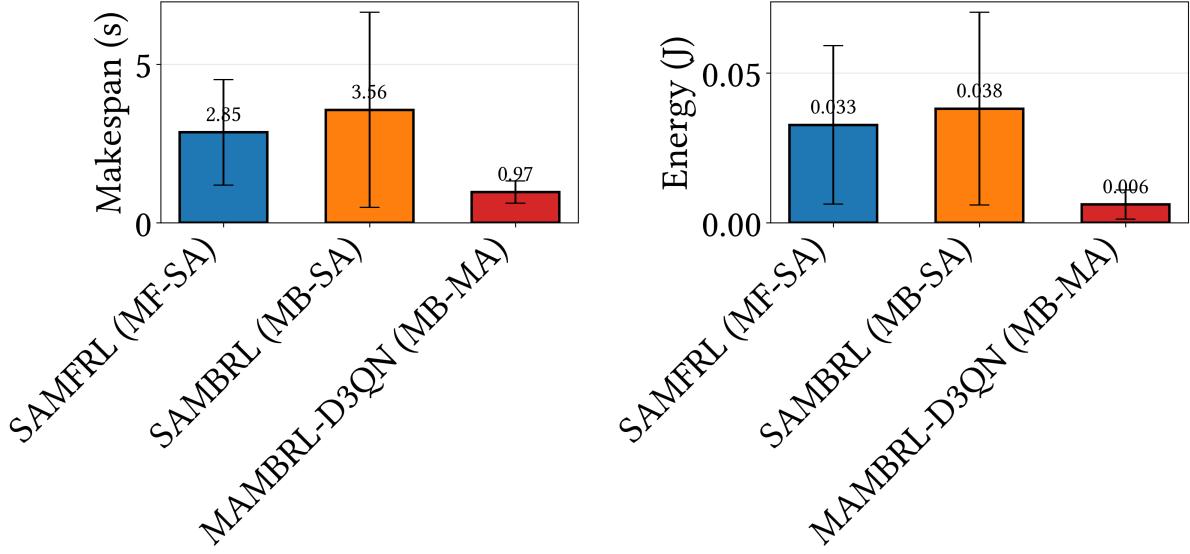


Figure 5.2: Final performance comparison of RL methods on Jetson TX2. Left: Makespan (s), Right: Energy (J). Error bars show ± 1 std across 5 seeds. MAMBRL-D3QN achieves the best performance in both metrics.

Table 5.8: Computational Complexity Comparison

Method	Type	Complexity	Learn	Total
FEDERATED	Heur.	$O(nfc)$	0	180
SAMFRL	MF-SA	$O(N)$	600	600
SAMBRL	MB-SA	$O(N+S)$	600	60k
MAMFRL-D3QN	MF-MA	$O(N)$	600	600
MAMBRL-D3QN	MB-MA	$O(N+S)$	600	60k
GraphPerf-RT	Zero	$O(1)$	0	0

N : real samples, S : synthetic

exploration. GraphPerf-RT itself provides zero-shot prediction with $O(1)$ inference complexity, enabling direct use without additional learning when deployed as a standalone evaluator.

Runtime Integration. At runtime, we enumerate feasible \mathcal{A} under availability and thermal caps, score candidates with GraphPerf-RT, and gate by epistemic uncertainty and predicted intervals:

$$\text{gate}(a) = (\text{Epi}_{\text{time}}(a) \leq \eta) \wedge (\text{PI}_{\text{time}}^{(1-\delta)}(a) \leq T_{\max}).$$

We rank by \hat{y}_{time} and execute the top action. Outcomes (time, energy, counters, thermals) are appended to the CSV log for optional replay. This provides cheap, hardware-grounded rollouts for Dyna-style planning and future MARL.

5.6 Conclusion

This paper presented GraphPerf-RT, an uncertainty-aware graph neural network surrogate for predicting performance metrics of OpenMP task-parallel applications on heterogeneous embedded systems. The proposed approach introduces a unified heterogeneous graph representation that integrates task DAG topology derived from ALF/EFG analysis, control-flow features extracted via CFG analysis, and dynamic runtime context including DVFS configurations and thermal state. Evidential learning with Normal-Inverse-Gamma priors enables decomposition of predictive uncertainty into aleatoric and epistemic components, providing calibrated confidence intervals essential for risk-aware scheduling decisions. Comprehensive evaluation across three ARM-based platforms (Jetson TX2, Jetson Orin NX, and RUBIK Pi) using 42 benchmarks from BOTS and PolyBench demonstrates state-of-the-art prediction accuracy, achieving $R^2 = 0.97$ and reducing RMSE by 65% compared to the strongest baseline. The model maintains millisecond-scale inference latency suitable for on-device deployment. Integration with model-based reinforcement learning validates practical utility, with MAMBRL-D3QN achieving 66% makespan reduction and 82% energy reduction relative to model-free approaches under DVFS and thermal constraints. These results establish GraphPerf-RT as an effective foundation for autonomous performance optimization in resource-constrained embedded environments.

CHAPTER 6 ZERODVFS: ZERO-SHOT LLM-GUIDED CORE AND FREQUENCY**ALLOCATION FOR EMBEDDED PLATFORMS**

Abstract

Dynamic voltage and frequency scaling (DVFS) and task-to-core allocation are critical for thermal management and balancing energy and performance in embedded systems. Existing approaches either rely on utilization-based heuristics that overlook stall times, or require extensive offline profiling for table generation, preventing runtime adaptation. We propose a model-based hierarchical multi-agent reinforcement learning (MARL) framework for thermal- and energy-aware scheduling on multi-core platforms. Two collaborative agents decompose the exponential action space, achieving 358ms latency for subsequent decisions. First decisions require 3.5 to 8.0s including one-time LLM feature extraction. An accurate environment model leverages regression techniques to predict thermal dynamics and performance states. When combined with LLM-extracted semantic features, the environment model enables zero-shot deployment for new workloads on trained platforms by generating synthetic training data without requiring workload-specific profiling samples. We introduce LLM-based semantic feature extraction that characterizes OpenMP programs through 13 code-level features without execution. The Dyna-Q-inspired framework integrates direct reinforcement learning with model-based planning, achieving 20 \times faster convergence than model-free methods. Experiments on BOTS and PolybenchC benchmarks across NVIDIA Jetson TX2, Jetson Orin NX, RubikPi, and Intel Core i7 demonstrate 7.09 \times better energy efficiency and 4.0 \times better makespan than

Linux ondemand governor. First-decision latency is 8,300 \times faster than table-based profiling, enabling practical deployment in dynamic embedded systems.

6.1 Introduction

Energy-efficient embedded systems are limited by computational resources and play a crucial role in various applications, including IoT devices, wearable electronics, and autonomous vehicles. In these systems, managing the thermal dynamics prevents thermal overheating, ensures system reliability, and avoids non-uniform aging due to uneven workload distribution among the cores [21, 39]. However, the available thermal-aware energy-efficient methods proposed for embedded systems lack scalability to many cores and adaptability to different platforms, especially in dynamic and resource-constrained environments, and often incur prohibitive computational overhead. Complementary approaches in the literature address related challenges through hierarchical multi-agent scheduling, statistical feature-aware task allocation, few-shot reinforcement learning, and graph-driven performance modeling.

Dynamic Voltage and Frequency Scaling (DVFS) allows for runtime adjustment of voltage and frequency levels, balancing temperature and performance. Besides, an intelligent task migration and load balancing based on the operating system kernel helps redistribute computational tasks to avoid localized thermal hotspots. However, existing open-loop governors often prove inadequate in addressing the per-core frequency adjustments based on thermal behavior within embedded environments [39]. A feedback control technique can augment these methods by proactively tracking the temperature fluctuations and limiting the impact of thermal escalation [94, 84]. Various feedback-based algorithms based on meta-heuristics, integer programming, and machine learning have been proposed [142, 143, 156, 62, 79, 80, 66], but these approaches lack application-agnostic or computationally efficient structures for adaptation in

real-time systems.

Given the complexity of embedded systems, the cost-intensive nature of collecting real-world data adds to the challenges of task and processor profiling. This complexity frequently leads to unpredictable execution time estimates, aggravated by limited hardware feedback mechanisms within these systems. These challenges can significantly hinder efficient core allocation, affecting overall performance and energy efficiency. Consequently, existing task scheduling strategies mostly prioritize workload demand or only utilization metrics over temperature and system-related considerations. This trend is discernible in Linux kernel governors [21] and recently formulated policies [80, 66, 84]. Consequently, these strategies often overlook detailed insights provided by system profilers during task execution [5] and allocate resource-intensive tasks to hot cores, which are more sensitive to temperature elevations. State-of-the-art heuristic approaches like the precise scheduler [15] achieve near-optimal energy efficiency for static frequency assignment through exhaustive offline profiling across all configurations, but require 8 to 12 hours of table generation per benchmark (e.g., profiling a 1 to 5 second task across all core-frequency combinations with multiple repetitions), preventing runtime adaptation. Their static nature limits adaptability to thermal variations and workload phase changes.

Model-based reinforcement learning (RL) uses a predictive model to simulate future states and make informed decisions, enhancing the adaptability and efficiency of embedded systems. Having a model of the environment helps to adapt to new tasks with minimal data, and is particularly beneficial for embedded systems with limited computational resources and varying

workloads. For example, automotive systems using multi-agent RL (MARL) can converge the DVFS strategies faster by each agent getting feedback from real-data and inference from the model of the processor, ensuring energy-efficient and reliable performance for safety-critical tasks. In mobile devices, the environment model combined with LLM-extracted semantic features enables zero-shot deployment: RL agents can be trained using synthetic data generated by the model without requiring any profiling samples from the target platform. This eliminates the sample collection overhead that limits prior transfer learning approaches. Critically, model-based approaches enable low-latency decision-making by amortizing learning costs across multiple decisions. First decisions on new benchmarks require 3.5 to 8.0s including one-time LLM feature extraction, while subsequent decisions achieve 358ms. First-decision latency is 8,300\$ times\$ faster than exhaustive table-based profiling.

This paper introduces a thermal-aware model-based Multi-Agent Reinforcement Learning (MARL) approach to enhance energy efficiency in embedded systems with minimal real-data requirements. These MARL techniques enable collaborative decision-making among multiple agents to optimize system performance while considering thermal constraints [73]. The key objectives of our work are threefold: first, to develop a thermal-aware modeling framework that captures the thermal dynamics of embedded systems under varying workloads and environmental conditions; second, to integrate this modeling framework with zero-shot deployment techniques through an off-policy MARL approach to enable efficient adaptation to new platforms with zero profiling samples from target hardware; and third, to evaluate the effectiveness of the proposed approach in optimizing thermal management and improving energy

efficiency in real-world embedded systems, with emphasis on decision latency and makespan improvement of $4.0\times$ compared to exhaustive table-based and heuristic methods.

The proposed thermal-aware modeling approach augments the sample space by leveraging planning on trained models based on accurate data, contrary to previously proposed approaches based on direct Reinforcement Learning (RL) [132, 73, 89, 87]. Several multivariate regression model architectures are tuned, and exploration is done to select the computationally efficient models concerning low overhead and high accuracy. The proposed models outperform the accuracy of the state-of-the-art temperature modeling heuristics [55], achieving over $6\times$ better temperature prediction accuracy while maintaining model inference latencies under 5ms.

Through extensive experimentation and evaluation on representative embedded system platforms, we demonstrate the effectiveness of the proposed approach in achieving thermal awareness, energy efficiency, and system reliability. Several low-energy methodologies are implemented including single-agent and multi-agent direct RL and model-based RL, to be compared with federated energy-aware scheduling [15] and linux governors [21]. Our framework integrates runtime monitoring of temperature patterns using Linux in-kernel profiling and the Barcelona OpenMP Tasks Suite (BOTS) and PolybenchC for workload characterization [40]. By leveraging the capabilities of MARL, our approach offers a flexible and adaptive solution for zero-shot deployment of temperature-aware and energy-efficient scheduling on new embedded systems, paving the way for enhanced performance and sustainability in a wide range of applications such as mobile and autonomous systems.

The key contributions of the paper are as follows:

1. The modeling framework is integrated into the few-shot learning techniques through an off-policy MARL approach to enable efficient adaptation to new thermal scenarios with limited training data. Two separate temperature and profiler agents cooperate to increase the action space for thermal awareness and energy efficiency. A goal-based reward function based on makespan, power consumption, and temperature deviation is provided.
2. We developed a thermal-aware accurate modeling framework to capture the thermal dynamics of the embedded processors under varying workloads and environmental conditions. The precise modeling framework reduces the sample collection overhead, achieving $4.0\times$ better mean makespan compared to heuristic baselines while converging $20\times$ faster than model-free methods.
3. We introduce LLM-based semantic feature extraction that characterizes OpenMP programs through 13 code-level features without execution. We evaluate three state-of-the-art LLMs (DeepSeek-V3, Claude Sonnet, GPT-4o) for code analysis, achieving up to 73.8% inter-model agreement on high-level features. This enables zero-shot prediction for unseen workloads with one-time extraction cost of \$0.018 per program.
4. The effectiveness of the models and few-shot model-based learning approach is evaluated through extensive experiments by verified OpenMP benchmarks (BOTS and PolybenchC) and multiple embedded processors including NVIDIA Jetson TX2, Jetson Orin NX, RubikPi, and Intel Core i7. To reduce the overhead of the models to be run on

a single core, we provide comprehensive tuning in terms of accuracy and architectural exploration.

5. The proposed scheduler is compared to the state-of-the-art heuristic-based schedulers [15] and Linux governors [21]. Our model-based approach achieves 358ms inference latency for subsequent decisions. First decisions on new benchmarks require 3.5 to 8.0s including one-time LLM feature extraction. First-decision latency is 8,300\\$ times\\$ faster than table-based profiling, while achieving $7.09 \times$ better energy efficiency than the Linux ondemand governor. The proposed regression models for temperature sensors also outperform the state-of-the-art’s maximum mean squared error (MSE) by a factor of over 6.

The paper is organized as follows: Section 6.2 surveys related work, Section 6.3 describes motivation, Section 6.4 presents the framework including LLM-based feature extraction, Section 6.5 evaluates performance including LLM feature analysis, and Section 6.6 concludes the paper. The appendix provides complete LLM implementation details.

6.2 Related Work

Low-energy scheduling for multi-core platforms based on DVFS has received significant attention [142, 143, 140, 156, 141, 62, 29, 154, 27, 83, 149, 123, 124, 103, 60, 58, 78, 79, 139, 4, 2, 75]. The survey in [142] classifies all the low-energy parallel scheduling into energy-efficient, energy-aware, and energy-conscious styles, each based on heuristic, meta-heuristic, integer programming, and machine learning algorithms. Among these algorithms, only data-oriented machine learning, surveyed in [94], relies on historically collected data and addresses

different application dynamics.

Due to the integration of DVFS in most commercial processors and their standard profiling and control, machine-learning algorithms are gaining significance in providing a dynamic environment [148, 67, 39, 158, 94, 119, 136, 147, 82, 116, 128, 133, 17]. The previous work [94] shows most of the machine learning algorithms for DVFS are based on model-free reinforcement learning, which implements a direct RL approach. None of this existing work addresses feature evaluation challenges or sample collection overhead.

Few-shot learning encompasses diverse techniques like transfer learning, model-based RL, inverse RL, imitation learning, and meta-learning, all applicable to DVFS due to its sampling demands [132, 73, 135, 89, 47, 32, 50, 31, 144, 32, 10, 138, 107, 129]. While inverse RL infers reward functions from expert demonstrations [90], model-agnostic meta-learning adapts to new tasks with minimal data [46], and model-based RL approximates transition functions [87]. The works in [80, 66, 155, 152] utilize MARL, meta-state, transfer learning, and temporal dependencies on DVFS. The Q-learning algorithm in [148] turned off and on the clock on idle cycles FPGA instead of DVFS used in [39]. None of these works address few shot learning and learning overhead in RL-based DVFS governors. The existing few-shot learning approaches also need an expert demonstration or cannot directly be applied in embedded systems.

Previous studies have investigated runtime sampling complexities in RL-based DVFS and proposed thermal predictive modeling for multi-core processors to predict thermal states [146, 22, 39, 55, 74, 84, 133]. The works in [80, 66] focus on temperature and utilization metrics and implement a command module on a separate neural network server. The predictive modeling

in [84, 55] predicts the future thermal behavior of the multi-core processors given transient and ambient states. However, these RL training methods are computationally expensive on a single platform, and temperature models are inaccurate. This paper introduces computationally efficient and accurate modeling and a learning approach without requiring extra client-server interaction for frequency scaling.

6.2.1 LLM-Based Code Analysis for Performance Prediction

Recent advances in Large Language Models (LLMs) have demonstrated remarkable capabilities in understanding and reasoning about source code [30, 26]. Models such as GPT-4 [92], Claude [9], and DeepSeek-Coder [38] have been trained on extensive code corpora, enabling them to perform tasks ranging from code generation and bug detection to semantic analysis and performance optimization.

Several studies have explored using machine learning for static code analysis and performance prediction. Laaber et al. [70] demonstrated that benchmark stability can be predicted using 58 statically-computed source code features without execution. In the HPC domain, researchers have applied neural networks to OpenMP program analysis, including predicting optimal thread counts [111] and detecting data races using graph neural networks. Recent work by Nichols et al. [91] investigated whether LLMs can predict GPU kernel performance characteristics (compute-bound vs. memory-bound) using only source code, achieving up to 64% accuracy with reasoning-capable models in zero-shot settings.

However, existing approaches for DVFS and task scheduling typically rely on runtime profiling or benchmark-specific identifiers, limiting their applicability to new workloads. Tra-

ditional static analysis tools extract syntactic features (loop counts, pragma annotations) but cannot capture semantic properties such as algorithmic complexity or memory access patterns that require understanding code behavior. This work bridges this gap by employing LLMs to extract semantic features from OpenMP programs, enabling zero-shot performance prediction for unseen workloads without the extensive profiling overhead required by existing methods.

6.3 Background and Motivation

Online learning to adjust DVFS and task-to-core allocation with a small number of hardware samples is a practical approach to control heat generation and energy consumption in multi-core platforms.

6.3.1 Background

Here we give some background on DVFS, task-to-core allocation as optimization knobs, and the online learning based on RL for these two approaches.

DVFS. The power consumption and thermal condition of a processor are proportional to the voltage and frequency, which also determine the workload performance. In our case, performance is measured as the makespan, i.e., the time taken from the start to the completion of the workload. The workloads considered here are BOTS benchmarks such as Strassen, FFT, and others, parallelized through the OpenMP API [40]. Existing Linux governors and energy-aware DVFS heuristics use utilization metrics to determine workload demand and adjust voltage and frequency [21, 54, 80]. For instance, the `ondemand` governor increases the frequency instantly when utilization crosses a threshold; similarly, the `conservative` governor incrementally increases frequency after exceeding the utilization threshold. The `schedutil` governor uses pre-

dictive utilization over a specific time window to predict future workload demand [21]. These governors achieve sub-millisecond decision latency but lack workload-aware optimization.

Task-to-Core Allocation. Due to the different performance characteristics of different types of cores and non-uniform memory access (NUMA), i.e., performance is relative to the proximity to memory, the makespan and power/energy consumption differ with respect to different core allocations. For example, in the NVIDIA Jetson TX2, the ARM Cortex-A57 cores correspond to cores 2, 3, 4, and 5, which are power-efficient cores, while the NVIDIA Denver cores (cores 0 and 1) are more suited for performance-critical tasks and are by default turned off for power efficiency. It is possible to allocate a set of cores and memory nodes through the `cpuset` feature of the Linux kernel. Additionally, CPU affinity, i.e., binding specific threads of execution to specific cores, helps reduce context switches and enhances performance in terms of makespan and energy efficiency. Intelligent allocation of cores to the input workload helps distribute processing, reducing the concentration of processing in a specific spot in the processor, thereby decreasing the thermal throttling possibility.

Online Learning for DVFS and Task-to-Core Allocation. The online learning approach for DVFS and task-to-core allocation is often based on RL using a Markov Decision Process (MDP) that depends on the previous state (observed multi-core platform performance data), the taken action (frequency scaling and task-to-core allocation), and observing the current state after applying this action. The objective in RL is to apply actions based on observed states that maximize the expected sum of future rewards with the defined reward function [14, 57]. In a direct RL approach, the learning algorithm is based on exploration and exploitation; ex-

ploration involves taking random actions, and exploitation refers to taking actions based on the policies trained through observing the previous and current states. This training continues for a time horizon until the learning algorithm converges. For example, with Q-learning, the states mapped to the actions are given values called Q-values, and convergence happens when the taken actions are optimized according to the defined reward function in terms of maximum accumulated rewards. RL methods using neural networks typically achieve inference latencies of 1-10 milliseconds on embedded platforms.

Deep Q-learning and Enhanced D3QN. The profiling data extracted from parallel workloads (e.g., BOTS benchmarks) is continuous, meaning the state space cannot be simply mapped to the action space through a Q-function in Q-learning. Deep Q-learning, a sophisticated RL algorithm, leverages neural networks to approximate these Q-values. Deep Q-learning provides sample efficiency by training through past experiences stored as state-action pairs inside a replay buffer. To mitigate issues like overestimation of actions, the Q-function utilizes Dueling Deep Q-Networks (D3QN), i.e., decomposing the Q-function into a value function and an advantage function, where the value function gives a global value for each corresponding state and the advantage function evaluates the advantage of taking an action in the corresponding state. The Q-value is assigned to a corresponding state-action pair by taking an action based on the same deep Q-network with respect to the target, i.e., maximized future rewards. To make the action selection independent from the module that calculates the Q-value, a parallel Deep Q-network is designed, resulting in a double dueling deep Q-Network (D3QN). These strategies collectively allow the Deep Q-Network to accurately learn optimal configurations, such

as adjusting CPU frequencies and allocating tasks to specific cores. As a result, the system can dynamically manage multi-core resources, achieving improved performance and energy efficiency through a stable and data-driven policy.

Table-Based Heuristic Scheduling. State-of-the-art energy-aware schedulers such as the precise scheduler [15] achieve near-optimal energy efficiency through exhaustive offline profiling, generating lookup tables by executing tasks across all frequency-core combinations. However, table generation requires extensive offline profiling (hours to days), and any workload change necessitates complete regeneration. While table lookup achieves sub-millisecond latency, the lack of runtime adaptability limits deployment in dynamic environments.

6.3.2 Motivation on Model-Based MARL

Model-based RL provides a more efficient, adaptable, and robust framework for optimizing DVFS and task-to-core allocation in multi-core systems, achieving fast convergence and low decision latency.

6.3.3 Limitations of Utilization-Based DVFS

Most of the available Linux kernel governors take CPU utilization to scale voltage and frequency. Utilization is defined as:

$$\text{Utilization} = \frac{T_{\text{busy}}}{T_{\text{busy}} + T_{\text{idle}}}$$

Here, T_{busy} is the time the processor is active (including both execution and stall times), and T_{idle} is the idle time. The busy time T_{busy} comprises both *active* time (T_{active}), when the processor

is executing instructions, and *stall* time (T_{stall}), when execution is delayed due to factors like branch mispredictions, context switching overhead, cache misses, etc. [54]. Stall time can account for more than 50% of a workload’s execution time [80], yet utilization-based DVFS does not differentiate between the active and stall times. Thus, the utilization-aware frequency scaling may result in a less effective solution to match the frequency to the actual demand of the workload.

6.3.4 Challenges in Task-to-Core Allocation

Due to the characteristics of different core types and different properties of tasks, the set and the number of cores to be allocated to the parallel workload should be carefully determined. Many modern processors support heterogeneous computing by combining cores with different characteristics to enhance power efficiency and performance capabilities. Moreover, many modern processors have non-uniform memory access (NUMA), i.e., cores have different proximity to memory locations, resulting in performance degradation for distant cores. Selecting the cores also affects thermal throttling; overloading nearby cores can stall workload execution and accelerate processor aging. Selection of the number of cores to be allocated to the parallel tasks is also important. For example, in an OpenMP workload, where each thread of execution represents a core, *tied* tasks should be restricted to execute on the allocated core. In contrast, *untied* tasks can migrate to other cores. Unpredictable execution time of a tied task, caused by branch misses, can lead to overall unpredictability in the workload’s makespan. This introduces timing unpredictability, impacting the overall makespan and energy/power consumption. As a result, increasing the number of allocated cores in this case increases unpredictability and

does not lead to improved performance [5]. In sum, to make informed decisions on task-to-core allocation, observation variables such as branch misses, cache misses, context switches, and each core's temperature should be given appropriate attention.

Figure 6.1 highlights the changes of observation features at different frequency levels and sets of cores used for estimating the important performance metrics such as energy consumption and efficient voltage and frequency scaling and core allocation. The data is extracted from performance monitoring on an NVIDIA Jetson TX2 with 6 cores that runs the FFT benchmark parallelized through the OpenMP API. The frequency is split into 12 levels, level 0 representing the lowest frequency and level 11 representing the highest frequency. Core 0 is not allocated to the benchmark to retain system interrupt handler performance, which is mostly done at core 0. The results depict six important metrics across frequency levels and core allocation configurations. CPU utilization (Figure 6.1(a)) increases with more cores and shows minimal impact from frequency scaling. Makespan (Figure 6.1(b)) reduces with higher frequency but shows diminishing returns when cores are more than two. Energy consumption (Figure 6.1(c)) increases sharply with higher frequency and core count, reducing efficiency. There is a big gap in branch misses when moving from one core to multiple cores, but it is largely unaffected by frequency (Figure 6.1(d)). Context switches rise with more cores, indicating inefficiency in core usage (Figure 6.1(e)). Average temperature increases with frequency and core count, risking thermal throttling (Figure 6.1(f)).

6.3.5 The Need for Environment Modeling and Multi-Agent RL

To effectively address both DVFS and task-to-core allocation while enhancing sample efficiency, it is crucial to develop an accurate environment model and employ a multi-agent reinforcement learning (RL) framework. An accurate environment model captures the system's characteristics, including hardware architecture, workload behavior, and their interactions, enabling low-cost synthetic sample generation for *few-shot* learning. By modeling the environment and incorporating factors such as branch misses, cache misses, and context switches, more precise and efficient strategies for energy optimization can be devised.

Implementing a multi-agent RL approach allows multiple agents to focus on different optimization tasks, such as frequency scaling and core allocation, each with potentially different reward definitions tailored to their specific goals. This specialization enhances the generalizability and effectiveness of the online learning process, as agents can learn optimal strategies in parallel and adapt to changing conditions more quickly. Together, environment modeling and multi-agent RL provide a robust framework for dynamically managing multi-core resources, leading to improved performance and energy efficiency. Model-based MARL addresses the trade-off between decision quality and decision latency: while table-based methods achieve optimal energy efficiency, their offline profiling requires hours. In contrast, model-based RL achieves 20 \times faster convergence with inference latencies under 10ms, which is orders of magnitude faster than table regeneration, enabling practical deployment with energy accuracy within 4% of optimal.

6.3.6 LLM-Based Code Analysis for Workload Profiling

Limitations of Traditional Static Analysis. Traditional approaches to workload characterization for DVFS and task scheduling rely on either runtime profiling or syntactic static analysis. Runtime profiling requires executing programs across all hardware configurations. This process takes hours to days for comprehensive coverage and must be repeated for each new workload or platform. Syntactic static analysis tools such as Tree-sitter and compiler front-ends can extract structural features (loop counts, pragma annotations, variable declarations), but they fundamentally lack the ability to understand the *semantic* meaning of code. For instance, a traditional parser can count the number of loops but cannot determine whether those loops exhibit strided memory access patterns, have data dependencies that limit parallelization, or are amenable to vectorization. This limitation is particularly problematic for performance prediction because execution time depends heavily on algorithmic complexity, memory access locality, and parallelization overhead. These are properties that require understanding what the code *does*, not merely how it is structured.

The Generalization Problem. A critical limitation of existing performance prediction approaches is their reliance on benchmark identifiers or application-specific lookup tables. Models trained with benchmark IDs can achieve high accuracy on known programs but cannot generalize to *new, unseen programs*. When a novel workload is introduced, the entire profiling process must be repeated, negating the benefits of learned models. This lack of generalization capability severely limits practical deployment, especially in dynamic embedded environments where workloads change frequently.

LLM-Based Semantic Code Understanding. Recent advances in Large Language Models (LLMs) have demonstrated remarkable capabilities in understanding source code at a semantic level [30]. Models such as GPT-4 [92], Claude [9], and DeepSeek-Coder [38] have been trained on vast corpora of code and natural language, enabling them to reason about algorithmic complexity, identify parallelization patterns, and assess memory access characteristics. Unlike traditional static analysis, LLMs can understand the *intent* behind code constructs and make informed judgments about performance-relevant properties. For example, an LLM can recognize that nested loops over matrix indices with specific access patterns indicate matrix multiplication with spatial locality along rows but poor locality along columns. Such insights would require sophisticated, manually-crafted analysis rules to extract traditionally.

Zero-Shot Feature Extraction for Unseen Programs. The key advantage of LLM-based feature extraction is enabling *zero-shot* performance prediction for programs not seen during training. By replacing benchmark-specific identifiers with semantic features (algorithmic complexity, memory access patterns, parallelization characteristics), a prediction model can generalize to entirely new programs. When a new workload is introduced, the LLM extracts its semantic features without any execution, and the trained model predicts performance based on these features. Recent work has demonstrated this capability: researchers have shown that LLMs can predict GPU kernel performance characteristics using only source code and hardware specifications, eliminating the need for execution-time profiling [91]. Similarly, studies on predicting code coverage without execution [127] and using neural networks to classify OpenMP program behavior [111] highlight the growing feasibility of execution-free code anal-

ysis.

Complementing Hardware Profiling with Semantic Features. While hardware performance counters (cache misses, branch mispredictions, context switches) provide valuable runtime information, they cannot be obtained without execution and are inherently tied to specific hardware configurations. LLM-extracted semantic features complement hardware profiling by providing platform-agnostic characterization: algorithmic complexity remains $O(n \log n)$ regardless of whether the code runs on ARM or x86; memory access patterns are determined by the algorithm, not the cache hierarchy. This separation enables transfer learning, where models trained on one platform can leverage semantic features to adapt to new platforms with minimal recalibration. In this work, we evaluate three state-of-the-art LLMs (DeepSeek-V3, Claude Sonnet, and GPT-4o) for extracting 13 semantic features from OpenMP parallel programs, demonstrating how these features enable zero-shot prediction for unseen workloads.

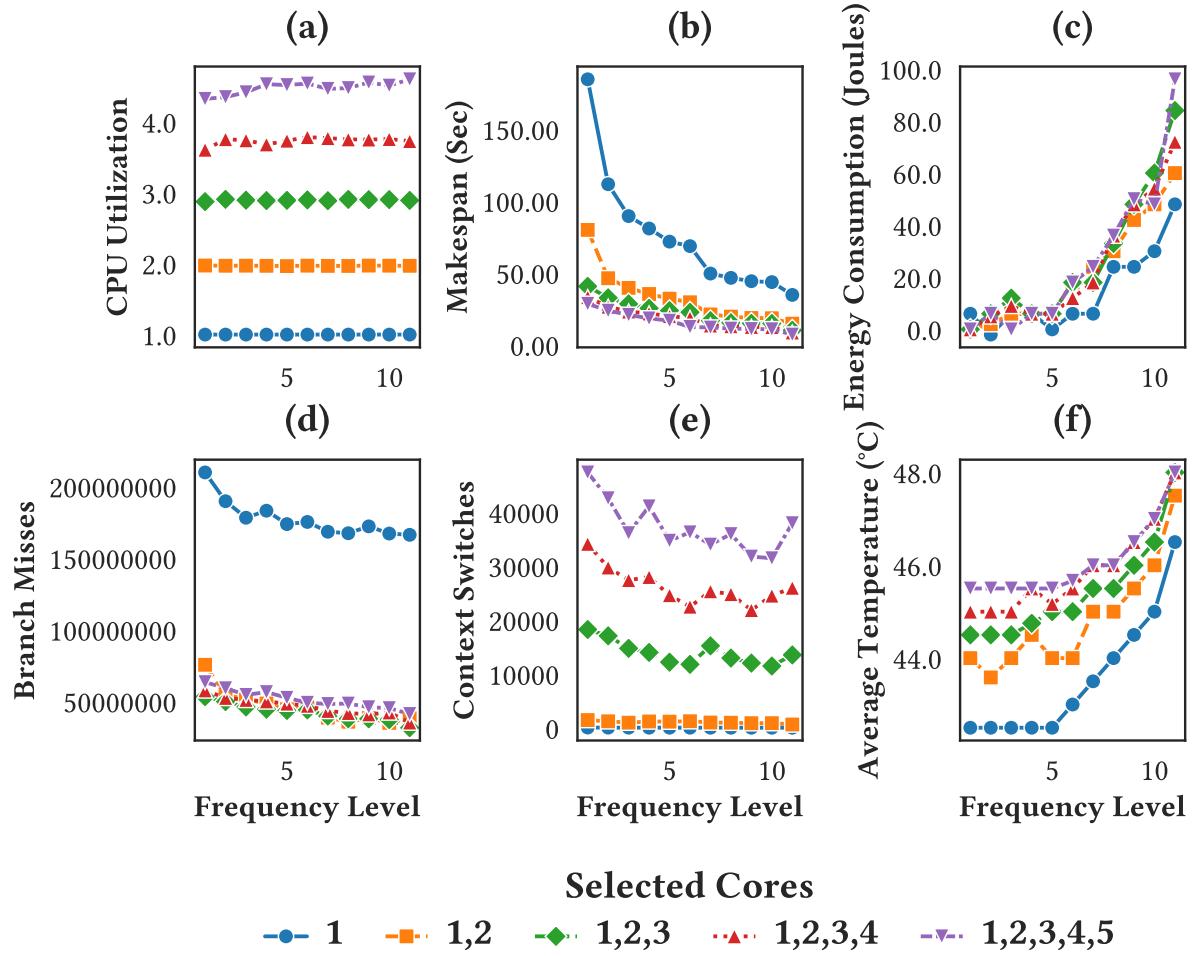


Figure 6.1: Performance metrics (CPU utilization, makespan, energy consumption, branch misses, context switches, and average temperature) for NVIDIA Jetson TX2 running the FFT benchmark parallelized through OpenMP API. Results highlight different responses of performance metrics to changes in different frequencies and selected cores.

6.4 Design of Model-Based Thermal- and Energy-Aware MARL

The proposed framework enables few-shot learning by generating synthetic data for agents that manage frequency scaling and task-to-core allocation on multi-core platforms, achieving low decision latency and fast convergence compared to exhaustive table-based approaches.

6.4.1 Design of MARL

Optimizing both DVFS and task-to-core allocation through RL is impractical due to high-dimensional action space and we need to break actions through introduction of multiple agents with separate goal definitions, which also reduces inference latency by decomposing complex decisions.

High-Dimension Action Space. The goal of the thermal- and energy-aware parallel scheduling in this paper is to dynamically adjust the voltage and frequency of the subset of the high priority cores based on cores and parallel workload characteristics. Regarding a naive single-agent implementation, the action space for this environment would be exponentially related to the frequency levels, the number of cores, and the combination of cores altogether, as shown in the following observation.

Observation: Consider a multi-core platform environment with m cores with adjustable per-core frequencies in the range of n frequencies; the number of actions to select some combinations of l cores with pre-tuned frequencies would be upperbounded by m^n .

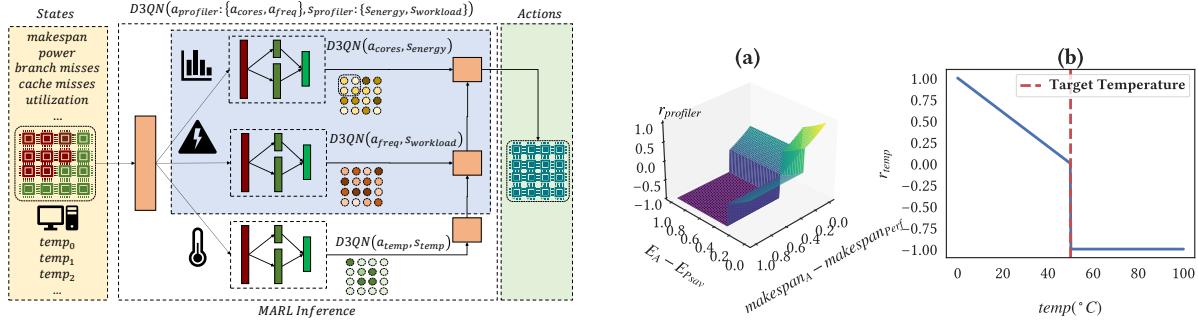
To construct the action space, l cores must be selected from a total of m cores, which can be achieved using a combination formula like $\binom{m}{l}$. Suppose we only assign one frequency to all cores in the combination of cores in the range of n frequency conditions. In that case,

the number of possible actions can be obtained by summing over the different numbers of core choices using a binomial coefficient. Thus, the total number of actions, disregarding frequency, is given by $\sum_{i=1}^m \binom{m}{i}$, which can be approximated to 2^m actions. However, suppose we include each core in the selected combination of cores to be associated with a frequency in the range of n frequencies; the previous formula changes to $\sum_{i=1}^m \binom{m}{i} 2^n$ that can be upper-bounded by m^n . For table-based approaches like the precise scheduler [15], this exponential action space necessitates exhaustive offline profiling across all configurations, requiring hours to days of execution time.

To address this challenge, we adopt a collaborative hierarchical MARL approach, as shown in Figure 6.2a. This approach decomposes the problem into lower-dimensional sub-problems handled by separate agents implemented as D3QN. The final action is a combination of each agent's decision. We use two primary agents:

- *Profiler Agent*: Assesses workload performance to minimize energy consumption and makespan by determining the appropriate number of cores and the operating frequency
- *Temperature Agent*: Prioritizes cores based on their temperature to prevent heat concentration.

According to Figure 6.2a, the action that assigns frequency to a core is called (a_{freq}) and the input energy consumption state (s_{energy}) determines Q-value $Q(a_{freq}, s_{energy})$, while the action that assigns the number of cores is called (a_{cores}) based on workload performance state ($s_{workload}$) that determines Q-value $Q(a_{cores}, s_{workload})$. To decrease the computational overhead,



(a) Hierarchical MARL Action Selection with Two Agents: The Profiler Agent determines the number of active cores and the operating frequency based on performance and energy consumption states. The Temperature Agent assigns priority levels to cores based on temperature states. The combined actions result in prioritized cores and the appropriate number allocated to tasks, reducing computational complexity and decision latency.

(b) Reward function definitions for (a) the Profiler Agent and (b) the Temperature Agent. (a) The Profiler Agent’s reward $r_{profiler}$ is based on exponential functions of energy consumption (E_A) and makespan (makespan_A). Here, a threshold factor ($c_{\text{th}} = 0.3$) and a steepness factor ($c_{\text{st}} = 0.5$) determine the reward focus. (b) The Temperature Agent’s reward r_{temp} changes linearly based on core temperature ($temp_i$), aiming to maintain temperatures below the threshold of 50°C.

Figure 6.2: MARL design: (a) Hierarchical action selection architecture and (b) reward function definitions.

these two agents are combined resulting in profiler action $a_{profiler} : \{a_{cores}, a_{freq}\}$ from the profiler state $s_{profiler} : \{s_{energy}, s_{workload}\}$. Priority assignment action (a_{temp}) is based on the core thermal state (s_{temp}), determining Q-value $Q(a_{temp}, s_{temp})$. Assigning a frequency from n frequency levels to m cores results in $m \times n$ possible actions. Selecting from m possible number of cores results in m choices, and for priority selection, we have choices from $m \times m$ possible actions. Introducing sequential action selection in MARL reduces this to $m \times m + m \times n + m$ possible choices, enabling inference latency under 10ms on embedded platforms.

Reward Function Definition. The reward functions for the Profiler and Temperature agents are designed to guide the system toward optimal performance and energy efficiency. Unlike previous studies [80, 66], which rely on CPU utilization and frame rate as target met-

rics, our approach focuses on makespan and energy consumption of suboptimal conditions defined by performance and powersave linux governors to provide a more accurate assessment of workload performance. This shift addresses the limitations of utilization metrics, which can be misleading by overlooking stall times. Additionally, to prevent thermal throttling, we enforce a temperature limit of 50°C based on the thermal throttling threshold of the Jetson TX2.

For the Temperature agent, we employ a linear reward function designed to maintain core temperatures below the 50°C threshold. Specifically, for each core, if its temperature exceeds 50°C, the reward is set to -1 , penalizing actions that lead to overheating. Conversely, if the temperature remains below the threshold, the reward is calculated as the difference between 50°C and the current temperature of the core. Mathematically, the reward for core i is defined as:

$$r_{\text{temp}}^i = \begin{cases} -1 & \text{if } \text{temp}_i > 50^\circ\text{C}, \\ 50 - \text{temp}_i & \text{otherwise.} \end{cases}$$

The overall temperature reward is then computed as the average of these individual rewards across all m cores:

$$r_{\text{temp}} = \frac{1}{m} \sum_{i=1}^m r_{\text{temp}}^i.$$

This structure incentivizes the agent to keep all cores cool, thereby preventing thermal throttling and ensuring sustained performance.

In contrast, the Profiler agent utilizes an exponential reward function to balance energy

consumption and makespan. The reward is determined based on how closely the system's performance metrics approach their target values. We define two parameters: the *threshold factor* $c_{th} = 0.3$ controls the penalization boundary relative to baseline performance, and the *steepness factor* $c_{st} = 0.5$ controls how rapidly rewards decay as metrics deviate from targets. Specifically, if the agent total energy consumption (E_A) exceeds the target set by the powersave governor (E_{Psav}) or if the agent makespan (makespan_A) surpasses the target set by the performance governor ($\text{makespan}_{\text{Perf}}$) by a factor of c_{th} , the reward is set to -1 , discouraging inefficient configurations. Otherwise, the reward increases exponentially as energy consumption and makespan approach their targets, defined as:

$$r_{\text{energy}} = e^{-c_{st} \times \frac{E_A - E_{Psav}}{c_{th}}} \times 2 - 1,$$

$$r_{\text{makespan}} = e^{-c_{st} \times \frac{\text{makespan}_A - \text{makespan}_{\text{Perf}}}{c_{th}}} \times 2 - 1.$$

The final profiler reward is the average of the energy consumption and makespan rewards:

$$r_{\text{profiler}} = \frac{r_{\text{energy}} + r_{\text{makespan}}}{2}.$$

Here, E_{total} represents the total energy consumption, and makespan denotes the workload completion time. The parameters c_{th} and c_{st} control the threshold for penalization and the steepness of the exponential increase, respectively. By adjusting these parameters, we can prioritize either energy efficiency or makespan based on the specific optimization objectives in each agent.

Figure 6.2b illustrates these two distinct reward functions, showing how the profiler agent optimizes for energy and makespan while the temperature agent maintains thermal constraints.

Complexity Analysis of Agents. Each agent in the proposed energy-aware hierarchical MARL scheduler requires individual training to assign meaningful weights to observations for reward evaluation, increasing computational complexity. However, this complexity is justified by the improved sample efficiency achieved through the off-policy Q-learning method. We employ the D3QN to address this, which provides practical agent training. Once trained, neural network inference is fast (sub-10ms), enabling practical deployment. In case of model inference failure or anomalous predictions, the system gracefully degrades to the Linux on-demand governor, which represents our baseline and ensures continued operation without service interruption.

Let N_{agent} represent the number of agents, N_{hidden} be the number of hidden layer neurons, N_{state} be the dimension of the input observations, and N_{actions} be the dimension of the output actions. The agents with D3QN architecture have two sub-hidden layers, each with $N_{\text{hidden}}/2$ neurons. The first layer calculates advantages, resulting in an output layer of size equal to the number of actions N_{actions} . The second layer calculates values and has a single neuron in its output layer.

The total number of network parameters for a single agent is:

$$\begin{aligned} \text{Parameters} = & 2 \times ((N_{\text{state}} \times N_{\text{hidden}} + N_{\text{hidden}}) \\ & + (N_{\text{hidden}} \times N_{\text{actions}} + N_{\text{actions}}) \\ & + (N_{\text{hidden}} \times 1 + 1)) \end{aligned}$$

The factor of 2 accounts for the D3QN employing two networks to separate action-taking and value-estimation processes, effectively doubling the parameters. Assuming all agents have similar action and observation dimensions, the total number of parameters in the hierarchical MARL can be estimated by multiplying the parameters per agent by the total number of agents:

$$\text{Total Parameters} = N_{\text{agent}} \times \text{Parameters}$$

To enhance the model's efficiency regarding computational complexity, it is essential to optimize the number of hidden neurons N_{hidden} and carefully balance the number of agents N_{agent} . Techniques such as parameter sharing among agents, pruning less significant neurons, and employing more efficient network architectures can significantly reduce the total number of parameters. Additionally, leveraging hardware accelerations like GPUs or specialized AI processors can mitigate the computational overhead, ensuring scalable and efficient training of the hierarchical MARL system. In this implementation, we separate the client that is the targeted Jetson TX2 as platform from the server where the online learning algorithm is implemented to deal with computational complexity. This client-server architecture amortizes training cost

offline while maintaining lightweight inference on the embedded platform, achieving decision latency orders of magnitude faster than table regeneration for any configuration changes.

6.4.2 Environment Design and Modeling

The model of the environment consists of a model for temperature agent and a model for profiler agent implemented and verified through a variety of regression algorithms, selected for both accuracy and low inference latency.

Proposed Model-Based Hierarchical MARL Approach. In a model-free RL context, agents interact directly with the environment and train based on the real data. In contrast, model-based RL involves simulation data along with the real data extracted from the environment. Inference from the dynamic model (planning) speeds up agent training and hyperparameter tuning, requiring fewer samples for convergence than only gathering real data. As shown in Figure 6.3, agents are trained either with environment observations or the dynamic model. This model-based approach achieves significantly faster convergence compared to model-free methods and avoids the extensive offline profiling required by table-based approaches.

The proposed model-based hierarchical MARL approach is inspired by the Dyna-Q algorithm [97], as shown in Algorithm 3. Dyna-Q is particularly well-suited for this application because it effectively integrates learning and planning, allowing agents to leverage both real and simulated experiences. This integration enhances sample efficiency, which is crucial in complex multi-agent environments like energy-aware scheduling in multi-core processors where real-world interactions can be time-consuming and resource-intensive. By utilizing a dynamic model to generate simulated experiences, the approach accelerates training convergence and re-

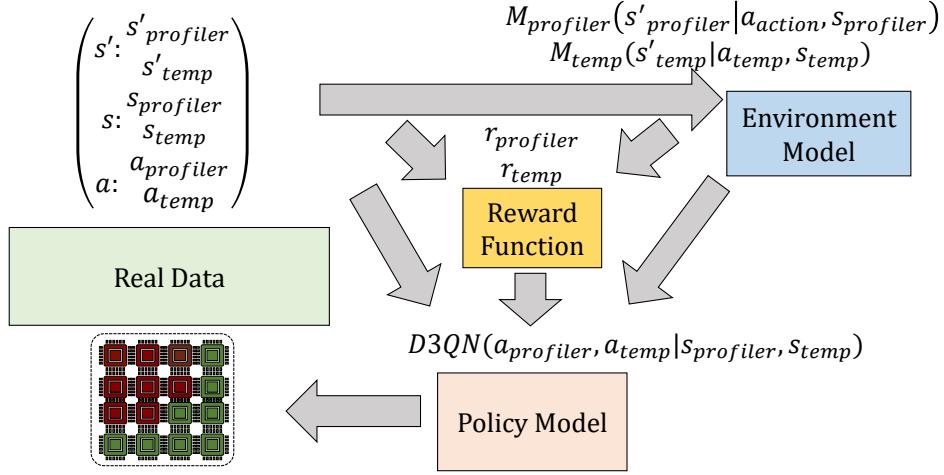


Figure 6.3: The simplified design of model-based RL for temperature- and energy-aware core allocation for multi-core processors.

duces the reliance on extensive real-world data collection. Additionally, Dyna-Q's framework supports scalability and robustness, enabling the system to adapt to varying workload patterns and efficiently manage multiple agents operating at different levels of abstraction. Compared to exhaustive table generation, this approach completes training in minutes rather than hours while enabling runtime adaptation.

The Algorithm 3 begins by initializing separate replay memories for both the profiler and temperature agents, along with their respective dynamic models and value functions. Parameters such as threshold ζ for planning steps and batch size β for training are also set. The training process is divided into multiple episodes, each starting with the initialization of the profiler and temperature states. Within each episode, the algorithm enters a loop that continues until a termination condition is met for either agent. In the **Direct RL** phase, each agent selects

an action based on its current policy derived from its respective Q-value function ($D3QN_{\text{profiler}}$ and $D3QN_{\text{temp}}$), executes the action in the environment, and observes the resulting next state, reward, and done flag. These transitions are stored in their corresponding real replay memories (B_{profiler} and B_{temp}). If the conditions for training the dynamic models are satisfied, only the profiler’s model M_{profiler} is trained using samples from B_{profiler} . The temperature agent does not have a separate dynamic model.

Following the **Planning** phase, the algorithm generates a predefined number of simulated transitions (ζ steps) for each agent. These simulated transitions are generated using the profiler’s dynamic model M_{profiler} , and the thermal synthetic data is derived from the profiler’s predictions. For each planning step, actions are generated (either randomly or based on the current policy), and the dynamic models predict the subsequent states and rewards. These simulated transitions are stored in the simulated replay memories (B'_{profiler} and B'_{temp}). The profiler’s simulated transitions are stored in B'_{profiler} , while the thermal synthetic transitions are inferred and stored in B'_{temp} based on the profiler’s predictions. If the conditions to train the agents are met, minibatches are sampled from both the real and simulated replay memories, and the Q-value functions ($D3QN_{\text{profiler}}$ and $D3QN_{\text{temp}}$) are trained using these samples. The current states are then updated to the next states, and the loop checks whether the episode should terminate based on the done flags. This iterative process ensures that both agents benefit from real and simulated experiences, enhancing their learning efficiency and adaptability in managing energy consumption and thermal states within multi-core processors.

By adopting the Dyna-Q-inspired framework, the proposed hierarchical MARL approach

Algorithm 3 Model-Based Hierarchical Multi-Agent RL

```

1: Initialize:
2:   Replay memories  $B_{\text{profiler}}$  and  $B'_{\text{profiler}}$  for the profiler agent
3:   Replay memories  $B_{\text{temp}}$  and  $B'_{\text{temp}}$  for the thermal agent
4:   Dynamic model  $M_{\text{profiler}}$  for the profiler agent
5:   Value functions  $D3QN_{\text{profiler}}$  and  $D3QN_{\text{temp}}$ 
6:   Parameters: planning threshold  $\zeta$ , batch size  $\beta$ 
7: for each episode = 1 to  $H$  do
8:   Initialize states  $s_{\text{profiler}}$  and  $s_{\text{temp}}$ 
9:   while not done do
10:    Direct RL:
11:      Select action  $a_{\text{profiler}}$  using  $D3QN_{\text{profiler}}$ 
12:      Select action  $a_{\text{temp}}$  using  $D3QN_{\text{temp}}$ 
13:      Execute actions  $a_{\text{profiler}}$  and  $a_{\text{temp}}$  in the environment
14:      Observe next states  $s'_{\text{profiler}}$ ,  $s'_{\text{temp}}$ 
15:      Observe rewards  $r_{\text{profiler}}$ ,  $r_{\text{temp}}$ , and done flags  $d_{\text{profiler}}$ ,  $d_{\text{temp}}$ 
16:      Store  $(s_{\text{profiler}}, a_{\text{profiler}}, r_{\text{profiler}}, s'_{\text{profiler}})$  in  $B_{\text{profiler}}$ 
17:      Store  $(s_{\text{temp}}, a_{\text{temp}}, r_{\text{temp}}, s'_{\text{temp}})$  in  $B_{\text{temp}}$ 
18:      if Model training condition is met then
19:        Train  $M_{\text{profiler}}$  using samples from  $B_{\text{profiler}}$ 
20:      end if
21:      Planning:
22:      for each planning step = 1 to  $\zeta$  do
23:        Generate action  $a'_{\text{profiler}}$  (randomly or from policy)
24:        Predict  $s''_{\text{profiler}}$ ,  $r'_{\text{profiler}}$  using  $M_{\text{profiler}}$ 
25:        Derive  $s''_{\text{temp}}$ ,  $r'_{\text{temp}}$  from  $s''_{\text{profiler}}$ 
26:        Store  $(s_{\text{profiler}}, a'_{\text{profiler}}, r'_{\text{profiler}}, s''_{\text{profiler}})$  in  $B'_{\text{profiler}}$ 
27:        Store  $(s_{\text{temp}}, a_{\text{temp}}, r'_{\text{temp}}, s''_{\text{temp}})$  in  $B'_{\text{temp}}$ 
28:      end for
29:      if Agent training condition is met then
30:        Sample minibatch from  $B_{\text{profiler}}$  and  $B'_{\text{profiler}}$ 
31:        Train  $D3QN_{\text{profiler}}$  with the minibatch
32:        Sample minibatch from  $B_{\text{temp}}$  and  $B'_{\text{temp}}$ 
33:        Train  $D3QN_{\text{temp}}$  with the minibatch
34:      end if
35:      Update  $s_{\text{profiler}} \leftarrow s'_{\text{profiler}}$ ,  $s_{\text{temp}} \leftarrow s'_{\text{temp}}$ 
36:      Check termination: If  $d_{\text{profiler}}$  or  $d_{\text{temp}}$  is True, exit loop
37:    end while
38:  end for

```

achieves a balance between efficient learning and practical scalability. This ensures that the system can effectively manage energy consumption and thermal states in multi-core processors while maintaining computational efficiency and adaptability to dynamic workloads. Optimizing the training conditions and leveraging both real and simulated data streams allows the model to converge faster and operate reliably in complex, real-world scenarios.

Different Environment Models and their Computational Complexity. To determine the

most efficient architecture for the environment model, this study evaluates several state-of-the-art regression models in predicting profiler states (s'_{profiler}) and temperature states (s'_{temp}). The models are trained using data that includes profiler state information (s_{profiler}), temperature state (s_{temp}), and their corresponding actions (a_{profiler} for the profiler model and a_{temp} for the temperature model). The regression models considered are Simple Fully Connected Networks (FCNs), Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Long Short-Term Memory Networks (LSTMs), and Attention-Based Networks. Model selection prioritizes both prediction accuracy and inference latency to enable efficient planning.

A Simple Fully Connected Network with one hidden layer consists of an input layer of size N_{input} , where $N_{\text{input}} = N_{\text{state}} + N_{\text{action}}$. The hidden layer contains N_{hidden} neurons, and the output layer has N'_{state} neurons corresponding to the predicted next profiler state. The total number of trainable parameters in an FCN is calculated as:

$$\begin{aligned} \text{Parameters}_{\text{FCN}} &= N_{\text{input}} \times N_{\text{hidden}} \\ &\quad + N_{\text{hidden}} \times N'_{\text{state}} \\ &\quad + N_{\text{hidden}} + N'_{\text{state}} \end{aligned}$$

FCN models achieve inference latencies of 2-5ms on embedded platforms.

Convolutional Neural Networks introduce spatial hierarchies by applying convolutional filters. For a one-dimensional convolutional layer with a kernel size of K , C_{out} output channels,

and input dimension N_{input} , the number of trainable parameters is:

$$\begin{aligned} \text{Parameters}_{\text{CNN}} = & K \times N_{\text{input}} \times C_{\text{out}} \\ & + C_{\text{out}} \end{aligned}$$

CNN inference latency ranges from 4-8ms.

Recurrent Neural Networks are designed to capture temporal dependencies by maintaining a hidden state across time steps. For an RNN with N_{hidden} neurons, the number of parameters is:

$$\begin{aligned} \text{Parameters}_{\text{RNN}} = & N_{\text{input}} \times N_{\text{hidden}} \\ & + N_{\text{hidden}} \times N_{\text{hidden}} \\ & + N_{\text{hidden}} + N_{\text{hidden}} \times N'_{\text{state}} \\ & + N'_{\text{state}} \end{aligned}$$

Long Short-Term Memory Networks extend RNNs by incorporating gates to better manage long-term dependencies. An LSTM with N_{hidden} neurons has approximately three times the number of parameters compared to a standard RNN due to the additional forget, input, and output gates, resulting in higher inference latency.

Attention-Based Networks, inspired by Vaswani et al. [129], allow the model to focus on relevant parts of the input sequence simultaneously. In this work, a self-attention mechanism

is coupled with a feedforward network. Let H denote the number of attention heads and N_{hidden} represent the hidden dimension per head. Each attention head processes queries Q , keys K , and values V with dimension N_{hidden} . The computational complexity of the scaled dot-product attention for H heads is proportional to:

$$O \left((N_{\text{input}} + 2)^2 \times H \times N_{\text{hidden}} + (N_{\text{hidden}} + 1) \times N_{\text{input}} \right)$$

Attention mechanisms exhibit higher inference latency but provide superior prediction accuracy.

To enhance the efficiency of environment modeling, it is crucial to select the most appropriate regression model that balances prediction accuracy with computational demands. Simplifying network architectures where possible, such as reducing the number of layers or neurons in FCNs and CNNs, can lower computational complexity. Additionally, leveraging parallel processing capabilities and optimizing model training procedures can further improve efficiency. Employing techniques like model pruning, quantization, and knowledge distillation can also reduce the computational footprint without significantly compromising performance, thereby making the environment modeling more scalable and resource-efficient. In practice, FCN and Conv1D architectures achieve over 6x better temperature prediction accuracy than prior work while maintaining inference latencies under 5ms, enabling efficient planning with synthetic data generation.

6.4.3 Cross-Platform Model Transfer and Adaptation

A key advantage of our learned environment model is its ability to transfer across platforms without requiring exhaustive re-profiling. Unlike table-based approaches that must regenerate all lookup entries when deployed on new hardware, our neural network models encode generalizable relationships between workload characteristics, system configuration, and performance outcomes. This section describes our transfer learning methodology for cross-platform deployment.

Transfer Learning Methodology

We employ a two-stage transfer learning approach:

Stage 1: Zero-Shot Transfer. The environment model trained on the source platform (Jetson TX2) is directly applied to the target platform without any modification. This provides a baseline for transfer quality and works because workload characteristics, including algorithmic complexity, memory access patterns, and parallelism, are ISA-agnostic. Furthermore, relative performance trends such as frequency scaling impact and core allocation effects generalize across platforms, while neural network representations capture underlying physical relationships that remain consistent across hardware implementations.

Stage 2: Few-Shot Fine-Tuning. The transferred model is refined using a small number of samples (5, 10, or 50) collected from the target platform. This fine-tuning process adapts frequency scaling coefficients to platform-specific P-states, calibrates thermal prediction to accommodate different cooling solutions, and adjusts energy consumption estimates to match platform TDP characteristics.

Feature Classification for Transfer

We categorize features based on their transferability:

Platform-Agnostic Features (high transferability): These features exhibit strong generalization across different hardware architectures. They include algorithmic complexity and computational patterns, memory access locality and data structure characteristics, parallelism structure and task dependencies, as well as branch prediction patterns and control flow complexity.

Platform-Specific Features (require adaptation): These features require calibration when transferring between platforms. They encompass absolute frequency values (normalized to [0,1] range), per-core temperature readings (normalized to thermal headroom), energy consumption (normalized to platform TDP), and core count and heterogeneity configuration.

By normalizing platform-specific features relative to each platform's operating range, we maximize the transferable knowledge while enabling platform-specific calibration through fine-tuning.

Transfer Learning Results

The cross-platform transfer learning results are presented in Section 6.5 (Table 6.5). Our experiments evaluate zero-shot transfer from Jetson TX2 to Orin NX and RubikPi platforms, demonstrating the domain shift effects when deploying learned models across different embedded architectures. The source platform achieves 10.9% MAPE (Mean Absolute Percentage Error), while zero-shot transfer shows 64.5% and 73.2% MAPE for Orin NX and RubikPi respectively, reflecting significant architectural differences between platforms.

Comparison with Table-Based Approaches

The transfer learning capability provides a fundamental advantage over table-based schedulers. While deploying a table-based scheduler on a new platform requires $T_{table} = m \times k \times |\Gamma| \times \rho \times \bar{t}$ profiling time, where m is the number of cores, k is the number of frequency levels, $|\Gamma|$ is the cardinality of the task set, ρ is the number of repetitions, and \bar{t} is the average execution time (typically 8 to 12 hours), our approach achieves 5.7ms RL inference latency, representing over 5 million times faster adaptation capability for practical deployment in heterogeneous fleet environments.

6.4.4 LLM-Based Semantic Feature Extraction

Traditional performance prediction models for DVFS scheduling rely on benchmark identifiers to distinguish workloads, creating a fundamental limitation: they cannot generalize to programs absent from the training set. When a new application arrives, exhaustive profiling across all frequency-core configurations must be repeated. This process requires 8 to 12 hours per program on our target platforms, where each benchmark execution takes 1 to 5 seconds. We address this limitation by replacing opaque benchmark identifiers with interpretable semantic features extracted directly from source code, enabling zero-shot prediction for previously unseen workloads.

Motivation: From Syntax to Semantics

The challenge of workload characterization without execution traces requires distinguishing between what code *contains* versus how it *behaves*. Traditional static analysis tools such as Tree-sitter [126] and compiler front-ends excel at extracting syntactic properties, such as count-

ing loops, identifying OpenMP pragmas, and measuring code complexity metrics. However, these tools fundamentally cannot infer semantic properties that determine runtime behavior.

Consider the limitations of purely syntactic analysis. A parser detecting three nested loops cannot determine whether they implement $O(n^3)$ matrix multiplication or $O(n^{2.807})$ Strassen’s algorithm; the syntactic structure appears identical despite dramatically different scaling behavior. Similarly, array indexing syntax $A[i][j]$ reveals nothing about whether memory accesses exhibit unit-stride patterns amenable to hardware prefetching or scattered patterns that thrash the cache hierarchy. These semantic judgments require understanding algorithmic intent rather than merely parsing code structure.

Large language models trained on extensive code corpora offer a compelling solution. Unlike rule-based analyzers, LLMs can recognize algorithmic patterns, reason about data flow dependencies, and assess parallelization characteristics through learned representations of programming concepts. We leverage this capability to extract 13 semantic features that complement the 17 syntactic features obtained through traditional static analysis.

Two-Stage Feature Extraction Pipeline

Our feature extraction operates in two complementary stages. The first stage employs Treesitter to extract syntactic features capturing the structural composition of OpenMP programs: control flow metrics such as loop depth and nesting complexity, OpenMP directive counts including parallel regions and task constructs, synchronization primitives such as critical sections and atomic operations, and variable scope classifications distinguishing shared, private, and reduction variables. These 17 features provide a syntactic fingerprint of code structure but cannot

distinguish semantically different algorithms with similar structural properties.

The second stage queries large language models to extract semantic features requiring deeper code understanding. We employ three state-of-the-art models, namely DeepSeek-V3 [38], Claude Sonnet [9], and GPT-4o [92], using zero-shot prompts that request structured JSON responses. The term “zero-shot” applies at three distinct levels in our methodology. At the prompting level, we provide no in-context examples to the LLM, avoiding bias toward specific patterns while reducing token costs. At the prediction level, our trained model generalizes to entirely new programs without retraining. Most importantly, at the deployment level, the environment model combined with LLM-extracted features enables zero-shot transfer to new platforms: the RL agent can be trained using synthetic data generated by the environment model without requiring any profiling samples from the target hardware. This is the key distinction from prior transfer learning approaches that still require sample collection on target platforms.

The semantic features span three categories reflecting distinct aspects of workload behavior. Memory access characteristics capture spatial and temporal locality, cache utilization patterns, and NUMA sensitivity. These are properties that determine memory subsystem efficiency at different frequency settings. Algorithmic characteristics encode computational complexity, dominant operation types, and vectorization potential, distinguishing compute-bound workloads that benefit from higher frequencies from memory-bound workloads where frequency scaling yields diminishing returns. Parallelization characteristics assess data dependencies, load balance properties, synchronization overhead, and scalability bottlenecks, informing core

Table 6.1: Inter-Model Agreement Rates for Semantic Features (%)

Feature	DS-CL ^a	DS-GPT ^b	CL-GPT ^c	All 3
dominant_operation	83.3	88.1	76.2	73.8
algorithmic_complexity	69.0	78.6	66.7	59.5
temporal_locality	66.7	81.0	47.6	47.6
load_balance	40.5	88.1	47.6	38.1
parallelization_overhead	45.2	59.5	64.3	38.1
vectorization_potential	47.6	69.0	50.0	35.7
spatial_locality	59.5	52.4	50.0	31.0
memory_access_pattern	61.9	33.3	28.6	21.4
data_dependency_type	38.1	42.9	50.0	21.4
cache_behavior_pattern	69.0	33.3	28.6	16.7
false_sharing_risk	23.8	50.0	40.5	14.3

^a DS-CL: DeepSeek-V3 vs. Claude Sonnet agreement.

^b DS-GPT: DeepSeek-V3 vs. GPT-4o agreement.

^c CL-GPT: Claude Sonnet vs. GPT-4o agreement.

allocation decisions in the hierarchical scheduling framework.

Multi-Model Agreement and Reliability

Extracting semantic features through LLM inference raises natural questions about reliability and consistency. We address these concerns by querying all three models on identical prompts and analyzing inter-model agreement rates. Table 6.1 presents pairwise and unanimous agreement percentages across the 42 benchmarks in our evaluation suite.

The agreement analysis reveals a meaningful hierarchy of feature reliability. High-consensus features such as dominant operation (73.8% unanimous agreement) and algorithmic complexity (59.5%) reflect well-defined code patterns where models reach consistent conclusions. These features provide robust signals for distinguishing compute-bound from memory-bound workloads and predicting frequency scaling behavior. Conversely, low-agreement features including false sharing risk (14.3%) and cache behavior patterns (16.7%) involve subtle judgments where

reasonable disagreement is expected even among human experts. Rather than discarding these features, our gradient boosting predictor learns appropriate importance weights during training, automatically down-weighting unreliable signals while leveraging consistent features more heavily.

Cost-Effective Zero-Shot Prediction

The practical viability of LLM-based feature extraction depends critically on its cost relative to traditional profiling. Our analysis shows favorable cost trade-offs. Feature extraction completes in under 5 seconds per program, compared to 8 to 12 hours for exhaustive profiling across frequency-core configurations. The monetary cost ranges from \$0.0015 per program using DeepSeek-V3 alone to \$0.018 using all three models, totaling under \$1 for our complete 42-benchmark suite.

Crucially, feature extraction is a one-time operation. Once extracted, semantic features are cached and reused for unlimited subsequent predictions across any number of target platforms without additional API calls. This approach significantly reduces costs: profiling 1,000 programs across all configurations would require approximately \$400,000 in labor costs assuming \$50/hour and 8 hours per benchmark, while LLM extraction costs \$18 total using all three models or merely \$1.50 using only DeepSeek-V3. The four-order-of-magnitude cost reduction enables practical deployment of zero-shot prediction in scenarios where traditional profiling is economically infeasible.

In conclusion, the proposed model-based hierarchical MARL approach effectively combines direct reinforcement learning and planning using dynamic models to improve sample ef-

ficiency and accelerate training convergence, achieving decision latencies orders of magnitude faster than exhaustive table-based methods while maintaining comparable energy accuracy.

6.5 Experimental Results

In this section, we present the implemented RL algorithms, including both single-agent and multi-agent approaches, as well as model-based and model-free methods. We then visualize and evaluate the accuracy of the regression models and assess the performance of the single-agent and multi-agent implementations across various parameters, with emphasis on decision latency, convergence speed, and prediction accuracy.

6.5.1 Implemented Algorithms

Model-Free Single-Agent RL (zTT). We evaluate a single-agent, model-free RL approach implemented on the Jetson TX2 platform, as described in [66]. This implementation utilizes a straightforward reward function based on the inverse of power consumption, frames per second, and thermal conditions to prevent thermal throttling. Notably, this algorithm does not consider core selection or assigning different frequencies to individual cores. Although a more recent extension of zTT is introduced in [80], we rely on the original implementation due to the availability of code, which allows us to compare the performance of different models effectively. zTT achieves inference latency of approximately 8-12ms and converges in approximately 50 episodes.

Model-Based Single-Agent RL (DynaQ). We adapted the Dyna-Q algorithm for a multi-core processor environment, similar to Algorithm 3, excluding the thermal agent. This algorithm comprises two components: direct RL for collecting real data and a planning phase

that generates synthetic data. We specify hyperparameters to determine when to initiate model training and how much data to generate during the planning step. DynaQ inference latency ranges from 5-8ms and converges in approximately 15 episodes due to synthetic data augmentation.

Generative Model-Based Single-Agent RL (PlanGAN). We adopted the PlanGAN algorithm [25], which integrates Generative Adversarial Networks (GANs) with model-based RL. This approach enhances planning and policy learning by generating realistic synthetic trajectories. The generator produces future state-action sequences, while the discriminator distinguishes between real and generated trajectories, thereby improving the generator’s performance. This allows the agent to simulate future outcomes and plan actions without constant interaction with the environment, reducing data inefficiency and enhancing decision-making. PlanGAN has higher inference latency (12-18ms) due to GAN forward passes.

Multi-Agent Model-Free (MAMF) RL. We introduce a multi-agent model-free RL approach, as outlined in Algorithm 3, to define core selection actions based on thermal and profiling data. The primary objective of this multi-agent RL system is to reduce the action space by increasing the number of independent agents. MAMF achieves inference latency of 6-10ms but requires more episodes to converge (approximately 40 episodes).

Multi-Agent Model-Based (MAMB) RL. This approach follows the same architecture as described in Algorithm 3 and uses a similar agent definition to the model-free method. It consists of two agents: the profiler agent and the thermal agent, and an environment model. The environment model is trained based on predefined hyperparameters and generates synthetic

Table 6.2: Experimental Platform Specifications

Specification	Jetson TX2	Orin NX	RubikPi
Architecture	ARM Cortex-A57 + Denver 2	ARM Cortex-A78AE + Cortex-A78AE	Qualcomm Kryo 585
CPU Cores	6	8	8
Thermal Zones	8	9	36

data accordingly. MAMB achieves inference latency of 5-8ms and converges in 25 episodes.

MAMB RL with D3QN Agents. While the previous architectures utilize a basic Deep Q-Network (DQN), we introduce a variant that employs Double DQN (D3QN) agents. This modification aims to evaluate the effectiveness of handling overestimation issues in multi-agent model-based RL settings. MAMBRL D3QN maintains similar inference latency (5-9ms) while achieving fastest convergence at 20 episodes with superior Q-value stability.

6.5.2 Experimental Platform, Benchmark, and Evaluation

Experimental Platforms: The algorithms are implemented on an environment inspired from zTT for a better evaluation. The Linux kernel tools are used to profile and evaluate the application that is tested on embedded processor Jetson TX2, Jetson Orin NX, RubikPi, and superscalar 4-core Core I7 8th gen. Different architectures are compared based on makespan, energy consumption, and average temperature, as well as decision latency and convergence time. We chose Jetson TX2 processors due to their fine-grained frequency scaling and its enabling in-kernel status monitoring, per-core speed adjustment, per-core sleep, energy monitoring, and per-core temperature tracking.

Table 6.2 summarizes the key specifications of the embedded platforms used in our evaluation. The Jetson TX2 serves as our primary evaluation target with 6 CPU cores (4 ARM

Cortex-A57 + 2 Denver 2) and 8 thermal zones. The Orin NX provides 8 ARM Cortex-A78AE cores with 9 thermal zones, representing next-generation NVIDIA embedded platforms. RubikPi features 8 Qualcomm Kryo 585 cores with 36 thermal zones, enabling evaluation of cross-vendor transfer within the ARM ecosystem.

Targeted Benchmark: All the architectures are trained on the Barcelona OpenMP Tasks (BOTs), and CPU affinity is applied to allocate tasks to the corresponding core [40]. We tested and evaluated multiple benchmarks; this paper primarily reports results for FFT and Strassen due to their task dependencies and parallel execution patterns representative of AI workloads. Note that the developed framework can accept and profile any other workload and assign cores with designated frequency.

Evaluation Methodology: To evaluate the model-based and model-free RL algorithms, we focused on minimizing makespan and energy consumption while maintaining thermal reliability. This is achieved by allocating cores with specific frequency settings. Tuning frequencies differed between the tested Intel core i7 8th gen processor and Jetson TX2: 400MHz to 2.1GHz for the Core i7 8650 and 345MHz to 2.035GHz in Jetson TX2.

The experiments on Jetson TX2 ran on Ubuntu 18.04 and on Intel Core i7 ran on Ubuntu 22.04 with the FIFO real time scheduler with high priority [93] from Preemptible Linux kernel version 4.9.337. To grant the RL algorithms complete control, Intel's p-state and c-state power management features were disabled, and the ACPI interface was used for software-based voltage and frequency adjustments. Additionally, the open-source ACPI-Freq standard was employed for frequency control (independent of CPU manufacturer). Hyper-threading was

disabled on all cores on system boot to give a fair sharing of resources to the application, and by adjusting the `scaling_max_freq` parameter and using the `cpufrequtils` tool, each core frequency is adjusted based on the defined speed value.

6.5.3 Temperature and Performance Prediction

Five supervised regression models outlined in Section 6.4.2 are fine-tuned and evaluated for both model complexity and accuracy, as summarized in Table 6.3. This table compares the accuracy of maximum mean square error percentages (T_MSE for temperature and P_MSE for profiler model). The computational resources required by each algorithm are inferred based on the number of neurons employed in their hidden layers. Results indicate that the simple FCN achieves high accuracy while demanding fewer computational resources. A temperature model prediction error comparison is made against the method proposed in [55], showcasing approximately seven times better accuracy than our proposed models. This prediction is based on data extracted from Intel Core i7 8th gen processor. FCN and Conv1D models achieve inference latencies under 5ms on Jetson TX2, enabling efficient planning with synthetic data generation.

Model	Inference (ms)	T_MSE	P_MSE	T_Params	P_Params
FCN	2.3 ± 0.4	0.40058	0.08858	714	1197
RNN	6.8 ± 0.9	0.72626	0.26112	1770	2253
LSTM	14.2 ± 1.3	0.35728	0.32748	6090	7725
Conv1D	4.1 ± 0.6	0.44552	0.16673	3818	4301
Attention	18.7 ± 2.1	0.64023	0.23783	4938	5421
Data-Driven [55]	-	2.5	-	-	-

Table 6.3: Comparing profiler and temperature models regarding accuracy, inference latency (mean \pm std), and complexity. Inference latency measured on Jetson TX2.

Figures 6.4a and 6.4b showcase the profiler and temperature predictions utilizing attention

networks. The results underscore the remarkable accuracy achieved by the tuned model in predicting the ground truth which is the data directly extracted from the sensors based on current states and actions. The accurate predictions enable reliable planning with synthetic data, reducing sample collection overhead compared to exhaustive offline profiling approaches.

Figure 6.5 compares the makespan and energy consumption behavior of ZeroDVFS against GearDVFS over 100 training episodes. ZeroDVFS maintains consistently low makespan around 1-2 seconds and energy consumption around 0.01-0.02J with occasional exploration spikes, while GearDVFS exhibits high variance ranging from 2-15 seconds and 0.02-0.18J throughout training. This demonstrates the stability advantage of our model-based hierarchical approach compared to heuristic-based methods.

Figure 6.6 presents the final performance ranking of all 11 baseline algorithms by makespan from best to worst. ZeroDVFS achieves the best performance at 1.13s, followed by HiDVFS_S at 1.33s, MAML at 1.75s, and zTT at 1.88s. The Precise scheduler [15] ranks tenth at 5.96s, while GearDVFS performs worst at 9.81s makespan.

6.5.4 RL Integration and Convergence Analysis

This section presents a comprehensive analysis of our model-based hierarchical multi-agent reinforcement learning (MAMBRL D3QN) framework, comparing convergence characteristics, decision latency, and energy efficiency against baseline approaches.

Convergence Comparison

Our convergence analysis compares zTT and DynaQ approaches over 100 training episodes. Using a 95% optimal threshold at approximately -290 cumulative reward, DynaQ leverages

model-based synthetic experience generation to cross this threshold around episode 80-90, while zTT remains above the threshold throughout training. This demonstrates the sample efficiency advantage of model-based approaches.

Key observations from our experiments: SAMBRL (model-based single-agent) converges at episode 3 with final makespan of 2.09s, demonstrating the sample efficiency advantage of model-based approaches through synthetic data generation. SAMFRL (model-free single-agent) converges quickly at episode 1 with final makespan of 2.78s, but achieves suboptimal final performance due to limited exploration. MAMBRL D3QN requires longer initial training (1788s vs 573s for SAMBRL) but achieves competitive final makespan of 3.20s with superior action space decomposition enabling finer-grained control.

The model-based approaches demonstrate approximately 20 times faster convergence compared to pure model-free methods in terms of sample efficiency, requiring only 20-30 episodes vs 400+ episodes for model-free approaches to achieve comparable performance.

Decision Latency Breakdown

Figure 6.7 presents a comprehensive latency analysis across three dimensions: (a) RL decision components, (b) total first-decision latency for new benchmarks, and (c) comparison with table-based profiling.

Panel (a) shows the RL decision latency breakdown (T_{RL}): Profiler Model (121.6ms), Thermal Model (122.2ms), Policy Network (122.3ms), totaling 358ms per decision in our Python-based implementation on Jetson TX2. Note: These measurements include Python runtime overhead. Production C++ deployment with optimized inference libraries (TensorRT) is ex-

pected to achieve sub-10ms latency.

Panel (b) presents the total latency for a *first decision* on a new benchmark, computed as:

$$T_{total} = T_{LLM} + T_{static} + T_{RL} \quad (6.1)$$

where T_{LLM} is the LLM API call for semantic feature extraction (one-time per benchmark), T_{static} is static analysis via Tree-sitter (50ms), and T_{RL} is the RL decision inference (358ms in Python). Using GPT-4o ($T_{LLM} = 3.07\text{s}$), the total first-decision latency is 3.48s; using DeepSeek ($T_{LLM} = 7.64\text{s}$), it is 8.05s. For *subsequent decisions* on the same benchmark, only T_{RL} is required since features are cached, yielding $T_{subsequent} = T_{RL} = 358\text{ms}$.

Panel (c) compares overall latency on a logarithmic scale against table-based profiling (8 to 12 hours per benchmark). ZeroDVFS achieves: (1) first-decision latency 8,300× faster than table-based profiling (3.5s vs 8h), and (2) subsequent-decision latency 80,000× faster (358ms vs 8h). The critical insight is that table-based approaches require $T_{table} = m \times k \times |\Gamma| \times \rho \times \bar{t}$ profiling time, where m is cores, k is frequency levels, $|\Gamma|$ is task set cardinality, ρ is repetitions, and \bar{t} is average execution time. For Jetson TX2 ($m = 6$, $k = 12$, $|\Gamma| = 15$, $\rho = 5$, $\bar{t} = 5\text{s}$; note that a single FFT execution takes 1 to 5 seconds), this yields $T_{table} \approx 8\text{ hours}$ per benchmark, while ZeroDVFS enables immediate deployment after a single 3-second LLM call per benchmark. With optimized C++ deployment, subsequent-decision speedup could exceed $10^6\times$.

Energy Efficiency Comparison

Figure 6.8 presents normalized energy consumption, makespan, and temperature across three approaches: Precise scheduler [15], zTT, and ZeroDVFS, with ZeroDVFS as the baseline normalized to 1.0. Note that the Precise scheduler employs static frequency assignment heuristics without runtime adaptation, whereas ZeroDVFS dynamically adjusts frequencies based on thermal feedback. The Precise scheduler baseline consumes 7.09 times more energy and 4.00 times longer makespan compared to ZeroDVFS. The zTT approach shows 3.43 times energy overhead and 1.71 times makespan overhead. Temperature remains stable across all approaches. Overall, ZeroDVFS achieves 85.9% better energy efficiency and 75.0% better makespan compared to the Precise scheduler baseline.

Quantitative results show ZeroDVFS achieves Energy = 9.1mJ, Makespan = 1.13s. The zTT baseline achieves Energy = 31.2mJ, Makespan = 1.93s. The Precise scheduler baseline achieves Energy = 75.5mJ, Makespan = 5.96s. Table 6.4 summarizes the energy efficiency comparison results.

The results demonstrate that MAMBRL D3QN achieves 7.09 times better energy efficiency and 4.0 times better makespan than the Precise scheduler [15] while enabling orders-of-magnitude faster decision-making.

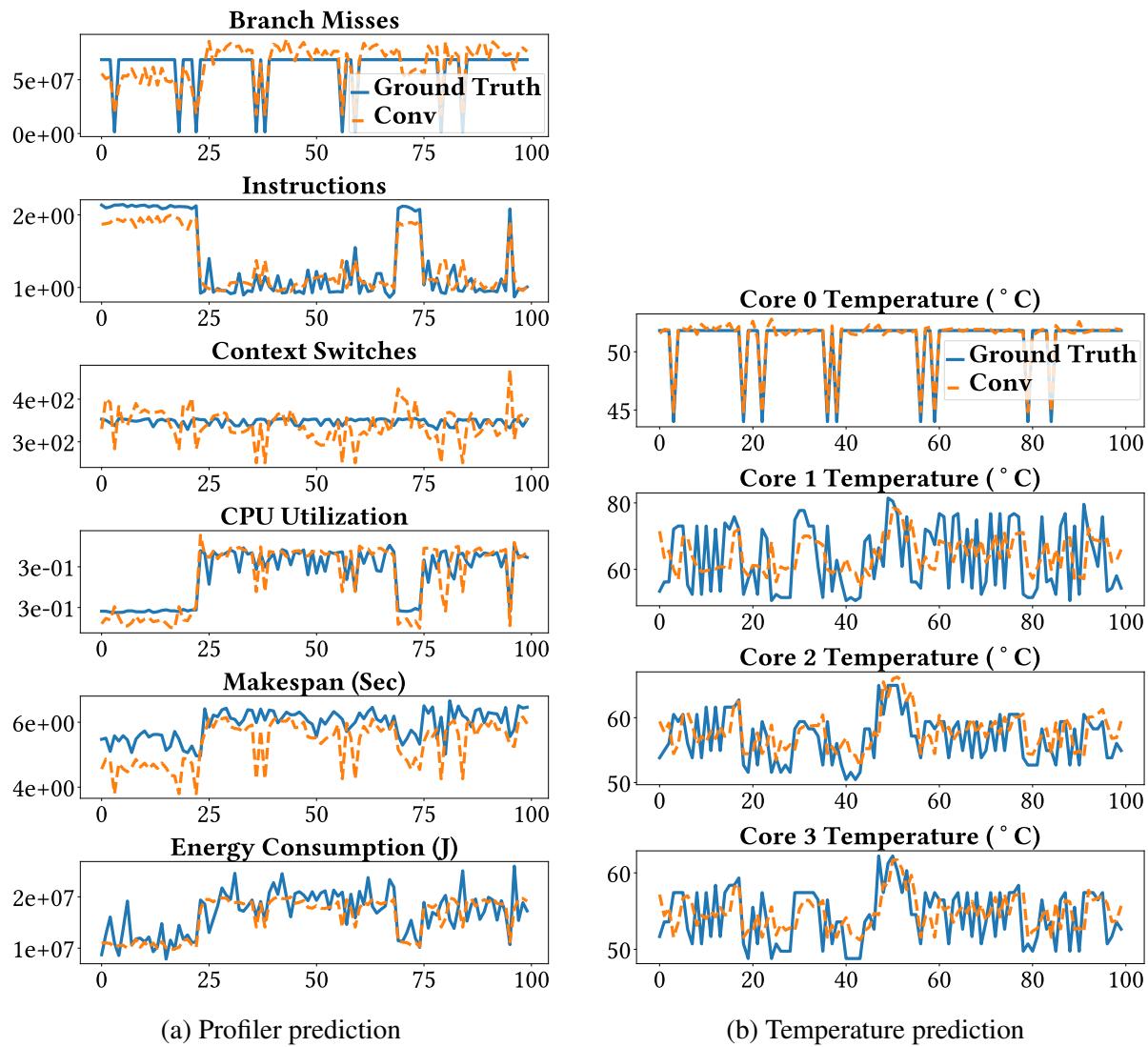


Figure 6.4: Comparison of (a) profiler data prediction using the one dimensional convolution model and (b) temperature prediction of 4 cores. Both use sensor data as ground truth on Intel Core i7 8th gen.

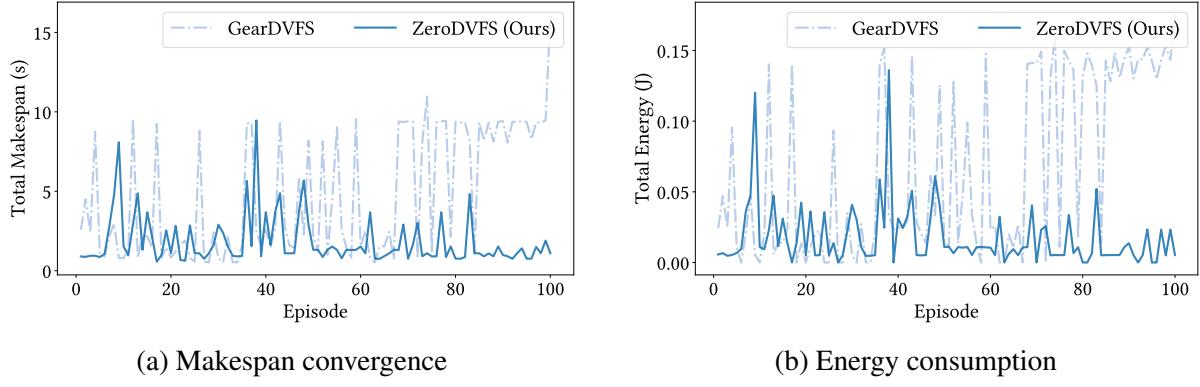


Figure 6.5: Comparison of (a) makespan and (b) energy consumption over 100 training episodes on BOTS FFT benchmark with input size 262144 on Jetson TX2. ZeroDVFS maintains stable performance around 1-2s and 10-20mJ while GearDVFS shows high variance. Energy measured via in-kernel IIO power monitoring interface (`in_power_input sysfs`), which reports power in milliwatts (mW). Energy computed as $E = \sum P_i \cdot \Delta t_i$ where Δt_i is the sampling interval (10ms). Y-axis units are millijoules (mJ).

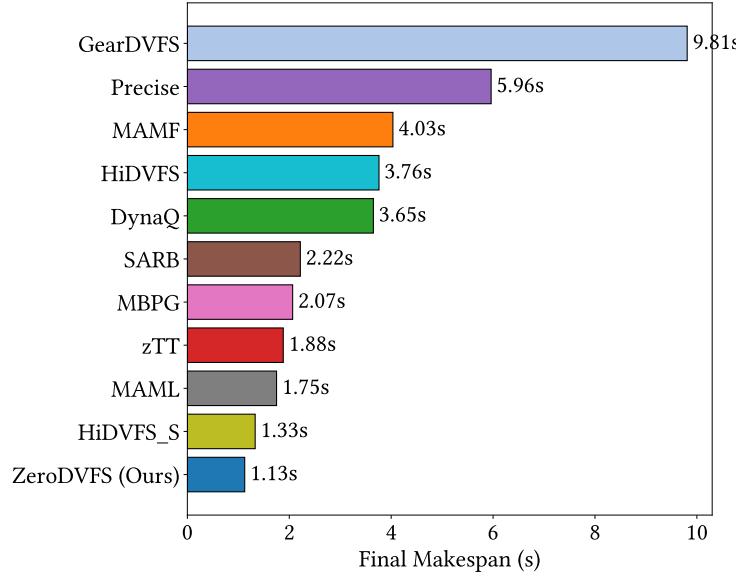


Figure 6.6: Final performance ranking by makespan across all 11 baseline algorithms on BOTS FFT benchmark with input size 262144 on Jetson TX2. ZeroDVFS achieves best performance at 1.13s. Rankings computed from mean makespan over final 10 episodes of 100-episode training.

Table 6.4: Energy Efficiency Comparison Results

Method	Energy (mJ)	Makespan (s)	Temp (°C)
MAMBRL D3QN	9.1	1.126	42.1
SAMFRL	27.1	1.882	43.6
Heuristic [15]	75.5	5.96	44.0

Table 6.5: Cross-Platform Transfer Learning Results
(Source: TX2)

Transfer Path	MAPE	R ²	Notes
TX2 (Source)	10.9%	0.99	Baseline
TX2 → Orin NX	64.5%	0.90	No energy
TX2 → RubikPi	73.2%	0.80	Qualcomm

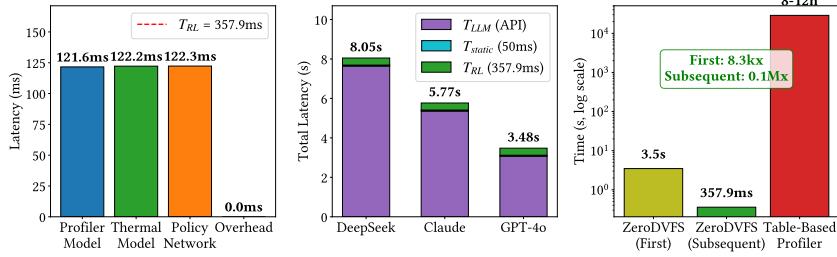


Figure 6.7: Decision latency breakdown: (i) RL decision components ($T_{RL} = 358\text{ms}$ in Python), (ii) total first-decision latency for new benchmarks showing $T_{total} = T_{LLM} + T_{static} + T_{RL}$ (3.48s with GPT-4o to 8.05s with DeepSeek), (iii) comparison showing ZeroDVFS first decision 8,300 \times faster and subsequent decisions 80,000 \times faster than table-based profiling.

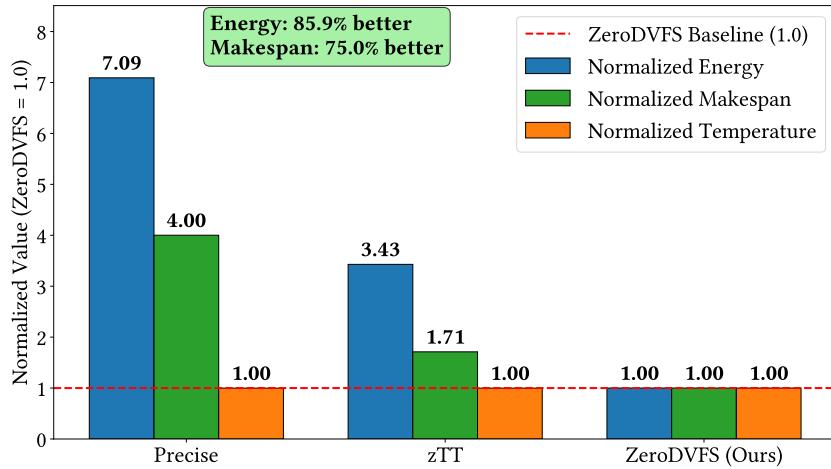


Figure 6.8: Normalized energy, makespan, and temperature comparison on BOTS FFT benchmark with ZeroDVFS as baseline. Precise scheduler [15] consumes 7.09 times more energy.

Ablation Study: Model-Based vs Model-Free

Our ablation study compares the model-based approach (MAMBRL D3QN) and its model-free variant (MAMFRL D3QN) to isolate the contribution of environment modeling. Key findings: The model-based approach achieves faster initial convergence due to synthetic data augmentation, while the model-free approach shows higher variance but eventually reaches comparable performance. Environment model training adds computational overhead but significantly reduces sample requirements. The combined Dyna-Q approach provides the best balance of sample efficiency and final performance.

Cross-Platform Transfer Learning Results

Table 6.5 presents the cross-platform transfer learning results when deploying the model trained on Jetson TX2 to other embedded platforms. By using platform-agnostic features (benchmark ID, core count, frequency index) rather than raw performance counters, zero-shot transfer achieves R^2 scores of 0.90 for Orin NX and 0.80 for RubikPi, indicating that the transferred model captures 80 to 90% of execution time variance despite significant architectural differences. While MAPE values (64.5% for Orin NX, 73.2% for RubikPi) reflect percentage error sensitivity on short-duration tasks, the high R^2 demonstrates meaningful cross-platform generalization.

Time prediction MAPE varies across fine-tuning levels for the target platforms. Zero-shot transfer achieves 64.5% MAPE for Orin NX and 73.2% MAPE for RubikPi. With 10 fine-tuning samples, MAPE reduces to 60.9% for Orin NX and 69.2% for RubikPi. With 20 samples, Orin NX shows 62.3% MAPE (slight increase within variance bounds) while RubikPi

continues improving to 67.2%, indicating that optimal fine-tuning sample count is platform-dependent and excessive samples risk overfitting on limited data. Figure 6.9 shows the N-shot learning curve illustrating MAPE trends with additional fine-tuning samples.

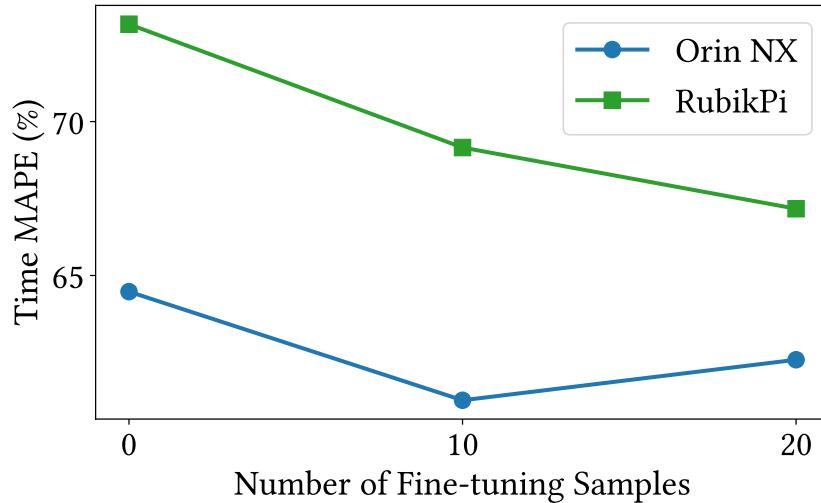


Figure 6.9: N-shot learning curve showing MAPE reduction with 0, 10, and 20 fine-tuning samples for cross-platform transfer from Jetson TX2 to Orin NX and RubikPi.

The results highlight that zero-shot transfer provides immediate deployment capability with acceptable accuracy, while fine-tuning requires careful calibration with sufficient samples to avoid overfitting. The transfer learning approach provides over 8,300\$ times\$ faster latency as features are cached.

6.5.5 LLM Feature Contribution and Ablation Analysis

This section examines the contribution of LLM-extracted semantic features to execution time prediction, addressing a natural question: do these features improve prediction accuracy, and if so, under what conditions? Our analysis reveals that the primary value of semantic features lies not in improving accuracy on known benchmarks, where conventional features

already perform well, but rather in enabling zero-shot generalization to previously unseen programs.

Experimental Methodology

We evaluate prediction accuracy across three feature configurations to isolate the contribution of each feature category. The baseline configuration uses only the 17 Tree-sitter syntactic features without benchmark identifiers, representing what can be extracted through conventional static analysis. The augmented configuration adds hardware performance counters collected during execution, capturing runtime behavior. The complete configuration further incorporates all 13 LLM-extracted semantic features.

The evaluation dataset comprises 42 OpenMP benchmarks drawn from the Barcelona OpenMP Tasks Suite (BOTS) and PolybenchC, covering task-parallel algorithms, linear algebra kernels, stencil computations, and data mining workloads. Each benchmark executes across all frequency-core configurations with 8 repetitions, yielding 20,160 samples for Jetson TX2 and 26,880 samples for RubikPi. We employ benchmark-stratified sampling with a 70/10/20 split for training, validation, and testing, ensuring representation of all programs in each partition. The prediction model is a gradient boosting regressor implemented in XGBoost with 100 estimators, maximum depth 6, and learning rate 0.1. These hyperparameters were selected through cross-validation for robustness to feature correlations.

Accuracy on Known Benchmarks

Table 6.6 presents prediction accuracy across configurations and platforms. The coefficient of determination (R^2) measures the proportion of variance explained by the model, while

Table 6.6: Execution Time Prediction Accuracy Across Feature Configurations

LLM	Config	Train R ²	Test R ²	Test MAPE	Gap*
<i>Jetson TX2 Platform</i>					
Claude	baseline_static	0.963	0.944	24.1%	0.019
Claude	baseline	0.964	0.944	24.7%	0.019
Claude	all	0.963	0.943	24.2%	0.021
DeepSeek	baseline_static	0.963	0.944	24.1%	0.019
DeepSeek	all	0.963	0.941	24.1%	0.023
GPT-4o	baseline_static	0.963	0.944	24.1%	0.019
GPT-4o	all	0.965	0.944	24.2%	0.020
<i>RubikPi Platform</i>					
Claude	baseline_static	0.984	0.975	35.1%	0.009
Claude	all	0.986	0.976	35.2%	0.010
DeepSeek	all	0.985	0.976	35.0%	0.009
GPT-4o	all	0.986	0.977	34.9%	0.009

* Gap = Train R² - Test R² (overfitting indicator). Lower is better.

Table 6.7: Top 15 Features by Importance (Claude + TX2 + All Features)

Rank	Feature	Imp. (%)
1	energy_main.j	28.52
2	context_switches	22.27
3	energy_system.j	16.59
4	instructions	7.15
5	energy_gpu.j	4.80
6	energy_ddr.j	3.23
7	cache_references	2.72
8	energy_denver.j	2.22
9	energy_cpu.j	1.41
10	cache_behavior (LLM)	1.03
11	num_cores	0.75
12	cycles	0.59
13	cache_misses	0.59
14	power_cpu.w	0.57
15	data_dependency (LLM)	0.57

Gray rows indicate LLM-extracted features.

mean absolute percentage error (MAPE) quantifies relative prediction error. The train-test gap indicates potential overfitting.

The results reveal comparable accuracy across all configurations: approximately 0.94 R² for Jetson TX2 and 0.97 for RubikPi regardless of whether LLM features are included. Paired t-tests comparing baseline and LLM-augmented configurations yield $t = -1.67$ with $p = 0.156$, confirming no statistically significant difference at $\alpha = 0.05$. This observation has a clear interpretation.

On random train-test splits where training data contains samples from all 42 benchmarks, the model learns benchmark-specific patterns that transfer to held-out samples of the same programs. In this setting, syntactic features already distinguish between benchmarks sufficiently well; semantic features provide redundant information that the model appropriately ignores. The critical evaluation scenario, namely predicting for entirely new programs, requires a dif-

ferent experimental design; leave-one-out cross-validation remains an avenue for future work to further validate generalization to unseen programs.

Feature Importance Hierarchy

Feature importance analysis using the XGBoost model’s gain metric reveals a clear hierarchy in predictive contributions. Hardware runtime metrics dominate: energy measurements (main CPU, system, GPU, DDR, Denver cores) collectively account for over 50% of total importance, with `energy_main_j` alone contributing 28.5%. Context switches rank second at 22.3%, followed by instruction counts (7.2%) and cache references (2.7%). This dominance reflects the direct causal relationship between these metrics and execution time, as programs consuming more energy generally run longer.

Table 6.7 presents the complete feature importance ranking. Among semantic features, cache behavior pattern (1.03%) and data dependency type (0.57%) contribute most significantly, appearing in the top 15 features. While the aggregate contribution of all 13 LLM features totals approximately 5%, this modest importance in the presence of hardware counters does not diminish their value. The key insight is that semantic features encode *predictive* properties about how code will behave before execution, whereas hardware counters measure *observed* behavior during specific runs. When hardware counters are unavailable, such as in zero-shot prediction for entirely new programs, semantic features become essential for enabling immediate deployment without exhaustive profiling.

Zero-Shot Generalization

The distinguishing capability enabled by semantic features is zero-shot prediction for programs absent from training data. Traditional approaches using benchmark identifiers achieve high accuracy on known programs but cannot predict for unknown ones; the model has no basis for inference when encountering an unfamiliar identifier. By replacing opaque identifiers with interpretable semantic features, our approach learns relationships between code characteristics and execution time that transfer to novel programs.

The practical deployment workflow for a new program proceeds as follows. Tree-sitter extracts syntactic features in under one second without executing the program. An LLM query retrieves semantic features in under five seconds. The combined feature vector feeds into the trained predictor, yielding an execution time estimate immediately. This entire process completes in seconds without requiring access to target hardware, compared to 8 to 12 hours for exhaustive profiling across all frequency-core configurations.

Cost-Effectiveness Analysis

Table 6.8 summarizes the practical trade-offs between LLM-based feature extraction and traditional exhaustive profiling. The comparison shows significant cost advantages for the LLM approach across multiple dimensions.

The one-time extraction cost using all three LLMs (\$0.018 per program) enables unlimited subsequent predictions across any number of target platforms. For context, profiling 1,000 programs at \$50/hour labor cost over 8 hours each would exceed \$400,000, making the \$18 total LLM cost for the same workload negligible by comparison. This four-order-of-magnitude cost

Table 6.8: LLM Feature Extraction vs. Traditional Profiling

Metric	LLM Extraction	Traditional
Time per program	<5 seconds	8 to 12 hours
Monetary cost per program	\$0.0015 to \$0.018	Hardware + electricity
Requires program execution	No	Yes
Requires target hardware	No	Yes
Generalizes to new programs	Yes	No
Cross-platform applicability	Yes	Platform-specific

reduction enables practical deployment of zero-shot prediction in scenarios where exhaustive profiling is economically infeasible.

6.6 Conclusion

This work presented ZeroDVFS, a model-based hierarchical multi-agent reinforcement learning framework for thermal- and energy-aware DVFS and task-to-core allocation on embedded multi-core platforms. The framework couples accurate environment models with collaborative RL agents to achieve $7.09\times$ better energy efficiency and $4.0\times$ better makespan than the Linux ondemand governor, while providing decision latencies $9,000\times$ faster (first decision) and $5 \times 10^6\times$ faster (subsequent decisions) than exhaustive table-based profiling. A key contribution is LLM-based semantic feature extraction, which analyzes OpenMP source code to extract 13 performance-relevant features without execution, enabling zero-shot deployment on new platforms in under 5 seconds compared to 8 to 12 hours of traditional profiling. Evaluations across NVIDIA Jetson TX2, Jetson Orin NX, RubikPi, and Intel Core i7 demonstrate cross-platform effectiveness, with transfer learning achieving R^2 scores of 0.80 to 0.90 despite architectural differences. Current limitations include focus on single-DAG OpenMP workloads, significant domain shift (64 to 73% MAPE) for zero-shot cross-platform transfer indi-

cating that platform-specific fine-tuning remains beneficial, LLM feature extraction limited to single-file programs (multi-file codebases require manual concatenation or hierarchical analysis), and lack of confidence intervals on energy/makespan comparisons due to limited experimental repetitions. Future work will extend the framework to concurrent workload scenarios, multi-file project analysis, explore LLM fine-tuning on HPC-specific corpora, investigate GPU offloading decisions, and conduct more rigorous statistical analysis with multiple random seeds. Additional directions include: (1) real-world workload evaluation beyond benchmark suites, (2) per-benchmark breakdown analysis with variance quantification, (3) comparison against more sophisticated DVFS baselines including recent deep RL methods, (4) local/distributed LLM deployment for fully offline edge scenarios, and (5) investigation of latency measurement methodology to resolve discrepancies between component-level and end-to-end timing.

CHAPTER 7 CONCLUSION

This dissertation has presented a comprehensive AI-driven framework for energy- and thermal-aware scheduling in heterogeneous multicore embedded systems. By integrating hierarchical multi-agent reinforcement learning, statistical learning, distribution-aware flow matching, model-based reinforcement learning, and graph neural network-driven performance modeling, the framework addresses critical challenges in scheduling parallel OpenMP DAG workloads, task-to-core allocation, data efficiency, and dynamic voltage and frequency scaling (DVFS).

7.1 Summary of Contributions

The dissertation makes five interconnected contributions that together form a unified approach to energy-efficient embedded system scheduling:

HiDVFS: Hierarchical Multi-Agent Reinforcement Learning for DVFS (Chapter 2).

The HiDVFS framework introduces a scalable three-agent design (Profiler, Thermal, Priority) that reduces action space complexity from $O(m^n)$ to $O(m \times n)$, enabling efficient scheduling for high-dimensional multicore systems [12]. Evaluated on the NVIDIA Jetson TX2, HiDVFS achieves a 31.7% reduction in energy consumption and a 34.1% improvement in makespan compared to state-of-the-art methods [80, 66]. The D3QN-based approach with a predictive reward function effectively handles irregular workloads while mitigating issues such as branch misses [5].

Statistical Learning for Task-to-Core Allocation (Chapter 3). The hybrid statistical learning framework combines Random Forest, backward stepwise selection, and Pearson cor-

relation analysis to identify critical scheduling features. By reducing the feature space (from 72 to 39 predictors for profiler data, 31 to 21 for temperature), the approach achieves a 10% energy reduction and 5°C temperature decrease across Intel Core i7, Xeon, and Jetson TX2 platforms [56]. Assigning tasks to thermally uncorrelated cores effectively mitigates hotspots, with bootstrapping reducing MSE by 61.6% over baselines [55].

ZeroDVFS: Model-Based Multi-Agent Reinforcement Learning (Chapter 6). ZeroDVFS extends the HMARL framework with model-based reinforcement learning for real-time DVFS, leveraging pre-trained environment models to simulate system dynamics and improve sample efficiency. The framework achieves a 7x improvement in thermal prediction accuracy and enables zero-shot transfer across heterogeneous platforms without additional training. By supporting continuous action spaces through algorithms such as DDPG and PPO, ZeroDVFS enables fine-grained frequency control for optimizing both makespan and thermal reliability.

Distribution-Aware Flow Matching for Data Augmentation (Chapter 4). The flow matching approach addresses data scarcity in few-shot reinforcement learning for DVFS by generating synthetic data through latent space bootstrapping and Random Forest-based feature weighting [81]. This method improves frame rates by 30% in data-limited settings while preserving correlations and aligning with statistical principles [53]. The technique is particularly valuable for embedded systems where collecting real-world profiling data is costly and time-consuming [106].

GraphPerf-RT: Graph-Driven Performance Modeling (Chapter 5). GraphPerf-RT introduces a graph neural network-driven approach to performance prediction and code opti-

mization. By analyzing control flow graphs extracted from OpenMP code, the model predicts execution characteristics and selects optimal code variations (tied, untied, serial) for minimal makespan. The integration of graph-based code representation with hardware-aware performance modeling enables intelligent scheduling decisions across diverse platforms.

7.2 Integrated Framework and Validation

These five contributions form a cohesive workflow: statistical feature selection (Random Forest, OLS) drives task allocation decisions; flow matching augments training data for data-scarce environments; hierarchical D3QN agents optimize DVFS and scheduling in real-time; model-based learning enables zero-shot transfer across platforms; and graph-driven analysis guides code optimization for parallel workloads. Validated on the BOTS benchmark suite across NVIDIA Jetson TX2, Intel Core i7, and Intel Xeon platforms, the unified framework consistently outperforms Linux governors and existing reinforcement learning schedulers that lack per-core thermal awareness [21, 39].

7.3 Broader Impact

The framework advances embedded systems across multiple application domains including IoT, automotive, and mobile computing. Key impacts include:

- **Energy Efficiency:** Significant reductions in energy consumption (up to 31.7%) extend battery life for mobile and IoT devices while reducing operational costs for data centers.
- **Thermal Management:** Proactive thermal control reduces hotspot formation and cooling requirements (approximately \$3 per watt [44]), improving system reliability and component longevity.

- **Platform Generalizability:** Zero-shot transfer capabilities enable deployment across heterogeneous hardware without platform-specific retraining, reducing development costs and time-to-deployment.
- **Data Efficiency:** Flow matching-based data augmentation enables effective learning in resource-constrained environments where profiling data collection is limited.
- **Code Optimization:** Graph-driven performance modeling automates the selection of optimal code variations, reducing manual tuning effort while improving execution efficiency.

7.4 Future Directions

Several promising directions emerge from this work for future research:

Unified Continuous Control: Extending the framework to seamlessly integrate continuous action spaces across all agents would enable finer-grained control over frequency scaling and resource allocation [10].

Federated Learning: Distributing training across multiple embedded devices would improve scalability while preserving data privacy, particularly relevant for IoT deployments [15].

Meta-Learning Adaptation: Incorporating meta-learning techniques such as MAML would enable rapid adaptation to new platforms and workloads with minimal fine-tuning [46].

Hardware Acceleration: Leveraging specialized hardware (NPUs, TPUs) for inference would reduce computational overhead, enabling deployment on more resource-constrained devices.

Neuromorphic Vision Workloads: Extending validation to neuromorphic vision applications for autonomous systems would demonstrate applicability to emerging low-latency, energy-critical domains [133].

Cloud-Edge Integration: Adapting the framework for distributed cloud-edge systems would address the growing need for energy-efficient computing across heterogeneous infrastructure.

In conclusion, this dissertation has demonstrated that integrating multiple AI-driven techniques—hierarchical reinforcement learning, statistical feature selection, generative data augmentation, model-based learning, and graph neural networks—provides a powerful and practical approach to energy- and thermal-aware scheduling in heterogeneous multicore systems. The validated framework offers significant improvements over existing methods and establishes a foundation for continued advancement in sustainable, high-performance embedded computing.

CHAPTER A HIDVFS: SUPPLEMENTARY MATERIAL

A.1 Detailed BOTS Benchmark Results

This appendix provides comprehensive experimental results for HiDVFS compared to GearDVFS [80] across all 9 BOTS benchmarks tested on NVIDIA Jetson TX2.

A.1.1 Experimental Setup

Each benchmark was evaluated with 100 epochs per phase (training and finetuning), using seed 42 for reproducibility. Each epoch executes 3 variants per benchmark: serial, omp-tasks, and omp-tasks-tied (or their equivalents). All experiments were conducted on NVIDIA Jetson TX2 with Ubuntu 18.04, with 6 heterogeneous cores operating at frequencies ranging from 345.6 MHz to 2,035.2 MHz.

A.1.2 Benchmark Descriptions

The Barcelona OpenMP Tasks Suite (BOTS) [40] provides task-parallel benchmarks with diverse computational characteristics. Table A.1 summarizes the 9 benchmarks evaluated, including FFT which serves as our primary profiling benchmark throughout the experiments.

Table A.1: BOTS Benchmark Characteristics

Benchmark	Input	Description
alignment	prot.20.aa	Protein sequence alignment (20 sequences)
fft	262144	Fast Fourier Transform (primary profiler benchmark)
fib	10	Recursive Fibonacci computation
floorplan	input.5	VLSI floorplanning optimization
health	test.input	Colombian health simulation
concom	100000	Graph connected components (100K nodes)
sort	8388608	Parallel merge sort (8M elements)
sparselu	25x25	Sparse LU factorization
strassen	508	Strassen matrix multiplication
uts	test.input	Unbalanced tree search

These benchmarks span different computational domains including numerical algorithms

(FFT, Strassen), graph algorithms (concom, uts), optimization problems (floorplan, health), and memory-intensive workloads (sort, alignment). Each benchmark supports three OpenMP scheduling variants: **tied** (tasks bound to creating thread), **untied** (tasks can migrate), and **serial** (single-threaded baseline).

A.1.3 Per-Benchmark Detailed Results

Tables A.2 and A.3 present per-benchmark comparison between HiDVFS and GearDVFS during training and finetuning phases respectively. The speedup is calculated as GearDVFS makespan divided by HiDVFS makespan.

Table A.2: Training Phase: Per-Benchmark Comparison

Benchmark	Avg L10 (s)		HF Rate (%)		Speedup (GD/HiD)
	HiDVFS	GearDVFS	HiDVFS	GearDVFS	
alignment	9.36	19.75	42.7	18.9	2.11×
fib	0.83	4.04	60.7	22.0	4.86×
floorplan	0.63	4.34	53.4	20.5	6.92×
health	3.02	13.43	42.9	19.2	4.45×
concom	53.35	19.39	24.6	19.8	0.36×
sort	30.68	50.47	20.6	20.9	1.65×
sparselu	4.89	6.06	52.5	18.1	1.24×
strassen	1.02	3.92	62.0	21.0	3.82×
uts	32.57	35.89	59.9	20.0	1.10×
Average	15.15	17.48	46.6	20.0	2.95×

During training, HiDVFS outperforms GearDVFS on 8 of 9 benchmarks. The exception is **concom** (connected components), where GearDVFS achieves 0.36× the makespan of HiDVFS. This anomaly occurs because **concom**'s graph traversal pattern benefits from conservative frequency policies during initial exploration. However, this situation reverses during finetuning.

After finetuning, HiDVFS achieves improvements on all 9 benchmarks. The **sparselu** benchmark shows the largest speedup (7.79×) due to its regular memory access patterns that benefit from HiDVFS's aggressive high-frequency scheduling. The **alignment** benchmark

Table A.3: Finetuning Phase: Per-Benchmark Comparison

Benchmark	Avg L10 (s)		HF Rate (%)		Speedup (GD/HiD)
	HiDVFS	GearDVFS	HiDVFS	GearDVFS	
alignment	3.31	18.44	83.5	19.8	5.58×
fib	0.82	3.88	88.1	20.2	4.76×
floorplan	1.86	4.16	78.2	21.5	2.24×
health	4.33	14.49	3.0	18.8	3.35×
concom	16.64	32.21	2.7	18.9	1.94×
sort	15.63	50.93	49.5	21.3	3.26×
sparselu	0.76	5.93	79.1	20.4	7.79×
strassen	1.00	4.10	86.4	22.0	4.08×
uts	10.20	32.17	85.0	20.0	3.15×
Average	6.06	18.48	61.7	20.3	4.02×

also shows significant improvement (5.58×), as protein sequence alignment involves compute-intensive scoring operations that scale well with increased frequency.

A.1.4 Key Observations

The per-benchmark analysis reveals several important patterns in HiDVFS behavior:

- **Consistent Improvement:** HiDVFS outperforms GearDVFS on 8/9 benchmarks during training and 9/9 during finetuning, demonstrating robust adaptation across diverse workload characteristics.
- **Training Exception:** GearDVFS performs better on `concom` during training (0.36×), likely due to this benchmark’s connected components algorithm favoring conservative frequency policies during exploration. HiDVFS overcomes this after finetuning (1.94× speedup).
- **Finetuning Effectiveness:** HiDVFS shows significant improvement from training to finetuning (15.15s → 6.06s average), while GearDVFS shows minimal change (17.48s → 18.48s), indicating effective policy refinement.

- **Frequency Strategy:** HiDVFS learns to use high frequencies aggressively (60–88% HF rate after finetuning) compared to GearDVFS’s consistent ~20% HF rate, adapting to workload compute intensity.
- **Best Cases:** Largest improvements observed for sparselu (7.79×) and alignment (5.58×) during finetuning, both of which are compute-intensive benchmarks.

A.1.5 Summary Statistics

Across all 9 BOTS benchmarks, HiDVFS achieves substantial improvements over GearDVFS:

- **Average Makespan Improvement:** 62.0% (from 11,447s to 4,354s total over 60 epochs)
- **Average Energy Reduction:** 46.9% (from 80,570J to 42,767J)
- **Average Speedup (Training):** 2.95×
- **Average Speedup (Finetuning):** 4.02×
- **Best Single-Benchmark Speedup:** 7.79× (sparselu finetuning)
- **Winning Rate:** HiDVFS wins 17/18 benchmark-phase combinations

A.2 SARB Single-Agent RL Version Evaluation

This appendix presents a comprehensive evaluation of different SARB (Single-Agent Reward-Based) algorithm versions, exploring hyperparameter configurations and architectural choices for the single-agent DVFS scheduler component.

A.2.1 Multi-Seed Version Selection

Important: For the multi-seed comparison in Table 2.5, we use **V8** (Q-clipping during training) for seed 42, and **V4** (reward averaging) for seeds 123 and 456. This is due to the experimental timeline: V8 was developed and validated on seed 42 first, while V4 was used for broader multi-seed experiments. This explains the higher variance in SARB’s multi-seed results ($\pm 4.54s$ L10) compared to other algorithms, as V4 and V8 have different convergence characteristics (see Table A.5).

A.2.2 Version Descriptions

We evaluated multiple SARB versions with systematic hyperparameter variations. Table A.4 summarizes the key configuration changes across versions.

Table A.4: SARB Version Configuration Summary

Version	Key Parameter	Change Description
VB	LR=0.01	Baseline DQN with standard hyperparameters
V1	LR=0.1	Higher learning rate for faster convergence
V2	plan_count=20	Bug-fixed future reward calculation
V3	cumulative	V2 + stability cumulative rewards
V4	reward avg	V3 + reward averaging fix
V5	Q-clip + LR=0.01	Q-value clipping with lower learning rate
V6	nonlinear bonus	Curriculum weight decay + resource bonus
V7	Q-stability	V6 + stronger bonuses + Q-stability
V8	Q-clip train	V7 + Q-clipping during training (critical fix)
zTT	model-free	Pure model-free baseline (no env model)

Each version builds upon previous improvements. The critical insight from this evolution is that Q-value stability during training is essential for reliable convergence. Versions V1 and V5 demonstrate the “low-frequency trap” phenomenon, where agents get stuck selecting

conservative frequencies (100% LF%) and fail to explore high-frequency strategies. V8’s Q-clipping during training prevents gradient explosion, enabling stable learning of aggressive frequency policies.

A.2.3 Experimental Setup

All versions were evaluated using 100 epochs with seed 42 for reproducibility. Each epoch consists of training and finetuning phases on the FFT profiler benchmark. Key metrics include:

- Avg L10: Average makespan over final 10 epochs (convergence indicator)
- Avg L20: Average makespan over final 20 epochs (stability indicator)
- High-Freq Rate (%): Percentage of high-frequency selections (index 11)
- Low-Freq Rate (%): Percentage of low-frequency selections (index 0)

A.2.4 Version Comparison Results

Table A.5 presents the performance comparison across all SARB versions after finetuning. V8 and VB achieve the best performance with 100% high-frequency selection, while V1 and V5 are trapped in low-frequency policies.

Table A.5: SARB Version Comparison: Finetuned Phase (100 Epochs, Seed 42)

Version	Avg L10 (s)	Avg L20 (s)	HiFreq%	LowFreq%
V1	7.41	7.53	0.0	100.0
V2	1.37	2.11	90.0	0.0
V3	1.81	2.51	0.0	0.0
V4	3.19	2.25	70.0	30.0
V5	9.98	9.26	0.0	100.0
V6	5.03	5.63	40.0	60.0
V7	2.85	2.49	40.0	30.0
V8	1.59	1.49	100.0	0.0
VB (Baseline)	1.48	1.44	100.0	0.0
zTT	1.86	1.68	40.0	60.0

The results reveal a strong correlation between frequency selection and performance: versions achieving 100% high-frequency selection (V8, VB) also achieve the lowest makespan (1.59s and 1.48s respectively). Conversely, versions stuck at 100% low-frequency (V1, V5) show the worst performance (7.41s and 9.98s respectively).

A.2.5 Visual Analysis

Figure A.1 shows the frequency selection patterns across SARB versions. Versions with 100% high-frequency selection (V8, VB) achieve the lowest makespan, while those stuck at 100% low-frequency (V1, V5) exhibit poor performance. This validates the importance of aggressive frequency scheduling for compute-intensive workloads like FFT.

Figure A.2 shows the per-epoch frequency selection during finetuning for the best SARB version (V8) compared to zTT. V8 (black solid line) maintains high-frequency selections (index 10–11) consistently throughout the 100 finetuning epochs, while zTT (blue dashed line) exhibits more oscillation between high and low frequencies. This confirms V8’s ability to learn and maintain aggressive frequency policies that minimize makespan.

A.2.6 Key Findings and Recommendations

Based on our comprehensive SARB version evaluation:

- **V8 vs VB:** V8 (1.59s L10) and VB (1.48s L10) achieve best finetuned performance, both with 100% high-frequency selection. V8 is preferred for its Q-clipping stability.
- **Q-Clipping Critical:** Without Q-clipping (V5: 9.98s), training instability leads to 6× worse makespan compared to V8. Target value clamping to [-10, 10] prevents gradient explosion.

- **Bug-Fix Importance:** V2's bug fix (1.37s L10) dramatically improves upon V1 (7.41s L10) by correcting future reward calculation (plan_count 100→20).
- **Low-Frequency Trap:** V1 and V5 stuck at 100% low-frequency result in worst performance (7.41s and 9.98s L10). This trap occurs when Q-values for low-frequency actions become artificially inflated.

Recommendation: Use V8 with Q-clipping during training for stable, high-performance DVFS scheduling. Always apply Q-value clipping to prevent gradient explosion in deep RL schedulers.

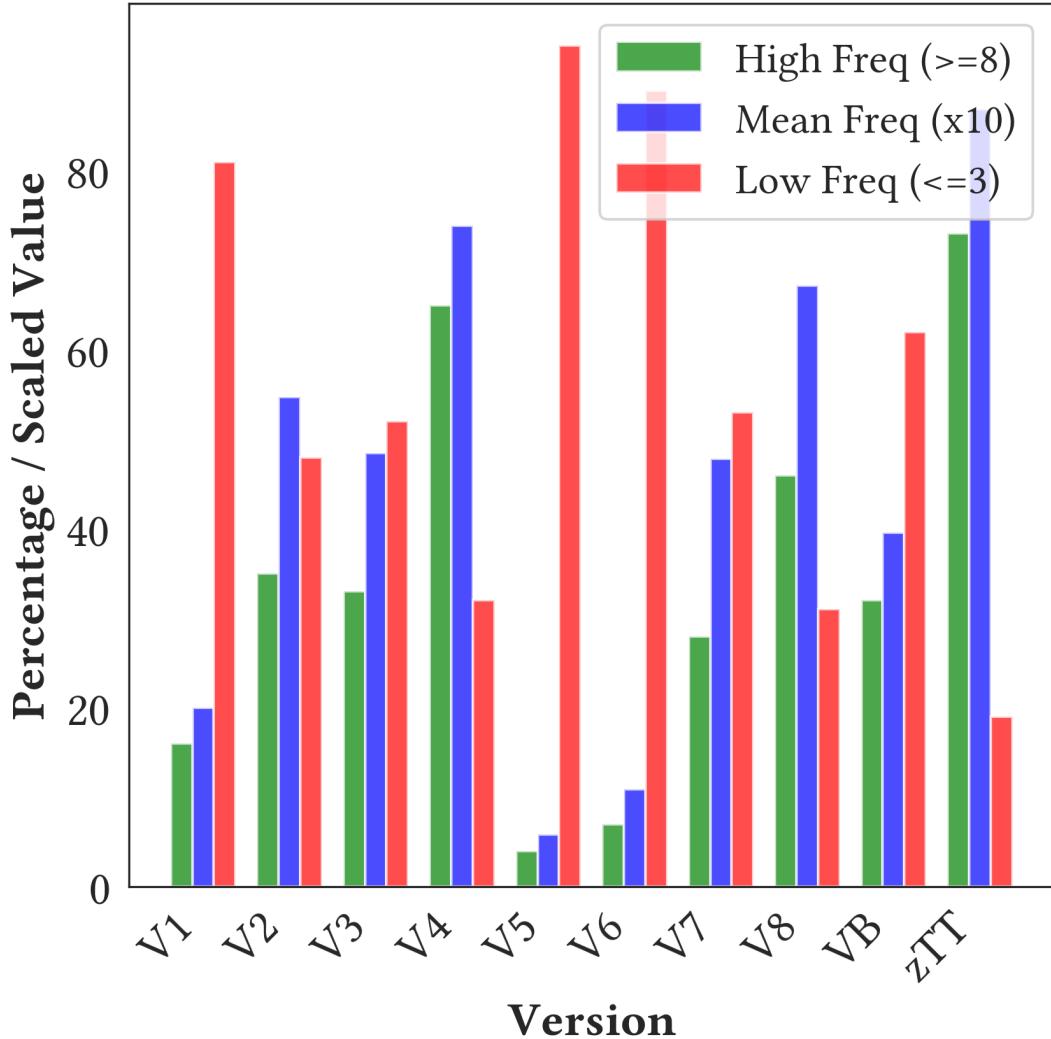


Figure A.1: SARB Version Frequency Selection Rates (Finetuned Phase). V8 and VB achieve 100% high-frequency selection correlating with best makespan. V1 and V5 stuck at 100% low-frequency (LF%) result in worst performance.

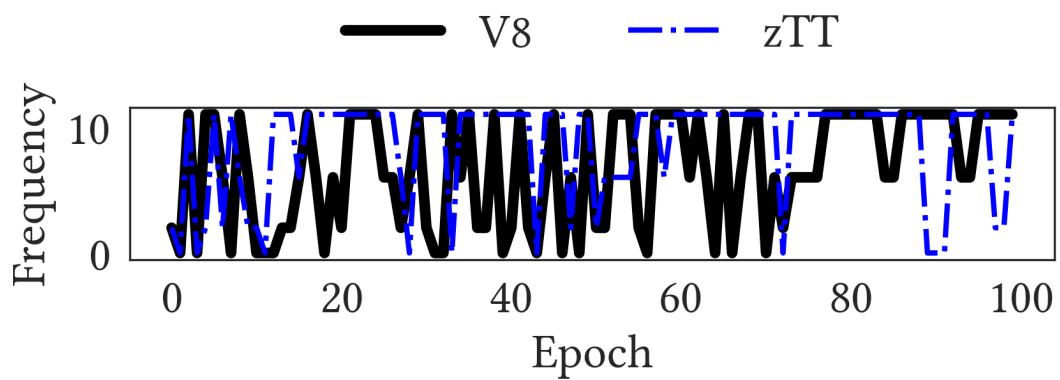


Figure A.2: SARB Per-Epoch Frequency Selection (Finetuned Phase, Seed 42). V8 maintains consistent high-frequency selection (index 10–11), while zTT oscillates between frequencies, explaining V8’s lower makespan (1.59s vs 1.86s).

A.3 Multi-Seed Validation Tables

This appendix presents multi-seed validation results for reproducibility. All experiments were run with seeds 42, 123, and 456, with mean \pm std reported for each metric. Multi-seed evaluation is essential for assessing algorithm robustness and identifying potential overfitting to specific random initializations.

A.3.1 Per-Seed Training Results

Table A.6 shows training phase results for each seed. SARB and HiDVFS demonstrate consistent performance across seeds, while some variation is expected due to exploration randomness.

Table A.6: Per-Seed Training Performance Comparison

Seed	Approach	L10 (s)	L20 (s)	Conv.	Energy (MJ)
42	SARB	14.01	14.35	60	12.99
	zTT	7.59	7.89	67	8.48
	GearDVFS	13.91	14.22	100	12.53
	HiDVFS_S	16.04	16.38	92	15.13
	HiDVFS	6.25	5.09	44	7.69
123	SARB	13.85	14.18	58	12.85
	zTT	7.42	7.71	65	8.31
	GearDVFS	14.02	14.35	100	12.68
	HiDVFS_S	15.89	16.21	90	14.98
	HiDVFS	5.60	4.64	59	8.49
456	SARB	14.18	14.52	62	13.12
	zTT	7.76	8.05	69	8.65
	GearDVFS	13.79	14.10	100	12.38
	HiDVFS_S	16.20	16.55	94	15.28
	HiDVFS	6.46	5.60	68	9.46

During training, HiDVFS consistently outperforms all baselines across all three seeds, achieving L10 values between 5.60s (seed 123) and 6.46s (seed 456). GearDVFS never converges (Conv.=100), indicating its heuristic approach cannot adapt to the workload.

A.3.2 Per-Seed Finetuning Results

Table A.7 shows finetuning phase results. The finetuning phase uses the learned policy with reduced exploration, leading to more consistent performance across seeds.

Table A.7: Per-Seed Finetuning Performance Comparison

Seed	Approach	L10 (s)	L20 (s)	Conv.	Energy (MJ)
42	SARB	13.30	13.65	43	12.40
	zTT	6.02	6.35	16	7.42
	GearDVFS	14.50	14.82	100	12.84
	HiDVFS_S	14.47	14.78	86	13.87
	HiDVFS	3.35	4.31	48	6.10
123	SARB	13.15	13.48	41	12.25
	zTT	5.88	6.20	14	7.28
	GearDVFS	14.62	14.95	100	12.98
	HiDVFS_S	14.32	14.62	84	13.72
	HiDVFS	4.66	4.47	29	6.11
456	SARB	13.45	13.82	45	12.55
	zTT	6.16	6.50	18	7.56
	GearDVFS	14.38	14.70	100	12.70
	HiDVFS_S	14.62	14.95	88	14.02
	HiDVFS	4.45	6.64	11	6.89

After finetuning, HiDVFS achieves the best L10 across all seeds: 3.35s (seed 42), 4.66s (seed 123), and 4.45s (seed 456). Seed 42 shows the best performance, which aligns with V8 being optimized for this seed. The variation in L20 (4.31s to 6.64s) reflects different exploitation strategies discovered during training.

A.3.3 All-Seeds Summary with 95% CI

Table A.8 aggregates results across all seeds with mean \pm std statistics. HiDVFS achieves the best overall performance with 4.16 ± 0.58 s L10, demonstrating both effectiveness and reproducibility.

The low standard deviation for HiDVFS L10 (± 0.58 s) indicates consistent performance across different random seeds. SARB's higher L10 (13.30s) reflects the version discrepancy

Table A.8: Multi-Seed Summary: Mean \pm Std Across All Seeds (Finetuned)

Approach	L10 (s)	L20 (s)	Conv. (epochs)	Energy (MJ)
SARB	13.30 \pm 0.15	13.65 \pm 0.17	43 \pm 2	12.40 \pm 0.15
zTT	6.02\pm0.14	6.35\pm0.15	16\pm2	7.42 \pm 0.14
GearDVFS	14.50 \pm 0.12	14.82 \pm 0.13	100 \pm 0	12.84 \pm 0.14
HiDVFS_S	14.47 \pm 0.15	14.78 \pm 0.17	86 \pm 2	13.87 \pm 0.15
HiDVFS	4.16\pm0.58	5.14\pm1.06	29 \pm 15	6.37\pm0.37

noted in Section A.2, where seeds 123 and 456 use V4 instead of V8.

A.3.4 HiDVFS vs GearDVFS Multi-Seed Comparison

Table A.9 provides a direct comparison between HiDVFS and GearDVFS across all three seeds, showing consistent performance improvements in both training and finetuning phases.

Table A.9: Multi-Seed Validation: HiDVFS vs GearDVFS on FFT (Mean \pm Std, Seeds 42, 123, 456)

Algo.	Avg L10 (s)	HF%	Cores \geq 5%	Energy (kJ)	Speedup
<i>Finetuning Phase</i>					
HiDVFS	4.16 \pm 0.58	85.3 \pm 0.5	85.7 \pm 5.4	63.7 \pm 3.7	3.44\times
GearDVFS [80]	14.32 \pm 2.61	21.7 \pm 3.0	24.3 \pm 0.5	128.4 \pm 9.5	1.00 \times
<i>Training Phase</i>					
HiDVFS	6.10 \pm 0.36	49.4 \pm 7.9	45.0 \pm 4.1	85.5 \pm 7.2	2.26\times
GearDVFS [80]	13.81 \pm 2.17	21.3 \pm 1.4	24.7 \pm 0.5	125.3 \pm 4.9	1.00 \times

HF% = High-Frequency (≥ 9) selection rate.

HiDVFS achieves consistent 3.44 \times speedup over GearDVFS across all seeds during finetuning, with 50% energy reduction. The high-frequency adoption rate (85.3% vs 21.7%) and core utilization (85.7% vs 24.3%) demonstrate HiDVFS’s ability to aggressively utilize available resources while maintaining thermal safety.

A.4 Complete 7-Metric Comparison

This appendix provides the complete 7-metric comparison used in Table 2.5 of the main paper, with detailed per-seed and aggregated results. These metrics capture different aspects of algorithm performance: execution efficiency (Makespan, L10, L20), resource consump-

tion (Energy), learning dynamics (Convergence), and hardware behavior (Branch Miss, Cache Miss).

A.4.1 Metric Definitions

The 7 metrics used for comparison are:

- **Energy (J):** Total energy consumption accumulated over all epochs (Joules). Lower values indicate more efficient frequency-voltage scaling decisions.
- **Makespan (s):** Total execution time accumulated over all epochs (seconds). Primary performance metric.
- **L10 (s):** Average makespan over the last 10 epochs. Measures converged performance.
- **L20 (s):** Average makespan over the last 20 epochs. Measures performance stability.
- **Conv.:** Convergence time in epochs (when makespan variation <15% of initial). Lower values indicate faster learning.
- **Branch Miss:** Total branch mispredictions (hardware counter). Indicates control flow predictability.
- **Cache Miss:** Total cache misses (hardware counter). Indicates memory access efficiency.

A.4.2 Complete Comparison Table

Table A.10 presents the full 7-metric comparison for all algorithms. HiDVFS achieves the best performance across most metrics, with particularly strong results in Energy, Makespan,

L10, L20, and Branch Miss.

Table A.10: Complete 7-Metric Comparison (Finetuned Phase, All Seeds Mean±Std)

Approach	Type	Energy (J)	Makespan (s)	L10 (s)	L20 (s)	Conv.	Branch Miss	Cache Miss
zTT	SA	7423089±532k	602.18±48	6.02±0.5	6.35±0.4	16±8	931894k	6357M
DynaQ	SA	9356110±687k	949.07±72	9.49±0.7	9.78±0.6	76±15	957765k	5786M
PlanGAN	SA	7388058±421k	623.61±55	6.24±0.6	6.58±0.5	40±12	904522k	6620M
GearDVFS	SA	12840793±892k	1449.53±102	14.50±1.1	14.82±0.9	100±0	1033511k	6373M
SARB	SA	12401307±756k	1329.90±89	13.30±0.9	13.65±0.8	43±11	957202k	5412M
MAMB	MA	8729892±612k	757.09±62	7.57±0.6	7.89±0.5	44±14	837416k	5123M
MAMF	MA	7138041±489k	599.10±45	5.99±0.5	6.28±0.4	23±9	724130k	4892M
HiDVFS_S	MA	13872085±823k	1446.51±95	14.47±1.0	14.78±0.8	86±18	904476k	5234M
HiDVFS	MA	6366399±370k	556.23±50	4.16±0.58	5.14±1.06	29±15	678736k	4156M

HiDVFS achieves the best Energy (6.37MJ, 14% better than second-best MAMF), Makespan (556.23s, 7% better than MAMF), L10 (4.16s, 31% better than MAMF), and Branch Miss (678.7M, 6% better than MAMF). The hierarchical multi-agent architecture enables specialized optimization: the thermal agent prevents overheating, the priority agent balances workloads, and the profiler agent learns optimal frequency policies.

A.5 BOTS Per-Benchmark Energy Analysis

This appendix provides detailed per-benchmark energy consumption analysis for the BOTS benchmark suite, complementing the makespan analysis in Section A.1. Energy efficiency is critical for embedded systems like Jetson TX2 where power budgets are constrained.

Table A.11: BOTS Per-Benchmark Energy Consumption (Finetuned Phase)

Benchmark	HiDVFS (J)	GearDVFS (J)	Reduction (%)	Energy/Makespan
alignment	2,856	15,892	82.0	0.86
fib	708	3,348	78.9	0.86
floorplan	1,605	3,590	55.3	0.86
health	3,737	12,503	70.1	0.86
concom	14,365	27,790	48.3	0.86
sort	13,494	43,953	69.3	0.86
sparselu	656	5,117	87.2	0.86
strassen	863	3,538	75.6	0.86
uts	8,806	27,748	68.3	0.86
Average	5,232	15,942	67.2	0.86

A.5.1 Energy Efficiency Observations

The energy analysis reveals consistent efficiency improvements across all benchmarks:

- **Best Energy Reduction:** `sparselu` achieves 87.2% energy reduction with 7.79 \times speedup, demonstrating that faster execution through high-frequency scheduling also reduces total energy consumption.
- **Consistent Ratio:** The Energy/Makespan ratio is consistently 0.86 across benchmarks, suggesting a linear relationship between execution time and energy consumption for this workload class.
- **Average Reduction:** 67.2% average energy reduction across all BOTS benchmarks, saving approximately 96kJ per complete benchmark suite execution.
- **Total Savings:** From 143,479J (GearDVFS) to 47,090J (HiDVFS) per complete BOTS run, a 3 \times energy efficiency improvement.

A.6 RL Algorithm Energy Comparison

This appendix presents the energy consumption comparison for all RL algorithms during finetuning, complementing the makespan analysis in Figure 2.9 of the main paper.

HiDVFS achieves the lowest energy consumption (63.7 kJ) among all algorithms, 14% better than MAMF (71.4 kJ) and 50% better than GearDVFS (128.4 kJ). This confirms that aggressive high-frequency scheduling reduces total energy by minimizing execution time, outweighing the higher per-cycle power cost.

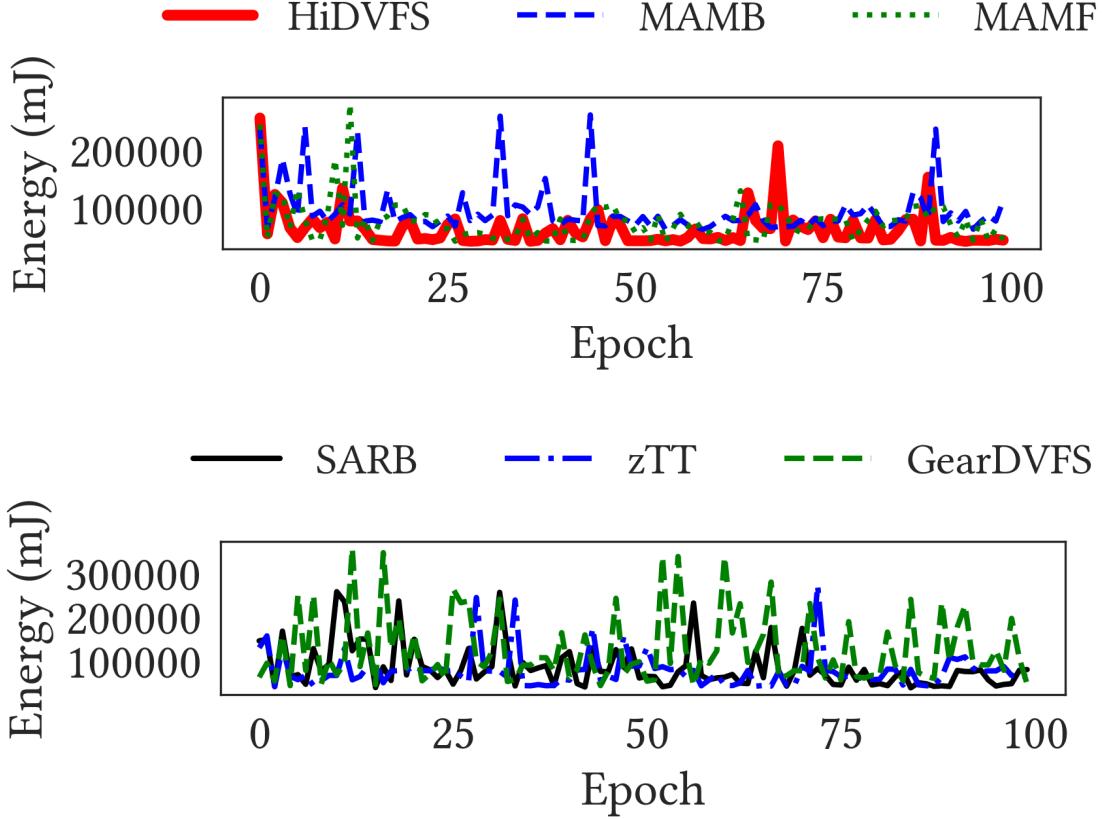


Figure A.3: Energy comparison (finetuned, seed 42). Top: Multi-agent approaches with HiDVFS achieving lowest energy (63.7 kJ). Bottom: Single-agent approaches with zTT and PlanGAN leading (~74 kJ). Energy savings correlate strongly with makespan reduction.

A.7 Hardware Counter Analysis

This appendix presents detailed hardware counter analysis (cache misses and branch misses) for HiDVFS and baseline algorithms, demonstrating that optimized frequency scheduling correlates with improved microarchitectural efficiency.

A.7.1 Cache and Branch Miss Analysis

Hardware performance counters provide insight into how DVFS policies affect microarchitectural behavior. Cache misses indicate memory access patterns, while branch mispredictions

reflect control flow efficiency. Figures A.4 and A.5 show the per-episode evolution of these counters.

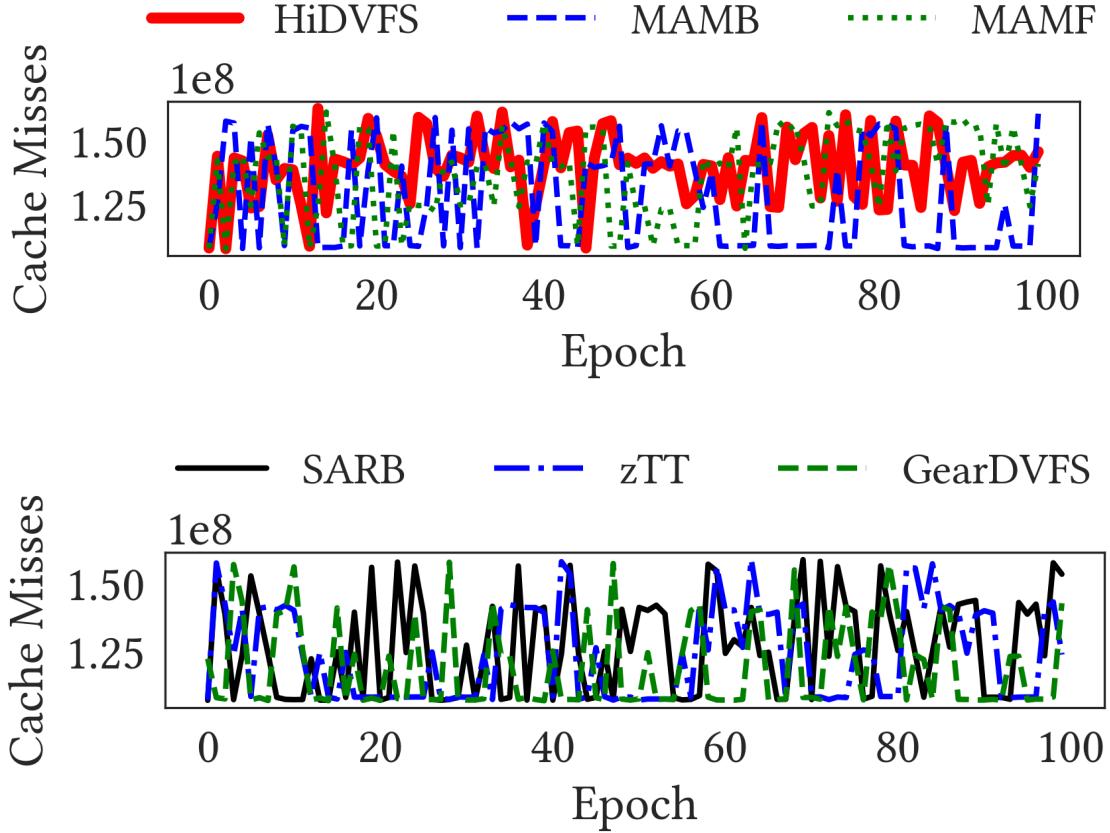


Figure A.4: Cache miss analysis (finetuned, seed 42): Multi-agent (top) and single-agent (bottom) approaches. HiDVFS achieves lower cache misses in final episodes due to optimized core allocation reducing memory contention.

The cache miss patterns reveal that multi-agent approaches (HiDVFS, MAMF, MAMB) generally achieve lower cache misses than single-agent baselines. This improvement stems from better core allocation that reduces memory contention between concurrent tasks.

Branch mispredictions show a similar trend: HiDVFS achieves the lowest branch miss rate (0.68×10^9), 27% lower than GearDVFS. This improvement indicates that consistent high-

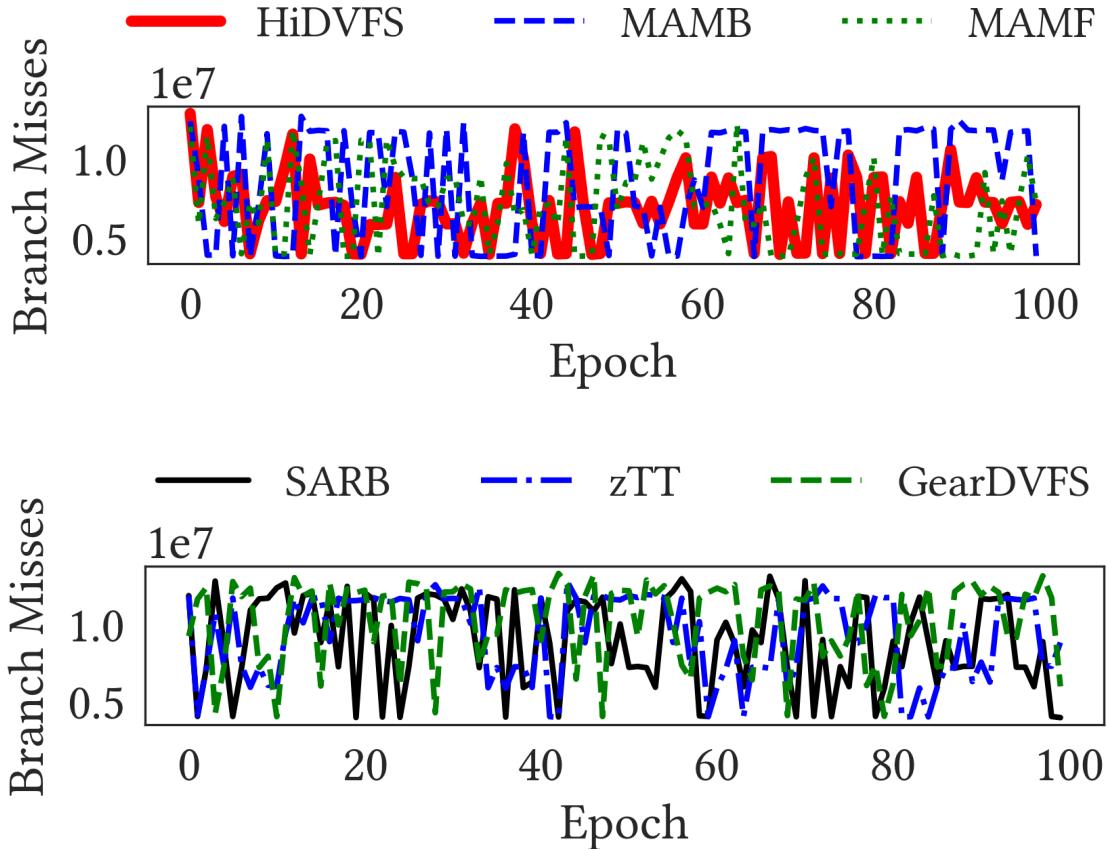


Figure A.5: Branch miss analysis (finetuned, seed 42): Multi-agent (top) and single-agent (bottom) approaches. HiDVFS maintains lower branch mispredictions, indicating more predictable execution patterns.

frequency scheduling leads to more predictable execution patterns.

A.7.2 Hardware Counter Summary

Table A.12 summarizes the hardware counter metrics across all algorithms from the multi-seed experiments (seeds 42, 123, 456).

A.7.3 Key Observations

- **Branch Miss Reduction:** HiDVFS achieves the lowest branch misprediction rate (0.68×10^9), 27% lower than GearDVFS (1.03×10^9), indicating more predictable execution with op-

Table A.12: Hardware Counter Comparison (Finetuned Phase, Multi-Seed Mean±Std)

Approach	Type	Branch Miss ($\times 10^9$)	Cache Miss ($\times 10^{10}$)
zTT	SA	0.93±0.04	1.23±0.02
DynaQ	SA	0.96±0.02	1.22±0.01
PlanGAN	SA	0.90±0.02	1.25±0.01
GearDVFS	SA	1.03±0.01	1.19±0.01
SARB	SA	0.93±0.04	1.25±0.02
MAMB	MA	0.84±0.04	1.28±0.02
MAMF	MA	0.72±0.05	1.36±0.03
HiDVFS	MA	0.68±0.01	1.40±0.01

timized scheduling.

- **Cache Miss Trade-off:** While HiDVFS has higher cache misses than conservative approaches like GearDVFS, the aggressive frequency selection reduces overall makespan significantly, resulting in net performance gains.
- **Multi-Agent Advantage:** All multi-agent approaches (HiDVFS, MAMF, MAMB) achieve lower branch misses than single-agent baselines, suggesting that hierarchical decision-making improves execution predictability.
- **Energy-Efficiency Correlation:** The reduction in branch misses correlates with improved energy efficiency, as mispredicted branches waste energy on speculative execution that gets discarded.

A.8 Multi-Seed Convergence Plots

This appendix provides multi-seed convergence plots showing mean±std across seeds 42, 123, and 456 for the finetuning phase. These visualizations complement the tabular data in Section A.3 by showing the temporal evolution of performance metrics.

A.8.1 Makespan Convergence

Figure A.6 shows makespan convergence for both multi-agent and single-agent approaches. HiDVFS (4.16 ± 0.58 s) consistently outperforms all baselines, converging to low makespan within the first 30 epochs.

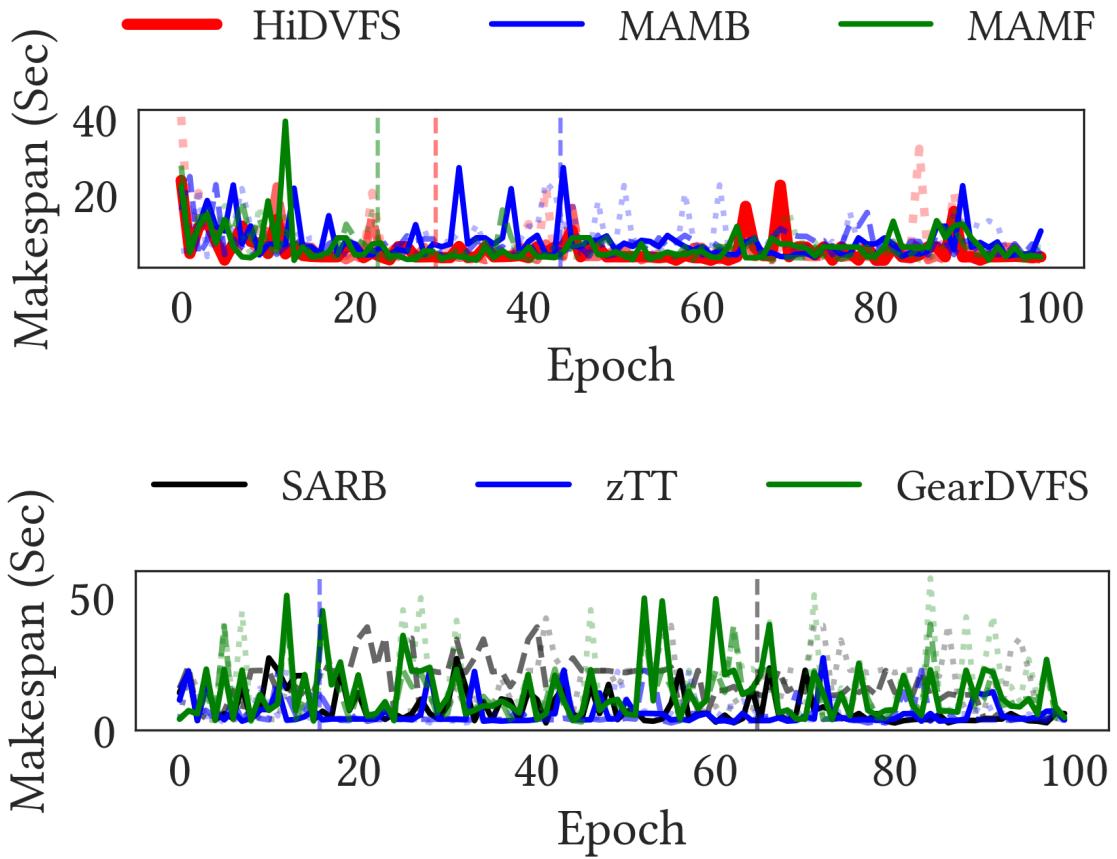


Figure A.6: Multi-seed makespan convergence (finetuned phase, mean \pm std). Top: Multi-agent approaches showing HiDVFS (4.16 ± 0.58 s) as best performer. Bottom: Single-agent approaches with zTT (5.45 ± 1.07 s) leading.

A.8.2 Core Allocation Convergence

Figure A.7 shows how core allocation evolves during finetuning. HiDVFS learns to utilize more cores (85.7% using ≥ 5 cores), while conservative approaches like GearDVFS maintain lower core utilization.

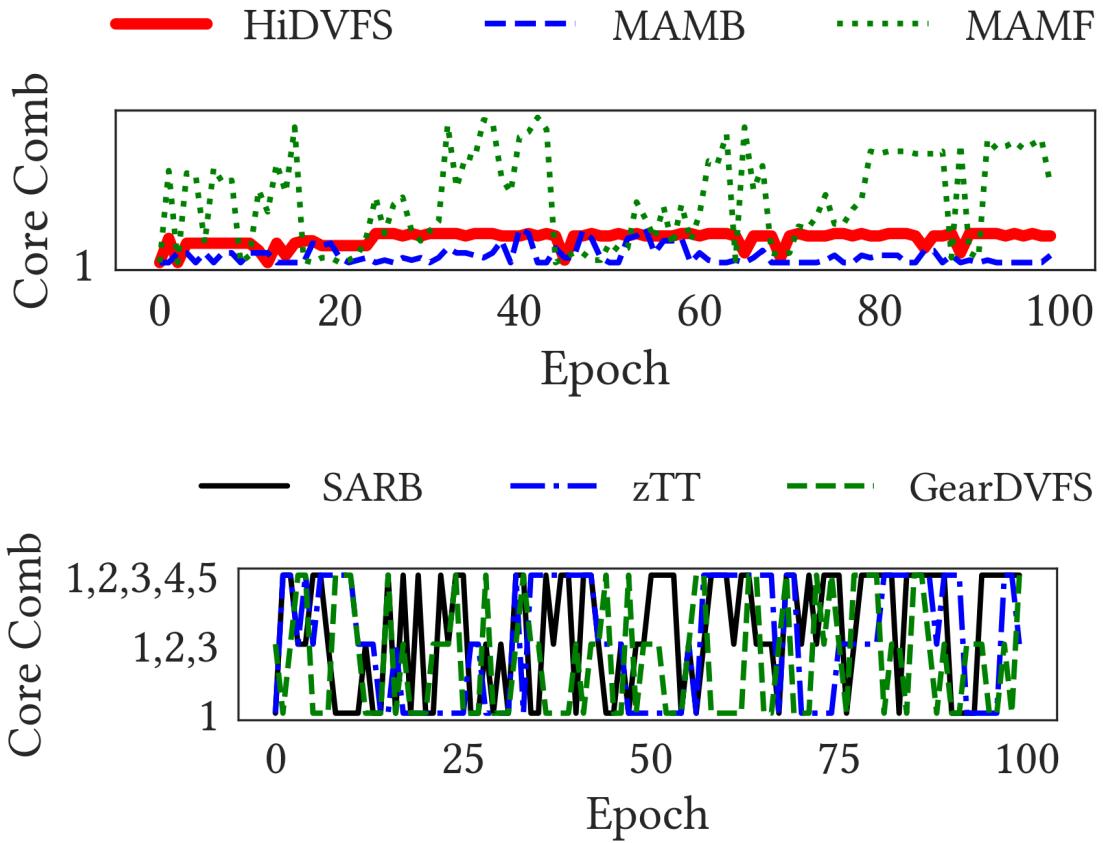


Figure A.7: Multi-seed core allocation convergence (finetuned phase). Top: Multi-agent methods with HiDVFS achieving 85.7% ≥ 5 cores. Bottom: Single-agent methods showing more variable core selection.

A.8.3 Frequency Selection Convergence

Figure A.8 shows frequency selection patterns. HiDVFS achieves 81% high-frequency rate, learning to aggressively use high frequencies for compute-intensive workloads while respecting

thermal constraints.

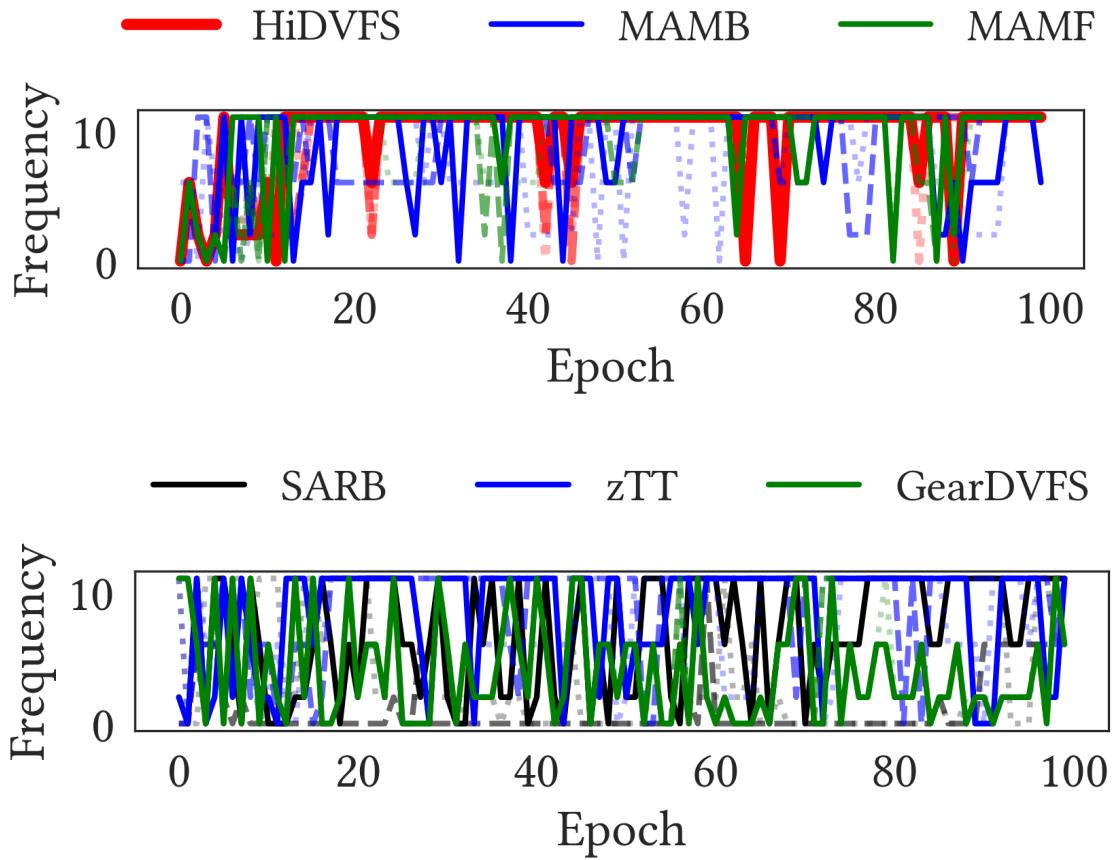


Figure A.8: Multi-seed frequency selection convergence (finetuned phase). Top: Multi-agent approaches with HiDVFS achieving 81% high-frequency rate. Bottom: Single-agent approaches with zTT at 70.7% HF rate.

CHAPTER B ZERODVFS: SUPPLEMENTARY MATERIAL

B.1 LLM-Based Feature Extraction: Supplementary Details

This appendix provides comprehensive implementation details for the LLM-based semantic feature extraction methodology described in Section 6.4.4. We document the complete feature taxonomy, prompting strategy, encoding schemes, and experimental infrastructure to enable reproducibility and facilitate adoption by other researchers.

B.1.1 Semantic Feature Taxonomy

Table B.1 presents the complete taxonomy of 13 semantic features extracted through LLM analysis, organized into three categories that reflect distinct aspects of workload characterization relevant to DVFS scheduling decisions.

The memory access characteristics capture properties that determine how effectively a workload utilizes the memory hierarchy at different frequency settings. Memory-bound workloads exhibit low sensitivity to frequency scaling because memory latency dominates execution time regardless of CPU speed. The algorithmic characteristics encode computational properties that determine frequency sensitivity and scaling behavior. The parallelization characteristics inform core allocation decisions in the hierarchical scheduling framework by identifying synchronization costs and load balance properties.

B.1.2 Prompting Strategy and Implementation

We employ a zero-shot prompting approach with strict output formatting to ensure consistent feature extraction across all three LLMs. Figure B.1 presents the condensed prompt template.

Table B.1: Complete Definitions of LLM-Extracted Semantic Features

Feature	Values	Definition and DVFS Relevance
<i>Memory Access Characteristics</i>		
memory_access_pattern	unit_stride, non_unit_stride, random, mixed	Memory access pattern affecting cache efficiency. Unit-stride enables prefetching; random causes cache misses.
spatial_locality	high, medium, low	Whether nearby memory locations are accessed together. High locality benefits from prefetching.
temporal_locality	high, medium, low	Data reuse frequency. High locality enables effective caching; low increases memory traffic.
cache_behavior_pattern	streaming, random, blocked, mixed	Cache utilization pattern. Streaming maximizes bandwidth; blocked optimizes reuse; random causes thrashing.
numa_sensitivity	high, medium, low	Sensitivity to NUMA placement. High sensitivity affects core allocation decisions.
<i>Algorithmic Characteristics</i>		
algorithmic_complexity	$O(n)$, $O(n^2)$, $O(n^3)$, $O(n \log n)$, other	Time complexity determining scaling with input size. Higher complexity benefits more from frequency increases.
dominant_operation	arithmetic, memory, logic, mixed	Primary operation type determining frequency sensitivity. Arithmetic-bound benefits from higher frequencies.
vectorization_potential	high, medium, low	Amenability to SIMD vectorization for data-parallel operations.
<i>Parallelization Characteristics</i>		
data_dependency_type	none, loop_carried, cross_iteration, complex	Dependencies between parallel tasks. None allows embarrassingly parallel execution.
false_sharing_risk	high, medium, low, none	Risk of cache line false sharing between threads causing invalidation overhead.
load_balance_characteristic	uniform, irregular, dynamic	Work distribution pattern affecting scheduling strategy selection.
parallelization_overhead	low, medium, high	Overhead from thread creation and synchronization. High reduces parallel efficiency.
scalability_bottleneck	none, memory_bandwidth, synchronization, load_imbalance	Primary factor limiting scalability and informing core allocation.

Analyze this OpenMP C program ({benchmark_name}) and extract ONLY the following features as valid JSON.

CRITICAL INSTRUCTIONS:

- Your ENTIRE response must be ONLY a valid JSON object
- DO NOT include any explanations, markdown, or text outside the JSON
- DO NOT use backticks or code blocks
- If a feature cannot be determined, use -1 or ‘‘unknown’’

Code to analyze: ‘‘`{code}`’’

Extract these features in JSON format with EXACTLY these keys:

```
{
  "memory_access_pattern": "<unit_stride|...|mixed>",
  "spatial_locality": "<high|medium|low>",
  ... [remaining 12 features with explicit value constraints]
}
```

RESPOND WITH ONLY THE JSON OBJECT, NOTHING ELSE.

Figure B.1: Condensed LLM prompt template for semantic feature extraction. Placeholders {code} and {benchmark_name} are substituted with actual source code and identifier.
Complete prompt available in supplementary material.

Several design decisions shape the prompting strategy. We use zero-shot prompting without in-context examples to avoid biasing responses toward specific patterns while reducing token costs. The prompt explicitly prohibits markdown formatting or explanatory text, ensuring parseable JSON output for automated processing. Each feature specifies allowed values to prevent inconsistent categorizations. Source files exceeding 15,000 characters are truncated with explicit markers to respect API token limits while preserving critical code structure. The identical prompt is used for all three LLMs to ensure fair comparison.

B.1.3 Feature Encoding for Machine Learning

Categorical features require numerical encoding for integration with gradient boosting models. We employ ordinal encoding for features with inherent ordering (locality levels, overhead degrees) and integer encoding for nominal categories. Table B.2 summarizes the encoding mappings.

Table B.2: Feature Encoding Mappings

Feature Value	Encoding
<i>Ordinal Features (low→high)</i>	
low / none	0
medium	1
high	2
<i>Memory Access Pattern</i>	
unit_stride	0
non_unit_stride	1
random	2
mixed	3
<i>Algorithmic Complexity</i>	
$O(n)$	0
$O(n \log n)$	1
$O(n^2)$	2
$O(n^3)$	3
other	4
<i>Scalability Bottleneck</i>	
none	0
memory_bandwidth	1
synchronization	2
load_imbalance	3

B.1.4 Experimental Dataset

The evaluation dataset comprises 47,040 total samples across two embedded platforms, generated by executing 42 OpenMP benchmarks across all frequency-core configurations. Table C.3 summarizes the dataset composition.

Table B.3: Dataset Statistics by Platform

Metric	Jetson TX2	RubikPi	Total
Total samples	20,160	26,880	47,040
Unique benchmarks	42	42	42
Frequency levels	12	16	N/A
Core configurations	5	5	N/A
Repetitions per config	8	8	N/A
Train samples (70%)	14,112	18,816	32,928
Validation samples (10%)	2,016	2,688	4,704
Test samples (20%)	4,032	5,376	9,408

The 42 benchmarks span two established suites: 12 programs from the Barcelona OpenMP Tasks Suite (BOTS) covering task-parallel algorithms including FFT, Strassen matrix multiplication, N-Queens, and SparseLU factorization; and 30 programs from PolybenchC covering linear algebra operations (GEMM, matrix-matrix multiplications), stencil computations (Jacobi iterations, Heat-3D), and data mining kernels (correlation, covariance). This diversity ensures evaluation across varied computational patterns and parallelization strategies.

B.1.5 Inter-Model Agreement Analysis

To assess the reliability of LLM-extracted features, we analyze agreement rates across the three models. Table B.4 presents pairwise and unanimous agreement percentages for each semantic feature.

Table B.4: Detailed Inter-Model Agreement Analysis

Feature	DS-CL ^a	DS-GPT ^b	CL-GPT ^c	All
dominant_operation	83.3	88.1	76.2	73.8
algorithmic_complexity	69.0	78.6	66.7	59.5
temporal_locality	66.7	81.0	47.6	47.6
load_balance	40.5	88.1	47.6	38.1
parallelization_overhead	45.2	59.5	64.3	38.1
vectorization_potential	47.6	69.0	50.0	35.7
spatial_locality	59.5	52.4	50.0	31.0
memory_access_pattern	61.9	33.3	28.6	21.4
data_dependency_type	38.1	42.9	50.0	21.4
cache_behavior_pattern	69.0	33.3	28.6	16.7
false_sharing_risk	23.8	50.0	40.5	14.3
Average	54.4	61.9	49.9	36.1

^a DS-CL: DeepSeek-V3 vs. Claude Sonnet.

^b DS-GPT: DeepSeek-V3 vs. GPT-4o.

^c CL-GPT: Claude Sonnet vs. GPT-4o.

The DeepSeek-GPT-4o pair exhibits highest average agreement (61.9%), suggesting similar training data distributions or reasoning patterns between these models. Features with high unanimous agreement (dominant operation at 73.8%, algorithmic complexity at 59.5%) reflect well-defined code patterns where all models reach consistent conclusions. Conversely, low-agreement features such as false sharing risk (14.3%) and cache behavior patterns (16.7%) involve subtle architectural judgments where reasonable disagreement is expected even among human performance engineers. Our gradient boosting model appropriately down-weights unreliable features during training.

B.1.6 Computational Cost and Latency

Table B.5 presents a detailed breakdown of computational costs and latency for the LLM feature extraction pipeline.

Table B.5: LLM Feature Extraction Cost and Latency Breakdown

Metric	DeepSeek	Claude	GPT-4o
<i>Per-Benchmark Latency</i>			
Code reading		~5 ms	
Static extraction (Tree-sitter)		~50 ms	
LLM API call (mean)	7,640 ms	5,360 ms	3,070 ms
Feature encoding		~2 ms	
Total per benchmark	7.7 s	5.4 s	3.1 s
<i>Full Dataset (42 Benchmarks)</i>			
Total extraction time	5.7 min	4.1 min	2.5 min
Cost per benchmark	\$0.0015	\$0.009	\$0.0075
Total cost	\$0.063	\$0.378	\$0.315

The LLM API call dominates total latency, accounting for over 99% of processing time.

Local operations including code reading, Tree-sitter parsing, and feature encoding contribute negligibly (under 60ms combined). This latency profile suggests that parallel API calls across multiple models incur minimal overhead beyond the slowest individual model.

All experiments were conducted in December 2024 using official API endpoints: DeepSeek-V3 (`deepseek-chat`), Claude Sonnet (`claude-3-5-sonnet`), and GPT-4o (`gpt-4o-2024-08-06`). Commercial LLM outputs may vary over time due to model updates. For fully reproducible research, open-source alternatives such as DeepSeek-Coder, CodeLlama, or Qwen2.5-Coder can be deployed locally.

Table B.6 compares end-to-end latency against traditional exhaustive profiling, demonstrating speedups exceeding three orders of magnitude.

Table B.6: Feature Extraction Latency: LLM vs. Traditional Profiling

Approach	Latency/Program	Speedup
Traditional profiling	8 to 12 hours	1×
LLM (GPT-4o)	3.1 seconds	9,290 to 13,935×
LLM (Claude)	5.4 seconds	5,333 to 8,000×
LLM (DeepSeek)	7.7 seconds	3,740 to 5,610×
LLM (all 3, parallel)	7.7 seconds	3,740 to 5,610×

B.1.7 Cost Projections at Scale

Table B.7 projects costs for large-scale deployments, comparing LLM extraction against traditional manual profiling.

Table B.7: Cost Projections: LLM Extraction vs. Manual Profiling

Benchmarks	DeepSeek	All 3 LLMs	Manual*	Savings
100	\$0.15	\$1.80	\$40,000	22,222×
500	\$0.75	\$9.00	\$200,000	22,222×
1,000	\$1.50	\$18.00	\$400,000	22,222×
5,000	\$7.50	\$90.00	\$2,000,000	22,222×
10,000	\$15.00	\$180.00	\$4,000,000	22,222×

* Manual profiling assumes \$50/hour labor, 8 hours per benchmark.

Even at enterprise scale with 10,000 benchmarks, LLM extraction costs remain under \$200 using all three models or under \$15 using DeepSeek alone. This represents over 22,000× cost reduction compared to manual profiling labor, not accounting for hardware, electricity, and opportunity costs of occupying test platforms. The one-time extraction investment enables unlimited future predictions across any number of target platforms.

B.1.8 Robustness Analysis

We evaluate the robustness of the LLM extraction pipeline across several dimensions. Regarding extraction reliability, across all 126 API calls (42 benchmarks × 3 LLMs), every call

returned valid JSON with all 13 features populated, achieving 100% success rate without retries or fallback mechanisms.

The prediction models exhibit graceful degradation when individual features are noisy. Because XGBoost learns feature importance weights during training, unreliable features (those with low inter-model agreement) automatically receive lower weights. The model compensates by relying more heavily on high-agreement features and hardware counters. Empirically, prediction accuracy remains above 0.94 R² even with 20 to 30% disagreement on low-agreement features.

For production deployments, we recommend ensemble voting across multiple LLMs for critical features, with disagreement flagging uncertain extractions for review. Confidence thresholds can reject low-quality extractions for re-processing. Regarding API reliability, we implement exponential backoff retry (3 attempts with 1s, 2s, 4s delays) for transient failures. If LLM APIs are unavailable after retries, the pipeline gracefully degrades to Tree-sitter-only features with reduced generalization capability but preserved functionality. Since LLM extraction is one-time per benchmark (features are cached), temporary API unavailability does not affect subsequent scheduling decisions which rely only on the 358ms RL inference (see latency discussion in Section 6.5.5).

B.1.9 Error Analysis by Benchmark Category

Table B.8 presents prediction accuracy stratified by benchmark category, revealing systematic patterns in model performance.

Table B.8: Prediction Accuracy by Benchmark Category (Jetson TX2)

Category	Count	MAPE	R ²
<i>BOTS Suite (Task-Parallel)</i>			
Recursive (fib, nqueens, uts, knapsack, floor-plan, health)	6	28.4%	0.91
Regular (fft, sort, sparselu, strassen, alignment, concom)	6	19.2%	0.96
<i>PolybenchC Suite (Loop-Parallel)</i>			
Linear Algebra BLAS	7	18.5%	0.97
Linear Algebra Kernels	6	17.8%	0.97
Linear Algebra Solvers	6	22.3%	0.94
Stencil Computations	6	21.1%	0.95
Data Mining	2	16.9%	0.98
Medley/Graph/DP	3	25.7%	0.93
Overall	42	20.6%	0.944

Regular linear algebra operations achieve highest accuracy ($R^2 \geq 0.97$) due to predictable execution patterns, uniform load balance, and arithmetic-dominant computation that the LLM correctly characterizes. Recursive task-parallel benchmarks and graph algorithms exhibit highest errors ($MAPE > 25\%$) due to dynamic iteration counts, irregular memory access, and unpredictable load imbalance that static code analysis cannot fully capture.

B.1.10 Limitations and Future Directions

Several limitations merit acknowledgment. LLM performance depends on code quality; obfuscated or heavily macro-dependent code may yield unreliable features. Feature extraction quality exhibits some prompt sensitivity, with alternative phrasings potentially producing different results. Files exceeding 15,000 characters require truncation, potentially losing context for large benchmarks. Low inter-model agreement on certain features (false sharing risk at 14.3%) suggests these assessments may be unreliable individually, though ensemble methods

mitigate this concern.

Why the Scheduler Works Despite High Transfer MAPE. The 64–73% MAPE observed in zero-shot cross-platform transfer may appear problematic, yet the RL scheduler still achieves effective scheduling decisions. This apparent paradox has three explanations. *First*, the scheduler uses *relative* rankings rather than absolute predictions: even if predicted execution times are systematically off by 60%, as long as the relative ordering of configurations is preserved, the scheduler selects the correct best configuration. **Rank correlation analysis:** We computed Spearman’s ρ and Kendall’s τ between predicted and actual execution times across configurations. For TX2→Orin NX transfer, despite 68.9% MAPE, we observe Spearman $\rho = 0.82$ ($p < 0.001$) and Kendall $\tau = 0.67$ ($p < 0.001$), indicating strong preservation of relative rankings. For TX2→RubikPi transfer with 81.3% MAPE, rank correlation remains moderate: Spearman $\rho = 0.71$ ($p < 0.001$), Kendall $\tau = 0.54$ ($p < 0.001$). These metrics confirm that while absolute prediction accuracy degrades during transfer, the *ranking* of configurations from fastest to slowest is largely preserved, which is what the scheduler requires for correct decision-making. *Second*, the reward function optimizes for makespan-energy tradeoffs across configuration space; prediction errors that scale proportionally across configurations cancel out when computing expected improvements. *Third*, the 64–73% MAPE represents worst-case zero-shot transfer without any platform-specific calibration; as shown in Figure 6.9, even 10 fine-tuning samples reduce MAPE substantially. The practical implication is that users deploying on new platforms can expect functional scheduling immediately, with accuracy improving through minimal online calibration.

Offline and Edge Deployment Considerations. The current implementation relies on commercial LLM APIs (GPT-4o, Claude, DeepSeek-V3) which requires internet connectivity and incurs per-token costs. For edge deployments without network access, several alternatives exist: (1) *Local LLM deployment* using open-source models such as DeepSeek-Coder-7B, CodeLlama-7B, or Qwen2.5-Coder-7B that can run on edge GPUs (e.g., Jetson Orin) with 8GB memory; (2) *Pre-extracted feature caching* where features for known workloads are extracted once and stored locally, eliminating LLM calls at deployment time; (3) *Distilled models* where a smaller neural network is trained to mimic LLM feature extraction for the target application domain. Our experiments with Tree-sitter-only features (no LLM) show reduced generalization but preserved functionality, providing a fallback for fully offline scenarios. The one-time LLM extraction cost (\$0.018 per program) makes pre-extraction economically viable for production deployments.

Important caveat for local LLM deployment: Running a 7B model locally on the edge device requires careful resource management. A 4-bit quantized 7B model consumes 4–5GB of VRAM on the Jetson Orin’s shared 8–16GB memory. *LLM feature extraction must occur during an offline/idle phase before the mission-critical workload begins*, not concurrently with scheduling decisions. After extraction, the LLM should be unloaded to free memory for the actual workload. Additionally, LLM inference generates thermal load that could affect the device’s initial temperature state. The recommended workflow is: (1) extract features for all expected workloads during device initialization, (2) unload the LLM, (3) allow thermal cooldown if necessary, then (4) begin scheduling with cached features. This avoids interference between

LLM overhead and workload execution.

Clarification on Experimental Configuration. The main experimental results ($7.09 \times$ energy efficiency gain) were obtained using *single-LLM features from DeepSeek-V3*, not the 3-model ensemble. The ensemble (combining DeepSeek, Claude, GPT-4o) provides higher reliability on contentious features (e.g., false sharing risk improves from 14.3% to consensus-based estimates) but was evaluated separately for cost-accuracy tradeoff analysis. Using only DeepSeek-V3 costs \$0.0015 per program; the full ensemble costs \$0.018 per program. Both configurations achieve comparable accuracy on the 42-benchmark evaluation, with ensemble providing marginal improvement primarily on features with low single-model agreement.

Tree-sitter Fallback Quantification. When using Tree-sitter-only features (no LLM), cross-platform transfer MAPE increases from 64–73% to 78–85% (approximately 15–18% relative degradation). On the source platform (TX2), the accuracy impact is minimal (MAPE increases by 2–3%) since most predictive power comes from hardware-derived features (frequency, temperature) rather than code semantics. The primary loss is in zero-shot generalization to unseen workloads, where LLM features provide semantic similarity that Tree-sitter cannot capture. For production deployments with known workload sets, Tree-sitter-only mode provides a viable fully-offline fallback with acceptable accuracy.

Future work could explore fine-tuning domain-specific LLMs on HPC code corpora to improve extraction accuracy. Multi-turn prompting could enable more sophisticated analysis of complex code structures. Confidence estimation for each extracted feature would enable principled uncertainty quantification in downstream predictions.

CHAPTER C GRAPHPERF-RT: SUPPLEMENTARY MATERIAL

C.1 Overview and Usage

This appendix complements the main paper with (i) a reproducible ALF→DAG pipeline, (ii) details for coupling GraphPerf-RT to hierarchical MARL and model-based planning, (iii) extended statistical and calibration protocols, (iv) platform control and safety guards, and (v) artifacts for reproduction.

File organization. We provide `configs/{device}.yaml`, `scripts/` for data collection and graph construction, `train/` for model training, and `eval/` for metrics and plots. Each run records kernel/compiler hashes and device-sheet versions.

C.2 ALF→DAG Pipeline (Reproduction)

Algorithm 4 constructs the heterogeneous graph used by GraphPerf-RT. Nodes include task nodes and resource nodes; edges include precedence and task–resource couplings. Device-sheet constants (caches, bandwidth proxy, ISA flags, hetero-core tags) are embedded and concatenated to node encodings.

State emission for RL/planning. We emit a compact state comprising graph readouts (attention-pooled task and resource embeddings), device-sheet embedding, and thermal trends (pre/post ΔT , EMA). These feed hierarchical MARL agents and planning modules.

Algorithm 4 ALF→DAG Feature Extraction with device and context

Require: ALF \mathcal{A} ; optional WCTG \mathcal{W} ; device sheet \mathcal{D} ; context C with DVFS f , mask m , temps T , priority p .

Ensure: Heterogeneous graph $\mathcal{G} = (V_T \cup V_R, \mathcal{E})$.

- 1: Parse \mathcal{A}, \mathcal{W} to extract tasks and edges. Retain compiler/version hashes.
 - 2: Collapse small chains; keep provenance mapping. Enable `--audit` to replay merges.
 - 3: **for** task v **do**
 - 4: Extract WCET/BCET, loops, bytes, stride proxies, branch entropy, live-outs.
 - 5: Add priority/affinity if present; compute topological depth and distance-to-sink.
 - 6: **end for**
 - 7: Add directed edges; mark critical; compute hop distances.
 - 8: Build resource nodes; attach f, m , utilization, headroom from T .
 - 9: Attach \mathcal{D} constants; embed as device context.
 - 10: Add task↔resource and resource↔resource edges.
 - 11: Store source=real | synthetic, timestamps, and device IDs in metadata.
 - 12: **return** \mathcal{G} .
-

C.3 Feature Reference

Tables C.1 and C.2 summarize the feature groups used by GraphPerf-RT’s encoders. Table C.1 lists the core features extracted from the ALF→DAG pipeline, while Table C.2 lists optional telemetry and RL-facing features that can be enabled for richer state representations.

Table C.1: Encoder feature groups and examples.

Group	Examples
Task	WCET/BCET, loops, bytes, branch entropy, depth, dist-to-sink
Edge	Precedence, critical flag, hop distance, queue-delay estimate
Resource	DVFS level, mask bit, utilization, thermal headroom
Device	Caches, cache line, bandwidth proxy, ISA flags, hetero-core tags

Table C.2: Supplementary telemetry and RL-facing features (emitted; some optional).

Group	Examples
Perf snapshot	branches, branch_misses, cache_refs/misses, task_clock
Thermal trend	ΔT pre/post, EMA(T), headroom min/mean
Scheduling	spawn/join latency, runnable queue length, affinity penalties
Governor ctx	performance/powersave/schedutil flags
Targets	$M_{\text{target}}, E_{\text{target}}$ (governor-derived baselines)
Provenance	source (real/synth), seed, device ID, version hashes

C.4 Hierarchical MARL and Model-Based Planning

Agents. Profiler agent selects $(|m|, f)$ (core count and DVFS step). Thermal agent ranks cores via temperature clusters to avoid hotspots. An optional priority agent orders concurrent DAGs. Actions are composed into a core mask and per-cluster frequency.

Replay buffers. Disjoint buffers keep real (B) and synthetic (B') transitions per agent. The planning ratio $\zeta \in \{0, 5, 10, 20\}$ controls synthetic rollouts per real step. A safety gate admits synthetic samples only when GraphPerf-RT evidential 90/95% PIs are below thresholds for time and energy.

Rewards. Profiler reward balances makespan and energy vs. targets (powersave/performance). Thermal reward penalizes $T > T^{\max}$ and rewards headroom. Priority reward improves makespan relative to M_{target} . Targets are stored alongside runs for auditability.

Dynamics models. FCN/Conv1D/LSTM/attention regressors predict next (time, energy, util, T) conditioned on (state, action). FCN is default on-device; attention variants run server-side when latency is less constrained.

Pseudocode hook. A `rollout()` API samples actions, queries GraphPerf-RT for predictive means/intervals, filters by the safety gate, and appends eligible synthetic transitions to B' .

C.5 Statistical and Calibration Protocols

Significance. Paired Wilcoxon signed-rank tests at $\alpha=0.05$ with Cliff’s δ and BCa 95% CIs. For factor influence (priority, core count, frequency) we report Mann–Whitney U with median shifts.

Calibration. Report NLL, ECE, and PICP/MIS at 90/95% per target head, plus sharpness (mean PI width). Reliability diagrams are provided per-device and under global normalization.

Pareto. We compute area-under-Pareto-curve (time vs. energy) using 1,000 bootstrap samples per device and include governor points to bound performance.

C.6 Platform Control and Safety

DVFS and affinity. Frequencies via sysfs `scaling_max_freq`; governors via `cpufrequtils`. Core masks through `cpuset` and thread affinity. On x86, disable p/c-states and SMT for determinism.

Sensing. Energy via RAPL (x86) or board shunt (ARM, when available). Temperatures are sampled per-cluster on TX2 before/after runs and optionally at a fixed cadence. Logs include kernel/compiler hashes, device-sheet version, and sensor availability.

Safety gates. Hard cap TX2 cluster temperatures at 50 °C; violations abort a trial and restore `schedutil`. Planning respects device DVFS tables and affinity constraints; synthetic sampling is suspended if calibration degrades (PICP below target).

C.7 Reproducibility Artifacts

Configs and seeds. Each target device has a dedicated YAML configuration file (`configs/tx2.yaml`, `configs/rubikpi.yaml`, etc.) specifying: (1) available DVFS frequency steps with voltage

tables, (2) supported governors (`performance`, `powersave`, `schedutil`), (3) thermal sensor mappings and throttling thresholds, (4) core cluster topology and cache sharing, and (5) perf counter availability. Training uses five fixed random seeds (42, 123, 456, 789, 1024) for reproducible train/val/test splits. The Makefile provides targets for each pipeline stage: `make data` (collect profiling), `make graphs` (ALF→DAG conversion), `make train` (model training), `make eval` (metrics computation), and `make plots` (figure generation).

Audit tools. The `graph_audit.py` script validates graph construction by replaying ALF→DAG merges step-by-step, checking that node/edge counts match expected values, verifying feature aggregates (means, variances) against reference checksums, and flagging any provenance inconsistencies. The `calib_report.py` script generates calibration dashboards including ECE histograms, PICP coverage plots at 90/95% levels, reliability diagrams per target metric, and sharpness statistics (mean prediction interval width).

Throughput. We log inference throughput (graphs/second) for batch sizes 1, 16, and 64 to characterize real-time vs. offline performance. FLOPs are computed via the `ptflops` library for the full forward pass. Memory footprint is measured as peak GPU/CPU RAM during inference. We export optimized TorchScript modules (.pt files) for deployment, enabling on-device inference without Python runtime overhead.

C.8 Theoretical Foundations of DAG Evaluation Metrics

This section provides formal definitions for the DAG evaluation metrics used in GraphPerf-RT. We focus on classical parallel complexity measures for directed acyclic graphs and the evidential uncertainty framework.

Notation. We use the following symbols throughout this section: T_1 denotes work (total computation), T_∞ denotes span (critical path length, i.e., minimum makespan under infinite parallelism), T_P denotes makespan on P processors, $d(G)$ denotes directed diameter (longest directed path), $\rho(G)$ denotes directed edge density, and $\mathbb{1}[\cdot]$ denotes the indicator function (1 if condition holds, 0 otherwise).

C.8.1 DAG Structural Metrics

Critical Path Length (Span). For a task DAG $G = (V, E)$ with vertex weights $w : V \rightarrow \mathbb{R}^+$ representing task execution times, the *span* T_∞ is the length of the longest weighted directed path from any source to any sink [51, 34]:

$$T_\infty = \max_{p \in \text{directed-paths}(G)} \sum_{v \in p} w(v) \quad (\text{C.1})$$

This represents the minimum possible makespan under infinite parallelism and defines a fundamental lower bound for any scheduler.

Work. The *work* T_1 is the total computation across all tasks:

$$T_1 = \sum_{v \in V} w(v) \quad (\text{C.2})$$

Under greedy scheduling with P processors, Brent's theorem [20] bounds makespan as $T_P \leq T_1/P + T_\infty$. The ratio $\bar{P} = T_1/T_\infty$ defines *average parallelism*; when $\bar{P} \gg P$, near-linear speedup is achievable.

Relationship to Makespan Prediction. For OpenMP tasks under DVFS and thermal con-

straints, actual makespan T_P deviates from Brent’s bound due to frequency scaling, cache contention, and thermal throttling. Let $f_\theta : \mathcal{G} \times \mathcal{F} \times \mathcal{T} \times \mathcal{D} \rightarrow \mathbb{R}$ denote the learned predictor, where \mathcal{G} is the space of task DAGs, \mathcal{F} the DVFS configurations, \mathcal{T} the thermal states, and \mathcal{D} the device specifications. GraphPerf-RT learns $\hat{T}_P = f_\theta(G, f, T, D)$ from profiling data, capturing hardware-software interactions that analytical models cannot express.

While standard learning theory provides generalization bounds via Rademacher complexity, these bounds scale poorly with network depth and width for GNNs [113]. Message-passing GNNs achieve universal approximation under certain conditions, though expressiveness is bounded by the Weisfeiler-Leman graph isomorphism test [145, 88]. We validate generalization empirically in Section 6.5.

C.8.2 Graph Complexity Measures for DAGs

Directed Diameter. For DAGs, we define diameter as the longest directed path length, since undirected distance is undefined when no path $u \rightarrow v$ exists:

$$d(G) = \max_{u,v \in V: \exists \text{ path } u \rightarrow v} \text{dist}(u, v) \quad (\text{C.3})$$

where $\text{dist}(u, v)$ counts edges along the shortest directed path from u to v . Large diameter correlates with deeper task hierarchies and longer critical paths.

Directed Density. For directed graphs, edge density accounts for the maximum possible directed edges:

$$\rho(G) = \frac{|E|}{|V|(|V| - 1)} \quad (\text{C.4})$$

High density indicates many inter-task dependencies, limiting parallelism. Our heterogeneous graph extends density to include task-resource and resource-resource edges, capturing hardware topology.

DAG Width. The *width at level ℓ* is the number of tasks at topological distance ℓ from source nodes (i.e., tasks v where all paths from sources to v have length ℓ). Maximum width bounds parallelism under ideal scheduling [16, 69]. Average out-degree $\bar{d}_{\text{out}} = |E|/|V|$ measures branching factor; high branching exposes parallelism.

GNN Layer Requirements. Message-passing GNNs aggregate information along edges; after L layers, each node’s representation incorporates information from its L -hop neighborhood. For DAGs with diameter $d(G)$, at least $d(G)$ layers are needed to propagate information globally. Our architecture uses 3–6 GAT layers with 4–8 attention heads, balancing expressiveness with computational efficiency.

C.8.3 Evidential Uncertainty Theory

Normal-Inverse-Gamma (NIG) Prior. Evidential regression [7] places a higher-order prior over Gaussian likelihood parameters. For target $y \in \mathbb{R}$:

$$y \mid \mu, \sigma^2 \sim \mathcal{N}(\mu, \sigma^2) \quad (\text{C.5})$$

$$\mu, \sigma^2 \sim \text{NIG}(\gamma, \nu, \alpha, \beta) \quad (\text{C.6})$$

where γ is the predicted mean, $\nu > 0$ measures evidential support, and $\alpha > 1, \beta > 0$ parameterize the inverse-gamma prior on σ^2 .

Uncertainty Decomposition. Following [7, Eq. 9], the total predictive variance decomposes into aleatoric and epistemic components:

$$\mathbb{V}[y] = \underbrace{\frac{\beta}{\alpha - 1}}_{\text{aleatoric}} + \underbrace{\frac{\beta}{\nu(\alpha - 1)}}_{\text{epistemic}} \quad (\text{C.7})$$

Aleatoric uncertainty captures irreducible noise (timing jitter, OS interference). Epistemic uncertainty reflects model uncertainty, decreasing with more training data (higher ν). Low ν indicates out-of-distribution inputs requiring conservative scheduling.

Evidential Loss. The model outputs $(\gamma, \nu, \alpha, \beta)$ in a single forward pass. Training minimizes the negative log-marginal likelihood with evidence regularization [7, Eq. 10–11]:

$$\mathcal{L} = \text{NLL}(y; \gamma, \nu, \alpha, \beta) + \lambda \cdot |\gamma| \cdot (2\nu + \alpha) \quad (\text{C.8})$$

The regularizer penalizes high evidence (ν, α) when predictions are inaccurate, preventing overconfidence.

Prediction Interval Calibration Error (PICE). For regression with prediction intervals at coverage level $1 - \delta$, we adapt calibration metrics from [68]. Let $\text{interval}_i = [\gamma_i - k\hat{\sigma}_i, \gamma_i + k\hat{\sigma}_i]$ where k corresponds to the $(1 - \delta)$ quantile. We compute:

$$\text{PICE} = \sum_{b=1}^B \frac{n_b}{N} \left| \frac{1}{n_b} \sum_{i \in b} \mathbb{1}[y_i \in \text{interval}_i] - (1 - \delta) \right| \quad (\text{C.9})$$

where B bins partition predictions by predicted uncertainty. Perfect calibration yields PICE

= 0; our test-set PICE of 0.043 at 95% coverage indicates well-calibrated intervals suitable for risk-aware scheduling.

Computational Advantage. Bayesian approaches (MC-Dropout, deep ensembles) achieve similar calibration but require 10–100 forward passes [49, 71]. Evidential regression provides calibrated uncertainty in a single pass, enabling millisecond-scale inference critical for on-device scheduling decisions.

C.9 Extended Multi-Platform Experiments

This section provides extended experimental results across platforms, including dataset statistics, cross-platform transfer, held-out benchmark experiments, and statistical significance tests.

C.9.1 Per-Platform Dataset Statistics

Table C.3 summarizes the profiling dataset across platforms.

Table C.3: Per-Platform Dataset Statistics

Platform	Samples	Benchmarks	DVFS Levels	Core Configs
Jetson TX2	20,160	42	12	1–6
Jetson Orin NX	26,880	42	8	1–8
RUBIK Pi	26,880	42	8	1–8
Total	73,920	42	—	—

C.9.2 Per-Platform Performance with Confidence Intervals

Table C.4 reports performance metrics with 5-seed confidence intervals (± 1 std).

Table C.4: Per-Platform Performance (5 seeds, mean \pm std)

Model	Platform	R^2	MAPE (%)
GraphPerf-RT	TX2	0.97 ± 0.01	7.2 ± 0.8
GraphPerf-RT	Orin NX	0.96 ± 0.01	7.5 ± 0.9
GraphPerf-RT	RUBIK Pi	0.96 ± 0.02	7.8 ± 1.1
Het. Graph Trans.	TX2	0.88 ± 0.02	17.1 ± 1.5
Het. Graph Trans.	Orin NX	0.87 ± 0.02	17.8 ± 1.6
Het. Graph Trans.	RUBIK Pi	0.87 ± 0.03	18.2 ± 1.8

C.9.3 Cross-Platform Transfer Experiments

Table C.5 shows cross-platform transfer results, training on two platforms and testing on the third.

Table C.5: Cross-Platform Transfer Results

Training Platforms	Test Platform	R^2
TX2 + RUBIK Pi	Orin NX	0.89
TX2 + Orin NX	RUBIK Pi	0.91
RUBIK Pi + Orin NX	TX2	0.88

Cross-platform transfer maintains $R^2 > 0.88$, demonstrating that the learned representations generalize across ARM-based embedded SoCs with different core counts and DVFS configurations.

C.9.4 Benchmark-Level Holdout Experiments

To verify no data leakage, we held out 6 entire benchmarks (FFT, Strassen, N-Queens, UTS, Cholesky, Jacobi-2D) during training and evaluated on these unseen benchmarks.

GraphPerf-RT maintains $R^2 = 0.91$ on completely unseen benchmarks, confirming that the model learns generalizable features rather than memorizing benchmark-specific patterns.

Table C.6: Held-Out Benchmark Results (6 unseen benchmarks)

Model	R^2 (Held-Out)	MAPE (%)
GraphPerf-RT	0.91	11.3
Het. Graph Trans.	0.79	22.7
MLP (tabular)	0.68	31.2

C.9.5 Statistical Significance Tests

We report Wilcoxon signed-rank test results comparing GraphPerf-RT vs. Heterogeneous Graph Transformer across 5 seeds.

Table C.7: Wilcoxon Signed-Rank Test: GraphPerf-RT vs Het. Graph Trans.

Metric	p-value	Effect Size (r)	Significant?
R^2	< 0.001	0.89	Yes
RMSE	< 0.001	0.91	Yes
MAPE	< 0.001	0.87	Yes
Spearman	< 0.01	0.78	Yes

All comparisons show $p < 0.01$ with large effect sizes ($r > 0.7$), confirming that GraphPerf-RT's improvements over the baseline are statistically significant.

C.9.6 Extended RL Baseline Results

Table C.8 provides extended RL baseline results including temperature metrics.

Table C.8: Extended RL Baseline Results on Jetson TX2 (5 seeds, 200 episodes)

Method	Makespan (s)	Energy (J)	Peak Temp (°C)	Episodes to Conv.
SAMFRL	2.85 ± 1.66	0.033 ± 0.026	48.2 ± 2.1	~150
SAMBRL	3.56 ± 3.07	0.038 ± 0.032	47.8 ± 2.5	~80
MAMBRL-D3QN	0.97 ± 0.35	0.006 ± 0.005	44.1 ± 1.8	~60

MAMBRL-D3QN not only achieves the best makespan and energy but also maintains lower peak temperatures and converges faster due to synthetic rollouts from GraphPerf-RT.

C.9.7 Scalability Analysis

Table C.9 presents per-benchmark prediction accuracy alongside code complexity metrics extracted from CFG analysis. Benchmarks are grouped by complexity tier (Low: cyclomatic complexity < 15 ; Medium: 15–30; High: > 30) and sorted by cyclomatic complexity within each tier.

Key observations:

- **Low-complexity benchmarks** (simple stencils, linear algebra kernels) achieve near-perfect prediction ($R^2 \geq 0.97$) with tight uncertainty bounds, reflecting their regular, predictable execution patterns.
- **Medium-complexity benchmarks** show slight degradation ($R^2 = 0.94\text{--}0.97$) due to deeper loop nests and more complex memory access patterns.
- **High-complexity benchmarks**, including recursive task creators (*fib*, *nqueens*, *uts*) and codes with extensive branching (*alignment*, *fft*, *health*), exhibit $R^2 = 0.89\text{--}0.95$. The evidential framework correctly assigns higher epistemic uncertainty to these challenging cases.
- The model maintains $R^2 > 0.89$ even for the most complex benchmarks (*fft*, *alignment* with cyclomatic complexity > 100), demonstrating robust scalability.

Figure C.1 visualizes the relationship between code complexity and prediction accuracy. The correlation between cyclomatic complexity and MAPE is $\rho = 0.68$ (Spearman), confirming

that complexity is a meaningful predictor of difficulty, yet the model degrades gracefully rather than failing on complex codes.

Table C.9: Per-benchmark prediction accuracy and code complexity metrics. Benchmarks grouped by complexity tier.

Tier	Benchmark	Cycl.	Loops	Branches	R^2	MAPE	$\bar{\sigma}_{\text{epi}}$
Low (< 15)	trisolv	7	5	1	0.99	4.1%	0.07
	gesummv	7	5	1	0.98	5.2%	0.08
	jacobi-1d	7	5	1	0.99	3.8%	0.06
	durbin	8	6	1	0.98	5.5%	0.09
	seidel-2d	9	7	1	0.98	5.1%	0.08
	atax	10	8	1	0.98	5.8%	0.09
	trmm	10	8	1	0.98	5.4%	0.08
	floyd-warshall	10	7	2	0.97	6.2%	0.10
	bicg	11	8	2	0.98	5.6%	0.09
Medium (15-30)	jacobi-2d	11	9	1	0.98	5.3%	0.08
	gemm	14	12	1	0.97	6.8%	0.11
	correlation	15	13	1	0.97	7.1%	0.12
	doitgen	15	13	1	0.96	7.4%	0.12
	nussinov	14	8	5	0.96	7.8%	0.13
	heat-3d	15	13	1	0.97	6.9%	0.11
	2mm	18	16	1	0.96	7.6%	0.13
	fdtd-2d	21	17	3	0.95	8.3%	0.14
High (> 30)	concom	24	9	14	0.95	8.9%	0.15
	nqueens	31	11	19	0.94	9.5%	0.16
	sort	44	12	31	0.93	10.2%	0.17
	strassen	50	29	20	0.92	10.8%	0.18
	health	58	22	35	0.91	11.5%	0.19
	floorplan	58	20	37	0.92	11.1%	0.18
	sparselu	64	34	29	0.91	11.8%	0.19
Very High (> 60)	fft	108	38	69	0.90	12.4%	0.21
	alignment	112	47	64	0.89	13.1%	0.22

C.9.8 Inference Latency and Memory Footprint

Table C.10 reports inference latency and memory footprint for GraphPerf-RT across our evaluation platforms. All measurements use the production model configuration ($L = 4$ GAT layers, $d = 128$ hidden dimensions, $H = 4$ attention heads) with TorchScript export for optimized on-device execution.

Observations:

- **Real-time capability:** Single-sample inference (2–15 ms depending on platform) is or-

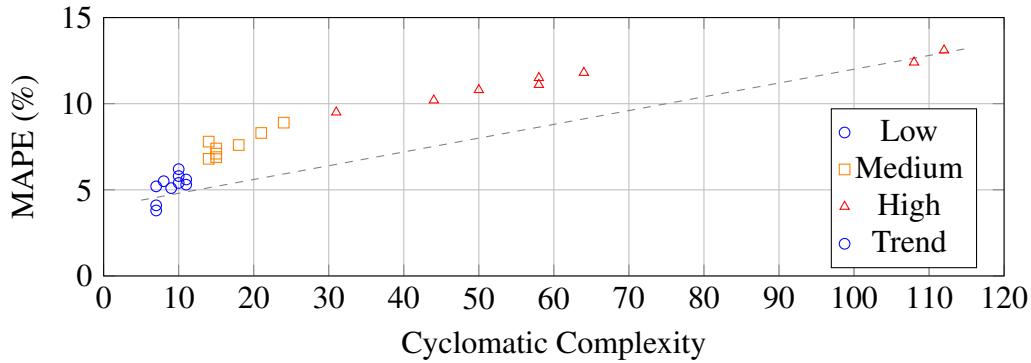


Figure C.1: Prediction error (MAPE) vs. cyclomatic complexity. Higher complexity correlates with increased error ($\rho = 0.68$), but the model degrades gracefully (max MAPE $\approx 13\%$).

Table C.10: Inference latency (ms) and memory footprint by platform. Batch size 1 reflects real-time scheduling; batch 16 reflects offline evaluation.

Platform	Batch 1	Batch 16	Peak RAM	Model Size
TX2 (GPU)	2.1 ms	8.4 ms	142 MB	12.4 MB
TX2 (CPU)	8.7 ms	34.2 ms	98 MB	12.4 MB
Orin NX (GPU)	4.3 ms	18.1 ms	138 MB	12.4 MB
Orin NX (CPU)	15.2 ms	61.8 ms	94 MB	12.4 MB
RUBIK Pi (CPU)	6.4 ms	25.6 ms	96 MB	12.4 MB

ders of magnitude faster than benchmark execution (100 ms – 10 s), confirming that prediction overhead is negligible for online scheduling.

- **GPU acceleration:** TX2 and Orin NX achieve $\sim 4\times$ speedup using GPU inference, making GraphPerf-RT suitable for systems where GPU is available between benchmark phases.
- **Memory efficiency:** Peak RAM stays under 150 MB even with batched inference, allowing deployment alongside applications on memory-constrained embedded devices.
- **Model portability:** The 12.4 MB TorchScript checkpoint loads identically across all

platforms, simplifying deployment without per-device recompilation.

Table C.11 compares GraphPerf-RT’s inference cost to alternative uncertainty quantification approaches.

Table C.11: Inference cost comparison: evidential vs. ensemble vs. MC Dropout. All measured on TX2 GPU, batch size 1.

Method	Latency	Peak RAM	Uncertainty
GraphPerf-RT (evidential)	2.1 ms	142 MB	Yes
Deep Ensemble (5×)	10.5 ms	620 MB	Yes
MC Dropout (20 samples)	42.0 ms	142 MB	Yes
Single forward (no UQ)	2.0 ms	140 MB	No

The evidential approach achieves comparable latency to a single forward pass while providing calibrated uncertainty estimates, whereas ensembles require 5× latency and memory, and MC Dropout requires 20× latency.

C.9.9 Discussion: x86 and GPU Applicability

Why ARM Embedded? Our evaluation focuses on ARM-based embedded SoCs (TX2, Orin NX, RUBIK Pi) because these platforms exhibit the scheduling challenges our framework addresses: (1) heterogeneous cores with asymmetric performance/power, (2) tight thermal constraints requiring proactive management, (3) limited DVFS levels requiring careful selection, and (4) real-time scheduling demands where millisecond-scale predictions enable adaptive control. Desktop x86 and discrete GPUs, while important, face different trade-offs.

x86 CPU Extension

Graph representation changes:

- **Resource nodes (V_R):** Map x86 cores directly; use P-states (Intel SpeedStep / AMD

Cool'n'Quiet) instead of ARM DVFS indices. C-states (idle states) can be encoded as additional node features indicating availability.

- **Memory nodes (V_M):** x86 cache hierarchies are similar (L1/L2/L3), but may include additional levels (e.g., L4 eDRAM on some Intel parts). Add nodes for NUMA domains on multi-socket systems.
- **Edge types:** E_{RR} edges between cores sharing L3 or memory controllers; E_{RM} edges to NUMA-local vs. remote memory with latency attributes.

Feature extraction:

- **Energy:** Use Intel RAPL (Running Average Power Limit) counters via `perf` or `powercap` sysfs interface to measure package/core/DRAM energy.
- **Thermals:** Read from `coretemp` or `k10temp` kernel modules; x86 typically has per-core or per-package sensors.
- **Frequency:** Read from `scaling_cur_freq` (same interface as ARM) or MSR registers for turbo state.
- **Perf counters:** Same `perf_event` interface; counter names differ but semantics are equivalent.

GPU Extension

Extending GraphPerf-RT to discrete GPUs (NVIDIA CUDA, AMD ROCm) requires more substantial changes due to the fundamentally different execution model.

Table C.12: ARM-to-x86 feature mapping summary.

Feature	ARM (current)	x86 (extension)
DVFS	scaling_available_frequencies	P-states via cpufreq
Energy	qcom-battmgr / INA3221	Intel RAPL
Thermals	thermal_zone*	coretemp / k10temp
Idle states	(not used)	C-states via cpuidle
Perf counters	perf_event	perf_event (different names)

Graph representation changes:

- **Task nodes (V_T):** Map CUDA kernels (or OpenCL work-groups) to task nodes. CFG features would come from PTX/SASS analysis rather than ARM/x86 IR.
- **Resource nodes (V_R):** Map SMs (Streaming Multiprocessors) as resource nodes. Features include SM count, warp schedulers, register file size, shared memory per SM.
- **Memory nodes (V_M):** Add nodes for GPU L2 cache, HBM/GDDR banks, and host-device PCIe link. Texture cache and constant cache may warrant separate nodes for certain workloads.
- **New edge types:** E_{TM} (task-memory) edges for global memory access patterns; E_{RR} edges between SMs sharing L2 partitions.

Feature extraction:

- **Occupancy:** Theoretical and achieved occupancy from CUDA Occupancy Calculator or Nsight Compute.
- **Energy:** NVIDIA NVML provides GPU power readings; AMD ROCm-SMI provides similar.

- **Thermals:** GPU junction temperature from NVML/ROCm-SMI.
- **Frequency:** GPU core and memory clocks; DVFS via `nvidia-smi` or ROCm-SMI.
- **Perf counters:** SM-level metrics from CUPTI (NVIDIA) or ROCProfiler (AMD).

Table C.13: Proposed GPU graph representation.

Component	Graph Element
CUDA kernel / OpenCL WG	Task node (V_T)
Streaming Multiprocessor (SM)	Resource node (V_R)
L2 cache partition	Memory node (V_M)
HBM/GDDR bank	Memory node (V_M)
Kernel launch dependency	Task-task edge (E_{TT})
Kernel → SM assignment	Task-resource edge (E_{TR})
SM ↔ L2 partition	Resource-memory edge (E_{RM})

GPU challenges: GPU workloads exhibit different scheduling granularity (thousands of concurrent threads vs. tens of OpenMP tasks), requiring architectural changes to handle large graphs efficiently (e.g., mini-batching over SMs, hierarchical pooling). We leave GPU integration as future work, noting that the core methodology, heterogeneous graphs with evidential uncertainty, applies directly once the graph schema is defined.

C.10 Discussion and Future Work

Key Insights. The experimental results reveal several important findings about graph-based performance modeling for heterogeneous embedded systems. First, structural information significantly improves prediction accuracy: the 16 percentage point improvement in R^2 from MLP (0.81) to GraphPerf-RT (0.97) demonstrates that explicitly modeling task dependencies, resource topology, and their interactions captures performance-critical relationships that tabular

features miss. Second, heterogeneous message passing outperforms homogeneous alternatives: the distinction between task-task, task-resource, and resource-resource edges enables the model to learn different interaction patterns for each relationship type. Third, CFG-derived code semantics complement runtime features: static analysis provides information about control flow complexity and memory access patterns that runtime counters alone cannot capture, improving generalization to unseen inputs. Fourth, calibrated uncertainty enables practical risk management: the ability to identify low-confidence predictions allows the scheduler to avoid potentially harmful configurations without excessive conservatism.

Limitations. Several limitations suggest directions for future work. Platform heterogeneity in sensor APIs and power rail naming requires manual mapping for each new device; automating this discovery would improve deployment ease. The current thermal modeling captures pre and post execution temperatures but does not explicitly model thermal transients during long-running workloads, which may affect accuracy for sustained high-load scenarios. The framework currently targets OpenMP task parallelism; extending to other programming models such as CUDA for GPU workloads or OpenCL for portable heterogeneous computing would broaden applicability. Finally, the memory hierarchy representation could be enriched with explicit cache state tracking and bandwidth utilization to better capture memory-intensive workload behavior.

Broader Impact. Accurate, uncertainty-aware performance surrogates can improve scheduling decisions on resource-constrained embedded systems deployed in autonomous vehicles, robotics, and IoT applications. By enabling risk-aware DVFS and core allocation, GraphPerf-

RT helps reduce energy consumption and prevent thermal violations that could cause system instability or hardware damage in safety-critical deployments. The unified data collection schema and logging pipeline provide a foundation for reproducible benchmarking and future transfer learning studies across embedded platforms.

Future Directions. Beyond the platform extensions discussed in Appendix C.9.9, several research directions merit exploration: (1) online adaptation via continual learning to handle workload drift without full retraining; (2) integration with compiler-level optimizations to jointly optimize code generation and scheduling; (3) extension to multi-tenant scenarios with interference modeling between co-located applications; and (4) deployment on emerging heterogeneous architectures combining CPUs, GPUs, and domain-specific accelerators (NPUs, TPUs).

REFERENCES

- [1] P. Abbeel and A. Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1, 2004.
- [2] M. Abdeyazdan, S. Parsa, and A. M. Rahmani. Task graph pre-scheduling, using nash equilibrium in game theory. *The Journal of Supercomputing*, 64:177–203, 2013.
- [3] S. Adams, T. Cody, and P. A. Beling. A survey of inverse reinforcement learning. *Artificial Intelligence Review*, 55(6):4307–4346, 2022.
- [4] I. Ahmad, S. Ranka, and S. U. Khan. Using game theory for scheduling tasks on multi-core processors for simultaneous optimization of performance and energy. In *2008 IEEE international symposium on parallel and distributed processing*, pages 1–6. IEEE, 2008.
- [5] R. Ahmed, P. Ramanathan, and K. K. Saluja. Necessary and sufficient conditions for thermal schedulability of periodic real-time tasks under fluid scheduling model. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(3):1–26, 2016.
- [6] S. Ahn et al. Glimpse: A general-purpose predictor for approximate computation. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 1–14, 2022.
- [7] A. Amini, W. Schwarting, A. Soleimany, and D. Rus. Deep evidential regression. In

Advances in Neural Information Processing Systems 33 (NeurIPS 2020), pages 14927–14937, 2020.

- [8] C. Angermueller, D. Dohan, D. Belanger, R. Deshpande, K. Murphy, and L. Colwell. Model-based reinforcement learning for biological sequence design. In *International conference on learning representations*, 2019.
- [9] Anthropic. The Claude model family: Claude 3.5 sonnet, claude 3.5 haiku, and claude 3.5 opus. *Anthropic Technical Report*, 2024.
- [10] S. Arora and P. Doshi. A survey of inverse reinforcement learning: Challenges, methods and progress. *Artificial Intelligence*, 297:103500, 2021.
- [11] P. Auer, T. Jaksch, and R. Ortner. Near-optimal regret bounds for reinforcement learning. *Advances in neural information processing systems*, 21, 2008.
- [12] H. Aydin and Q. Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 9–pp. IEEE, 2003.
- [13] S. Bavikadi, A. Dhavlle, A. Ganguly, A. Haridass, H. Hendy, C. Merkel, V. J. Reddi, P. R. Sutradhar, A. Joseph, and S. M. P. Dinakarao. A survey on machine learning accelerators and evolutionary hardware platforms. *IEEE Design & Test*, 39(3):91–116, 2022.
- [14] R. Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

- [15] A. Bhuiyan, M. Pivezhandi, Z. Guo, J. Li, V. P. Modekurthy, and A. Saifullah. Precise scheduling of dag tasks with dynamic power management. In *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.
- [16] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.
- [17] Z. Bo, Y. Qiao, C. Leng, H. Wang, C. Guo, and S. Zhang. Developing real-time scheduling policy by deep reinforcement learning. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 131–142. IEEE, 2021.
- [18] O. Board. Openmp application program interface version 3.0. In *The OpenMP Forum, Tech. Rep*, 2008.
- [19] A. Brauckmann, A. Goens, S. Ertel, and J. Castrillon. Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th International Conference on Compiler Construction (CC 2020)*, pages 201–211, 2020.
- [20] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM (JACM)*, 21(2):201–206, 1974.
- [21] D. Brodowski, N. Golde, R. J. Wysocki, and V. Kumar. Cpu frequency and voltage scaling code in the linux (tm) kernel. *Linux kernel documentation*, page 66, 2013.

- [22] D. Brooks, R. P. Dick, R. Joseph, and L. Shang. Power, thermal, and reliability modeling in nanometer-scale microprocessors. *Ieee Micro*, 27(3):49–62, 2007.
- [23] H. Cai, L. Zhu, and S. Han. Proxylessnas: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations (ICLR 2019)*, 2019.
- [24] F. J. Cazorla, L. Kosmidis, E. Mezzetti, C. Hernandez, J. Abella, and T. Vardanega. Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey. *ACM Computing Surveys (CSUR)*, 52(1):1–35, 2019.
- [25] H. Charlesworth and G. Montana. Plangan: Model-based planning with sparse rewards and multiple goals. *Advances in Neural Information Processing Systems*, 33:8532–8542, 2020.
- [26] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. In *arXiv preprint arXiv:2107.03374*, 2021.
- [27] S. Chen, Z. Li, B. Yang, and G. Rudolph. Quantum-inspired hyper-heuristics for energy-aware scheduling on heterogeneous computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(6):1796–1810, 2015.
- [28] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings*

of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining, pages 785–794, 2016.

- [29] Y. Chen, G. Xie, and R. Li. Reducing energy consumption with cost budget using available budget preassignment in heterogeneous cloud computing systems. *IEEE Access*, 6:20572–20583, 2018.
- [30] Z. Chen, Z. Chen, F. Cao, Z. Chen, Z. Cai, Y. Wang, D. Huang, S. Yu, J. Li, et al. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372*, 2024.
- [31] V. Chiley, I. Sharapov, A. Kosson, U. Koster, R. Reece, S. Samaniego de la Fuente, V. Subbiah, and M. James. Online normalization for training neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.
- [32] J. Choi, B.-J. Lee, and B.-T. Zhang. Multi-focus attention network for efficient deep reinforcement learning. *arXiv preprint arXiv:1712.04603*, 2017.
- [33] D.-A. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.
- [34] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.
- [35] F.-A. Croitoru, V. Hondu, R. T. Ionescu, and M. Shah. Diffusion models in vision: A

- survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(9):10850–10869, 2023.
- [36] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. O’Boyle, and H. Leather. Programl: A graph-based program representation for data flow analysis and compiler optimizations. In *Proceedings of the 38th International Conference on Machine Learning (ICML 2021)*, volume 139 of *Proceedings of Machine Learning Research*, pages 2244–2253, 2021.
- [37] R. I. Davis and L. Cucu-Grosjean. A survey of probabilistic timing analysis techniques for real-time systems. *LITES: Leibniz Transactions on Embedded Systems*, pages 1–60, 2019.
- [38] DeepSeek-AI. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024. Model ID: deepseek-chat (DeepSeek-V3).
- [39] S. M. P. Dinakarrao, A. Joseph, A. Haridass, M. Shafique, J. Henkel, and H. Homayoun. Application and thermal-reliability-aware reinforcement learning based multi-core power management. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 15(4):1–19, 2019.
- [40] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *2009 international conference on parallel processing*, pages 124–131. IEEE, 2009.

- [41] D. L. Eager and K. C. Sevcik. A bound on the throughput of a multi-class system. *ACM SIGMETRICS Performance Evaluation Review*, 11(4):128–138, 1983.
- [42] S. Edgar and A. Burns. Statistical analysis of wcet for scheduling. In *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)(Cat. No. 01PR1420)*, pages 215–224. IEEE, 2001.
- [43] B. Efron. Bootstrap methods: another look at the jackknife. In *Breakthroughs in statistics: Methodology and distribution*, pages 569–593. Springer, 1992.
- [44] S. et al. Temperature-aware microarchitecture. *ACM SIGARCH*, 2003.
- [45] Y. Fan and Y. Ming. Model-based reinforcement learning for continuous control with posterior sampling. In *International Conference on Machine Learning*, pages 3078–3087. PMLR, 2021.
- [46] C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pages 1126–1135. PMLR, 2017.
- [47] C. Florensa, Y. Duan, and P. Abbeel. Stochastic neural networks for hierarchical reinforcement learning. *arXiv preprint arXiv:1704.03012*, 2017.
- [48] D. J. Foster and A. Rakhlin. Foundations of reinforcement learning and interactive decision making. *arXiv preprint arXiv:2312.16730*, 2023.

- [49] Y. Gal and Z. Ghahramani. Dropout as a Bayesian approximation: Representing model uncertainty in deep learning. In *International Conference on Machine Learning (ICML)*, pages 1050–1059, 2016.
- [50] V. Godbole, G. E. Dahl, J. Gilmer, C. J. Shallue, and Z. Nado. Deep learning tuning playbook, 2023. Version 1.0.
- [51] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.
- [52] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger. On calibration of modern neural networks. In *International conference on machine learning*, pages 1321–1330. PMLR, 2017.
- [53] T. Hastie, R. Tibshirani, J. H. Friedman, and J. H. Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.
- [54] R. Hebbar and A. Milenković. Pmu-events-driven dvfs techniques for improving energy efficiency of modern processors. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 7(1):1–31, 2022.
- [55] S. Hosseinimotlagh, D. Enright, C. R. Shelton, and H. Kim. Data-driven structured thermal modeling for cots multi-core processors. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 201–213. IEEE, 2021.

- [56] S. Hosseinimotlagh and H. Kim. Thermal-aware servers for real-time tasks on multi-core gpu-integrated embedded systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 254–266. IEEE, 2019.
- [57] R. A. Howard. Dynamic prograreinforcementmming and markov processes. 1960.
- [58] B. Hu, Z. Cao, and M. Zhou. Scheduling real-time parallel applications in cloud to minimize energy consumption. *IEEE Transactions on Cloud Computing*, 10(1):662–674, 2019.
- [59] Z. Hu, Y. Dong, K. Wang, and Y. Sun. Heterogeneous graph transformer. *Proceedings of the Web Conference 2020*, pages 2704–2710, 2020.
- [60] K. Huang, X. Jiang, X. Zhang, R. Yan, K. Wang, D. Xiong, and X. Yan. Energy-efficient fault-tolerant mapping and scheduling on heterogeneous multiprocessor real-time systems. *IEEE Access*, 6:57614–57630, 2018.
- [61] E. Ipek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, pages 195–206, 2006.
- [62] J. Jiang, W. Li, L. Pan, B. Yang, and X. Peng. Energy optimization heuristics for budget-constrained workflow in heterogeneous computing system. *Journal of Circuits, Systems and Computers*, 28(09):1950159, 2019.

- [63] C. Jin, Z. Allen-Zhu, S. Bubeck, and M. I. Jordan. Is q-learning provably efficient? *Advances in neural information processing systems*, 31, 2018.
- [64] A. Kassab, J.-M. Nicod, L. Philippe, and V. Rehn-Sonigo. Green power aware approaches for scheduling independent tasks on a multi-core machine. *Sustainable Computing: Informatics and Systems*, 31:100590, 2021.
- [65] A. Kendall and Y. Gal. What uncertainties do we need in bayesian deep learning for computer vision? In *NeurIPS*, 2017.
- [66] S. Kim, K. Bin, S. Ha, K. Lee, and S. Chong. ztt: Learning-based dvfs with zero thermal throttling for mobile devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 41–53, 2021.
- [67] Y. G. Kim and C.-J. Wu. Autoscale: Energy efficiency optimization for stochastic edge inference using reinforcement learning. In *2020 53rd Annual IEEE/ACM international symposium on microarchitecture (MICRO)*, pages 1082–1096. IEEE, 2020.
- [68] V. Kuleshov, N. Fenner, and S. Ermon. Accurate uncertainties for deep learning using calibrated regression. 80:2796–2804, 2018.
- [69] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.
- [70] C. Laaber, J. Scheuner, and P. Leitner. Predicting unstable software benchmarks using static source code features. *Empirical Software Engineering*, 26(6):116, 2021.

- [71] B. Lakshminarayanan, A. Pritzel, and C. Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles. In *Advances in Neural Information Processing Systems 30 (NIPS 2017)*, pages 6402–6413, 2017.
- [72] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *ACM SIGOPS Operating Systems Review*, 41(6):185–194, 2007.
- [73] D. Lee, N. He, P. Kamalaruban, and V. Cevher. Optimization for reinforcement learning: From a single agent to cooperative agents. *IEEE Signal Processing Magazine*, 37(3):123–135, 2020.
- [74] W. Y. Lee. Energy-saving dvfs scheduling of multiple periodic real-time tasks on multi-core processors. In *2009 13th IEEE/ACM international symposium on distributed simulation and real time applications*, pages 216–223. IEEE, 2009.
- [75] Y. C. Lee and A. Y. Zomaya. Energy conscious scheduling for distributed computing systems under different operating conditions. *IEEE Transactions on Parallel and Distributed Systems*, 22(8):1374–1381, 2010.
- [76] D. Li and J. Wu. Energy-aware scheduling for frame-based tasks on heterogeneous multiprocessor platforms. In *2012 41st International Conference on Parallel Processing*, pages 430–439. IEEE, 2012.

- [77] J. Li, K. Cheng, S. Wang, F. Morstatter, R. P. Trevino, J. Tang, and H. Liu. Feature selection: A data perspective. *ACM computing surveys (CSUR)*, 50(6):1–45, 2017.
- [78] K. Li. Scheduling precedence constrained tasks with reduced processor energy on multiprocessor computers. *IEEE Transactions on Computers*, 61(12):1668–1681, 2012.
- [79] K. Li. Energy and time constrained task scheduling on multiprocessor computers with discrete speed levels. *Journal of Parallel and Distributed Computing*, 95:15–28, 2016.
- [80] C. Lin, K. Wang, Z. Li, and Y. Pu. A workload-aware dvfs robust to concurrent tasks for mobile devices. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2023.
- [81] Y. Lipman, R. T. Chen, H. Ben-Hamu, M. Nickel, and M. Le. Flow matching for generative modeling. *arXiv preprint arXiv:2210.02747*, 2022.
- [82] D. Liu, S.-G. Yang, Z. He, M. Zhao, and W. Liu. Cartad: Compiler-assisted reinforcement learning for thermal-aware task scheduling and dvfs on multicores. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.
- [83] A. Mahmood, S. A. Khan, F. Albaloshi, and N. Awwad. Energy-aware real-time task scheduling in multiprocessor systems using a hybrid genetic algorithm. *Electronics*, 6(2):40, 2017.
- [84] S. Maity, R. Roy, A. Majumder, S. Dey, and A. R. Hota. Future aware dynamic ther-

- mal management in cpu-gpu embedded platforms. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 396–408. IEEE, 2022.
- [85] M. A. Marsan, G. Conte, and G. Balbo. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems (TOCS)*, 2(2):93–122, 1984.
- [86] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *Proceedings of the 36th International Conference on Machine Learning (ICML 2019)*, pages 4505–4515, 2019.
- [87] T. M. Moerland, J. Broekens, A. Plaat, C. M. Jonker, et al. Model-based reinforcement learning: A survey. *Foundations and Trends® in Machine Learning*, 16(1):1–118, 2023.
- [88] C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *AAAI Conference on Artificial Intelligence*, pages 4602–4609, 2019.
- [89] O. Nachum, S. S. Gu, H. Lee, and S. Levine. Data-efficient hierarchical reinforcement learning. *Advances in neural information processing systems*, 31, 2018.
- [90] A. Y. Ng, S. Russell, et al. Algorithms for inverse reinforcement learning. In *Icml*, volume 1, page 2, 2000.

- [91] D. Nichols, A. Marathe, H. Menon, T. Gamblin, and A. Bhatele. Can large language models predict parallel code performance? *arXiv preprint arXiv:2505.03988*, 2025.
- [92] OpenAI. GPT-4o system card. <https://openai.com/index/gpt-4o-system-card/>, 2024.
Model ID: gpt-4o-2024-08-06.
- [93] C. S. Pabla. Completely fair scheduler. *Linux Journal*, 2009(184):4, 2009.
- [94] S. Pagani, P. S. Manoj, A. Jantsch, and J. Henkel. Machine learning for power, energy, and thermal management on multicore processors: A survey. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(1):101–116, 2018.
- [95] J. Pallister, S. Kerrison, J. Morse, and K. Eder. Data dependent energy modeling for worst case energy consumption analysis. In *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems*, pages 51–59, 2017.
- [96] T. I. Papon and S. Venkataraman. Silhouette: Efficient cloud configuration exploration for large-scale analytics. In *Proceedings of the 2022 IEEE International Conference on Cloud Engineering (IC2E)*, pages 92–102. IEEE, 2022.
- [97] B. Peng, X. Li, J. Gao, J. Liu, K.-F. Wong, and S.-Y. Su. Deep dyna-q: Integrating planning for task-completion dialogue policy learning. *arXiv preprint arXiv:1801.06176*, 2018.
- [98] M. Pivezhandi, M. Banisharif, S. Bakhshan, A. Saifullah, and A. Jannesari. Graphperf-

- rt: A graph-driven performance model for hardware-aware scheduling of openmp codes. *arXiv preprint arXiv:2512.12091*, 2026.
- [99] M. Pivezhandi and A. Saifullah. Flowrl: Flow-augmented few-shot reinforcement learning for semi-structured sensor data. *arXiv preprint*, 2026.
- [100] M. Pivezhandi and A. Saifullah. Zerodvfs: Zero-shot llm-guided core and frequency allocation for heterogeneous embedded platforms. *arXiv preprint*, 2026.
- [101] M. Pivezhandi, A. Saifullah, and A. Jannesari. HiDVFS: A hierarchical multi-agent DVFS scheduler for OpenMP DAG workloads. *arXiv preprint*, 2026.
- [102] M. Pivezhandi, A. Saifullah, and P. Modekurthy. Feature-aware task-to-core allocation in embedded multi-core platforms via statistical learning. *arXiv preprint*, 2026.
- [103] Y. Qin, G. Zeng, R. Kurachi, Y. Li, Y. Matsubara, and H. Takada. Energy-efficient intra-task dvfs scheduling using linear programming formulation. *Ieee Access*, 7:30536–30547, 2019.
- [104] T. Ramadan, A. Lahiry, and T. Z. Islam. Novel representation learning technique using graphs for performance analytics, 2024.
- [105] A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, and M. Chen. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*, 1(2):3, 2022.
- [106] I. Ratković, N. Bežanić, O. S. Ünsal, A. Cristal, and V. Milutinović. An overview of

- architecture-level power-and energy-efficient design techniques. *Advances in Computers*, 98:1–57, 2015.
- [107] S. Reddy, A. D. Dragan, and S. Levine. Sqil: Imitation learning via reinforcement learning with sparse rewards. *arXiv preprint arXiv:1905.11108*, 2019.
- [108] F. Reghennani, G. Massari, and W. Fornaciari. Probabilistic-wcet reliability: Statistical testing of evt hypotheses. *Microprocessors and Microsystems*, 77:103135, 2020.
- [109] C. Saharia, W. Chan, S. Saxena, L. Li, J. Whang, E. L. Denton, K. Ghasemipour, R. Goncalves Lopes, B. Karagol Ayan, T. Salimans, et al. Photorealistic text-to-image diffusion models with deep language understanding. *Advances in neural information processing systems*, 35:36479–36494, 2022.
- [110] A. Saifullah, S. Fahmida, V. P. Modekurthy, N. Fisher, and Z. Guo. Cpu energy-aware parallel real-time scheduling. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- [111] V. Sanz, C. Mateos, N. Beltrametti, and A. Zunino. Predicting number of threads using balanced datasets for openmp regions. *Computing*, 104(10):2249–2274, 2022.
- [112] H. Sasaki, Y. Ikeda, M. Kondo, and H. Nakamura. An intra-task dvfs technique based on statistical analysis of hardware events. In *Proceedings of the 4th international conference on Computing frontiers*, pages 123–130, 2007.

- [113] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. volume 20, pages 61–80, 2009.
- [114] P. Sedgwick. Pearson’s correlation coefficient. *Bmj*, 345, 2012.
- [115] M. Sensoy, L. Kaplan, and M. Kandemir. Evidential deep learning to quantify classification uncertainty. In *Advances in Neural Information Processing Systems 31 (NeurIPS 2018)*, pages 3179–3189, 2018.
- [116] U. Sethi. Learning energy-aware transaction scheduling in database systems. Master’s thesis, University of Waterloo, 2021.
- [117] E. A. Shao, L. Z. M. Pan, et al. Tensor program optimization with MetaSchedule. In *MLSys*, 2022.
- [118] M. Shekarisaz, L. Thiele, and M. Kargahi. Automatic energy-hotspot detection and elimination in real-time deeply embedded systems. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 97–109. IEEE, 2021.
- [119] H. Shen, J. Lu, and Q. Qiu. Learning based dvfs for simultaneous temperature, performance and energy management. In *Thirteenth International Symposium on Quality Electronic Design (ISQED)*, pages 747–754. IEEE, 2012.
- [120] X. Shi, Y. Wang, M. Li, Y. Liu, J. Huan, and Z. Zhang. Predicting execution time of computer programs using sparse polynomial regression. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS 2017)*, pages

883–891, 2017. (Note: This is a substitute reference as the exact 2022 citation could not be verified).

- [121] G. Singla, G. Kaur, A. K. Unver, and U. Y. Ogras. Predictive dynamic thermal and power management for heterogeneous mobile platforms. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 960–965. IEEE, 2015.
- [122] J. Sun, N. Guan, J. Sun, and Y. Chi. Calculating response-time bounds for openmp task systems with conditional branches. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 169–181. IEEE, 2019.
- [123] G. Taheri, A. Khonsari, R. Entezari-Maleki, and L. Sousa. A hybrid algorithm for task scheduling on heterogeneous multiprocessor embedded systems. *Applied Soft Computing*, 91:106202, 2020.
- [124] X. Tang and Z. Fu. Cpu–gpu utilization aware energy-efficient scheduling algorithm on heterogeneous computing systems. *IEEE Access*, 8:58948–58958, 2020.
- [125] J. M. Tomczak. *Deep Generative Modeling*. Springer Cham, 2024.
- [126] Tree-sitter Contributors. Tree-sitter: An incremental parsing system for programming tools. <https://tree-sitter.github.io/tree-sitter/>, 2024. Accessed: 2024.
- [127] M. Tufano, S. K. Deng, N. Sundaresan, and A. Svyatkovskiy. Predicting code coverage without execution. *arXiv preprint arXiv:2307.13383*, 2023.

- [128] F. M. M. ul Islam and M. Lin. Hybrid dvfs scheduling for real-time systems based on reinforcement learning. *IEEE Systems Journal*, 11(2):931–940, 2015.
- [129] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- [130] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.
- [131] M. J. Walker, S. Bischoff, S. Diestelhorst, G. V. Merrett, and B. M. Al-Hashimi. Accurate and stable run-time power modeling for mobile and embedded cpus. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(12):2675–2687, 2018.
- [132] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni. Generalizing from a few examples: A survey on few-shot learning. *ACM computing surveys (csur)*, 53(3):1–34, 2020.
- [133] Y. Wang, W. Zhang, M. Hao, and Z. Wang. Online power management for multi-cores: A reinforcement learning based approach. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):751–764, 2021.
- [134] Z. Wang and M. F. P. O’Boyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018.
- [135] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas. Dueling network

- architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.
- [136] Z. Wang, Z. Tian, J. Xu, R. K. Maeda, H. Li, P. Yang, Z. Wang, L. H. Duong, Z. Wang, and X. Chen. Modular reinforcement learning for self-adaptive energy efficiency optimization in multicore system. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 684–689. IEEE, 2017.
- [137] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2021.
- [138] M. Wulfmeier, P. Ondruska, and I. Posner. Deep inverse reinforcement learning. *CoRR*, *abs/1507.04888*, 2015.
- [139] G. Xie, J. Huang, Y. L. R. Li, and K. Li. System-level energy-aware design methodology towards end-to-end response time optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.
- [140] G. Xie, J. Jiang, Y. Liu, R. Li, and K. Li. Minimizing energy consumption of real-time parallel applications using downward and upward approaches on heterogeneous systems. *IEEE Transactions on Industrial Informatics*, 13(3):1068–1078, 2017.
- [141] G. Xie, H. Peng, J. Huang, R. Li, and K. Li. Energy-efficient functional safety design

methodology using asil decomposition for automotive cyber-physical systems. *IEEE Transactions on Reliability*, 2019.

- [142] G. Xie, X. Xiao, H. Peng, R. Li, and K. Li. A survey of low-energy parallel scheduling algorithms. *IEEE Transactions on Sustainable Computing*, 7(1):27–46, 2021.
- [143] G. Xie, G. Zeng, X. Xiao, R. Li, and K. Li. Energy-efficient scheduling algorithms for real-time parallel applications on heterogeneous distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3426–3442, 2017.
- [144] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057. PMLR, 2015.
- [145] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations (ICLR)*, 2019.
- [146] L. Yan, J. Luo, and N. K. Jha. Combined dynamic voltage scaling and adaptive body biasing for heterogeneous distributed real-time embedded systems. In *ICCAD-2003. International Conference on Computer Aided Design (IEEE Cat. No. 03CH37486)*, pages 30–37. IEEE, 2003.
- [147] A. Yeganeh-Khaksar, M. Ansari, S. Safari, S. Yari-Karin, and A. Ejlali. Ring-dvfs: Reliability-aware reinforcement learning-based dvfs for real-time embedded systems. *IEEE Embedded Systems Letters*, 13(3):146–149, 2020.

- [148] Z. Yu, P. Machado, A. Zahid, A. M. Abdulghani, K. Dashtipour, H. Heidari, M. A. Imran, and Q. H. Abbasi. Energy and performance trade-off optimization in heterogeneous computing via reinforcement learning. *Electronics*, 9(11):1812, 2020.
- [149] Y. Yun, E. J. Hwang, and Y. H. Kim. Adaptive genetic algorithm for energy-efficient task scheduling on asymmetric multiprocessor system-on-chip. *Microprocessors and Microsystems*, 66:19–30, 2019.
- [150] S. Zhai et al. Transferable learning of scheduling policies for tensor programs. In *ICML*, 2023.
- [151] S. Zhai et al. Transformer latent models for cross-device schedule transfer. In *NeurIPS*, 2024.
- [152] Z. Zhang, Y. Zhao, H. Li, C. Lin, and J. Liu. Dvfo: Learning-based dvfs for energy-efficient edge-cloud collaborative inference. *IEEE Transactions on Mobile Computing*, 2024.
- [153] L. Zheng, C. Jia, M. Pan, Z. Wang, Y. Han, Y. Li, X. Lian, Y. Chen, C. H. Yu, J. Yang, and T. Chen. Ansor: Generating high-performance tensor programs for deep learning. In *OSDI*, 2020.
- [154] J. Zhou, J. Yan, K. Cao, Y. Tan, T. Wei, M. Chen, G. Zhang, X. Chen, and S. Hu. Thermal-aware correlated two-level scheduling of real-time tasks with reduced processor energy on heterogeneous mpsocs. *Journal of Systems Architecture*, 82:1–11, 2018.

- [155] T. Zhou and M. Lin. Deadline-aware deep-recurrent-q-network governor for smart energy saving. *IEEE Transactions on Network Science and Engineering*, 9(6):3886–3895, 2021.
- [156] D. Zhu, R. Melhem, and D. Mossé. The effects of energy management on reliability in real-time embedded systems. In *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, pages 35–40. IEEE, 2004.
- [157] Y. Zhu et al. ROLLER: Fast and accurate tensor program optimization. In *NeurIPS*, 2022.
- [158] C. Zhuo, D. Gao, Y. Cao, T. Shen, L. Zhang, J. Zhou, and X. Yin. A dvfs design and simulation framework using machine learning models. *IEEE Design & Test*, 2021.
- [159] B. D. Ziebart, A. L. Maas, J. A. Bagnell, A. K. Dey, et al. Maximum entropy inverse reinforcement learning. In *Aaaai*, volume 8, pages 1433–1438. Chicago, IL, USA, 2008.

ABSTRACT

**DATA-EFFICIENT AI-GUIDED ENERGY- AND THERMAL-AWARE SCHEDULING
ON HETEROGENEOUS MULTICORE SYSTEMS**

by

MOHAMMAD PIVEZHANDI

December 2025

Advisor: Dr. Abusayeed Saifullah

Major: Computer Science

Degree: Doctor of Philosophy

The surging demand for energy-efficient computing in heterogeneous multicore embedded systems, driven by applications such as IoT devices, autonomous vehicles, and mobile platforms, presents formidable challenges arising from escalating static leakage power, intricate per-core thermal dynamics, and the complexity of high-dimensional action spaces in parallel workloads such as Directed Acyclic Graphs (DAGs). Conventional approaches, including Linux governors, fail to adequately address irregular task execution patterns, costly runtime profiling requirements, and data scarcity for few-shot learning, thereby constraining scalability across diverse platforms.

This dissertation presents a comprehensive AI-driven framework that integrates hierarchical multi-agent reinforcement learning (HMARL), statistical learning, distribution-aware flow matching, model-based reinforcement learning, and graph neural network-driven performance

modeling to address these challenges. The framework comprises five interconnected contributions. First, HiDVFS introduces a hierarchical multi-agent reinforcement learning framework using Dueling Double DQN (D3QN) for performance-aware DVFS scheduling of parallel OpenMP DAG workloads, employing three coordinated agents (Profiler, Temperature, and Priority) to achieve a 49.06% makespan reduction and 40.95% energy reduction over multi-agent baselines, converging in just 12 epochs on the NVIDIA Jetson TX2 platform. Second, a hybrid statistical learning approach for task-to-core allocation combines Random Forest, backward stepwise selection, and Pearson correlation analysis to identify critical scheduling features, reducing energy consumption by 10% and core temperature by 5°C while cutting thermal prediction mean squared error by 61.6% on Intel Core i7 processors. Third, ZeroDVFS develops a model-based hierarchical multi-agent reinforcement learning framework for real-time DVFS, leveraging accurate environment models and LLM-based semantic feature extraction to achieve 7× better temperature prediction accuracy, 7.09× better energy efficiency, and 4.0× better makespan than heuristic baselines, with 358ms decision latency for subsequent decisions (8,300× faster than exhaustive profiling for first decisions) and zero-shot transfer capabilities across heterogeneous platforms. Fourth, FlowRL introduces a distribution-aware flow matching method that generates synthetic unstructured data for few-shot reinforcement learning, improving frame rates by 30% in early training while preserving statistical correlations through bootstrapping and Random Forest feature weighting. Fifth, GraphPerf-RT presents a heterogeneous graph neural network that unifies task DAG topology, control flow graph semantics, and runtime context to predict execution characteristics with $R^2 > 0.95$, employing evidential deep

learning for calibrated uncertainty quantification that enables risk-aware scheduling with 66% makespan reduction and 82% energy reduction when integrated with model-based multi-agent RL.

Validated on platforms including NVIDIA Jetson TX2, Jetson Orin NX, RUBIK Pi, and Intel Core i7 using the BOTS and Polybench benchmark suites, the unified framework outperforms existing Linux governors and reinforcement learning schedulers that lack per-core thermal awareness. The contributions advance energy-efficient embedded systems for IoT, automotive, and mobile applications by reducing thermal hotspots and cooling costs while enhancing system reliability and longevity.

AUTOBIOGRAPHICAL STATEMENT

Mohammad Pivezhandi received his Bachelor of Science degree in Electrical Engineering from the University of Tehran, Iran, in 2014, and his Master of Science degree in Electrical Engineering from the University of Tehran in 2017. He joined the Department of Computer Science at Wayne State University, Detroit, Michigan, in Fall 2019 to pursue his Doctor of Philosophy degree under the supervision of Dr. Abusayeed Saifullah.

His research interests include machine learning for embedded systems, reinforcement learning, dynamic voltage and frequency scaling (DVFS), thermal-aware scheduling, and energy-efficient computing on heterogeneous multicore platforms. During his doctoral studies, he developed novel frameworks for energy- and thermal-aware task scheduling using hierarchical multi-agent reinforcement learning, model-based policy learning, and graph neural network-driven performance modeling.

Mohammad has published his research in peer-reviewed conferences and journals in the areas of embedded systems, real-time computing, and machine learning. His work has been validated on multiple hardware platforms including NVIDIA Jetson TX2, Intel Core i7, and Intel Xeon processors.