TimelyHLS: A Framework for Timing-Aware and Architecture-Specific FPGA HLS Optimization

***Abstract*—Achieving timing closure and design-specific optimizations in FPGA-targeted High-Level Synthesis (HLS) remains a significant challenge due to the intricate interaction between architectural constraints, resource utilization, and the absence of automated support for platform-specific pragmas. In this work, we propose TimelyHLS, a novel framework integrating Large Language Models (LLMs) with Retrieval-Augmented Generation (RAG) to automatically generate and iteratively refine HLS code optimized for FPGA-specific timing and performance requirements. TimelyHLS is driven by a structured architectural knowledge base containing FPGA-specific features, synthesis directives, and pragma templates. Given a kernel, TimelyHLS generates HLS code annotated with both timing-critical and design-specific pragmas. The synthesized RTL is then evaluated using commercial toolchains, and simulation correctness is verified against reference outputs via custom testbenches. TimelyHLS iteratively incorporates synthesis logs and performance reports into the LLM engine for refinement in the presence of functional discrepancies. Experimental results across multiple FPGA architecture targets demonstrate that TimelyHLS significantly reduces the number of manual iterations required to achieve timing closure and optimization, while consistently generating functionally correct and performant hardware designs. This work highlights the potential of LLM-driven, architecture-aware synthesis frameworks in closing the gap in FPGA design automation.**

***Index Terms*—FPGA, Large Language Models, High-Level Synthesis, Timing Closure, Retrieval-Augmented Generation**

I. Introduction

FPGAs provide a flexible platform for accelerating diverse algorithms, but achieving timing closure on FPGAs remains a notorious challenge [1]. Timing closure entails ensuring that all signal paths meet the FPGA's timing constraints (setup/hold times, clock delays, etc.), and it is critical for correct operation at the target clock frequency [2]. In practice, reaching timing closure is an iterative and complex process, hindered by high clock rates, large and interconnected designs, and physical effects like routing delays [3]. Modern FPGA design flows often demand manual tuning and optimization to meet timing, especially when using High-Level Synthesis (HLS) tools to generate hardware from C/C++ code [4], [5].

HLS tools such as Xilinx Vitis HLS [6] and Intel HLS [7] raise the design abstraction to high-level code, but they still rely heavily on user guidance to produce efficient, timing-compliant hardware [8]. In particular, performance-critical microarchitectural optimizations (e.g. loop pipelining, loop unrolling, memory partitioning) are typically controlled by pragmas or directives embedded in the HLS source [9]. Selecting the right combination of pragmas for a given design and FPGA is a non-trivial task that often requires deep hardware expertise [10].

Currently, there is a lack of automated support within HLS tools for platform-specific pragmas (i.e. directives tailored to a specific FPGA architecture or vendor). Each FPGA platform introduces unique resources and constraints, which often necessitates different pragmas or coding styles. Designers must manually adapt and tune their HLS code for each target device, since a pragma that works well (or is even recognized) on one toolchain may not apply on another.

To ease the challenges of HLS optimization, researchers have developed automated methods like DSE frameworks [11], [12], which search large pragma spaces to find configurations with good Quality of Results (QoR). Tools such as AutoDSE [13] use heuristic searches but require many HLS runs, making them slow. Analytical methods, like formulating pragma selection as a non-linear optimization problem [9], reduce this cost by pruning poor choices. ML-based tools (e.g., HARP [14]) predict performance to guide optimizations more efficiently. However, models often struggle to generalize to new designs or architectures. Despite progress, fully automated, widely adopted solutions remain lacking, especially for timing-driven optimization, leaving designers to rely on manual tuning.

Recent advances in Large Language Models (LLMs) have opened new possibilities in automating HLS optimization for FPGA design [8], [15], [16]. Tools like LIFT [15] fine-tune LLMs with Graph Neural Network (GNN) analyses to insert pragmas, achieving up to 3.5× speedup over prior methods. HLSPilot [8] uses in-context learning and retrieval from vendor docs, combining DSE and profile-guided refinement to produce results comparable to expert designs. However, prompting general LLMs without domain-specific context can significantly degrade performance [17], demonstrating the need for integrated domain knowledge and feedback.

Despite recent progress, HLS-based FPGA design continues to face two critical challenges: (i) *ensuring that synthesized designs meet stringent timing requirements*, which is complicated by factors such as deep logic pipelines, routing congestion, and long critical paths that are difficult to predict and optimize at a high level; and (ii) *adapting optimizations to the unique characteristics of each FPGA architecture*, where vendor-specific toolchains, resource constraints, and low-level hardware features often necessitate custom pragma configurations and design patterns that do not generalize across platforms. These issues collectively contribute to a persistent "closure gap" that current tools struggle to overcome without significant manual intervention.

To address these limitations, we propose TimelyHLS, a framework that combines LLMs with retrieval-augmented generation (RAG) and iterative refinement. TimelyHLS is guided by a structured knowledge base encoding FPGA-specific features, pragmas, and optimization heuristics. The LLM queries this knowledge during inference to generate HLS code tailored to the target architecture. This initial generation is followed by an iterative loop: synthesized RTL is evaluated using commercial tools, testbenches verify functional correctness, and synthesis logs are fed back into the model. The LLM then revises the code based on this feedback until the design achieves timing closure and correctness.

We evaluate TimelyHLS on various FPGA devices and benchmark kernels. Results show that TimelyHLS reduces manual iterations, consistently meets timing constraints, and delivers performance comparable to hand-tuned designs. Our contributions include:

- **LLM + RAG for FPGA HLS:** We propose *TimelyHLS*, the first framework to integrate an LLM with retrieval-augmented generation (RAG) for FPGA-specific HLS code generation. By grounding the model in a curated knowledge base of FPGA-specific architectural features, synthesis directives, and pragma strategies, TimelyHLS generates HLS code tailored to the target device.
- **Iterative Refinement with Tool Feedback:** TimelyHLS employs an iterative refinement loop where synthesis reports, timing analysis, and functional verification feedback are reintegrated into the LLM. This enables the model to progressively resolve timing violations and improve design quality in a closed-loop, minimizing the need for manual tuning.
- **Effective Timing Closure and Optimization:** Extensive experiments show that TimelyHLS consistently achieves timing closure across diverse FPGAs while significantly reducing manual intervention. It delivers performance and QoR on par with or exceeding expert-optimized HLS designs.

II. Related Works

A. Heuristic and Analytical Approaches for HLS Optimization

Early work on automating HLS optimization used heuristic search and static modeling to explore the vast design space of synthesis directives (pragmas) [18]. Exhaustive search is infeasible (the combination of loop unrolling, pipelining, memory partitioning, etc. grows combinatorially), so researchers applied heuristics and meta-heuristics to guide design space exploration (DSE) without brute force. OpenTuner [18], a general autotuning framework adapted for HLS that orchestrates many search strategies (greedy, genetic algorithms, simulated annealing, etc.) via a multi-armed bandit approach. By dynamically choosing among strategies, OpenTuner effectively navigated large pragma spaces, often finding better solutions than any single algorithm alone. AutoDSE [13] , a DSE tool that iteratively tunes one pragma at a time, always addressing the current performance bottleneck. By focusing on the most critical optimization step-by-step, AutoDSE achieved expert-level results with far fewer directives. Many DSE frameworks employed simulated annealing [19] or hill-climbing [20] to automate pragma selection. These rule-based searches marked a big step in reducing engineering effort. However, purely heuristic approaches can miss global optima or get stuck in local optima, especially in very large parameter spaces.

Complementing those search techniques, analytical modeling approaches sought to speed up exploration by predicting HLS outcomes without full synthesis. Tools like Lin-Analyzer [21], COMBA [22], and more recently ScaleHLS [23], use static code analysis (e.g., loop dependence graphs, pipeline initiation interval formulas) to estimate a design's latency and resource usage under different pragma choices. By evaluating design points with these mathematical models, unpromising configurations can be pruned orders of magnitude faster than actually synthesizing them. In practice, these models often need re-tuning for each tool or hardware target, and they handle only a subset of possible pragmas or code structures (to keep the formulas tractable). As a result, analytical estimation is usually used as a component in a larger system.

B. Machine Learning and LLM-Based Approaches

Recent work has explored learning-based methods to enhance HLS automation by guiding optimization more intelligently than heuristic-driven approaches. ML frameworks like HARP [14] use surrogate models, often GNNs to predict performance from code and pragma configurations, enabling rapid design-space exploration without full synthesis. Bayesian optimization (BO) tools like Sherlock [16] further improve efficiency by prioritizing high-potential candidates through probabilistic modeling, excelling in multi-objective tuning.

Reinforcement learning (RL) frames HLS as sequential decision-making, where agents learn to apply transformations or insert pragmas to maximize performance. Methods like AutoAnnotate [24] report up to 4× speedup, though RL faces challenges with large design spaces and long training times. Hybrid solutions like AutoHLS [25] combine neural prediction with BO to prune poor candidates early, achieving up to 70× speedup.

LLM-based approaches aim to generate optimized HLS code directly. While general-purpose models like GPT-4 struggle without domain grounding, LIFT [15] improves results by fine-tuning on 40k+ annotated HLS samples with graph-based features, achieving 3.5× speedup over baselines. However, it lacks adaptability to new platforms and toolchains. HLSPilot [8] uses retrieval-augmented generation (RAG), guiding the LLM with relevant HLS examples and vendor-specific rules at inference time. It enables structural code transformations and rivals expert performance. Building on this, TimelyHLS enhances adaptability by dynamically querying an evolving, platform-aware knowledge base during generation. This ensures that all optimizations are context-aware, verifiable, and tailored to current toolchains, avoiding hallucinated or outdated directives.

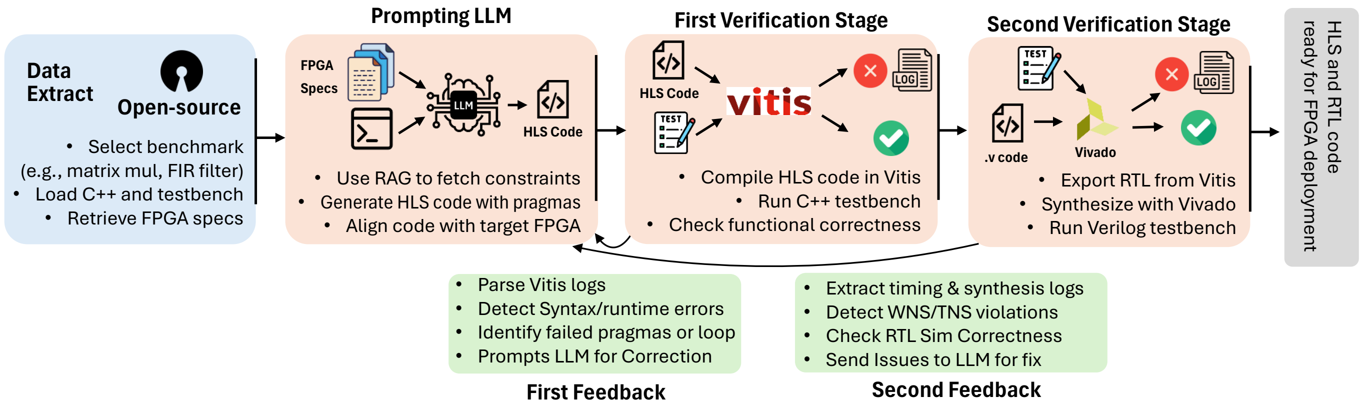
Data Extract
Open-source
Select benchmark (e.g., matrix mul, FIR filter)
Load C++ and testbench
Retrieve FPGA specs
Prompting LLM
FPGA Specs
LLM
HLS Code
Use RAG to fetch constraints
Generate HLS code with pragmas
Align code with target FPGA
First Verification Stage
HLS Code
TEST
vitis
LOG
Compile HLS code in Vitis
Run C++ testbench
Check functional correctness
Second Verification Stage
TEST
.v code
Vivado
LOG
Export RTL from Vitis
Synthesize with Vivado
Run Verilog testbench
HLS and RTL code ready for FPGA deployment
Parse Vitis logs
Detect Syntax/runtime errors
Identify failed pragmas or loop
Prompts LLM for Correction
First Feedback
Extract timing & synthesis logs
Detect WNS/TNS violations
Check RTL Sim Correctness
Send Issues to LLM for fix
Second Feedback

Fig. 1

III. Overview of TimelyHLS

IV. Methodology

A. Dataset Collection

To evaluate our framework, we curated a dataset of real-world HLS design examples gathered from open-source repositories, including CHStone [26], LegUp benchmarks [27], and MachSuite [28]. These repositories provide diverse C/C++ programs widely used in FPGA research, covering a range of computational domains. We selected 10 representative HLS applications that reflect common optimization bottlenecks in FPGA synthesis, including timing violations, long critical paths, inefficient pipelining, and suboptimal resource allocation. A detailed description of each benchmark and its associated synthesis challenge is provided in Table I.

For each application, we developed corresponding HLS C++ source files as well as custom testbenches to verify functional correctness during simulation and synthesis. All designs were compiled and analyzed using the Xilinx Vitis HLS toolchain. To ensure architectural diversity and practical relevance, we evaluated each design across 10 distinct FPGA targets spanning multiple device families.

For each architecture, we collected synthesis reports, timing summaries, and resource utilization logs. Additionally, we captured the output of testbench simulations to verify functional correctness. This comprehensive dataset, which includes source code, testbenches, and tool-generated logs across multiple architectures, forms the basis for evaluating the effectiveness of our proposed framework, TimelyHLS.

B. TimelyHLS

The TimelyHLS framework automates timing-aware HLS code generation through a combination of LLM, RAG, and iterative synthesis feedback. The workflow operates in two main verification stages, HLS-level and RTL-level, and leverages FPGA-specific knowledge to guide the model toward platform-compliant and timing-closure-friendly designs.

C. Prompting LLM for Initial Code Generation

For each sample in our dataset, we craft a task-specific prompt that describes the functionality and performance objectives of the design (e.g., loop behavior, target throughput, or memory access constraints). This prompt is paired with target FPGA metadata (i.e. device family, number of DSPs, BRAMs, LUTs, and timing constraints as part of a RAG pipeline). The architectural specifications are embedded from datasheets and vendor tool documentation into a structured knowledge base.

TABLE I: Benchmark HLS applications and their corresponding synthesis challenges.

Application	Optimization Challenge
Matrix Multiplication	Long critical path due to nested loops; loop pipelining inefficiencies.
Convolution	Timing violations caused by inefficient memory access and computation overlap.
Vector Dot Product	Underutilized resources and insufficient parallelism.
Vector Addition	Suboptimal loop unrolling with moderate timing slack.
Bitonic Sort	Deep logic pipelines leading to routing congestion and critical path delays.
Viterbi Decoder	Complex control dependencies causing resource contention.
Adaptive FIR Filter (LMS)	Feedback loop latency and failed timing closure due to iteration dependencies.
CORDIC Algorithm	Inefficient pipelining due to iterative data dependencies.
Matrix-Vector Multiplication	Memory partitioning bottlenecks leading to timing degradation.
Needleman–Wunsch (DP)	Irregular memory access patterns causing critical path delay and low throughput.

We then query the LLM (e.g., Code LLaMA or GPT-4) with the prompt and retrieved FPGA constraints to generate HLS-compliant C/C++ code. The model is expected to insert relevant synthesis directives (pragmas) such as #pragma HLS pipeline, unroll, or array_partition, aligned with the resource capabilities of the target device.

D. HLS-Level Verification and Correction

The generated HLS code is compiled and simulated using Xilinx Vitis HLS, paired with the testbench we previously authored for each benchmark. This first stage ensures that the model's output is functionally correct and synthesizable at the C level.

If the compilation fails or the functional simulation does not produce expected results, we extract relevant information from Vitis logs (e.g., syntax errors, resource binding issues, or pipeline depth violations) and return this feedback to the LLM in the form of an augmented prompt. The LLM is asked to revise its output to address the specific failures. This process is repeated iteratively until the design passes both HLS synthesis and functional simulation.

E. RTL-Level Verification and Timing Evaluation

Once the HLS design passes the first verification stage, we export the RTL (Verilog) output and move to the second stage using Xilinx Vivado. In this phase, we generate the corresponding Verilog testbenches and synthesize the design for the selected FPGA target.

Vivado's post-synthesis reports are then used to evaluate timing closure (e.g., Worst Negative Slack, Total Negative Slack), resource utilization, and syntactic validity of the RTL. We also simulate the design at the RTL level using the generated testbenches to verify behavioral equivalence with the HLS-level outputs.

If Vivado fails to synthesize the design or simulation results deviate from the expected output, we extract detailed logs—including synthesis errors, critical path reports, and functional mismatches and pass them as feedback to the LLM. This closes the second loop of iterative refinement, allowing the model to correct deeper architectural or low-level issues that were not visible during HLS-level synthesis.

This two-stage refinement loop continues until the design satisfies the following criteria:

- Passes functional simulation in both HLS and RTL levels.
- Is synthesizable by Vivado for the target FPGA architecture.
- Meets timing closure requirements with no negative slack.

By integrating FPGA-specific architectural guidance into generation and leveraging compiler logs as structured feedback, TimelyHLS transforms the traditional trial-and-error-based HLS optimization into an automated, LLM-driven pipeline.

V. Experimental Setup

VI. Results and Discussion

VII. Conclusion and Future Work

References

[1] Q. Yanghua, N. Kapre, H. Ng, and K. Teo, "Improving classification accuracy of a machine learning approach for fpga timing closure," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2016, pp. 80–83.

[2] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "Fpga hls today: successes, challenges, and opportunities," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 4, pp. 1–42, 2022.

[3] M. W. Numan, B. J. Phillips, G. S. Puddy, and K. Falkner, "Towards automatic high-level code deployment on reconfigurable platforms: A survey of high-level synthesis tools and toolchains," *IEEE Access*, vol. 8, pp. 174 692–174 722, 2020.

[4] E. Ustun, S. Xiang, J. Gui, C. Yu, and Z. Zhang, "Lamda: Learning-assisted multi-stage autotuning for fpga design closure," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 74–77.

[5] Q. Sun, T. Chen, S. Liu, J. Chen, H. Yu, and B. Yu, "Correlated multi-objective multi-fidelity optimization for hls directives design," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 27, no. 4, pp. 1–27, 2022.

[6] AMD, "AMD Vitis HLS, High-Level Synthesis Tool," https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html.

[7] Intel Corporation, "Intel® High Level Synthesis (HLS) Compiler — Quartus Prime," https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[8] C. Xiong, C. Liu, H. Li, and X. Li, "Hlspilot: Llm-based high-level synthesis," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–9.

[9] S. Pouget, L.-N. Pouchet, and J. Cong, "Automatic hardware pragma insertion in high-level synthesis: A non-linear programming approach," *ACM Transactions on Design Automation of Electronic Systems*, vol. 30, no. 2, pp. 1–44, 2025.

[10] Y. Chi, W. Qiao, A. Sohrabizadeh, J. Wang, and J. Cong, "Democratizing domain-specific computing," *Communications of the ACM*, vol. 66, no. 1, pp. 74–85, 2022.

[11] S. Liu, F. C. Lau, and B. C. Schafer, "Accelerating fpga prototyping through predictive model-based hls design space exploration," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[12] L. Ferretti, G. Ansaloni, and L. Pozzi, "Lattice-traversing design space exploration for high level synthesis," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018, pp. 210–217.

[13] A. Sohrabizadeh, C. H. Yu, M. Gao, and J. Cong, "Autodse: Enabling software programmers to design efficient fpga accelerators," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 27, no. 4, pp. 1–27, 2022.

[14] A. Sohrabizadeh, Y. Bai, Y. Sun, and J. Cong, "Robust gnn-based representation learning for hls," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.

[15] N. Prakriya, Z. Ding, Y. Sun, and J. Cong, "Lift: Llm-based pragma insertion for hls via gnn supervised fine-tuning," *arXiv preprint arXiv:2504.21187*, 2025.

[16] Q. Gautier, A. Althoff, C. L. Crutchfield, and R. Kastner, "Sherlock: A multi-objective design space exploration framework," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 27, no. 4, pp. 1–20, 2022.

[17] B. Peng, M. Galley, P. He, H. Cheng, Y. Xie, Y. Hu, Q. Huang, L. Liden, Z. Yu, W. Chen *et al.*, "Check your facts and try again: Improving large language models with external knowledge and automated feedback," *arXiv preprint arXiv:2302.12813*, 2023.

[18] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 303–316.

[19] Z. Ding, A. Sohrabizadeh, W. Li, Z. Qin, Y. Sun, and J. Cong, "Efficient task transfer for hls dse," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–9.

[20] N. K. Pham, A. K. Singh, A. Kumar, and M. M. A. Khin, "Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 157–162.

[21] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, "Lin-analyzer: A high-level performance analysis tool for fpga-based accelerators," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[22] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "Comba: A comprehensive model-based analysis framework for high level synthesis of real applications," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 430–437.

[23] H. Ye, C. Hao, J. Cheng, H. Jeong, J. Huang, S. Neuendorffer, and D. Chen, "Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation," in *2022 IEEE international symposium on high-performance computer architecture (HPCA)*. IEEE, 2022, pp. 741–755.

[24] H. Shahzad, A. Sanaullah, S. Arora, U. Drepper, and M. Herbordt, "Autoannotate: Reinforcement learning based code annotation for high

level synthesis," in *2024 25th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2024, pp. 1–9.

[25] M. R. Ahmed, T. Koike-Akino, K. Parsons, and Y. Wang, "Autohls: Learning to accelerate design space exploration for hls designs," in *2023 IEEE 66th International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE, 2023, pp. 491–495.

[26] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "Chstone: A benchmark program suite for practical c-based high-level synthesis," in *2008 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2008, pp. 1192–1195.

[27] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2, pp. 1–27, 2013.

[28] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2014, pp. 110–119.