A Limit Study of State-Space-Free Model for Basic Block Latency Attribution

Chengyi Lux Zhang

***Abstract*—Traditional performance models rely on tracking detailed architectural state to predict execution behavior, requiring sequential simulation and extensive manual calibration—an approach that scales poorly for large systems with long instruction streams. In this work, we explore the capabilities and limitations of *state-space-free* models that attribute per-instruction latencies within a basic block using only static structural embeddings and aggregate basic block time. Our results demonstrate that, while state-space-free learning-based models can recover approximate latency attributions for a subset of basic blocks with modest complexity, they exhibit error in complex dynamic contexts with deeper global state. We conclude with a discussion of hybrid extensions that incorporate partial state or execution history and provide recommendations for trace features that future hardware should expose to support lightweight, accurate attribution.**

***Index Terms*—Performance Modeling, Processor Tracing**

I. Introduction

Understanding which instruction sequences stall and why is critical for post-silicon software optimization. Traditional performance models rely on simulators that maintain a detailed architectural state. Although accurate, these approaches require sequential execution and extensive parameter tuning, limiting scalability for long instruction streams and complex multicore interactions. Moreover, determining which state components to model is often heuristic and ad hoc.

In this work, we investigate a complementary direction: using *state-space-free* (SSF) learning-based models that infer performance attribution directly from the static structural embeddings of a basic block (BB), combined with its observed cycle count. These models allow for scalable, batch inference, but raise fundamental questions about attribution accuracy in the absence of dynamic context.

This paper presents a limit study of such models. Our contributions are as follows.

- A modeling pipeline that maps basic blocks and cycle counts to per-instruction latency estimates;
- A systematic evaluation of accuracy across diverse instruction sequences;
- A characterization of when state-space-free models succeed or fail at latency attribution;
- An exploratory experiment probing into the expressiveness of the embedding space of the model;
- A discussion of hybrid modeling strategies and trace features to enable better profiling in future designs.

Our findings clarify the boundaries of stateless attribution models and offer guidance for designing scalable, low-overhead profiling and tracing systems.

II. Background

A. Processor Tracing

Modern hardware platforms often include dedicated processor tracing units that record execution events, such as retired instructions, branch outcomes, and exceptions, in a compact stream. Examples include Intel Processor Trace (PT) [1], ARM Embedded Trace Macrocell (ETM) [2], and RISC-V E- [3] and N-trace extensions [4]. These mechanisms enable post-hoc reconstruction of continuous execution paths with extremely low overhead, unlike intrusive binary instrumentation.

Typical trace formats intentionally exclude fine-grained timing information to save bandwidth, which precludes precise timing analysis. To address this, we implemented *TACIT* (Timestamp Annotated Core Instruction Trace), a lightweight hardware extension that embeds delta-encoded cycle timestamps at each control-flow event, enabling the low-overhead extraction of per-basic-block timing. Implementation details of the TACIT prototype are beyond the scope of this report.

B. Performance Modeling

Cycle-accurate simulators such as gem5 [5] provide fine-grained insights by modeling microarchitectural state at the expense of low throughput and heavyweight calibration. Recently, deep-learning-based approaches like TAO [6] use extensive feature engineering and an autoregressive transformer to estimate latencies, providing a 1000x speedup compared to gem5 at the expense of explanability and accuracy. Notice that TAO keeps a latent state space that is updated per instruction token, making its inference still sequential and stateful.

Stateless performance modeling methods are used for adjacent fields, like compiler cost models. Ithemal [7] provides a coarse-grained BB throughput estimation via a stateless hierarchical RNN. It uses Long Short-Term Memory networks (LSTM) within a BB, so a latent space state exists. However, each basic block is inferred individually, making it stateless across BBs. Such granularity is sufficient for compiler heuristics but not for micro-architecture evaluation.

III. Implementation

We implemented an SSF model prototype, FireFlower. Figure 1 represents the high-level modeling pipeline of FireFlower.

A. Canonicalization

The first step is to canonicalize an instruction trace to a common tokenized form. Existing RISC-V disassemblers are

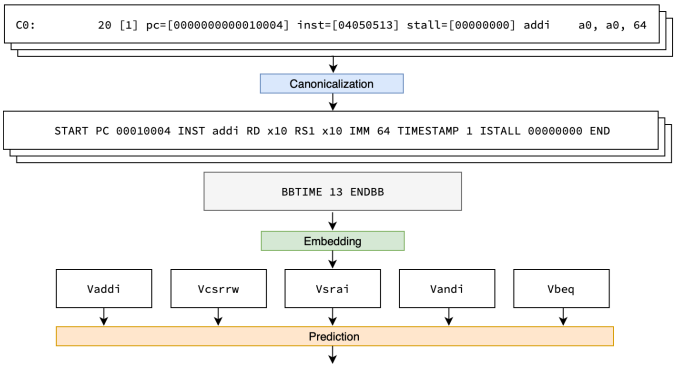

Fig. 1. FireFlower System Architecture

designed to be human-friendly. They extensively use pseudo-instructions and outputs to string formats, preventing them from being used for canonicalization. As a result, we implemented *rvdasm*, a canonical RISC-V disassembler written in Rust. It has three main features.

- **Correctness**. It is code-generated from the upstream riscv-opcodes specification for correctness guarantee.
- **Programmer-Friendly**. It decodes instructions into a cloneable struct with no pseudocode and no register name aliases, making it easy to manipulate and canonicalize.
- **Complete**. It fully supports RISC-V I, M, A, C, F, D, V, and zicsr extensions, and distinguishes XLEN of 32 bits and 64 bits.

B. Embedding Design

FireFlower constructs a unified, learnable embedding for each instruction by encoding its constituent features and supplying them to a Transformer backbone. First, categorical fields—**opcodes** and **operands**—are each mapped through dedicated embedding tables: an opcode vocabulary and a register file dictionary produce discrete indices, which are then projected into dense vectors via learnable look-up matrices. Because these fields are non-ordinal, embedding tables allow the model to learn distinct representations without imposing spurious numeric relationships. By contrast, **immediate** values, which are inherently continuous, are fed through a linear projection layer to align with the embedding dimension. The per-field embeddings are then concatenated and passed through a linear fusion layer to yield a single instruction representation.

To capture instruction order, we add a **positional embedding** to each fused vector. Additionally, we broadcast the basic block's total cycle count (**BB time**) to every instruction position and project it into the same embedding space, enabling the model to jointly reason about global timing context and local instruction structure.

C. Model Backbone

We chose a four-layer Transformer encoder with eight attention heads per layer as the backbone. Self-attention is essential because it directly considers the relationship between any two instructions in the block, capturing long-range data and control

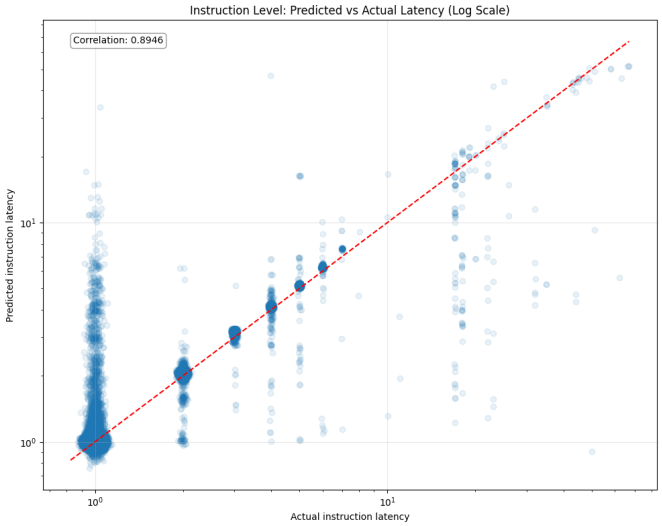

Fig. 2. Evaluation Results: FireFlower per-instruction predicted vs actual latency, both axes shown in log scale.

dependencies without requiring manual feature engineering. Multiple attention heads allow the model to specialize. Some focus on register producer–consumer flows, others on memory interactions, thereby modeling diverse microarchitectural effects in parallel. Empirically, four layers provided sufficient representational power to identify stalls without overfitting: fewer layers are not expressive enough, while deeper stacks takes longer to train and converge.

Finally, a lightweight linear head projects the transformer's output embedding to a scalar latency. We chose a regression head because instruction latency, while measured in discrete cycles, is fundamentally an arbitrarily large quantity. Using a classification head would hence complicate loss design.

D. Training

Our training set comprises per-instruction retirement cycles collected from an RTL simulation of the Rocket Core [8] running the official RISC-V benchmark suite. The raw simulation log occupies 368 MB; after canonicalizing, the processed dataset shrinks to 156 MB. We then shuffle the examples and split them 90 / 10 into training and validation subsets to ensure unbiased performance evaluation. We train for 20 epochs with a validation patience of 3 epochs.

The loss combines a weighted mean-squared error (MSE) over valid per-instruction latency predictions with a block-level MSE on the sum of those predictions against the true basic-block time. Formally, letting m_i be a mask indicating valid instructions, $\hat{a}_i$ the predicted latency for instruction i, a_i its ground truth, and λ_{reg} and λ_{bb} being weight terms, the loss function is:

$$\mathcal{L} = \lambda_{\text{reg}} \frac{1}{\sum_i m_i} \sum_i m_i \left(\hat{a}_i - a_i\right)^2 + \lambda_{\text{bb}} \left(\sum_i m_i \hat{a}_i - \sum_i m_i a_i \right)^2 .$$

IV. EVALUTATION

Overall, FireFlower 's predictions align closely with the true latencies. Specifically, the Pearson correlation coefficient between the predicted and ground-truth values is 0.89, indicating a strong linear relationship. An ordinary least squares fit of predicted versus actual latencies yields a slope of 0.963 and an intercept of 0.029, showing that the model slightly underestimates true values, but remains extremely close on average cases. Finally, the normalized root-mean-square error (NRMSE) is 0.00948, meaning that the standard deviation of the prediction error is under 1% of the observed latency range—evidence of very low relative error.

The per-instruction latency predictions versus ground truth are plotted in Figure 2. Most points lie close to the $y = x$ line, indicating generally accurate estimates. However, two primary error modes stand out. First, many instructions with a true latency of 1 cycle are predicted above one, producing false positives: non-stalling instructions are misclassified as stalling, which can mislead engineers hunting performance bottlenecks to non-existent performance bugs. Second, for true latencies above 10 cycles, the model systematically underestimates stall durations. These high-latency points scatter broadly along the y-axis rather than clustering, resulting in false negatives where genuine stalls are under-attributed. Both issues can obscure the actual sources of system slowdowns.

The following sections will provide examples of working and not-working predicted blocks.

A. Working Cases

TABLE I
DATA CACHE MISS CASE

Pos.	Opcode	Pred. (cyc)	Actual (cyc)	Delta
0	lw	4.1384	4.0000	0.1384
1	c.mv	0.9990	1.0000	-0.0010
2	c.addi	0.9947	1.0000	-0.0053
3	blt	0.9969	1.0000	-0.0031

In table I, we see an example where the memory operation is correctly attributed for the stalling the pipeline. In this case, the `lw` instruction likely triggered a L1 Data Cache miss and caused the pipeline stall. FireFlower is capable of successfully identifying this instruction for the stall.

TABLE II
LOAD-USE INTERLOCK CASE

Pos.	Opcode	Pred. (cyc)	Actual (cyc)	Delta
0	sltu	1.0097	1.0000	0.0097
1	sw	1.0070	1.0000	0.0070
2	csrrs	1.0072	1.0000	0.0072
3	and	3.2134	3.0000	0.2134
4	c.or	1.0066	1.0000	0.0066
5	beq	1.0030	1.0000	0.0030

In table II, we see an example where the load-use interlock is correctly attributed for the stall. An `and` instruction is `and x15, x15, x18`, which depends on the results of the `csrrs` instruction, `csrrs x15, x0, CSR768`, which leads to a true write after read data dependency that could not be resolved by the processor. There are no remainig useful work, so the compiler did not reorder anyhow. This stall is hence true cycles stalled due to data dependency. FireFlower is capable of successfully identifying this pattern, likely due to the attention mechanism, and attributing latency to the dependent `and`, despite the existence of a `sw` that could potentially also stall the pipeline for data cache miss.

B. Misattribution

TABLE III
MIATTRIBUTION CASE

Pos.	Opcode	Pred. (cyc)	Actual (cyc)	Delta
0	lh	46.8364	4.0000	42.8364
1	c.li	0.9803	50.0000	-49.0197
2	c.andi	1.0133	1.0000	0.0133
3	bne	2.3458	1.0000	1.3458

Despite being capable of correctly attributing some latencies with local context, FireFlower demonstrates incompletence on more complex blocks requireing deeper dynamic global state. In table III, the `lh` instruction is predicted to take 46+ cycles, and `bne` is also attributed with a 1 cycle stall. However, in reality, the majority of the stall comes from `c.li` instruction. It has no dependency with previous instructions, and also is not a memory operation. However, its PC `0x80001b80` happens to be on the instruction cache block boundary, indicating that this likely is an I$ miss, which further triggered an LLC miss, leading to this long latency. An I$ miss cannot be captured by a model without state space, as such effects are compounded across a large trace of thousands of instruction fetch outside of the basic block context. This demonstrates how FireFlower will significantly mislead the users when blindly attributing latencies without considering a deeper global state.

C. Lack of Explainability

TABLE IV
A SHORTER-LATENCY XORI BLOCK

Pos.	Opcode	Pred. (cyc)	Actual (cyc)	Delta
0	xori	2.0593	2.0000	0.0593
1	c.addiw	1.0027	1.0000	0.0027
2	and	1.0015	1.0000	0.0015
3	c.slli	1.0023	1.0000	0.0023
4	c.slli	1.0006	1.0000	0.0006
5	beq	1.0013	1.0000	0.0013

TABLE V
A LONGER-LATENCY XORI BLOCK

Pos.	Opcode	Pred. (cyc)	Actual (cyc)	Delta
0	xori	5.1045	5.0000	0.1045
1	c.addiw	1.0002	1.0000	0.0002
2	and	0.9993	1.0000	-0.0007
3	c.slli	0.9991	1.0000	-0.0009
4	c.slli	0.9972	1.0000	-0.0028
5	beq	0.9973	1.0000	-0.0027

In this example, compare table IV with table V. It seems that FireFlower does a good job on predicting `xori` instruction is

the source of stalling here. However, FireFlower provides no insight in why this is the case. Why would an `xori` instruction stall anyways? One potential explanation is the preceding BB has a write on the source operands of `xori`, and the previous block's trailing branch got correctly speculated. Still, why would these two `xori` stall for different periods? FireFlower, as a data-driven model, although correctly attributes latencies, fails to provide insights and potential fixes into the stalling events.

V. Case Study

To study the capability of FireFlower on the shortcomings we demonstrate in section IV-B and section IV-C, we conduct an exploratory study to investigate an SSF model's ability boundaries to predict global-state correlated events. Similar to how Time Proportional Event Analysis [9] performs event profiling, we modified Rocket Core to tag each instruction with tracking information on what events in the pipeline stall it. We specifically track for I$ misses event. This tracking information eventually shows up in the oracle training trace.

We hence modify FireFlower to support the prediction of I$ miss event. We add a special layer of `PC` embedding as it is the most relevant feature for predicting I$ misses. We add two additional branches of 2-layer transformers after the original FireFlower transformer. This allows for separate attention to focus on predicting I$ miss and the original latency attribution task. We then attach a linear projection head to each transformer output embedding, one as the original attribution, and one as logits for predicting whether I$ missed or not. The new FireFlower is essentially a multi-tasked model, but shares some early embedding space as the tasks are correlated. We add a `BCEWithLogitsLoss` loss to the total loss, and train the new model on the same dataset.

The resulting model has no substantial difference in performing the latency attribution. On the new task of predicting I$ stalls, it produced 7 false positives, 14 false negatives, and only 4 true positives.

TABLE VI
A MISATTRIBUTED STALL FOR FIREFLOWER +

Pos.	Opcode	Pred	Actual	Pred I$	Actual I$
0	remu	11.8060	22.0000	0.0	1.0
1	addiw	8.9859	1.0000	0.0	0.0
2	c.addi4spn	7.4786	1.0000	0.0	0.0
3	c.li	6.7753	1.0000	0.0	0.0
4	c.swsp	45.7851	63.0000	1.0	0.0
5	bltu	4.8629	23.0000	0.0	0.0

In Table VI, the modified FireFlower yields one false positive and one false negative. Notably, the store instruction `c.swsp` at PC `0x800049fc` resides off any I$ block boundary and cannot incur an I$ miss, yet the model predicts a long latency consistent with an I$ stall and flags it as such. This misclassification demonstrates that the learned embedding does not internally distinguish specific microarchitectural events. This probing verifies that the network's learned embedding space relies solely on top-down, statistical correlations of aggregated timing effects.

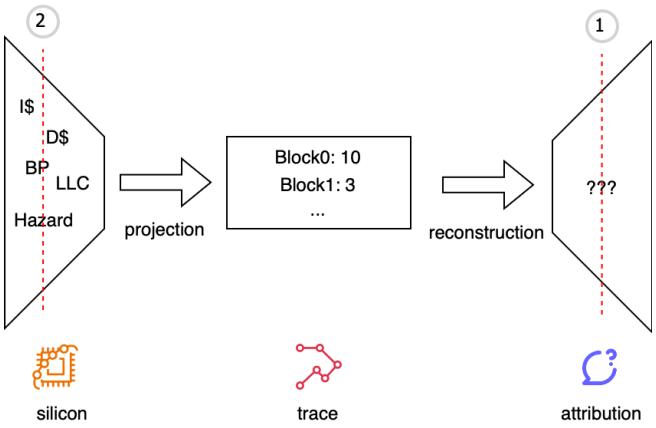

Fig. 3. A view on the lifetime of performance analysis

VI. Conclusion & Future Work

Fundamentally, the shortfall identified in Section V stems from the limited expressiveness of state-space-free (SSF) models, which in turn reflects the information bottleneck in timestamp-annotated traces. As illustrated in Figure 3, a wide variety of microarchitectural events occur on silicon, and a timed instruction trace captures only a lossy projection of their net effect. The conciseness of trace is a fundamental limitation due to on-chip bandwidth limitation, but it also imposes a hard limit on what can be inferred after the fact. In signal-processing terms, the trace is a forward (stateful) projection of dynamic events; any stateless reconstruction algorithm—no matter how sophisticated—cannot fully invert that projection when the original process depends on hidden state.

We hence conclude with concrete evidence and reasoning that a pure SSF is not an adequate heuristic for disambiguating the performance events causing the overall stall seen, and propose two future directions.

One approach is to adopt a hybrid design that combines lightweight, stateless heuristics with selective stateful reconstruction. By replaying a brief "warm-up" trace segment, one can maybe recover an approximate architectural state sufficiently accurate to inform downstream attribution without incurring full sequential simulator overhead.

An alternative is to perform performance attribution on-chip, before the lossy project occurs. Extending existing hardware trace encoders is reasonable, as they can already observe microarchitectural events. By embedding lightweight analysis logic directly in silicon, the system can distill and export only the most salient metrics, rather than raw trace streams.

However, to match peak retire rates with the core, these algorithms must be extremely efficient, processing multiple instructions per cycle without added latency. Equally important, the on-chip analysis engine must incur minimal area and power overhead to remain practical for commercial implementation.

VII. Appendix

A. Technical Challenges

At the beginning of the project, I was trying to use a BERT model - a classification model - to perform this task. However, due to how most instructions can maintain an IPC of 1, BERT overfits to only classify 1 instead of any other possible value. This leads to the conclusion that a regression head and a model built from scratch, with customized loss function, is necessary for this to produce any meaningful result.

B. Roles

I did everything covered in this report.

C. Class Contents

This project is closely tied to the sequential analysis section of the class. It is investigating whether a heuristic approach exists to adequately approximate a stateful, time-stepping algorithm that is sequential in nature, in a constrained and specific setting.

References

[1] Intel Corporation, *8th and 9th Generation Intel® Core™ Processor Specification Update*, Document ID 337346, Apr. 20, 2023. [Online]. Available: https://www.intel.com/content/www/us/en/content-details/337346/8th-and-9th-generation-intel-core-processor-specification-update.html

[2] ARM Ltd., *Embedded Trace Macrocell™ Architecture Specification (IHI0014Q)*, ARM Developer, 2011. [Online]. Available: https://developer.arm.com/documentation/ihi0014/latest/ :contentReference[oaicite:0]index=0

[3] RISC-V International, *RISC-V Processor Trace Specification*, Version 1.0, 2019. [Online]. Available: https://raw.githubusercontent.com/riscv/riscv-trace-spec/master/riscv-trace-spec.pdf :contentReference[oaicite:1]index=1

[4] RISC-V International, *RISC-V N-Trace (Nexus-based Trace) Specification*, Version 1.0-rc42, 2024. [Online]. Available: https://tools.cloudbear.ru/docs/riscv-n-trace-1.0-rc42-20240724.pdf :contentReference[oaicite:2]index=2

[5] J. Lowe-Power *et al.*, "The GEM5 Simulator: Version 20.0+," arXiv preprint arXiv:2007.03152, 2020. [Online]. Available: https://arxiv.org/abs/2007.03152

[6] S. Pandey, A. Yazdanbakhsh, and H. Liu, "TAO: Re-Thinking DL-based Microarchitecture Simulation," arXiv preprint arXiv:2404.10921, 2024. [Online]. Available: https://arxiv.org/abs/2404.10921

[7] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks," in Proc. 36th Int. Conf. Machine Learning (ICML), 2019. [Online]. Available: https://arxiv.org/abs/1808.07412

[8] K. Asanović, J. Bachrach, B. Richards, J. Shan, D. Patterson, and R. Michaud, "Rocket Chip: An Open-Source SoC Generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Feb. 2016. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.pdf

[9] B. Gottschall, L. Eeckhout, and M. Jahre, "TEA: Time-Proportional Event Analysis," in *Proc. 50th Int. Symp. Computer Architecture (ISCA)*, Orlando, FL, Jun. 2023, pp. 13–25. doi: 10.1145/3579371.3589058. [Online]. Available: https://doi.org/10.1145/3579371.3589058