MOSAIC: MORE LLMS, LESS COST THROUGH COLLABORATIVE REASONING FOR COMPILER OPTIMIZATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-guided compiler optimization has recently shown promise. However, current LLM-guided optimizers query a single LLM at every search step, so their cost grows with both that model's price and the search budget. A smaller model is cheaper but degrades the search: in our experiments, a recent single-LLM optimizer reaches 10.2% (6.0%) lower speedup on GPU (CPU) with gpt-5-mini than with GPT-5.2, which gives up the output quality that motivates using LLMs in the first place. Agentic multi-LLM frameworks add controllers, concurrent calls, and shared memory, whose overhead can offset the savings of cheaper models. This paper shows that the optimization search tree itself can coordinate heterogeneous LLMs: the tree already records which proposals improved the program, and we can augment the tree to record which model made them. That evidence can then be used to decide which model should propose next. To this end, we introduce MOSAIC, in which each Monte Carlo tree search (MCTS) node records a program and the LLM that expands it, and the active LLM proposes both a compiler transformation and the LLM for the next step. We also contribute an LLM-aware UCT that favors smaller models and a course-alteration rule that invokes the largest model only after repeated small-model regressions. Against that single-LLM optimizer run with GPT-5.2 alone and the same number of search iterations and tree parameters, on an NVIDIA GPU (Intel CPU), MOSAIC with eight LLMs improves the geometric-mean speedup on five kernels by 7.2% (14.4%) and the end-to-end Llama-3-8B speedup by 1.61× (1.41×), while reducing kernel compilation time by 1.95× (1.74×) and API cost by 4.47× (4.32×), with GPT-5.2 making only 23.1% (23.9%) of LLM calls. With the same eight models, random or round-robin model choice improves over the single-LLM baseline by only 2.5% on CPU, showing that the gains come from letting the LLMs choose the next model rather than from mixing the pool at random.

1 INTRODUCTION

The cost of model serving increasingly dominates modern AI systems, and the compiler determines how efficiently neural workloads run on target hardware through transformations such as scheduling, fusion, and layout changes (Patel & Ahmad, 2023; Li et al., 2020). Finding effective transformation sequences is difficult: the search space is combinatorial, hardware-dependent, and shaped by long-range interactions among transformations (Vasilache et al., 2018; Chen et al., 2018a; Zheng et al., 2020; Shao et al., 2022). LLMs are therefore entering the compiler optimization loop: LLM-guided optimizers condition each transformation proposal on the program, its transformation history, and observed outcomes, and they reach high-quality code in fewer samples than stochastic autotuners (Cummins et al., 2023; 2025; Deng et al., 2025; Pan et al., 2025; Tang et al., 2025).

However, this quality comes at a price. Existing LLM-guided optimizers query a single LLM at every search step (Pan et al., 2025; Tang et al., 2025), so their cost scales with both that model's price and the search budget: a recent single-LLM optimizer—Reasoning Compiler—with GPT-5.2 spends 2262 s and $14.07 to tune a single Llama-4-Scout MLP kernel on CPU (Appendix F), and a full model contains many such kernels. Using a smaller model alone does not solve this: in our experiments, Reasoning Compiler with gpt-5-mini reaches 10.2% (6.0%) lower speedup on GPU

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

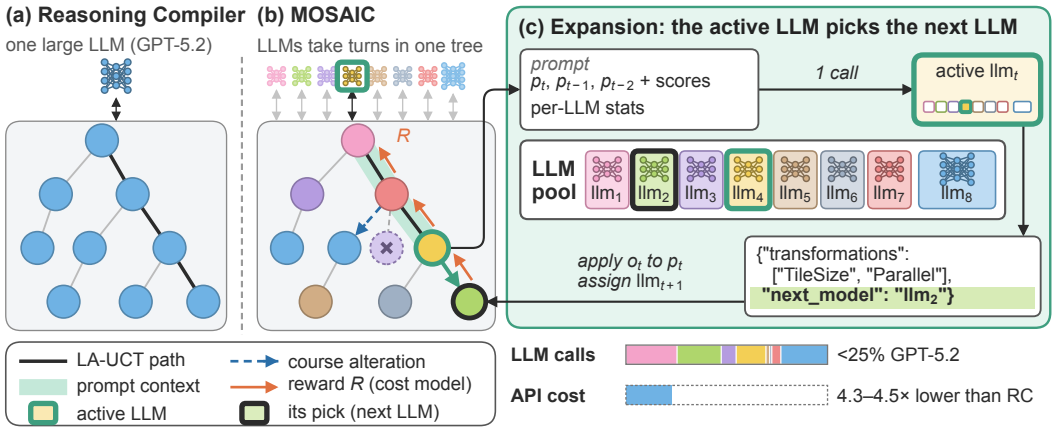


Figure 1: MOSAIC's LLMs collaborate only through one MCTS tree. (a) Reasoning Compiler uses one large LLM. (b) In MOSAIC, small LLMs and GPT-5.2 take turns (color = expanding LLM, green = active): the active LLM reads nodes others wrote (green band), LA-UCT favors small LLMs, GPT-5.2 proposes a replacement child after two consecutive small-LLM regressions (dashed), and R credits every LLM on the path. (c) One call to llm_t returns $\langle o_t, \text{llm}_{t+1} \rangle$, picking the next LLM (black) from the pool.

(CPU) than with GPT-5.2 (Figure 2), which gives up the output quality that motivates using LLMs at all. This cost recurs for every model, hardware target, and update, whereas the speedup of the compiled code pays off at every inference. As LLMs move into the compiler loop, their cost becomes a barrier alongside their quality.

Our goal in this work is to lower the cost of LLM-guided compiler optimization while retaining the output quality of a single large LLM, without fine-tuning, inter-model message passing, or agentic machinery. Combining several LLMs is the natural direction, but LLM multi-agent systems coordinate through explicit interaction patterns, such as peer-to-peer exchange, coordinator-executor hierarchies, shared memory, debate, voting, or verifier-based aggregation (Wu et al., 2023; Hong et al., 2024; Salemi et al., 2025; Du et al., 2024; Wang et al., 2023; 2025; Lifshitz et al., 2025), and their concurrent calls, controllers, and memory state add overhead (Smit et al., 2024; Cemri et al., 2025) that can negate the savings of cheaper models.

Our key insight is that the optimization search tree already records which proposals improved the program, and can easily be augmented to record which model made them. This evidence can therefore be used by an LLM to decide which model should propose next. To this end, we introduce MOSAIC, which makes model selection endogenous to one shared Monte Carlo tree search (MCTS): each search state pairs the current program with the active LLM, and each action jointly chooses a compiler transformation and the next LLM (Figure 1(c)), so collaboration happens inside the search rather than through an external router or agent controller.

The shared MCTS tree is the central collaboration substrate in MOSAIC (Figure 1(b)): heterogeneous LLMs take turns expanding it, one call at a time, building on common transformation prefixes, and value estimates backpropagated through the same tree let signals found by one model influence later decisions made by others. To reduce collaboration cost, we instantiate this idea with an LLM-aware Upper Confidence bounds applied to Trees (UCT) (Kocsis & Szepesvári, 2006) variant that biases model choice toward smaller models while preserving the compiler performance objective. We further add a course-alteration mechanism that selectively invokes the largest model when persistent small model regressions would otherwise propagate degraded values.

We compare MOSAIC against Reasoning Compiler (Tang et al., 2025), the single-LLM MCTS optimizer that MOSAIC extends (Figure 1(a)), run with GPT-5.2 alone and the same number of search iterations and tree parameters. On five kernels on an NVIDIA GPU (Intel CPU), MOSAIC(8 LLMs) improves the geometric-mean speedup by 7.2% (14.4%) while reducing compilation time by 1.95× (1.74×) and API cost by 4.47× (4.32×). On CPU, choosing the next model at random from the same pool improves speedup by only 2.5% and costs 1.27× more than MOSAIC (Table 4),

so the gains come from letting the active LLM choose the next model. Unlike LE-MCTS (Park et al., 2025), which picks models at random, and Multi-LLM AB-MCTS (Inoue et al., 2025) and ShinkaEvolve (Lange et al., 2025), which use bandits over per-model rewards, MOSAIC lets the active LLM choose the next model from the program context and per-model statistics.

2 MOSAIC: MULTI-LLM TREE SEARCH FOR COMPILER OPTIMIZATION

MOSAIC makes one shared MCTS tree the coordination substrate for heterogeneous LLMs, without inter-model communication, concurrent LLM calls, or agentic controllers.

2.1 JOINT SEARCH OVER COMPILER TRANSFORMATIONS AND LLM SELECTION

We consider the task of optimizing an initial program $p_1 \in P$ through a sequence of semantic-preserving compiler transformations (Touati & Barthou, 2006). Each transformation $o \in O$ maps one program to another, and the objective is to identify a transformation sequence $o_1, \ldots, o_{T-1}$ that maximizes a performance metric $f : P \to \mathbb{R}_{\geq 0}$ on a target hardware platform:

$$p_{t+1} = o_t(p_t), \qquad \max_{o_1,\ldots,o_{T-1} \in O} f(p_T).$$

Following standard formulations of compiler phase ordering, the underlying compiler environment can be modeled as a finite horizon Markov decision process (MDP) (Kulkarni & Cavazos, 2012): states are programs, actions are compiler transformations, and rewards are derived from the performance metric. Transitions are stochastic in our implementation: an LLM names a transformation (e.g., TileSize), and the compiler samples its concrete parameters.

Let LLMSet $= \{\text{llm}_1, \ldots, \text{llm}_N\}$ denote the available LLMs, which may differ in size, capability, and reliability. To make collaboration possible and lightweight, we make the LLM choice part of the same decision process as the compiler optimization. Thus, MOSAIC augments the compiler state with the identity of the active LLM. The joint state space is

$$\mathcal{S} = P \times \text{LLMSet}.$$

A state $\langle p_t, \text{llm}_t \rangle$ contains the current program p_t and the LLM llm_t responsible for proposing the next expansion. A joint action is a pair

$$\langle o_t, \text{llm}_{t+1} \rangle \in O \times \text{LLMSet},$$

where o_t is the compiler transformation applied to the current program and llm_{t+1} is the model assigned to the resulting child state. When the current state is $\langle p_t, \text{llm}_t \rangle$, the active model llm_t induces a context-dependent stochastic proposal distribution over such joint actions. Applying a sampled joint action produces the next state

$$\langle p_{t+1}, \text{llm}_{t+1} \rangle = \langle o_t(p_t), \text{llm}_{t+1} \rangle.$$

For notation, we write o_t as a single transformation, although in the implementation an LLM response may contain a short sequence of valid compiler transformations that is composed and applied as one edge expansion.

2.2 SHARED-TREE MCTS WITH ENDOGENOUS MODEL SELECTION

MOSAIC instantiates the joint search process above using MCTS. Figure 1(b) illustrates the optimization loop on the shared tree, and Figure 1(c) one expansion. Each tree node corresponds to a joint state $\langle p, \text{llm} \rangle \in P \times \text{LLMSet}$ and stores visit counts and accumulated rollout reward. Each MCTS iteration proceeds through selection, expansion, rollout, and backpropagation.

During selection, the search descends the tree using the LLM-aware tree policy described in §2.3. The selected node determines both the current program and the active LLM. During expansion, the active LLM is queried with the current optimization context and returns a joint proposal $\langle o, \text{llm}' \rangle$, specifying the transformation to apply and the LLM to assign to the child node. Applying o produces a new program state, yielding the child node $\langle o(p), \text{llm}' \rangle$.

After expansion, the search performs a short rollout consisting of randomly selected transformations. The terminal program produced by the rollout is evaluated using a cost model, which estimates the objective without requiring direct execution on the target hardware at every search step. The reward

predicted by the cost model is then backpropagated along the selected path, updating visit counts and value estimates. Since rewards depend only on resulting programs, credit assignment naturally spans both compiler transformations and the LLM selection decisions that led to them.

2.3 LLM-Aware Tree Policy

In MOSAIC, selecting a tree node determines both the program prefix to extend and the LLM that extends it, so the tree policy must weigh downstream program quality against LLM cost. We therefore introduce LA-UCT, an LLM-aware variant of UCT (Kocsis & Szepesvári, 2006) that scores each node by combining the compiler performance objective with a normalized preference for smaller LLMs.

As shown in Appendix A, at any fixed parent node treated as a stationary bandit, LA-UCT is equivalent to UCB1 applied to the transformed reward $(1-\lambda)R+\lambda\phi_{\text{small}}(\text{llm})$ and asymptotically concentrates visits on children maximizing the corresponding surrogate mean. For a child node corresponding to program p_t and model llm_t, with parent $\langle p_{t-1}, \text{llm}_{t-1}\rangle$, we define

$$\text{LA-UCT}\,(p_t, \text{llm}_t) = (1-\lambda)\,\frac{W(p_t, \text{llm}_t)}{N(p_t, \text{llm}_t)} + \lambda\,\phi_{\text{small}}(\text{llm}_t) + c\sqrt{\frac{\ln N(p_{t-1}, \text{llm}_{t-1})}{N(p_t, \text{llm}_t)}}.$$

Here, $N(p_t, \text{llm}_t)$ is the visit count of the child node, $W(p_t, \text{llm}_t)$ is its cumulative rollout reward, and c controls exploration. The first term estimates the downstream value of the program trajectory, while the second term favors smaller LLMs through

$$\phi_{\text{small}}(\text{llm}) = \frac{\log n_\theta(\text{llm}_{\max}) - \log n_\theta(\text{llm})}{\log n_\theta(\text{llm}_{\max}) - \log n_\theta(\text{llm}_{\min}) + \varepsilon} \in [0,1],$$

where $n_\theta(\text{llm})$ denotes the parameter count of llm, and $\text{llm}_{\min}$ and $\text{llm}_{\max}$ are the smallest and largest models in the candidate set. Smaller models receive larger ϕ_{small}, and ε prevents division by zero. The size term is a cost prior known before search; observed reliability enters separately, through the per-model hit rates and error counts that the active LLM sees when it chooses the next model (§2.4). We use public parameter counts for open-weight models and assumed sizes for closed models, whose sizes are not disclosed (GPT-5.2: 300B; gpt-5-mini: 20B). ϕ_{small} depends on the assumed sizes only through log-ratios, and with the open-weight Llama-3.3-70B-Instruct as the largest model, where only gpt-5-mini's size is assumed, the results show the same trends (§3.2).

The parameter $\lambda \in [0,1]$ controls the strength of the LLM size term. When $\lambda = 0$, selection reduces to reward-only UCT; as λ increases, a child expanded by a smaller LLM is preferred when its expected downstream reward is competitive. At $\lambda = 0.5$, a child assigned to a larger model wins exploitation only if its mean reward exceeds the smaller model's by more than their gap in ϕ_{small}, so larger models are reached mainly through exploration, the active LLM's next-model choice, and course alteration.

2.4 Contextual Prompting for Joint Transformation and LLM Selection

At each node expansion, the active LLM is queried for a joint proposal (Figure 1(c)): a sequence of compiler transformations and the LLM to assign to the child node.

Prompt context. For a selected leaf state $\langle p_t, \text{llm}_t\rangle$, the prompt includes the current program p_t and, when available, its parent p_{t-1} and grandparent p_{t-2}. For these local program variants, MOSAIC provides source code, transformation histories, and predicted performance scores. The prompt also includes the valid transformations O, the current leaf depth, and progress through the search budget. The same prompt exposes the candidate LLMs (LLMSet) with associated statistics collected during search. For each LLM, MOSAIC provides its size, invocation count, hit rate (the fraction of its expansions whose child scores higher than its parent under the cost model), and error count, where errors include invalid transformation names or invalid next model identifiers. The prompt also includes local model context, namely the models used to expand the current node and its recent ancestors.

Joint proposal. Given this combined context, the active LLM returns a structured proposal containing a transformation sequence and a single next model choice. The LLM selection instruction is size-aware: prefer the smallest model likely to support continued progress, while allowing larger models

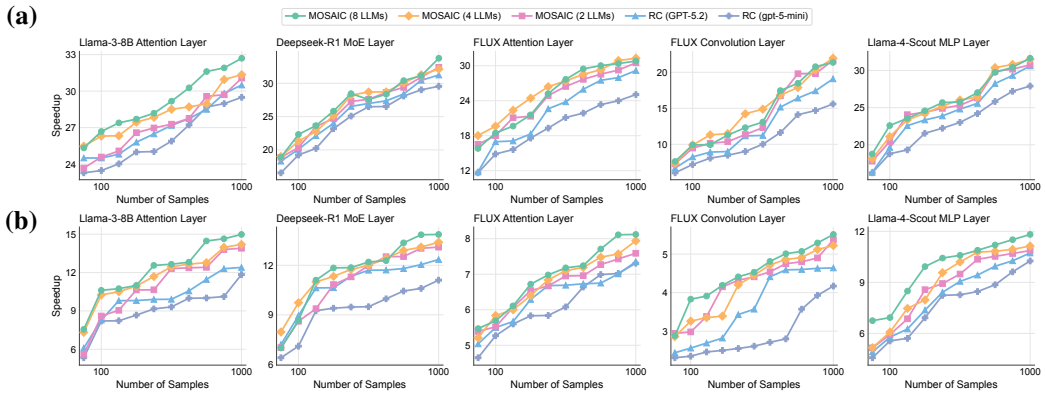


Figure 2: Speedup over unoptimized code as a function of the number of searched samples on (a) GPU and (b) CPU. RC is Reasoning Compiler (Tang et al., 2025) run with the named LLM alone; MOSAIC uses GPT-5.2 as its largest model. Both panels share the legend of (a); Appendix D lists the final values.

when the local program context or prior statistics suggest that additional capacity may be useful. The resulting child is then evaluated through the shared MCTS tree, so both the transformation proposal and the LLM recommendation receive credit only through their downstream effect on compiler reward. Appendix B provides a complete prompt template and an example output.

2.5 Course Alteration via Large Model Intervention

Because MOSAIC prefers smaller LLMs, one small model's errors can accumulate in the shared tree, so course alteration invokes the largest LLM when persistent small-model regressions are detected.

Diagnosing persistent small-LLM regressions. A regression is detected when a child has a reward score worse than that of its parent. A persistent small-model regression pattern is detected when two consecutive small-LLM expansions both produce regressions; expansions by the largest model neither count toward nor reset this streak. When this pattern is detected, MOSAIC asks the largest LLM for an alternative to the most recent regressive expansion, which replaces the regressive child unless it scores lower under the cost model, so a replaced child's degraded value estimate is never backpropagated through the tree.

Large-model course alteration. MOSAIC invokes the largest LLM from the same parent node using a dedicated course-alteration prompt (dashed arrow in Figure 1(b)). This prompt reuses the local program context, adds the failed small model proposal, and asks the largest LLM to revise the transformation sequence, the next LLM choice, or both. This prompt is shorter and more targeted than a regular prompt, which lowers the cost of each high-capacity intervention. An example course-alteration prompt is provided in Appendix B. The resulting child goes through the same rollout and backpropagation, after which the standard shared-tree search resumes. Discarding regressive children biases small-model value estimates upward; we accept this as a heuristic safeguard and quantify its effect in Table 4.

3 Evaluation

3.1 Experimental Setup

Benchmarks and platforms. We evaluate five kernels from production-scale neural networks: (1) a self-attention layer from Llama-3-8B (Llama Team, AI @ Meta, 2024), (2) a mixture-of-experts (MoE) layer from DeepSeek-R1 (DeepSeek-AI, 2025), (3) a self-attention layer and (4) a convolution layer from FLUX (Black Forest Labs, 2024), and (5) an MLP layer from Llama-4-Scout (Meta, 2025). We also compile end-to-end Llama-3-8B. We report results on an NVIDIA 2080 Ti GPU and an Intel Core i9 CPU.

Table 1: Compilation time and API cost reduction of MOSAIC relative to Reasoning Compiler run with the largest model alone. Column groups indicate the largest model. GPU/CPU results are reported; Llama-3.3-70B-Instruct results are CPU only.

Benchmark	Metric	GPT-5.2			Llama-3.3-70B-Instruct		
		MOSAIC (8 LLMs)	MOSAIC (4 LLMs)	MOSAIC (2 LLMs)	MOSAIC (8 LLMs)	MOSAIC (4 LLMs)	MOSAIC (2 LLMs)
Llama-3-8B Attention Layer	Comp. Time ↓ (×)	1.85/1.48	1.46/1.09	1.01/1.00	1.93	1.68	1.31
	API Cost ↓ (×)	4.35/3.06	3.30/2.57	1.87/2.64	2.04	1.38	1.14
DeepSeek-R1 MoE Layer	Comp. Time ↓ (×)	1.79/1.68	1.20/1.49	1.05/1.10	1.87	1.75	1.35
	API Cost ↓ (×)	5.18/4.18	2.88/3.21	2.55/2.42	2.35	1.59	1.15
FLUX Attention Layer	Comp. Time ↓ (×)	2.01/2.03	1.66/1.18	1.04/1.10	1.84	1.62	1.35
	API Cost ↓ (×)	5.37/4.59	3.22/2.89	1.87/2.37	1.91	1.39	1.16
FLUX Convolution Layer	Comp. Time ↓ (×)	2.09/1.73	1.47/1.22	1.06/1.02	2.23	1.53	1.31
	API Cost ↓ (×)	3.76/5.87	3.18/2.96	2.78/2.17	2.30	1.34	1.16
Llama-4-Scout MLP Layer	Comp. Time ↓ (×)	2.03/1.85	1.75/1.10	1.08/1.04	2.32	1.72	1.25
	API Cost ↓ (×)	3.91/4.38	3.10/2.56	2.63/2.23	2.67	1.42	1.12

Table 2: Invocation rates (%) of each model, averaged across the five benchmarks, for 2-, 4-, and 8-LLM configurations of MOSAIC on GPU/CPU (Llama-3.3-70B-Instruct: CPU only). C.A. denotes course-alteration calls.

Largest Model: GPT-5.2 GPU/CPU				Largest Model: Llama-3.3-70B-Instruct (CPU)			
Model	MOSAIC (8 LLMs)	MOSAIC (4 LLMs)	MOSAIC (2 LLMs)	Model	MOSAIC (8 LLMs)	MOSAIC (4 LLMs)	MOSAIC (2 LLMs)
GPT-5.2 (Regular)	9.8/10.7	16.1/17.8	26.2/24.5	Llama-3.3-70B-Inst (Reg.)	18.6	30.5	45.5
GPT-5.2 (C.A.)	13.3/13.2	12.2/12.0	10.6/10.5	Llama-3.3-70B-Inst (C.A.)	12.1	10.1	7.9
GPT-5.2 (Total)	23.1/23.9	28.3/29.8	36.7/35.1	Llama-3.3-70B-Inst (Total)	30.7	40.6	53.4
gpt-5-mini	1.8/7.8	33.3/36.5	63.3/64.9	gpt-5-mini	3.0	27.9	46.6
DS-R1-Qwen-7B	25.4/12.6	—	—	DS-R1-Qwen-7B	8.9	—	—
Llama-3.1-8B-Inst	7.4/11.4	29.2/15.1	—	Llama-3.1-8B-Inst	15.6	23.1	—
Qwen3-8B	21.9/35.2	—	—	Qwen3-8B	22.0	—	—
Qwen3-14B	14.8/4.7	—	—	Qwen3-14B	15.2	—	—
Devstral-Small-2505	1.1/2.4	—	—	Devstral-Small-2505	3.5	—	—
DS-R1-Qwen-32B	4.5/2.0	9.2/18.6	—	DS-R1-Qwen-32B	1.0	8.5	—

Baselines. Our single-LLM baseline is Reasoning Compiler (Tang et al., 2025), the LLM-guided MCTS compiler optimizer that MOSAIC extends. We run Reasoning Compiler in our environment with GPT-5.2 or gpt-5-mini as its only model and with the same search budget, cost model, exploration constant, branching factor, and final-program selection as MOSAIC. We do not compare against non-LLM autotuners: our question is how to lower the cost of LLM-guided optimization, whose sample-efficiency advantage over TVM MetaSchedule on these kernels was established on CPUs by Tang et al. (2025) in a separate campaign (Appendix D).

MOSAIC configurations. We evaluate three model sets of increasing size: (1) MOSAIC(2 LLMs) comprises GPT-5.2 and gpt-5-mini; (2) MOSAIC(4 LLMs) adds DeepSeek-R1-Distill-Qwen-32B (DeepSeek-AI, 2025) and Llama-3.1-8B-Instruct (Llama Team, AI @ Meta, 2024); and (3) MOSAIC(8 LLMs) further adds DeepSeek-R1-Distill-Qwen-7B (DeepSeek-AI, 2025), Qwen3-8B, Qwen3-14B (Team, 2025), and Devstral-Small-2505 (Mistral AI, 2025). Going from four to eight LLMs adds only open-weight models of 7B–24B parameters, none larger than the 32B model or GPT-5.2 already in the pool, so the gains of MOSAIC(8 LLMs) do not come from adding a larger model.

Table 3: End-to-end Llama-3-8B: final speedup improvement, compilation time reduction, and API cost reduction of MOSAIC over Reasoning Compiler run with the largest model alone. GPU/CPU results are reported; Llama-3.3-70B-Instruct results are CPU only.

	Largest Model: GPT-5.2			Largest Model: Llama-3.3-70B-Instruct		
MOSAIC Configuration	Speedup over RC ↑ (×)	Comp. Time ↓ (×)	API Cost ↓ (×)	Speedup over RC ↑ (×)	Comp. Time ↓ (×)	API Cost ↓ (×)
MOSAIC(8 LLMs)	1.61/1.41	1.52/1.71	3.94/4.44	1.34	1.45	1.96
MOSAIC(4 LLMs)	1.86/1.19	1.32/1.27	2.81/2.79	1.27	1.39	1.55
MOSAIC(2 LLMs)	1.11/1.28	1.22/1.10	2.19/2.21	1.23	1.28	1.36

Table 4: Where the gains come from, on CPU at 1000 samples. All MOSAIC variants use the same eight-LLM pool and the same shared tree; only the listed component changes, and indented rows modify MOSAIC(8 LLMs). RC is Reasoning Compiler. The last two columns give compilation time and API cost relative to MOSAIC (higher is more expensive).

Variant	L3 Attn.	MoE	FLUX Attn.	FLUX Conv.	L4 MLP	Geo-mean	vs. RC	Time	Cost
RC (GPT-5.2 only)	12.40	12.35	7.35	4.64	10.73	8.91	—	1.74×	4.32×
Random next LLM	12.82	12.54	7.37	4.93	10.83	9.12	+2.5%	1.37×	1.27×
Round-robin next LLM	12.96	12.69	7.42	4.82	10.80	9.13	+2.5%	1.38×	1.26×
MOSAIC(8 LLMs)	**14.98**	**13.87**	**8.12**	**5.51**	**11.82**	**10.19**	**+14.4%**	1.00×	1.00×
w/o size term (λ=0)	14.62	13.65	7.85	5.44	11.58	9.97	+12.0%	—	—
w/o course alteration	14.05	12.91	7.61	5.33	11.36	9.65	+8.3%	—	—
alteration after every regression	15.26	14.24	8.18	5.53	11.92	10.32	+15.9%	1.38×	1.58×

Implementation. We implement MOSAIC in TVM's MetaSchedule (Chen et al., 2018a; Shao et al., 2022) (Apache TVM v0.20.0 (Apache TVM Community, 2025)) by replacing its search routine, and we access models through the OpenAI and Nscale APIs; the code is in the Supplementary Material (Appendix C). LA-UCT uses $\lambda = 0.5$ (Appendix G), exploration constant $c = \sqrt{2}$, and branching factor $B = 2$, following prior work (Coulom, 2007; Auer et al., 2002; Tang et al., 2025). We set λ and the course-alteration threshold on the CPU kernels (Appendices G and H) and use them unchanged on GPU and with Llama-3.3-70B-Instruct. Each experiment is repeated 10 times, and we report the mean.

Samples, rewards, and measurement. One sample is one MCTS iteration: one node expansion by one regular LLM call, followed by a rollout, so every 1000-sample run makes exactly 1000 regular LLM calls; course-alteration calls are additional and are included in MOSAIC's compilation time, API cost, and call counts. Rollouts are scored by TVM's default XGBoost cost model (Chen et al., 2018a; Chen & Guestrin, 2016), which MetaSchedule trains online on candidates measured on the target hardware; Reasoning Compiler uses the same model, so neither method has a cost-model advantage. MOSAIC itself needs only a scalar prediction per program, which any regression-based cost model provides and which an order-preserving map (e.g., clipping or min–max scaling) brings into $[0, 1]$, as is standard in cost-model-guided autotuning (Chen et al., 2018b; Zheng et al., 2020; Shao et al., 2022). During search, the reward is the cost model's prediction clipped below at 0; the model regresses each measured candidate's throughput relative to the best measured candidate of the same workload, a target in $(0, 1]$, so rewards lie approximately in $[0, 1]$ but are not bounded above. All speedups in figures and tables, however, are latencies measured on the target hardware: speedup is the latency of the unoptimized IRModule divided by the lowest measured latency among the programs found within the given number of samples, averaged over the 10 runs.

Correctness. The LLMs never emit code. They name MetaSchedule schedule transformations (tile sizes, compute location, parallelization, and unrolling), whose parameters MetaSchedule samples and which change loop structure and memory placement but not the computation, and TVM rejects any proposal that is invalid for the program. Every candidate is therefore semantically equivalent to the input program by construction, up to floating-point reassociation from reordered reductions, as in MetaSchedule itself (Shao et al., 2022).

Cost accounting. Compilation time is the wall-clock time of the entire search, including LLM latency, rollouts, and hardware measurement. API cost sums, over all calls, the input and output tokens times each model's list price from its provider (OpenAI for GPT-5.2 and gpt-5-mini; Nscale for open-weight models). Both measures depend on the serving provider: wall-clock time includes queueing under variable load and network latency, and list prices differ across providers. They are nevertheless what users of hosted models pay, so we report them (Appendix F) along with per-model call counts (Table 2), which do not depend on the provider.

3.2 Experimental Results

MOSAIC retains the quality of a large LLM at a fraction of its cost. A small model alone degrades the search: at 1000 samples, Reasoning Compiler with gpt-5-mini trails Reasoning Compiler with GPT-5.2 by 10.2% (6.0%) on GPU (CPU) (Figure 2). MOSAIC recovers and exceeds the quality of the large model. On GPU (Figure 2a), MOSAIC(8 LLMs) reaches a geometric-mean speedup of 29.7× and improves over Reasoning Compiler with GPT-5.2 on all five kernels, by 7.2% on geometric mean and up to 11.9% on FLUX Convolution Layer (19.1× to 21.4×). On CPU (Figure 2b), MOSAIC(8 LLMs) reaches 10.2× and improves over Reasoning Compiler with GPT-5.2 on all five kernels, by 14.4% on geometric mean and up to 20.8% on Llama-3-8B Attention Layer (12.4× to 15.0×).

These gains come at lower cost: relative to Reasoning Compiler with GPT-5.2, MOSAIC(8 LLMs) reduces compilation time by 1.95× (1.74×) and API cost by 4.47× (4.32×) on GPU (CPU) (Table 1), since small models perform most expansions (Table 2).

On CPU, the LLM-chosen next model drives the speedup gains. Table 4 isolates each component of MOSAIC on CPU with the same eight-LLM pool and shared tree (full results in Appendices G–I). Random or round-robin next-LLM selection, i.e., a static mixture of the pool, improves over Reasoning Compiler by only 2.5%, and it costs 1.37× (1.38×) more compilation time and 1.27× (1.26×) more API cost than MOSAIC. Letting the active LLM choose the next model raises the gain to 14.4%, and removing the size term (λ=0, reward-only UCT) or course alteration lowers it to 12.0% or 8.3%, respectively. The two mechanisms are complementary: the LLM-chosen next model adds 11.9 points over random choice, and course alteration adds 6.1 points. Invoking course alteration after every regression gains only 1.5 more points but costs 1.38× more time and 1.58× more API cost. Every tested $\lambda \in \{0, 0.25, 0.5, 0.75, 1\}$ beats Reasoning Compiler on every kernel (+9.3% to +14.4%; Appendix G), and λ=0.5 beats both reward-only (λ=0) and size-only (λ=1) node selection, so both terms of LA-UCT contribute. We conclude that the benefit of MOSAIC comes from treating model choice as part of the joint MCTS action rather than from mixing the pool at random.

Scaling the LLM set improves the cost-quality tradeoff. Growing the pool from two to four to eight LLMs raises the geometric-mean final GPU speedup from 29.0× to 29.4× to 29.7×, and the compilation-time (API-cost) reduction from 1.05× (2.31×) to 1.50× (3.13×) to 1.95× (4.47×), since larger pools shift more expansions to small models. Eight LLMs are not the best everywhere, however: on GPU, MOSAIC(4 LLMs) beats MOSAIC(8 LLMs) on FLUX Attention Layer and FLUX Convolution Layer at 1000 samples and on end-to-end Llama-3-8B (1.86× vs. 1.61× over Reasoning Compiler; Table 3). Cost, in contrast, falls at each step from two to four to eight LLMs on both platforms (on CPU, 1.05×, 1.21×, and 1.74× lower compilation time). MOSAIC(8 LLMs) therefore offers the best cost-quality tradeoff rather than the highest speedup on every workload.

Small models perform most expansions. As Table 2 shows, the total GPT-5.2 invocation rate falls from 36.7% (35.1%) with MOSAIC(2 LLMs) to 23.1% (23.9%) with MOSAIC(8 LLMs) on GPU (CPU), and regular GPT-5.2 calls fall to only 9.8% (10.7%); the rest are course alterations, which use a shorter prompt.

End-to-end compilation. On end-to-end Llama-3-8B (Table 3), MOSAIC improves final speedup over Reasoning Compiler with GPT-5.2 by 1.11–1.86× on GPU and 1.19–1.41× on CPU, while reducing compilation time by 1.10–1.71× and API cost by 2.19–4.44×.

MOSAIC's gains hold with an open-weight largest LLM. With the open-weight Llama-3.3-70B-Instruct as the largest model (CPU; Appendix E), MOSAIC(8 LLMs) improves over Reasoning Compiler with Llama-3.3-70B-Instruct by 4.8–27.5% at 1000 samples on every kernel. It also reduces

compilation time by 1.84–2.32× and API cost by 1.91–2.67×, and it invokes the largest model for only 30.7% of calls.

3.3 LIMITATIONS

We evaluate five kernels, spanning attention, mixture-of-experts, convolution, and MLP operators from four model families, and one end-to-end model on one consumer GPU and one CPU within TVM; extending MOSAIC to data-center GPUs and other compilers is future work. The mechanism needs only a scalar reward per node, but we evaluate it only on compiler optimization. We report means over 10 runs; MOSAIC(8 LLMs) beats Reasoning Compiler with the same largest model on all 15 kernel settings (Appendix D), as does every λ and course-alteration setting on CPU (Appendices G–H). The size preference uses parameter count as a proxy for cost, and closed-model sizes are assumed (§2.3). We do not evaluate each small model alone, bandit-based model selection (Inoue et al., 2025; Lange et al., 2025; Zhu et al., 2025), cheap-first cascades (Aggarwal et al., 2024), separate per-model trees, or debate- and orchestrator-style multi-agent baselines (see also Appendix K).

4 RELATED WORK

Multi-LLM tree search. LE-MCTS (Park et al., 2025) ensembles LLMs inside MCTS for mathematical reasoning, choosing a model at random per expansion and scoring steps with a pre-trained process reward model. Multi-LLM AB-MCTS (Inoue et al., 2025) and ShinkaEvolve (Lange et al., 2025) instead select the LLM that expands a node or mutates a program with a bandit (Thompson sampling or UCB1) over per-model rewards, SYMPHONY (Zhu et al., 2025) assembles heterogeneous LLM agents in MCTS planning, and AlphaEvolve (Novikov et al., 2025) mixes a fast and a strong LLM for code evolution. MOSAIC differs in who chooses and why: the active LLM itself proposes the next model from the program context and per-model statistics, the tree policy prefers smaller models, and the largest model is escalated to after regressions, so the pool is used to cut cost rather than only to raise quality. Our random next-LLM ablation (Table 4) uses LE-MCTS's uniform model choice and improves over Reasoning Compiler by only 2.5%.

Adaptive LLM use under cost constraints. Confidence-based cascades and cost-aware routers balance LLM quality and cost per query (Chen et al., 2023; Ding et al., 2024; Aggarwal et al., 2024; Ong et al., 2024; Dekoninck et al., 2025; Jitkrittum et al., 2026; Zhang et al., 2025; Wang et al., 2026). MOSAIC instead assigns credit to model choices through their downstream effect on search reward, without a fixed or trained router.

Collaborative reasoning across LLMs. Multi-agent LLM systems coordinate through debate, critique, or message passing (Du et al., 2024; Lifshitz et al., 2025; Wang et al., 2025; Wu et al., 2023; Hong et al., 2024; Salemi et al., 2025; Wang et al., 2023). MOSAIC coordinates models only through the states and rewards of a shared MCTS tree, with one LLM call per expansion.

ML for compiler optimization. Compiler optimization has used both search and learning (Wang & O'Boyle, 2018), including MCTS without LLMs (Haj-Ali et al., 2020; Koo et al., 2021), LLMs (Cummins et al., 2023; Grubisic et al., 2024; Cummins et al., 2025; Deng et al., 2025; Cui et al., 2025; Pan et al., 2025), language models trained to generate tensor schedules (Zhai et al., 2024), and their combination in Reasoning Compiler (Tang et al., 2025).

5 CONCLUSION

We introduced MOSAIC, which turns a shared MCTS tree into the collaboration mechanism for heterogeneous LLMs, so that small models drive most decisions and MOSAIC finds faster code than a single large LLM at a fraction of its cost. LLM-guided optimizers need not scale by calling ever larger models, but by letting many inexpensive models share one search and calling the largest model mostly to correct course.

AI USE STATEMENT

In this work, we used generative AI tools for polishing writing, such as refining the language and improving readability. We have not used generative AI tools for retrieval and discovery, for research ideation or execution, for generating synthetic dataset, for drafting sections of the paper, or for proving mathematical claims. We take responsibility for the final content of this work, including text polished with the aid of generative AI.

REFERENCES

Pranjal Aggarwal, Aman Madaan, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, Shyam Upadhyay, Manaal Faruqui, and Mausam. AutoMix: Automatically mixing language models. In *NeurIPS*, 2024.

Apache TVM Community. Apache TVM v0.20.0. https://github.com/apache/tvm/releases/tag/v0.20.0, 2025.

Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2–3):235–256, 2002.

Black Forest Labs. Flux. https://github.com/black-forest-labs/flux, 2024.

Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? In *NeurIPS*, 2025.

Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *arXiv*, 2023.

Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *KDD*, 2016.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018a.

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *NeurIPS*, 2018b.

Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *CG*, 2007.

Tianming Cui, Pen-Chung Yew, Stephen McCamant, and Antonia Zhai. DeCOS: Data-efficient reinforcement learning for compiler optimization selection ignited by LLM. In *ICS*, 2025.

Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, and Hugh Leather. Large language models for compiler optimization. *arXiv*, 2023.

Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. LLM Compiler: Foundation language models for compiler optimization. In *CC*, 2025.

DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv*, 2025.

Jasper Dekoninck, Maximilian Baader, and Martin Vechev. A unified approach to routing and cascading for LLMs. In *ICML*, 2025.

Chaoyi Deng, Jialong Wu, Ningya Feng, Jianmin Wang, and Mingsheng Long. CompilerDream: Learning a compiler world model for general code optimization. In *KDD*, 2025.

Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Ruhle, Laks VS Lakshmanan, and Ahmed Hassan Awadallah. Hybrid LLM: Cost-efficient and quality-aware query routing. In *ICLR*, 2024.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. In *ICML*, 2024.

Dejan Grubisic, Chris Cummins, Volker Seeker, and Hugh Leather. Priority sampling of large language models for compilers. *arXiv*, 2024.

Ameer Haj-Ali, Hasan Genc, Qijing Huang, William Moses, John Wawrzynek, Krste Asanović, and Ion Stoica. ProTuner: Tuning programs with Monte Carlo tree search. *arXiv*, 2020.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *ICLR*, 2024.

Yuichi Inoue, Kou Misaki, Yuki Imajuku, So Kuroki, Taishi Nakamura, and Takuya Akiba. Wider or deeper? Scaling LLM inference-time compute with adaptive branching tree search. In *NeurIPS*, 2025.

Wittawat Jitkrittum, Harikrishna Narasimhan, Ankit Singh Rawat, Jeevesh Juneja, Congchao Wang, Zifeng Wang, Alec Go, Chen-Yu Lee, Pradeep Shenoy, Rina Panigrahy, Aditya Krishna Menon, and Sanjiv Kumar. Universal model routing for efficient LLM inference. In *ICLR*, 2026.

Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *ECML*, 2006.

Jaehoon Koo, Prasanna Balaprakash, Michael Kruse, Xingfu Wu, Paul Hovland, and Mary Hall. Customized Monte Carlo tree search for LLVM/Polly's composable loop optimization transformations. In *PMBS*, 2021.

Sameer Kulkarni and John Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. In *OOPSLA*, 2012.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards open-ended and sample-efficient program evolution. *arXiv*, 2025.

Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708–727, 2020.

Shalev Lifshitz, Sheila A. McIlraith, and Yilun Du. Multi-Agent Verification: Scaling test-time compute with multiple verifiers. In *COLM*, 2025.

Llama Team, AI @ Meta. The Llama 3 herd of models. *arXiv*, 2024.

Meta. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation. https://ai.meta.com/blog/llama-4-multimodal-intelligence/, 2025.

Mistral AI. Devstral. https://mistral.ai/news/devstral, 2025.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv*, 2025.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M. Waleed Kadous, and Ion Stoica. RouteLLM: Learning to route LLMs with preference data. *arXiv*, 2024.

Haolin Pan, Hongyu Lin, Haoran Luo, Yang Liu, Kaichun Yao, Libo Zhang, Mingjie Xing, and Yanjun Wu. Compiler-R1: Towards agentic compiler auto-tuning with reinforcement learning. In *NeurIPS*, 2025.

Sungjin Park, Xiao Liu, Yeyun Gong, and Edward Choi. Ensembling large language models with process reward-guided tree search for better complex reasoning. In *NAACL*, 2025.

Dylan Patel and Afzal Ahmad. The inference cost of search disruption – large language model cost analysis. https://newsletter.semianalysis.com/p/the-inference-cost-of-search-disruption, 2023.

Alireza Salemi, Mihir Parmar, Palash Goyal, Yiwen Song, Jinsung Yoon, Hamed Zamani, Tomas Pfister, and Hamid Palangi. LLM-based multi-agent blackboard system for information discovery in data science. *arXiv*, 2025.

Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. Tensor program optimization with probabilistic programs. In *NeurIPS*, 2022.

Andries Smit, Nathan Grinsztajn, Paul Duckworth, Thomas D. Barrett, and Arnu Pretorius. Should we be going MAD? a look at multi-agent debate strategies for LLMs. In *ICML*, 2024.

Annabelle Sujun Tang, Christopher Priebe, Rohan Mahapatra, Lianhui Qin, and Hadi Esmaeilzadeh. REASONING COMPILER: LLM-guided optimizations for efficient model serving. In *NeurIPS*, 2025.

Qwen Team. Qwen3 technical report. *arXiv*, 2025.

Sid-Ahmed-Ali Touati and Denis Barthou. On the decidability of phase ordering problem in optimizing compilation. In *CF*, 2006.

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor Comprehensions:

Framework-agnostic high-performance machine learning abstractions. *arXiv*, 2018.
Chenyu Wang, Zishen Wan, Hao Kang, Emma Chen, Zhiqiang Xie, Tushar Krishna, Vijay Janapa Reddi, and Yilun Du. SLM-MUX: Orchestrating small language models for reasoning. In *ICLR*, 2026.
Junlin Wang, Jue Wang, Ben Athiwaratkun, Ce Zhang, and James Zou. Mixture-of-agents enhances large language model capabilities. In *ICLR*, 2025.
Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *ICLR*, 2023.
Zheng Wang and Michael O'Boyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018.
Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv*, 2023.
Yi Zhai, Sijia Yang, Keyu Pan, Renwei Zhang, Shuo Liu, Chao Liu, Zichun Ye, Jianmin Ji, Jie Zhao, Yu Zhang, and Yanyong Zhang. Enabling tensor language model to assist in generating high-performance tensor programs for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.
Haozhen Zhang, Tao Feng, and Jiaxuan You. Router-R1: Teaching LLMs multi-round routing and aggregation via reinforcement learning. In *NeurIPS*, 2025.
Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *OSDI*, 2020.
Wei Zhu, Zhiwen Tang, and Kun Yue. SYMPHONY: Synergistic multi-agent planning with heterogeneous language model assembly. In *NeurIPS*, 2025.

A Local Equivalence and Concentration of LA-UCT

Proposition 1 formalizes the local behavior of the LA-UCT rule introduced in Section 2.3.

Proposition 1 (Local equivalence and concentration of LA-UCT)**.** Fix a parent node u with a fixed, finite set of children $v_1, \dots, v_K$, where v_j is expanded by model $\text{llm}_t^{(j)}$, and assume: (A1) the rewards $R_{j,s}$ observed at v_j are i.i.d. across visits with mean μ_j; (A2) $R_{j,s} \in [0,1]$; (A3) the maximizer $j_\lambda = \text{argmax}_j \widetilde{\mu}_j$ of $\widetilde{\mu}_j = (1-\lambda)\mu_j + \lambda\phi_{\text{small}}(\text{llm}_t^{(j)})$ is unique; and (A4) $c = \sqrt{2}$. Then selecting children of u by LA-UCT is UCB1 applied to the transformed rewards $Y_{j,s} = (1-\lambda)R_{j,s} + \lambda\phi_{\text{small}}(\text{llm}_t^{(j)})$, and every $j \neq j_\lambda$ satisfies $\mathbb{E}[N(v_j)] \leq 8\ln N(u)/(\Delta_j^\lambda)^2 + 1 + \pi^2/3$, where $\Delta_j^\lambda = \widetilde{\mu}_{j_\lambda} - \widetilde{\mu}_j$.

Scope. In MOSAIC, (A1) and (A2) hold only approximately: rewards come from a cost model that MetaSchedule retrains online, subtrees grow and course alteration can replace children, so child rewards are not stationary, and predictions are clipped below at 0 but not above (Section 3.1). With branching factor $B = 2$ and 1000 iterations, most parents also receive few visits. The proposition therefore describes the intended local behavior of the selection rule rather than a guarantee for our runs.

Proof. Let $u = \langle p_{t-1}, \text{llm}_{t-1}\rangle$ be a parent node. Suppose that u has finitely many children

$$v_j = \langle p_t^{(j)}, \text{llm}_t^{(j)}\rangle, \qquad j = 1, \dots, K.$$

Let $R_{j,s}$ be the reward observed the s-th time node v_j is selected; by (A2), $R_{j,s} \in [0,1]$. Any regression-based cost model's output can be brought into this range by an order-preserving map, such as clipping or min–max scaling. In our implementation, the predictions of TVM's XGBoost model approximate normalized throughput in $(0,1]$ and are clipped below at 0 (Section 3.1).

The LA-UCT score of child v_j is

$$\text{LA-UCT}(v_j) = (1-\lambda)\frac{W(v_j)}{N(v_j)} + \lambda\phi_{\text{small}}\big(\text{llm}_t^{(j)}\big) + c\sqrt{\frac{\ln N(u)}{N(v_j)}}.$$

Define the transformed reward

$$Y_{j,s} = (1-\lambda)R_{j,s} + \lambda\phi_{\text{small}}\big(\text{llm}_t^{(j)}\big).$$

Since $R_{j,s} \in [0,1]$ and $\phi_{\text{small}}\big(\text{llm}_t^{(j)}\big) \in [0,1]$, we have $Y_{j,s} \in [0,1]$.

Then

$$\begin{aligned}
\overline{Y}_j &= \frac{\sum_{s=1}^{N(v_j)} Y_{j,s}}{N(v_j)} \\
&= \frac{\sum_{s=1}^{N(v_j)} \Big((1-\lambda)R_{j,s} + \lambda\phi_{\text{small}}\big(\text{llm}_t^{(j)}\big)\Big)}{N(v_j)} \\
&= \frac{(1-\lambda)W(v_j) + N(v_j)\lambda\phi_{\text{small}}\big(\text{llm}_t^{(j)}\big)}{N(v_j)} \\
&= (1-\lambda)\frac{W(v_j)}{N(v_j)} + \lambda\phi_{\text{small}}\big(\text{llm}_t^{(j)}\big).
\end{aligned}$$

Therefore,

$$\text{LA-UCT}(v_j) = \overline{Y}_j + c\sqrt{\frac{\ln N(u)}{N(v_j)}}.$$

Thus, at a fixed parent node, LA-UCT is UCT applied to the transformed reward

$$Y_{j,s} = (1-\lambda)R_{j,s} + \lambda\phi_{\text{small}}\big(\text{llm}_t^{(j)}\big).$$

Define the transformed mean

$$\widetilde{\mu}_j = \mathbb{E}[Y_{j,s}] = (1-\lambda)\mathbb{E}[R_{j,s}] + \lambda\phi_{\text{small}}\big(\text{llm}_t^{(j)}\big).$$

Let $\mu_j = \mathbb{E}[R_{j,s}]$. Then $\widetilde{\mu}_j = \mathbb{E}[Y_{j,s}] = (1-\lambda)\mu_j + \lambda\phi_{\text{small}}\big(\text{llm}_t^{(j)}\big)$. By (A3), the maximizer $j_\lambda = \text{argmax}_j\, \widetilde{\mu}_j$ is unique.

For each suboptimal child $j \neq j_\lambda$, define the surrogate gap

$$\Delta_j^\lambda = \widetilde{\mu}_{j_\lambda} - \widetilde{\mu}_j > 0.$$

Since the transformed rewards satisfy

$$Y_{j,s} \in [0,1],$$

we can apply Theorem 1 of Auer et al. (2002) to the bandit with arm means $\widetilde{\mu}_j$, optimal arm j_λ, and pull counts $N(v_j)$. By (A1), (A2), and (A4), the theorem applies to UCB1 with index $\overline{Y}_j + \sqrt{2\ln N(u)/N(v_j)}$ and gives, for each suboptimal arm j,

$$\mathbb{E}[N(v_j)] \leq \frac{8\ln N(u)}{\big(\Delta_j^\lambda\big)^2} + 1 + \frac{\pi^2}{3}.$$

Therefore, LA-UCT concentrates its visits on the child maximizing

$$\widetilde{\mu}_j = (1-\lambda)\mu_j + \lambda\phi_{\text{small}}\big(\text{llm}_t^{(j)}\big).$$

At any fixed parent node, LA-UCT is equivalent to UCB1 applied to a transformed reward that combines expected downstream program reward with the normalized preference for smaller models. Therefore, the policy allocates a vanishing fraction of visits to children that are suboptimal with respect to this surrogate mean, and asymptotically concentrates on children that best balance downstream reward and model size preference. In this sense, the proposition formalizes the intended size-aware behavior of LA-UCT: smaller LLMs are favored when they are sufficient, while larger LLMs remain preferred whenever they provide sufficiently greater expected downstream value. □

B PROMPT

Regular Model Invocation Prompt

You are an AI scheduling assistant to help with a Monte Carlo Tree Search (MCTS) to find an optimal program in the search space starting from an unoptimized program.
In this MCTS, the current program is the leaf we are expanding, while immediate parent and grandparent refer to the ancestors in the tree.
Each program has:

- a piece of code
- a transformation history sequence
- a predicted performance score

You are given:

- Code of the current program
- Historical performance info of current/parent/grandparent (Pieces of code, transformation history sequences, predicted scores)
- A list of possible transformations that can be applied next
- Search context: leaf depth and trials progress
- Global per-model stats:
  - hit_rate: fraction of calls where score(child) > score(parent)
  - number of times the model is invoked
  - number of errors the model has made in past invocations (+1 for invalid transformation, +1 for invalid next_model name)
  - parameter counts
- Local model context:
  - model used to expand current/parent/grandparent

Task:

1. Compare code/transformation history/predicted performance scores to infer what changes might improve performance.
2. Propose a sequence of transformations from the provided list. You may repeat a transformation to explore different decisions.
3. Choose exactly one model from the provided model list as the next model to expand the child. Use the smallest model that could give best results. Prefer models with fewer errors.

Output a single valid JSON object in the EXACT format:

```
{
  "transformations": ["Fullname1", "Fullname2", "..."],
  "next_model": "..."
}
```

Historical Performance Info (Leaf, Parent, Grandparent)
Current Program:
Code:

```
@T.prim_func
def main(A, B, C):
    C_global = T.alloc_buffer((1, 16, 4096))
    for fused in T.parallel(2):
        for b_1, i_1, j_1 in T.grid(1, 2, 64):
            ...
            for k_0, ..., k_1, ... in T.grid(512, ..., 8, ...):
                for j_vec in T.vectorized(64):
                    with T.block("matmul_update"):
                        ...
```

```
                        C_global[...] = C_global[...] + A[...] * B
    [...]
        for ax0, ax1, ax2 in T.grid(1, 8, 4096):
            C[...] = C_global[...]
```

Transformation history:

```
sch.sample_perfect_tile(loop=j, decision=[1, 64, 1, 64])
sch.vectorize(...)
sch.cache_write(..., storage_scope="global")
sch.decompose_reduction(...)
...
```

Predicted score: 0.0739

Immediate Parent Schedule:

Code:

```
@T.prim_func
def main(A, B, C):
    C_global = T.alloc_buffer((1, 16, 4096))
    for fused in T.parallel(4):
        for b_1, i_1, j_1 in T.grid(1, 2, 32):
            ...
            for k_0, ..., k_1, ... in T.grid(512, ..., 8, ...):
                for j_vec in T.vectorized(32):
                    with T.block("matmul_update"):
                        ...
                        C_global[...] = C_global[...] + A[...] * B
    [...]
        for ax0, ax1, ax2 in T.grid(1, 8, 2048):
            C[...] = C_global[...]
```

Transformation history:

```
sch.sample_perfect_tile(loop=j, decision=[2, 32, 2, 32])
sch.vectorize(...)
sch.cache_write(..., storage_scope="global")
sch.decompose_reduction(...)
...
```

Predicted score: 0.136

Available Transformations

```
[
  "ComputeLocation",
  "Parallel",
  "Unroll",
  "TileSize"
]
```

Search Context

Leaf depth: 3

Trials progress: 10 / 300

Global Per-Model Stats

Model gpt-5-mini: params=20.0B, regular_calls=12, regular_hit_rate=0.364, errors=0

Model gpt-5.2: params=300.0B, regular_calls=11, regular_hit_rate=0.5, course_alteration_calls=0, course_alteration_hit_rate=0, errors=0

Local Model Context

Model used to expand the current node: gpt-5.2

Model used to expand the parent node: gpt-5.2

Model used to expand the grandparent node: N/A

Model Answer

```
"transformations": [ "TileSize", "TileSize", "ComputeLocation",
"Parallel", "Unroll" ], "next_model": "gpt-5-mini"
```

Course Alteration Prompt

You are the largest model invoked for course alteration in a Monte Carlo Tree Search (MCTS) for compiler optimization. A smaller model has proposed a sequence of transformations and a next model for expanding the child node. This proposal triggered course alteration because the predicted score of the resulting child is lower than the predicted score of the current program.
In this MCTS, the current program is the leaf being expanded, while immediate parent and grandparent refer to ancestors in the tree. Each program has:

- a piece of code
- a transformation history sequence
- a predicted performance score

You are given:

- Code of the current program
- Code of the parent program
- A list of possible transformations that can be applied next
- The smaller model's proposed transformations and proposed next model
- The predicted score of the current program and the predicted score after applying the smaller model's proposal
- Search context: leaf depth and trials progress
- Global per-model stats:
  - hit_rate: fraction of calls where score(child) > score(parent)
  - number of times the model is invoked
  - number of errors the model has made in past invocations (+1 for invalid transformation, +1 for invalid next_model name)
  - parameter counts
- Local model context:
  - model used to expand current/parent/grandparent

Task:

1. Modify the smaller model's proposal by changing the transformation sequence, the next model, or both. Compare code/predicted performance scores to infer what changes might improve performance.
2. Propose a sequence of transformations from the provided list. You may repeat a transformation to explore different decisions.
3. Choose exactly one model from the provided model list as the next model to expand the child. Use the smallest model that could give best results. Prefer models with fewer errors.

Output a single valid JSON object in the EXACT format:

```
{
  "transformations": ["Fullname1", "Fullname2", "..."],
  "next_model": "..."
}
```

Historical Performance Info (Leaf, Parent, Grandparent)
Current Program:
Code:

```
@T.prim_func
def main(A, B, C):
    C_global = T.alloc_buffer((1, 16, 4096))
    for fused in T.parallel(2):
        for b_1, i_1, j_1 in T.grid(1, 4, 1):
            ...
            for k_0, ..., k_1, ... in T.grid(80, ..., 64, ...):
                for j_vec in T.vectorized(32):
                    with T.block("matmul_update"):
                        ...
                        C_global[...] = C_global[...] + A[...] * B
    [...]
        for ax0, ax1, ax2 in T.grid(1, 16, 2048):
            C[...] = C_global[...]
```

Predicted score: 0.460
Immediate Parent Program:
Code:

```
@T.prim_func
def main(A, B, C):
    C_global = T.alloc_buffer((1, 16, 4096))
    for fused in T.parallel(64):
        for b_1, i_1, j_1 in T.grid(1, 8, 1):
            ...
            for k_0, ..., k_1, ... in T.grid(80, ..., 64, ...):
                for j_vec in T.vectorized(32):
                    with T.block("matmul_update"):
                        ...
                        C_global[...] = C_global[...] + A[...] * B
    [...]
        for ax0, ax1, ax2 in T.grid(1, 16, 256):
            C[...] = C_global[...]
```

Predicted score: 0.572
Smaller Model Proposal Triggering Course Alteration
Smaller model name: gpt-5-mini
Proposed transformations:

```
["TileSize", "Parallel", "Unroll"]
```

Proposed next model: gpt-5.2
Predicted current score: 0.460
Predicted child score from smaller model proposal: 0.028
Search Context
Leaf depth: 9
Trials progress: 179 / 300
Global Per-Model Stats
Model gpt-5-mini: params=20.0B, regular_calls=425, regular_hit_rate=0.494, errors=0
Model gpt-5.2: params=300.0B, regular_calls=232, regular_hit_rate=0.513, course_alteration_calls=73, course_alteration_hit_rate=0.282, errors=0
Local Model Context
Model used to expand the current node: gpt-5-mini
Model used to expand the parent node: gpt-5-mini
Model used to expand the grandparent node: gpt-5.2

Course Alteration Model Answer

```
"transformations": ["TileSize", "ComputeLocation"], "next_model":
"gpt-5-mini"
```

C CODE REPOSITORY

The code implementation of MOSAIC is available in the Supplementary Material.

D PER-KERNEL FINAL SPEEDUPS

Table 5 lists the speedups at 1000 samples plotted in Figure 2 and Figure 3. Tang et al. (2025) compare Reasoning Compiler with TVM MetaSchedule on these CPU kernels in a separate experimental campaign; absolute speedups differ between the two campaigns, so that comparison indicates relative sample efficiency rather than absolute speedups in our setting.

Table 5: Speedup over unoptimized code at 1000 samples. RC is Reasoning Compiler run with the named LLM alone. CPU (70B) uses Llama-3.3-70B-Instruct as the largest model.

Platform	Method	Llama-3 Attn.	DS-R1 MoE	FLUX Attn.	FLUX Conv.	L4-Scout MLP	Geomean
GPU	RC (gpt-5-mini)	29.49	29.56	25.05	15.60	27.90	24.86
	RC (GPT-5.2)	30.50	31.25	29.17	19.09	30.62	27.68
	MOSAIC(2 LLMs)	31.07	32.40	30.50	21.68	30.76	28.99
	MOSAIC(4 LLMs)	31.35	32.13	31.28	21.95	31.51	29.35
	MOSAIC(8 LLMs)	**32.70**	**33.77**	**30.80**	**21.37**	**31.64**	**29.67**
CPU	RC (gpt-5-mini)	11.84	11.10	7.31	4.17	10.26	8.37
	RC (GPT-5.2)	12.40	12.35	7.35	4.64	10.73	8.91
	MOSAIC(2 LLMs)	13.90	13.10	7.59	5.37	10.87	9.58
	MOSAIC(4 LLMs)	14.20	13.40	7.94	5.22	11.14	9.74
	MOSAIC(8 LLMs)	**14.98**	**13.87**	**8.12**	**5.51**	**11.82**	**10.19**
CPU (70B)	RC (gpt-5-mini)	11.84	11.10	7.31	4.17	10.26	8.37
	RC (Llama-3.3-70B)	12.50	10.82	7.42	3.82	10.80	8.38
	MOSAIC(2 LLMs)	12.95	11.49	7.53	4.32	10.97	8.81
	MOSAIC(4 LLMs)	12.73	11.91	7.72	4.38	11.72	9.03
	MOSAIC(8 LLMs)	**13.10**	**12.50**	**7.89**	**4.87**	**11.91**	**9.44**

E LLAMA-3.3-70B-INSTRUCT AS THE LARGEST MODEL

Figure 3 shows CPU speedup when Llama-3.3-70B-Instruct (Llama Team, AI @ Meta, 2024) replaces GPT-5.2 as the largest model, so that every model except gpt-5-mini is open-weight with a public parameter count. MOSAIC(8 LLMs) achieves the best final speedup on every kernel, improves over Reasoning Compiler with Llama-3.3-70B-Instruct by 4.8–27.5% at 1000 samples, and also gains early (e.g., 6.3× vs. 5.2× on FLUX Attention Layer at 100 samples). Reasoning Compiler with gpt-5-mini and Reasoning Compiler with Llama-3.3-70B-Instruct reach nearly the same geometric mean (8.37× vs. 8.38×; Table 5), and MOSAIC(8 LLMs) improves over both, to 9.44×.

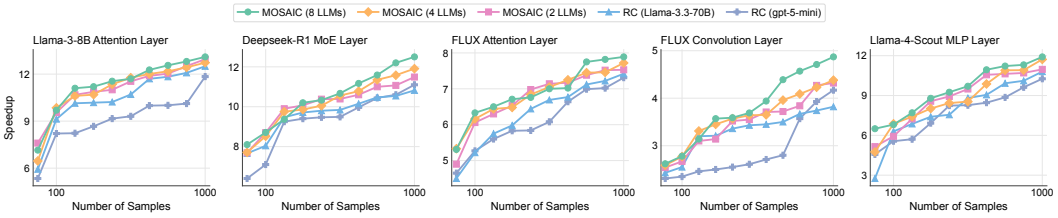


Figure 3: CPU speedup over unoptimized code as a function of the number of searched samples, using Llama-3.3-70B-Instruct as the largest model. RC is Reasoning Compiler run with the named LLM alone.

F Absolute Compilation Time and API Cost

Table 6: Absolute compilation time and API cost on CPU for Reasoning Compiler with GPT-5.2 and for MOSAIC with 2, 4, and 8 LLMs.

	Setting	Runtime (sec)	Runtime Imp.	API Cost	API Cost Imp.
Llama-3-8B Attention Layer	Reasoning Compiler (GPT-5.2)	2236	—	$13.49	—
	2 Models	2227	1.00	$5.11	2.64
	4 Models	2051	1.09	$5.25	2.57
	8 Models	1514	1.48	$4.41	3.06
	Setting	Runtime (sec)	Runtime Imp.	API Cost	API Cost Imp.
DeepSeek-R1 MoE Layer	Reasoning Compiler (GPT-5.2)	2493	—	$13.75	—
	2 Models	2270	1.10	$5.68	2.42
	4 Models	1675	1.49	$4.28	3.21
	8 Models	1488	1.68	$3.29	4.18
	Setting	Runtime (sec)	Runtime Imp.	API Cost	API Cost Imp.
FLUX Attention Layer	Reasoning Compiler (GPT-5.2)	2494	—	$13.78	—
	2 Models	2265	1.10	$5.82	2.37
	4 Models	2112	1.18	$4.77	2.89
	8 Models	1227	2.03	$3.00	4.59
	Setting	Runtime (sec)	Runtime Imp.	API Cost	API Cost Imp.
FLUX Convolution Layer	Reasoning Compiler (GPT-5.2)	2393	—	$16.73	—
	2 Models	2342	1.02	$7.71	2.17
	4 Models	1968	1.22	$5.65	2.96
	8 Models	1382	1.73	$2.85	5.87
	Setting	Runtime (sec)	Runtime Imp.	API Cost	API Cost Imp.
Llama-4-Scout MLP Layer	Reasoning Compiler (GPT-5.2)	2262	—	$14.07	—
	2 Models	2166	1.04	$6.32	2.23
	4 Models	2049	1.10	$5.49	2.56
	8 Models	1223	1.85	$3.21	4.38

G Ablations of Lambda Parameters

As shown in Table 7, at 1000 searched samples $\lambda = 0.5$ gives the highest final speedup on every benchmark (geometric mean $10.19\times$), and every tested value improves over Reasoning Compiler. $\lambda = 0.5$ provides the best empirical balance between the two components of the LA-UCT surrogate reward: the downstream program reward term and the normalized model size preference term. With this balance, LA-UCT can preserve the compiler performance objective while still encouraging efficient use of smaller models when they are sufficient, as shown in Table 8, allowing the search to allocate model choices more effectively and achieve the best overall speedup.

Table 7: Speedup over unoptimized code across varying numbers of samples for different choices of λ values.

Benchmark	λ Value / Number of Samples	50	100	250	500	750	1000
Llama-3-8B Attention Layer	0.00	7.36	10.22	10.93	12.27	12.50	14.62
	0.25	7.49	10.49	10.82	12.51	12.70	14.85
	0.50	7.56	10.61	10.99	12.65	12.81	14.98
	0.75	7.30	10.19	10.61	12.16	12.40	14.52
	1.00	7.09	9.87	10.78	11.77	12.09	14.15
DeepSeek-R1 MoE Layer	0.00	7.22	9.01	11.64	12.05	12.37	13.65
	0.25	7.08	8.82	11.71	12.12	12.34	13.80
	0.50	6.99	8.66	11.84	12.20	12.28	13.87
	0.75	6.97	8.62	11.54	11.99	12.24	13.56
	1.00	6.95	8.44	11.36	11.87	12.26	13.44
FLUX Attention Layer	0.00	5.26	5.64	6.43	6.99	7.07	7.85
	0.25	5.35	5.58	6.58	7.09	7.18	8.02
	0.50	5.47	5.69	6.72	7.18	7.24	8.12
	0.75	5.33	5.62	6.55	7.06	7.15	7.96
	1.00	5.19	5.55	6.40	6.95	7.03	7.74
FLUX Convolution Layer	0.00	2.84	3.76	4.13	4.51	4.93	5.44
	0.25	2.87	3.80	4.16	4.48	4.98	5.48
	0.50	2.88	3.83	4.19	4.53	5.01	5.51
	0.75	2.82	3.74	4.02	4.42	4.88	5.38
	1.00	2.77	3.67	4.10	4.29	4.76	5.26
Llama-4-Scout MLP Layer	0.00	6.43	6.78	9.62	10.31	10.80	11.58
	0.25	6.60	6.88	9.78	10.47	10.76	11.73
	0.50	6.75	6.93	9.93	10.59	10.89	11.82
	0.75	6.55	6.85	9.72	10.50	10.70	11.66
	1.00	6.12	6.33	9.18	10.10	10.38	11.31

Table 8: Invocation rates (%) of different models for different choices of λ values.

Benchmark	λ / Models	DeepSeek-R1-Distill-Qwen-7B	Llama-3.1-8B-Instruct	Qwen3-8B	Qwen3-14B	Devstral-Small-2505	gpt-5-mini	DeepSeek-R1-Distill-Qwen-32B	gpt-5.2	gpt-5.2 Course Alteration
Llama-3-8B Attention Layer	0.00	16.2%	18.7%	10.0%	7.4%	1.3%	5.8%	4.8%	23.1%	12.7%
	0.25	13.2%	6.5%	28.0%	7.2%	3.6%	7.2%	1.0%	19.8%	13.5%
	0.50	11.6%	7.6%	34.1%	5.1%	1.0%	5.6%	2.5%	18.3%	14.3%
	0.75	15.4%	22.4%	16.8%	7.8%	1.0%	5.3%	1.8%	14.4%	15.1%
	1.00	14.4%	9.2%	32.1%	3.8%	4.5%	6.1%	1.3%	12.8%	16.0%
DeepSeek-R1 MoE Layer	0.00	10.3%	20.9%	7.3%	11.1%	3.0%	21.7%	1.9%	12.4%	11.4%
	0.25	12.6%	16.0%	10.7%	11.7%	4.0%	20.2%	1.7%	10.9%	11.9%
	0.50	12.4%	23.5%	8.9%	1.8%	2.7%	27.3%	1.3%	9.6%	12.4%
	0.75	11.9%	25.7%	10.2%	1.6%	4.2%	22.0%	2.7%	9.0%	12.8%
	1.00	14.8%	25.7%	11.6%	3.9%	4.5%	16.4%	1.5%	9.4%	12.4%
FLUX Attention Layer	0.00	12.6%	13.8%	22.3%	11.9%	3.9%	3.3%	8.2%	11.5%	12.4%
	0.25	30.6%	14.5%	16.5%	2.7%	3.5%	2.5%	4.9%	12.0%	12.8%
	0.50	13.7%	14.9%	36.5%	2.3%	3.4%	1.7%	4.9%	9.2%	13.3%
	0.75	5.9%	28.3%	20.8%	10.7%	5.1%	3.0%	3.7%	8.5%	13.9%
	1.00	21.3%	16.0%	24.8%	7.5%	2.1%	0.9%	5.1%	7.4%	14.2%
FLUX Convolution Layer	0.00	11.6%	21.2%	25.2%	10.8%	4.1%	2.1%	0.8%	12.8%	11.4%
	0.25	5.9%	29.9%	25.0%	7.3%	5.1%	4.9%	0.4%	8.8%	12.5%
	0.50	6.2%	3.7%	54.5%	11.2%	4.3%	0.5%	0.3%	5.7%	13.5%
	0.75	23.9%	27.8%	12.7%	10.6%	5.2%	0.6%	1.0%	0.4%	14.1%
	1.00	23.1%	21.3%	22.6%	7.8%	2.4%	3.0%	1.4%	3.8%	14.6%
Llama-4-Scout MLP Layer	0.00	16.0%	30.4%	12.2%	6.5%	1.0%	5.6%	2.0%	14.2%	12.0%
	0.25	17.2%	31.8%	13.9%	2.0%	1.5%	7.5%	1.4%	12.3%	12.4%
	0.50	19.0%	7.3%	41.6%	3.1%	0.8%	4.2%	0.9%	10.6%	12.6%
	0.75	20.5%	24.7%	25.7%	2.7%	0.7%	3.4%	0.6%	8.9%	12.9%
	1.00	19.4%	31.6%	18.0%	4.9%	1.3%	2.7%	2.4%	6.4%	13.2%

H Course Alteration Ablations

Across the five benchmarks, triggering course alteration after every small model regression achieves the highest speedup, whereas disabling course alteration yields the lowest speedup. MOSAIC instead triggers course alteration after every two small model regressions, providing the best speedup-cost tradeoff. As shown in Table 10 and Table 11, after 1000 searched samples, MOSAIC(8 LLMs) lowers the largest model course alteration rate from 29.9% to 13.2%, while reducing compilation time by 1.38× and API cost by 1.58× (geometric means over the five benchmarks). This reduced course alteration rate incurs only a 1.3% mean relative final-speedup gap per kernel, as shown in Table 9 (1.5 points of geometric-mean gain over Reasoning Compiler; Table 4).

Table 9: Speedup over unoptimized code across varying numbers of samples for different course alteration settings.

Benchmark	Course Alteration / Number of Samples	50	100	250	500	750	1000
Llama-3-8B Attention Layer	No Course Alteration	6.91	10.02	10.19	11.78	12.01	14.05
	Every 1 Small Model Regression	7.60	10.68	11.16	12.88	13.07	15.26
	Every 2 Small Model Regressions	7.56	10.61	10.99	12.65	12.81	14.98
DeepSeek-R1 MoE Layer	No Course Alteration	6.92	8.57	10.88	11.85	11.91	12.91
	Every 1 Small Model Regression	7.56	9.29	12.09	12.31	12.40	14.24
	Every 2 Small Model Regressions	6.99	8.66	11.84	12.20	12.28	13.87
FLUX Attention Layer	No Course Alteration	5.07	5.58	6.55	6.94	7.02	7.61
	Every 1 Small Model Regression	5.50	5.74	6.82	7.23	7.32	8.18
	Every 2 Small Model Regressions	5.47	5.69	6.72	7.18	7.24	8.12
FLUX Convolution Layer	No Course Alteration	2.84	3.07	4.11	4.43	4.78	5.33
	Every 1 Small Model Regression	2.91	3.87	4.25	4.61	5.10	5.53
	Every 2 Small Model Regressions	2.88	3.83	4.19	4.53	5.01	5.51
Llama-4-Scout MLP Layer	No Course Alteration	5.57	6.18	8.53	9.54	10.55	11.36
	Every 1 Small Model Regression	6.87	7.08	10.02	10.78	10.98	11.92
	Every 2 Small Model Regressions	6.75	6.93	9.93	10.59	10.89	11.82

Table 10: Invocation rates (%) of different models for different course alteration settings.

Benchmark	Course Alteration / Models	DeepSeek-R1-Distill-Qwen-7B	Llama-3.1-8B-Instruct	Qwen3-8B	Qwen3-14B	Devstral-Small-2505	gpt-5-mini	DeepSeek-R1-Distill-Qwen-32B	gpt-5.2	gpt-5.2 Course Alteration
Llama-3-8B Attention Layer	No Course Alteration	13.8%	17.5%	30.2%	5.3%	1.4%	7.8%	2.6%	21.4%	0.0%
	Every 1 Small Model Reg.	13.8%	12.5%	14.8%	4.0%	2.2%	8.7%	1.5%	11.7%	30.7%
	Every 2 Small Model Reg.	11.6%	7.6%	34.1%	5.1%	1.0%	5.6%	2.5%	18.3%	14.3%
DeepSeek-R1 MoE Layer	No Course Alteration	18.9%	23.7%	10.5%	1.1%	2.7%	30.9%	1.4%	10.8%	0.0%
	Every 1 Small Model Reg.	11.7%	15.8%	6.5%	3.4%	2.9%	21.2%	1.2%	8.7%	28.4%
	Every 2 Small Model Reg.	12.4%	23.5%	8.9%	1.8%	2.7%	27.3%	1.3%	9.6%	12.4%
FLUX Attention Layer	No Course Alteration	15.9%	13.4%	22.6%	11.6%	8.1%	11.7%	1.3%	15.4%	0.0%
	Every 1 Small Model Reg.	17.6%	11.1%	8.6%	4.9%	2.6%	16.3%	0.8%	9.4%	28.7%
	Every 2 Small Model Reg.	13.7%	14.9%	36.5%	2.3%	3.4%	1.7%	4.9%	9.2%	13.3%
FLUX Convolution Layer	No Course Alteration	9.8%	25.2%	32.7%	13.4%	4.8%	5.7%	0.6%	7.8%	0.0%
	Every 1 Small Model Reg.	12.2%	17.0%	21.1%	9.0%	3.4%	0.8%	1.0%	5.7%	29.8%
	Every 2 Small Model Reg.	6.2%	3.7%	54.5%	11.2%	4.3%	0.5%	0.3%	5.7%	13.5%
Llama-4-Scout MLP Layer	No Course Alteration	21.1%	8.2%	34.9%	3.4%	9.2%	7.7%	0.9%	14.6%	0.0%
	Every 1 Small Model Reg.	15.1%	19.5%	14.3%	2.5%	4.0%	3.3%	2.5%	7.2%	31.7%
	Every 2 Small Model Reg.	19.0%	7.3%	41.6%	3.1%	0.8%	4.2%	0.9%	10.6%	12.6%

Table 11: Compilation time and API cost reduction of performing course alteration every two small model regression against performing course alteration every one small model regression.

	Llama-3-8B Attention Layer	DeepSeek-R1 MoE Layer	FLUX Attention Layer	FLUX Convolution Layer	Llama-4-Scout MLP Layer
Comp. Time ↓ (×)	1.29	1.34	1.53	1.37	1.38
API Cost ↓ (×)	1.30	1.57	1.69	1.80	1.58

I LLM SELECTION ABLATIONS

MOSAIC's gains are not due to LLM diversity alone. To isolate the role of endogenous LLM selection, we compare MOSAIC(8 LLMs) against two ablations that use the same eight LLM pool but replace MOSAIC's state-dependent routing with random or round-robin next model selection. Unlike these static policies, MOSAIC makes the next model choice part of the joint action: in each expansion, the active LLM considers the current program context, search progress and per model statistics, and proposes both a compiler transformation and the LLM that should expand the resulting child. As shown in Table 12, MOSAIC(8 LLMs) achieves the best speedup at every reported number of searched samples on every benchmark. For example, MOSAIC(8 LLMs) achieves 16.8% and 15.6% higher final speedup compared with random and round-robin LLM selection respectively on the Llama-3-8B Attention Layer. The same trend holds for tuning cost: Table 14 shows that MOSAIC(8 LLMs) reduces compilation time by 1.37×/1.38× and API cost by 1.27×/1.26× (geometric means over the five benchmarks) relative to random/round-robin selection. These results show that random and round-robin LLM selection fail to choose the LLMs with the best cost-quality tradeoffs. Therefore, on CPU, the benefit of MOSAIC over these static policies comes from treating model choice as part of the joint MCTS action, rather than simply exposing the compiler to a heterogeneous set of LLMs.

Table 12: Speedup over unoptimized code across varying numbers of samples for different LLM selection settings.

Benchmark	LLM Selection / Number of Samples	50	100	250	500	750	1000
Llama-3-8B Attention Layer	MOSAIC(8 LLMs)	7.56	10.61	10.99	12.65	12.81	14.98
	Random	6.20	8.55	10.05	10.60	11.10	12.82
	Round-Robin	6.30	8.70	10.18	10.55	11.25	12.96
DeepSeek-R1 MoE Layer	MOSAIC(8 LLMs)	6.99	8.66	11.84	12.20	12.28	13.87
	Random	5.28	7.62	10.72	11.90	11.98	12.54
	Round-Robin	5.06	6.60	10.89	11.96	11.93	12.69
FLUX Attention Layer	MOSAIC(8 LLMs)	5.47	5.69	6.72	7.18	7.24	8.12
	Random	4.61	5.53	6.43	6.80	6.84	7.37
	Round-Robin	3.74	5.01	6.36	6.85	6.88	7.42
FLUX Convolution Layer	MOSAIC(8 LLMs)	2.88	3.83	4.19	4.53	5.01	5.51
	Random	2.24	2.73	3.35	3.91	4.65	4.93
	Round-Robin	2.32	2.53	3.08	3.78	4.63	4.82
Llama-4-Scout MLP Layer	MOSAIC(8 LLMs)	6.75	6.93	9.93	10.59	10.89	11.82
	Random	3.76	4.81	7.60	9.36	10.08	10.83
	Round-Robin	3.92	4.85	7.62	9.28	9.90	10.80

Table 13: Invocation rates (%) of different models for different LLM selection settings.

Benchmark	LLM Selection / Models	DeepSeek-R1-Distill-Qwen-7B	Llama-3.1-8B-Instruct	Qwen3-8B	Qwen3-14B	Devstral-Small-2505	gpt-5-mini	DeepSeek-R1-Distill-Qwen-32B	gpt-5.2	gpt-5.2 Course Alteration
Llama-3-8B Attention Layer	MOSAIC(8 LLMs)	11.6%	7.6%	34.1%	5.1%	1.0%	5.6%	2.5%	18.3%	14.3%
	Random	10.5%	10.4%	11.5%	9.9%	11.2%	10.7%	10.6%	10.1%	15.0%
	Round-Robin	10.7%	10.7%	10.7%	10.7%	10.7%	10.7%	10.7%	10.7%	14.4%
DeepSeek-R1 MoE Layer	MOSAIC(8 LLMs)	12.4%	23.5%	8.9%	1.8%	2.7%	27.3%	1.3%	9.6%	12.4%
	Random	11.3%	10.2%	11.5%	10.0%	10.7%	11.0%	10.3%	11.1%	13.9%
	Round-Robin	10.9%	10.9%	10.9%	10.9%	10.9%	10.9%	10.9%	10.9%	12.9%
FLUX Attention Layer	MOSAIC(8 LLMs)	13.7%	14.9%	36.5%	2.3%	3.4%	1.7%	4.9%	9.2%	13.3%
	Random	11.7%	12.9%	12.7%	10.7%	10.2%	8.8%	10.7%	8.2%	14.1%
	Round-Robin	10.8%	10.8%	10.8%	10.8%	10.8%	10.8%	10.8%	10.8%	13.5%
FLUX Convolution Layer	MOSAIC(8 LLMs)	6.2%	3.7%	54.5%	11.2%	4.3%	0.5%	0.3%	5.7%	13.5%
	Random	10.7%	11.0%	11.1%	10.1%	10.5%	10.9%	10.6%	11.4%	13.7%
	Round-Robin	10.7%	10.7%	10.7%	10.7%	10.7%	10.7%	10.7%	10.7%	14.1%
Llama-4-Scout MLP Layer	MOSAIC(8 LLMs)	19.0%	7.3%	41.6%	3.1%	0.8%	4.2%	0.9%	10.6%	12.6%
	Random	11.5%	10.8%	11.0%	10.1%	10.7%	11.5%	10.2%	11.1%	13.2%
	Round-Robin	10.8%	10.8%	10.8%	10.8%	10.8%	10.8%	10.8%	10.8%	13.6%

Table 14: Compilation time and API cost reduction of MOSAIC(8 LLMs) against random / round-robin LLM selection settings.

	Llama-3-8B Attention Layer	DeepSeek-R1 MoE Layer	FLUX Attention Layer	FLUX Convolution Layer	Llama-4-Scout MLP Layer
Comp. Time ↓ (×)	1.17 / 1.18	1.21 / 1.24	1.39 / 1.45	1.49 / 1.44	1.64 / 1.62
API Cost ↓ (×)	1.05 / 1.03	1.13 / 1.15	1.29 / 1.31	1.55 / 1.53	1.38 / 1.36

J Number of Model Invocations

Table 15: Model call counts for 2-, 4-, and 8-LLM configurations of MOSAIC across five benchmarks on GPU when GPT-5.2 is the largest model.

Benchmark	MOSAIC Combination	Number of Regular Model Calls								Number of Course Alterations
		DeepSeek-R1-Distill-Qwen-7B	Llama-3.1-8B-Instruct	Qwen3-8B	Qwen3-14B	Devstral-Small-2505	gpt-5-mini	DeepSeek-R1-Distill-Qwen-32B	GPT-5.2	GPT-5.2
Llama-3-8B Attention Layer	MOSAIC(8 LLMs)	183	21	411	94	43	60	79	109	158
	MOSAIC(4 LLMs)	—	358	—	—	—	377	99	166	142
	MOSAIC(2 LLMs)	—	—	—	—	—	588	—	412	103
DeepSeek-R1 MoE Layer	MOSAIC(8 LLMs)	257	129	86	417	0	14	18	79	162
	MOSAIC(4 LLMs)	—	198	—	—	—	576	30	196	146
	MOSAIC(2 LLMs)	—	—	—	—	—	764	—	236	120
FLUX Attention Layer	MOSAIC(8 LLMs)	146	47	465	216	14	7	26	79	155
	MOSAIC(4 LLMs)	—	419	—	—	—	307	92	182	140
	MOSAIC(2 LLMs)	—	—	—	—	—	584	—	416	88
FLUX Convolution Layer	MOSAIC(8 LLMs)	316	196	130	72	3	13	112	158	124
	MOSAIC(4 LLMs)	—	350	—	—	—	392	78	180	137
	MOSAIC(2 LLMs)	—	—	—	—	—	813	—	187	142
Llama-4-Scout MLP Layer	MOSAIC(8 LLMs)	565	35	168	54	6	8	25	139	167
	MOSAIC(4 LLMs)	—	338	—	—	—	246	223	193	127
	MOSAIC(2 LLMs)	—	—	—	—	—	788	—	212	139

Table 16: Model call counts for 2-, 4-, and 8-LLM configurations of MOSAIC across five benchmarks on CPU when GPT-5.2 is the largest model.

Layer	Experiment	Number of Regular Model Calls								Number of Course Alterations
		DeepSeek-R1-Distill-Qwen-7B	Llama-3.1-8B-Instruct	Qwen3-8B	Qwen3-14B	Devstral-Small-2505	gpt-5-mini	DeepSeek-R1-Distill-Qwen-32B	GPT-5.2	GPT-5.2
Llama-3-8B Attention Layer	MOSAIC(8 LLMs)	135	89	398	59	12	65	29	213	167
	MOSAIC(4 LLMs)	—	163	—	—	—	229	365	243	128
	MOSAIC(2 LLMs)	—	—	—	—	—	776	—	224	110
DeepSeek-R1 MoE Layer	MOSAIC(8 LLMs)	142	268	102	21	31	312	15	109	141
	MOSAIC(4 LLMs)	—	346	—	—	—	181	301	172	137
	MOSAIC(2 LLMs)	—	—	—	—	—	743	—	257	125
FLUX Attention Layer	MOSAIC(8 LLMs)	158	172	421	27	39	20	57	106	154
	MOSAIC(4 LLMs)	—	77	—	—	—	536	208	179	139
	MOSAIC(2 LLMs)	—	—	—	—	—	727	—	273	114
FLUX Convolution Layer	MOSAIC(8 LLMs)	72	43	630	129	50	6	4	66	156
	MOSAIC(4 LLMs)	—	217	—	—	—	539	57	187	143
	MOSAIC(2 LLMs)	—	—	—	—	—	686	—	314	122
Llama-4-Scout MLP Layer	MOSAIC(8 LLMs)	217	84	476	35	9	48	10	121	144
	MOSAIC(4 LLMs)	—	55	—	—	—	587	128	230	136
	MOSAIC(2 LLMs)	—	—	—	—	—	698	—	302	118

Table 17: Model call counts for 2-, 4-, and 8-LLM configurations of MOSAIC across five benchmarks on CPU when Llama-3.3-70B-Instruct is the largest model.

Layer	Experiment	Number of Regular Model Calls								Number of Course Alterations
		DeepSeek-R1-Distill-Qwen-7B	Llama-3.1-8B-Instruct	Qwen3-8B	Qwen3-14B	Devstral-Small-2505	gpt-5-mini	DeepSeek-R1-Distill-Qwen-32B	Llama-3.3-70B-Instruct	Llama-3.3-70B-Instruct
Llama-3-8B Attention Layer	MOSAIC(8 LLMs)	51	54	372	108	88	92	6	229	144
	MOSAIC(4 LLMs)	—	214	—	—	—	329	122	335	94
	MOSAIC(2 LLMs)	—	—	—	—	—	495	—	505	87
DeepSeek-R1 MoE Layer	MOSAIC(8 LLMs)	98	88	213	368	20	5	2	206	139
	MOSAIC(4 LLMs)	—	416	—	—	—	159	69	356	122
	MOSAIC(2 LLMs)	—	—	—	—	—	539	—	461	91
FLUX Attention Layer	MOSAIC(8 LLMs)	116	298	114	93	37	8	6	328	125
	MOSAIC(4 LLMs)	—	230	—	—	—	379	49	342	119
	MOSAIC(2 LLMs)	—	—	—	—	—	549	—	451	93
FLUX Convolution Layer	MOSAIC(8 LLMs)	140	277	153	122	52	63	39	154	130
	MOSAIC(4 LLMs)	—	198	—	—	—	284	121	397	96
	MOSAIC(2 LLMs)	—	—	—	—	—	516	—	484	82
Llama-4-Scout MLP Layer	MOSAIC(8 LLMs)	103	172	400	176	2	2	4	141	153
	MOSAIC(4 LLMs)	—	224	—	—	—	399	109	268	129
	MOSAIC(2 LLMs)	—	—	—	—	—	429	—	571	75

K Limitations and Future Work

Richer LLM actions. MOSAIC uses LLMs to propose semantic-preserving schedule transformations, which guarantees correctness by construction and keeps each call short; both the transformation proposal and the next model choice receive credit only through downstream reward. A natural extension is to let LLMs act as verified schedule editors that emit richer TVM schedule programs or low-level schedule edits directly, with compiler-side checks before a candidate enters the shared search tree.

Scope of the evaluation. We evaluate five kernels and end-to-end Llama-3-8B on one consumer GPU and one CPU. Extending the evaluation to data-center GPUs, other accelerators, and compilers other than TVM is future work.

Model size as a cost proxy. The size term ϕ_{small} uses parameter count as a proxy for cost. Closed-model sizes are not disclosed, so we assume 300B for GPT-5.2 and 20B for gpt-5-mini; the configuration with the open-weight Llama-3.3-70B-Instruct as the largest model (Appendix E) reduces this assumption to gpt-5-mini.

Usage of External APIs. Our framework currently relies on external APIs (OpenAI and Nscale) to access LLMs. This dependency introduces challenges around reproducibility, cost, and long-term availability—particularly for closed-source models. Although MOSAIC reduces reliance on the largest model and shows strong results with open-weight models, practical deployment may require local hosting. Our method assumes a sufficient context window to encode hierarchical information (e.g., parent and grandparent schedules). This may not generalize effectively to models with shorter context lengths.