# MARCONI: PREFIX CACHING FOR THE ERA OF HYBRID LLMS

**Rui Pan** [1 *] **Zhuang Wang** [2] **Zhen Jia** [2] **Can Karakus** [2] **Luca Zancato** [2] **Tri Dao** [1] **Yida Wang** [2] **Ravi Netravali** [1]

## ABSTRACT

Hybrid models that combine the language modeling capabilities of Attention layers with the efficiency of Recurrent layers (e.g., State Space Models) have gained traction in practically supporting long contexts in Large Language Model serving. Yet, the unique properties of these models complicate the usage of complementary efficiency optimizations such as prefix caching that skip redundant computations across requests. Most notably, their use of in-place state updates for recurrent layers precludes rolling back cache entries for partial sequence overlaps, and instead mandates only exact-match cache hits; the effect is a deluge of (large) cache entries per sequence, most of which yield minimal reuse opportunities. We present Marconi, the first system that supports efficient prefix caching with Hybrid LLMs. Key to Marconi are its novel admission and eviction policies that more judiciously assess potential cache entries based not only on recency, but also on (1) forecasts of their reuse likelihood across a taxonomy of different hit scenarios, and (2) the compute savings that hits deliver relative to memory footprints. Across diverse workloads and Hybrid models, Marconi achieves up to 34.4× higher token hit rates (71.1% or 617 ms lower TTFT) compared to state-of-the-art prefix caching systems.

## 1 INTRODUCTION

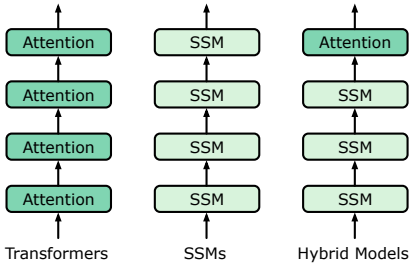
The emergence of large language models (LLMs) has empowered many applications including chatbots (cha, 2022), AI-powered search (per, 2023), coding assistants (git, 2022), and more. Owing to increasing workload complexity and the evolution of inference-time enhancements such as few-shot prompting (Brown, 2020) and chain-of-thoughts (Wei et al., 2022), recent years have witnessed a push towards *longer context windows* during serving. Indeed, the accuracy improvements from multi-step reasoning (Khattab et al., 2023; Yao et al., 2022), detailed prompt templates (Liu et al., 2023), and increased samples in few-shot prompting (Brown, 2020) all require expanded context sizes.

To keep pace with these trends, traditional foundation models like Transformers have been extended to support longer context windows, e.g., Gemini 1.5 (Team et al., 2023) and Claude 3 (cla, 2024b) both have 1M window size. However, their intrinsic reliance on the Attention mechanism harms long-context serving efficiency, particularly due to its quadratic compute complexity and large memory footprint for housing KV cache states (Wu et al., 2024; Lee et al., 2024). As a result, new recurrent and subquadratic model architectures like State Space Models (SSMs) (Gu & Dao, 2023; Dao & Gu, 2024) have emerged to offer lower compute and memory costs for long-context serving (Fig. 1c), albeit with mild accuracy degradations (Waleffe et al., 2024). These subquadratic layers are commonly mixed with several full Attention layers to produce *Hybrid LLMs* (Fig. 1a) that lower serving costs while maintaining Attention's superior recall and in-context learning capabilities (Waleffe et al., 2024; Lieber et al., 2024; Team et al., 2024; Glorioso et al., 2024).

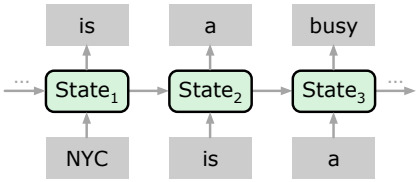
Although Hybrid LLMs improve per-request efficiency, they complicate the cross-request efficiency wins that prior efforts have shown to be especially effective in long-context scenarios, i.e., *prefix caching* that reuses model states for common prefixes across requests, thereby skipping computation without harming accuracy (Kwon et al., 2023; Zheng et al., 2023b; Gao et al., 2024; Abhyankar et al.). The primary challenge is that SSM model states are updated in place (Fig. 1b), so states at the end of a sequence cannot be rolled back to represent a prefix of the sequence. This presents a dilemma for serving systems. On one hand, maximizing reuse opportunities for future (arbitrary) workloads mandates caching fine-grained state checkpoints at regular intervals, e.g., every 256 tokens. On the other hand, increased checkpointing frequency inflates the number of cache entries generated per sequence, each of which is *large* (due to the sheer size of SSM states) but most of which present limited reuse opportunities, i.e., *sparsely-hit*. The net effect is cache thrashing with low-utility entries (Fig. 2), and a poor memory-vs-compute savings tradeoff (§3).

[*]Work done during internship at AWS. [1]Princeton University [2]AWS. Correspondence to: Rui Pan <ruipan@princeton.edu>.

*Proceedings of the $8^{th}$ MLSys Conference*, Santa Clara, CA, USA, 2025. Copyright 2025 by the author(s).




(a) Different model architectures. Hybrid models usually mix in one Attention layer for every 6-10 SSM layers.




(b) SSMs update their states recurrently and in place.

| | Attention | SSM |
|---|---|---|
| Computational Complexity | $O(L^2)$ | $O(L)$ |
| Inference-Time Memory | $O(L)$ | $O(1)$ |

(c) Prefill complexity comparison between quadratic Attention layers and subquadratic SSM layers.

*Figure 1.* Overview of Hybrid models.

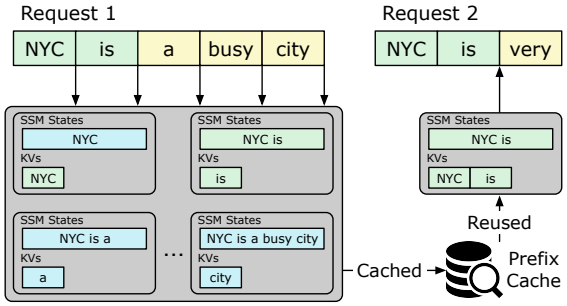



*Figure 2.* Prefix caching reuses model states of common prefixes (green) across requests, accelerating inference. Fine-grained checkpointing results in many sparsely-hit entries (blue).

We present ***Marconi***,[1] the first prefix caching system designed to support Hybrid LLMs. For a given sequence, Marconi simultaneously manages cache entries for both SSM states and KVs, ensuring that *all* of the preceding sequence states for each cached entry are also present to support reuse. Moreover, Marconi eschews traditional caching logic based on recency (Kwon et al., 2023; Zheng et al., 2023b; Srivatsa et al., 2024), and instead introduces novel admission and eviction strategies that value potential cache entries with awareness of the aforementioned SSM overheads; we describe these in turn.

To cope with the large size and "all or nothing" nature of SSM cache entries (i.e., hits only arise on exact sequence matches, unlike with KVs), Marconi adopts a judicious admission strategy, whereby only SSM candidates with high reuse likelihood are accepted. Our driving insight is that, despite lacking knowledge of upcoming requests, the reuse potential of each SSM state can be sufficiently estimated when assessed against the taxonomy of potential prefix reuse scenarios. Specifically, token redundancy arises across requests as either (a) purely-input shared prefixes, e.g., system prompts, or (b) a combination of input and output token sharing, e.g., conversation history. The former case typically arises across many requests, enabling Marconi to vet each candidate against previously-observed sequences. In contrast, for the latter case, Marconi assigns value only to SSM states that represent the last decoded token (which conversations typically resume from, as opposed to intermediate branching). We efficiently encapsulate this bookkeeping in a radix tree that highlights sequence overlap, and introduce a speculative insertion step prior to each prefill to determine the potential reuse opportunties for the upcoming sequence (and the resultant checkpointing required).

For eviction, our main observation is that, with Hybrid LLMs, when assigning value to the cache entry for a given sequence, we have to account for both KVs and SSM states which exhibit different tradeoffs between memory and compute savings. Specifically, whereas the size of KVs' entries for a sequence is (linearly) proportional to the compute savings from reusing that sequence, SSM state sizes are fixed and unrelated to sequence length and compute savings. To enable more holistic management of cache entries, Marconi introduces a new FLOP-aware eviction policy that assesses candidates for eviction based not only on recency/popularity, but also the potential compute savings they deliver (normalized against the space they consume in the cache). This inherently trades the hit rate of shorter sequences for longer ones – an especially desirable tradeoff given the superior efficiency of Hybrid models over Transformers.

We evaluated Marconi on a wide range of workloads, request arrival patterns, cache sizes, and Hybrid model architectures. Overall, we find that Marconi improves cache hit rates by an average of 4.5-34.4× compared to state-of-the-art caching systems (e.g., vLLM (Kwon et al., 2023), SGLang (Zheng et al., 2023b)) that are extended to support Hybrid models. The win in token hit rate translates to latency savings of 36.1-71.1% (103.3-617.0ms) for P95 TTFT. Microbenchmarks show that Marconi performs better in scenarios with longer contexts, higher ratios of SSM layers, and larger SSM state dimensions — trends that align with recent model developments. Marconi is open sourced at `https://github.com/ruipeterpan/marconi`.

[1] "Marconi plays the mamba, listen to the radio, don't you remember?" – Lyrics of *We Built This City*, song by Starship

## 2 BACKGROUND

In this section, we provide background on the efficient recurrent architectures (exemplified by SSMs), Hybrid models that contain these alternative architectures, and prefix caching for efficient LLM inference.

### 2.1 State Space Models and Hybrid Models

Token generation in LLM inference involves two main phases. First, the prefill phase processes an input sequence, generating the model's internal states (e.g., KV cache/KVs in Attention layers) for each layer of an LLM, and outputs the first new token. The decoding phase then utilizes the internal states to perform autoregressive token generation.

During prefill, generating the first token depends on all previous tokens. In Transformers, the self-attention mechanism calculates how each token in the sequence "attends" to every other token. Consequently, the Attention mechanism incurs quadratic computational complexity (Dao et al., 2022), which quickly bottlenecks GPU compute as sequence lengths scale (Agrawal et al., 2024). Furthermore, the size of the KVs in Attention layers grows linearly with sequence length, resulting in a large inference-time memory requirement (Wu et al., 2024; Lee et al., 2024).

State space models (SSMs) and more generally linear RNNs and linear attention, such as Mamba (Gu & Dao, 2023; Dao & Gu, 2024), address these inefficiencies by selectively "compressing" previous context into a recurrent and compact representation. The recurrent representation is used alongside the previous token to update the recurrent representation in place, as shown in Fig. 1b. Because the SSM states maintain a constant size, memory consumption remains fixed regardless of sequence length, and the computational complexity scales linearly, rather than quadratically, with the sequence length (Fig. 1c). Although pure SSM models outperform Transformers on many NLP tasks, they lag behind on certain workloads that require strong recall or in-context learning capabilities (Waleffe et al., 2024; nee, 2024). To balance inference efficiency and model capability, SSM-Attention *Hybrid models* (Fig. 1a) have been proposed. These models blend quadratic Attention and subquadratic SSM layers, typically interleaved in a specific ratio (commonly 1 Attention layer for every 6-10 SSM layers (Waleffe et al., 2024; Lieber et al., 2024; Glorioso et al., 2024)). When compared to Transformers of equivalent scale trained on the same datasets, Hybrid models demonstrate superior performance across a wide range of tasks while preserving most of the efficiency advantages of SSM layers (up to 8× faster) (Waleffe et al., 2024). Many Hybrid models have been productionized (car, 2024; Lieber et al., 2024; Team et al., 2024; Glorioso et al., 2024; zam, 2024a;b; Waleffe et al., 2024; Ren et al., 2024), with the largest being Jamba 1.5 at 398B parameters (Team et al., 2024).

### 2.2 Prefix Caching

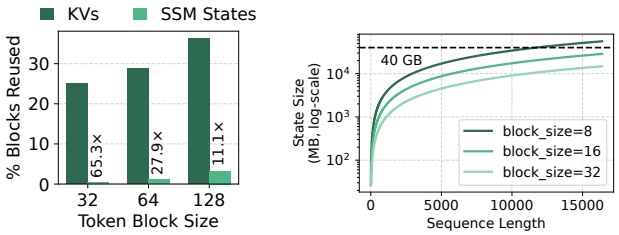
Shared prefixes across requests, including input tokens and sometimes output tokens, are common across many LLM applications. For example, question-answering workloads often share a detailed system prompt combined with few-shot examples that provide instructions and demonstrations (Yao et al., 2022). Similarly, coding agents interact with the environment in multiple rounds, where each new request consists of a trajectory of past environment interactions and new observations and actions (Yang et al., 2024a; Wang et al., 2024). Redoing the prefill of these shared prefixes for all requests leads to many redundant computations, hurting both throughput and latency. Prefix caching mitigates this by caching and reusing the model's internal states that represent the common prefixes (Fig. 2), achieving a lower time to first token (TTFT) latency, lower tail time per token (TPT) latency[2], and higher prefill throughput (measured in tokens/s).

Many research and production systems have been proposed to reap the benefits of prefix caching in Transformer inference (Kwon et al., 2023; Zheng et al., 2023b; Gao et al., 2024; Srivatsa et al., 2024; Abhyankar et al.; cla, 2024a; cha, 2024; Qin et al., 2024; ope, 2024). On startup, these systems provision blocks of GPU/CPU memory for caching the prefix states. Before prefilling a new sequence, the inference engine looks up the prefix cache for the longest matching prefix. Upon a cache hit, the corresponding prefix is fetched before prefilling. After decoding the final token of the sequence, to favor recency, the system admits the model states of all tokens of the new sequence into the cache. To reduce memory fragmentation, the KVs are usually partitioned into fixed-sized token blocks in the prefix cache, each housing the KVs of $x$ tokens where $x$ is the block size (Kwon et al., 2023), and existing token blocks are evicted to make room if the cache is full. Because KVs have a sequence dimension, managing and evicting model states is efficient and flexible. E.g., if we have the KVs of a sequence of tokens $1...q$ and want to retain the KVs of the prefix $1...p$ ($p < q$), tensor slicing can be performed on the sequence dimension of the full KVs.

## 3 CHALLENGES OF PREFIX CACHING WITH HYBRID LLMS

Compared to Attention, SSM's properties greatly benefit its per-request computational complexity and memory consumption. However, the very properties that make SSMs more efficient also complicate prefix caching, an important optimization for cross-request efficiency wins. The key

[2]Even though prefix caching is a prefill-only optimization, a lower prefill latency also reduces the tail TPT for high-throughput LLM inference engines (Agrawal et al., 2024).




(a) Token block reuse rate comparison between KVs and SSM states.

(b) Total size of a Hybrid model's cache entries as sequence length scales.

*Figure 3.* Fine-grained caching of token blocks results in many SSM states being cached. This creates sparsely-hit entries in which many SSM states are never reused (a), underutilizing the precious cache capacity. Worse, this creates a huge memory usage even for a single sequence of a 7B model (b), overwhelming and thrashing the cache.

reason lies in how SSM updates its internal states: SSMs recurrently and efficiently compress the history of tokens into a compact state and update the recurrent state in place by overwriting the previous states, keeping the memory usage constant. Further, while SSM states are generally smaller than the KVs of *whole sequences* in Attention, they aim to capture the same cross-token information as KVs do and thus are typically 10-100x larger than a *single token*'s KVs (Appendix A). In summary, SSM states exhibit the following key properties:

**SSM State Properties**

1. SSM states are constant-sized regardless of how many tokens they represent.
2. SSM states are updated in place, so a sequence's states cannot be rolled back to represent its prefixes.
3. SSM states are orders of magnitude larger than the KVs of a single token.

The core challenge of prefix caching in Hybrid LLM inference is that SSM states exhibit "all or nothing" reusability: To realize prefix reusing, we need all prefix tokens' KVs for each Attention layer and one SSM state that exactly matches all prefix tokens for each SSM layer. However, if future requests only use a prefix of a sequence, such as tokens $1...p$ from a sequence $1...q$ ($p < q$), the SSM layers cannot reuse states that represent $1...q$ (property ❷) and neither can the KVs be reused by Attention layers, as prefix reusing is bottlenecked by the layer with the least reusing opportunities. To maximize reuse opportunities for future requests that might reuse arbitrary prefixes of the most recent request, fine-grained checkpointing (every $x$ tokens) of many previously overwritten SSM states is required. This creates an equal-sized token block for each interval, containing KVs for $x$ tokens and SSM states that represent all prior tokens (Fig. 2). However, this elicits two challenges for existing prefix caching systems. To demonstrate this, we analyze one of the experiments in §5. No existing systems support prefix caching for Hybrid models, so we extend vLLM (Kwon et al., 2023) using its caching policy to support Hybrid models.

**Cache underutilization.** While reusing KVs requires accessing all prior tokens' KVs, reusing SSM states only needs the last token block. This results in sparsely-hit cache entries on a sequence level, with many low-utility SSM states never accessed after admission. Fig. 3a shows that with a block size of 32, 25.0% of the token blocks' KVs are reused by future requests, but a mere 0.4% of SSM states are reused, a 65.3× difference. Using larger blocks mitigates both issues by reducing the checkpointing granularity and the number of SSM states checkpointed. However, the issues persist (3.3% reuse rate for block size 128 which exceeds what vLLM natively supports), and larger block sizes lead to internal memory fragmentation of KVs within token blocks, impacting memory utilization (Kwon et al., 2023).

**High memory usage.** Worse, fine-grained checkpointing quickly leads to an excessive memory usage due to the SSM states' size. For example, with block size 16, the state size of an SSM layer is 4×[3] larger than the KVs of an Attention layer in the token block because of property ❸. Since there are many more SSM layers than Attention layers in Hybrid models for efficiency (Waleffe et al., 2024), their states quickly overwhelm the GPU HBM (and possibly CPU RAM): for a 7B model, a single sequence of 10K tokens consumes 17.4 GB (Fig. 3b), 3.3× bigger than a Transformer of the same size. Even if sequences are short and the states are evicted soon after a request finishes inference, they must still be admitted into the prefix cache, potentially requiring eviction of entries with higher utility, leading to frequent cache thrashing and low cache hit rate.

In conclusion, existing prefix caching systems face a dilemma in Hybrid LLM inference: maximizing reuse opportunities mandates fine-grained state checkpointing but doing so creates large yet sparsely-hit entries that overwhelm and thrash the limited cache capacity. Therefore, a judicious cache management scheme is needed to better reap the benefits of prefix caching in Hybrid LLM inference.

## 4 Design and Implementation

Marconi is the first prefix caching system designed to accommodate the unique characteristics of Hybrid models. It is designed to support models with arbitrary layer compositions, including Hybrid models, pure Transformers, and

[3] `d_state` /(2· `block_size`). The 2 stands for the key and value tensors in KVs. See Appendix A for more details.

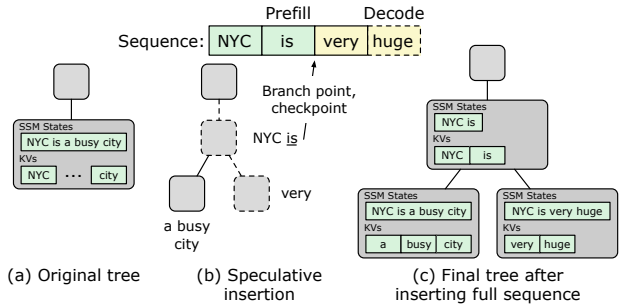



*Figure 4.* Marconi performs a speculative insertion to check if inserting the prefill segment of a sequence results in an intermediate node. If so, the SSM states at the branch point are checkpointed. States at the last decoded token are checkpointed in any case. For ease of visualization, we associate model states with nodes rather than edges.

pure SSMs. Its primary goal is maximizing the cache utilization to reduce redundant computation and minimize TTFT latency. To improve cache utilization and prevent the large SSM states from thrashing the cache, Marconi judiciously admits SSM states (§4.1), only accepting states with a high reuse likelihood based on a taxonomy of potential prefix reuse scenarios. The bookkeeping of requests is realized via a radix tree that represents past request overlap. Once admitted, as states of all layers need to work in conjunction and represent the same prefix tokens to realize prefix caching, Marconi opts to manage different types of model states holistically in the same tree (Fig. 4), where each node contains the SSM states and KVs of a sequence, rather than disaggregating the cache space for different types of layer states. For eviction (§4.2), to account for the different tradeoffs between memory and compute savings of different layers' states, Marconi introduces a FLOP-aware eviction policy that balances recency and potential compute savings delivered by reusing entries. We describe the implementation details in §4.3.

### 4.1 Cache Admission

Marconi aims to cache the states with high reuse likelihood during admission. Although prefix reuse patterns of future requests cannot be perfectly predicted, our key insight is that the reuse potential can be sufficiently estimated through a taxonomy of potential prefix reusing scenarios. Through extensive analysis of prefix reusing patterns in various real-world datasets and request traces (Qin et al., 2024; cha, 2024; Zheng et al., 2023b) that represent traffic of both dedicated inference deployments (Jimenez et al., 2023) and public-facing APIs (cha, 2022; sha, 2024), we classify the token composition of all reused prefixes into two types:

1. **Purely input:** The prefix is a part of the input sequence from a previous input, such as system prompts (cha, 2022), instructions (Yao et al., 2022), few-shot examples (Hendrycks et al., 2020), self-consistency (Wang et al., 2022), long-document QA (Li et al., 2023), etc.
2. **Input and output:** The prefix consists of previous input and output tokens, such as conversation history in chatbots (Gao et al., 2024; cha, 2024; 2022), past environment interactions for LLM agents (Yang et al., 2024a; Yao et al., 2022), etc.

This taxonomy allows Marconi to devise different mechanisms for identifying and caching the corresponding states. For the purely-input case, as the same prefix is usually shared across many requests, Marconi can observe and compare previous requests to identify these hot common prefixes. For the input-and-output case, Marconi only values SSM states that represent the last decoded token between conversation rounds, which conversations typically append to, as opposed to branch off from.

Leveraging the above insights, to estimate the reuse likelihood, Marconi uses a radix tree to bookkeep the request history, identify common prefixes, and map sequences to their states. A radix tree is a space-efficient prefix tree with edges labeled by sequences of varying lengths. Within the tree, each edge is associated with the KVs of tokens it represents and the SSM states that represent all tokens prior to the last token in the edge (Fig. 4c). Nodes represent branch-off points in sequences: those with multiple children represent prefixes that are "purely input", whereas nodes with $\leq 1$ child may represent "input and output" prefixes of future requests. Because nodes represent high-utility states, Marconi caches states judiciously by only admitting SSM states represented by the last token of edges. For input-and-output cases, Marconi simply checkpoints the state after the last decoding step. Because the last token's KVs include all prior tokens, all KVs of the whole sequence are still effectively cached, which is the same as existing systems. To identify "purely input" prefixes, prior to prefilling each sequence, Marconi employs a speculative insertion of the input tokens to see if new intermediate nodes will be created (Fig. 4). If so, Marconi caches the prefix's states during prefill.

**Obtaining states during prefill.** After prefilling KVs in Attention, they can be partitioned and trimmed to represent subsequences. In contrast, SSM states cannot be rolled back to represent a prefix, so Marconi needs a new mechanism previously unnecessary for Transformers inference. Marconi supports two main methods for checkpointing SSM states during prefill. Some SSMs (Dao & Gu, 2024; Sun et al., 2023; Yang et al., 2023) perform chunked state passing during prefill, where the input sequence is split into chunks to compute intra-chunk states. For these models, we simply materialize and cache the state of the second-to-last chunk in the prefix. For example, when prefilling a sequence of 100 tokens using chunk size 32, if we need to cache the

state at token 80, we can checkpoint the state at token 64. This approach may miss some prefix caching opportunities within a chunk but introduces minimal runtime overhead. Optionally, custom kernels can be developed to quickly roll the state forward by a few tokens to reach the exact location. For models that don't support chunked state passing (Gu & Dao, 2023; Lieber et al., 2024), Marconi performs a two-pass prefill to get the precise state at the prefix. For example, the first pass prefills the first 80 tokens to generate the prefix state, while the second pass starts from the prefix state and prefills the remaining 20 tokens.

**Tradeoffs.** Compared to fine-grained checkpointing, Marconi's judicious admission has slight drawbacks, but its tremendous benefits compensate for the drawbacks. Because the states of the last decoded token are immediately cached for all sequences, "input and output" prefixes can be reused instantaneously. In contrast, "purely input" prefixes only benefit from reusing starting from the third occurrence of the prefix, as Marconi uses the second occurrence to identify the prefix and checkpoint its states. While this approach sacrifices the benefit of reusing a "purely input" prefix on its second occurrence, these prefixes are typically shared across many requests. As a result, missing savings on a single request has a negligible impact on overall savings. On the other hand, judicious admission reduces coverage and slightly limits the potential reusability of arbitrary prefixes, as only up to two SSM states are admitted per sequence. However, due to the huge number of low-utility SSM states rejected from admission by Marconi, this altruistic approach significantly reduces the size and improves the utility of cached Hybrid model states, enhancing overall cache utilization.

**Comparison with SGLang.** SGLang (Zheng et al., 2023b) is an existing prefix caching system for Transformers that also leverages a radix tree for mapping sequences to tokens and their KVs. Different from SGLang, Marconi uses the past requests in the radix tree to determine which SSM states to cache by performing speculative insertions before prefilling an upcoming sequence. Further, Marconi's philosophy of judicious state admission fundamentally differs from SGLang, which admits the states of all tokens during admission and is no different from other prefix caching systems like vLLM (Kwon et al., 2023). We show how these differences yield superior cache efficiency in §5. Our system doesn't invent new data structures for KV cache management, and instead builds on the rich prior work in this space that also aims to manage prefixes' model states efficiently. Our main contribution lies in being the first system to redesign caching policies and their interactions with these data structures to practically address the unique properties of emerging Hybrid models.

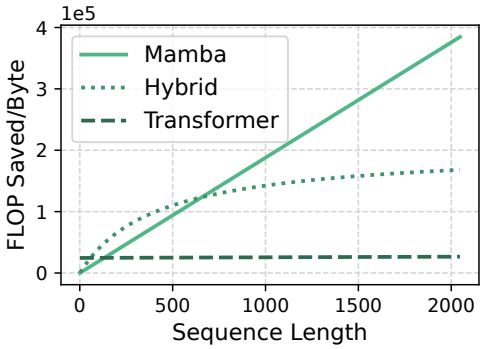



*Figure 5.* FLOP efficiency of the model states of different 7B models as the sequence length scales. The more SSM layers in the model, the steeper the increase in FLOP efficiency.

### 4.2 Cache Eviction

For eviction, our main observation is that KVs and SSMs in Hybrid model states exhibit different tradeoffs between memory and compute savings. Specifically, whereas the size of KVs for a sequence is linearly proportional to the sequence length and (approximately) the compute savings from reusing that sequence, SSM state sizes are fixed regardless of sequence length and compute savings. To quantify this difference, we propose a new metric, **FLOP efficiency**, to measure the compute savings (measured in the number of floating operations) per unit of memory achieved by reusing a prefix cache entry:

$$flop_efficiency = \frac{Total\ FLOPs\ across\ layers}{Memory\ consumption\ of\ all\ states} \tag{1}$$

Here, $Total\ FLOPs\ across\ layers$ denotes the sum of redundant compute across different types of layers (i.e., Attention, SSM, and MLP) circumvented by reusing the prefix entry, and $Memory\ consumption\ of\ all\ states$ denotes the total size of all stateful layers' states (i.e., Attention and SSM). More details on FLOP efficiency are in Appendix A. Fig. 5 compares the FLOP efficiency of model states in three different 7B models: Transformers (Attention-only), Mamba (SSM-only), and Hybrid (with an Attention:SSM ratio of 1:6). The increase of FLOP efficiency as sequence length scales is steeper for models with a higher ratio of SSM layers as larger portions of the model states become more FLOP efficient.

Traditional prefix caching systems designed for Transformers don't need to consider FLOP efficiency because KVs' FLOP efficiency is near-constant, and most systems only use recency for eviction. However, ignoring it in Hybrid LLM inference risks evicting states with higher FLOP efficiency but do not have the best recency. To enable more holistic management of cache entries, Marconi introduces a **FLOP-aware eviction policy** that assesses candidates for

eviction based not only on recency but also the potential compute savings they deliver (normalized against the space they consume in the cache). In addition to recency, Marconi accounts for the FLOP efficiency by computing a utility score $S$ that represents the utility for each radix node $n$:

$$S(n) = recency(n) + \alpha \cdot flop_efficiency(n) \quad (2)$$

This metric favors cache entries with higher recency, save more compute, and take less memory. Both the `recency` and `flop_efficiency` scores are normalized to the range $(0, 1)$ by comparing all nodes' last-accessed timestamps and FLOP saved/byte in the radix tree. During eviction, Marconi iteratively removes nodes with the lowest utility score until there is enough space to accommodate the new request's states. As such, child nodes' FLOP savings are calculated relative to parents' savings.

**Managing the balance.** Marconi manages the balance between favoring recency and FLOP efficiency by tuning $\alpha$. A higher $\alpha$ emphasizes FLOP efficiency, while setting $\alpha$ to 0 falls back to LRU. Marconi manages the balance by observing the workload and retrospectively setting the best configuration. On startup, Marconi sets $\alpha$ to 0 until the first eviction. Afterward, Marconi takes a snapshot of the radix tree and enters a bootstrap period, continuing to use LRU while bookkeeping token-level information of requests during the bootstrapping phase. The bootstrap period includes $5 - 15\times$ the number of requests seen before the first eviction, capturing a representative workload sample. Once sufficient requests have been observed, Marconi asynchronously launches a grid search over possible $\alpha$ values by replaying the bootstrap requests. This grid search is parallelized across CPU cores, significantly speeding up the tuning process, typically taking just a few seconds, often shorter than the time required to prefill and decode a single request. After the grid search, Marconi adopts the $\alpha$ value that maximizes the hit rate.

**Comparisons with existing size-based eviction algorithms.** Cost-aware cache eviction for objects with variable sizes is a well-studied problem (e.g., GDSF (Cherkasova, 1998)). The KV size in Attention layers scales with sequence length, serving as a proxy for compute savings from cache hits, whereas SSM states are fixed-sized irrespective of sequence length or compute savings. Thus, size fails as a proxy in Hybrid LLM inference, where longer sequences (with greater compute savings) are represented by equally sized SSM states. We note that our system and techniques are complementary to such prior schemes; just as we combine FLOP efficiency with recency, our focus is to augment the estimators for cache entry importance used in prior work with a factor tailored specifically for Hybrid model reuse benefits.

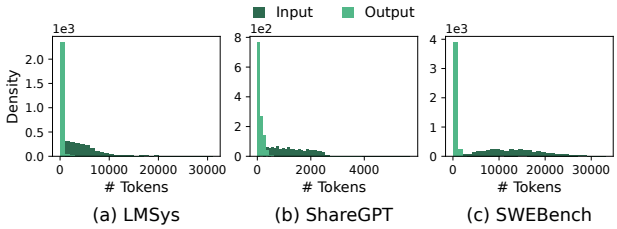



*Figure 6.* Input/output sequence length distributions per workload.

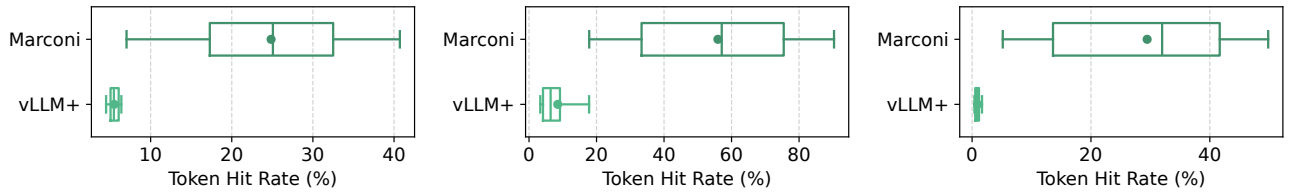
### 4.3 Implementation details

Alongside the custom admission and eviction policies, we made the following changes to accommodate Hybrid model states: (1) During eviction, all nodes with $\leq 1$ children are considered for eviction, not just leaf nodes. The reasoning is that nodes with multiple children represent the common prefixes shared by multiple requests and should not be evicted (unless they become stale, at which point all their children will be evicted first and the ex-parent nodes will become leafless nodes themselves, which make them subject for eviction), whereas intermediate nodes with a single child are unlikely to be reused more than once and still incur a memory cost for their SSM states. When an intermediate node is evicted, its SSM states are released, and its KVs are absorbed by its child node. (2) When a cache hit occurs, only the accessed node's timestamp is updated, unlike in existing systems where timestamps for all ancestor nodes are updated. In Marconi, previous SSM states are not reused (Fig. 4c), and although ancestors' KVs are accessed, their KVs will be subsumed by child nodes if evicted. Thus, not updating ancestors' timestamps doesn't affect recency tracking.

## 5 Evaluation

We evaluated the performance of Marconi under various workloads with different datasets, request arrival patterns, cache sizes, and model architectures. Our key findings are:

1. Overall, Marconi improves token hit rate by an average of 4.5-34.4$\times$ compared to fine-grained checkpointing, reducing the P95 TTFT by up to 71.1% (617.0 ms) compared to baseline prefix caching systems.
2. Compared to LRU, Marconi's FLOP-aware eviction improves the token hit rate by 19.0-219.7%. It achieves higher token hit rates and FLOP savings by trading off hit rate of shorter sequences to boost hit rate for longer sequences, a desirable tradeoff given the efficiency of Hybrid models over Transformers.
3. Marconi performs better when sequences are long, the SSM layer ratio is high, and the SSM state dimension is large – trends that align with recent model developments.




(a) LMSys (b) ShareGPT (c) SWEBench

*Figure 7.* Comparison with vLLM+. With judicious cache admission, Marconi utilizes the limited cache space to retain states with higher utility, improving the token hit rate significantly over vLLM+.

### 5.1 Methodology

**Baselines.** We compare Marconi with the following baselines. Note that neither vLLM nor SGLang natively supports prefix caching for Hybrid/SSM models, so we have extended both to support Hybrid models favorably.

- **Vanilla inference**: This baseline prefills all requests without doing prefix caching.
- **vLLM+** (Kwon et al., 2023): This baseline performs fine-grained checkpointing and caches a state for every token block. We use a token block size of 32, the largest size that vLLM supports (vll, 2024), which favors vLLM+ by minimizing the number of low-utility SSM states admitted (Fig. 3a) while reducing memory fragmentation of KVs within a token block.
- **SGLang+** (Zheng et al., 2023b): While SGLang also uses a radix tree, it doesn't judiciously checkpoint states during admission. We enhance it by applying the same judicious admission policy as Marconi; however, the eviction policy remains LRU, which does not account for FLOP efficiency.

**Metrics.** The main metric we evaluate is token hit rate (%), which represents the effectiveness of the cache and approximates the total compute saved (FLOP) well. We define token hit rate as the ratio of the number of tokens that skipped prefill over the total number of input tokens. We also evaluate different percentiles (P5, P50, and P95) of time to first token (TTFT, ms). FLOP saved is a reasonable proxy for compute and latency savings because prefill is easily compute-bottlenecked. We do not evaluate downstream metrics (e.g., F1 score) as prefix reusing is exact and does not change the LLM output.

**Workloads.** In our main results, we evaluate Marconi on two multi-turn conversational datasets: LMSys (Zheng et al., 2023a) and ShareGPT (sha, 2024) with different sequence lengths distributions. The LLM output sequences in LMSys are relatively long, often reaching thousands of tokens, whereas the LLM output sequences in ShareGPT are succinct and often take tens or hundreds of tokens. We also include an agentic workload: SWE-Agent (Yang et al., 2024a) on SWE-Bench (Jimenez et al., 2023), a benchmark for evaluating LLM agents on real-world software issues collected from GitHub. We plot the sequence length distribution in Fig. 6. These datasets contain multiple chat sessions, each with multiple rounds of requests. We vary the inter-session arrival time and inter-request arrival time to account for environment response time (e.g., human typing, interactions with the coding IDE) and queuing delay in the inference engine.

**Models.** We use a 7B Hybrid model with {4,24,28} {Attention,SSM,MLP} layers in our main results. For insights into our wins on TTFT, we use Jamba-1.5-Mini, an Attention-SSM Hybrid model with 12B active/52B total parameters, and serve it with state dimension 128 using the vLLM implementation on four A100-40GB GPUs. We use FP16 precision in all experiments. The results apply to models of different sizes because the size of the prefix cache needs to be scaled accordingly.

**Setup.** Experiments were run on a p4d.24xlarge AWS instance with eight A100-40GB GPUs, 96 Intel(R) Xeon(R) Platinum 8275CL CPUs, and 1152GB of DDR4 RAM.

### 5.2 End-to-End Results

Fig. 7 shows the token hit rate of vLLM+ and Marconi across different traces. The boxes show different quartiles of the data while the whiskers on the box plots show P5 and P95, allowing us to disregard extreme data and concentrate on the typical cases. With its judicious admission strategy, Marconi avoids wasteful caching decisions and optimizes the cache utilization, improving the hit rate by an average of 4.5×, 7.3×, and 34.4× for LMSys, ShareGPT, and SWEBench.

Fig. 8 compares Marconi with SGLang+ to highlight the benefits of FLOP-aware eviction over LRU. The win is the most significant on SWEBench, with a P95 win of 219.7%. The improvements on LMSys and ShareGPT are less pronounced, reaching a P95 win of 45.6% and 19.0%. This can be attributed to the difference in workload characteristics: as shown in Fig. 6, SWEBench has the widest input sequence length distribution, ranging from hundreds of tokens to tens of thousands of tokens. LMSys has a narrower distribution, with most sequences under 10K tokens, while ShareGPT

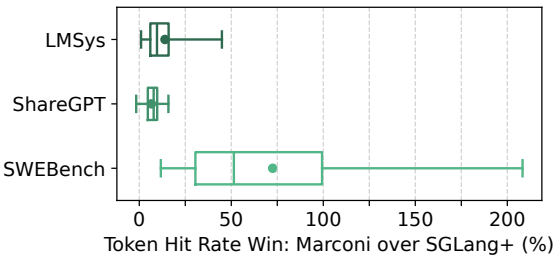



*Figure 8.* Comparison with SGLang+, which uses LRU as its eviction policy. Marconi balances recency and FLOPs efficiency, improving the token hit rate significantly, especially for workloads with longer context.

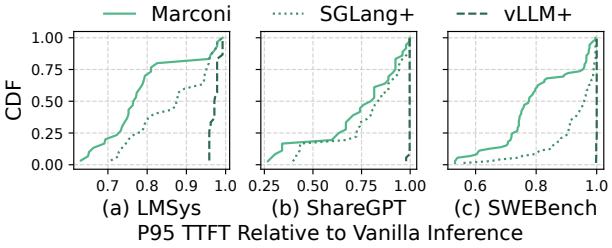



*Figure 9.* Distribution of P95 TTFT relative to no prefix caching.

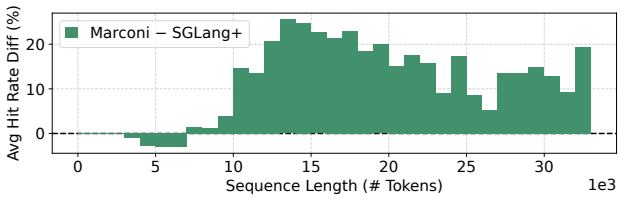
predominantly features sequences under 2K tokens. The longer the sequences, the more critical FLOP efficiency becomes in eviction decisions. For workloads dominated by shorter sequences, suboptimal eviction choices in terms of FLOP efficiency have a smaller impact on Marconi's performance gains.

Fig. 9 shows the distribution of P95 TTFT relative to no prefix caching across different traces. Marconi's token hit rate translates to TTFT reduction, reducing the P95 TTFT by up to 36.9%, 73.2%, and 46.8% (281.4 ms, 106.3 ms, and 617.0 ms) compared to vanilla inference without prefix caching. Compared to vLLM+, Marconi delivers up to 36.1%, 71.1%, and 46.8% (275.4 ms, 103.3 ms, and 617.0 ms) larger P95 TTFT reductions; these numbers are 17.2%, 12.8%, and 24.7% (131.1 ms, 18.5 ms, and 325.7 ms) when compared to SGLang+.

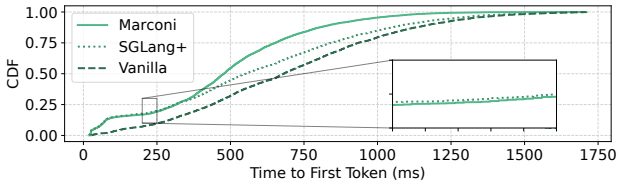
### 5.3 Fine-Grained Analysis of FLOP-Aware Eviction

To understand the performance improvement of FLOP-aware eviction over LRU, we compare the caching decisions of Marconi with SGLang+ using a request arrival trace from SWEBench. On this trace, SGLang+ achieves a 16.4% overall token hit rate, while Marconi achieves a significantly higher hit rate of 32.7%, an improvement of 99.4%.

In Fig. 10a, we categorize the requests by sequence length and plot the difference in average hit rate between SGLang+ and Marconi. Marconi shows a lower hit rate (up to -3.0%) for sequences with <7K tokens, while for sequences




(a) Difference between the average token hit rate of requests in Marconi and SGLang+, binned by the input sequence length.




(b) TTFT distribution of all requests in the trace.

*Figure 10.* Fine-grained analysis of FLOP-aware eviction. Marconi achieves a higher hit rate for longer sequences while sacrificing the hit rate for some shorter sequences (a), although the degradation in TTFT for shorter sequences is minimal (b).

with >7K tokens, it outperforms SGLang+ with a hit rate improvement of up to 25.5%. This is due to Marconi's FLOP-aware approach, which prioritizes caching entries with higher FLOP efficiency under contention. Since longer sequences cost more FLOP, Marconi demonstrates an overall improvement of 90.3% in FLOP saved compared to SGLang+.

Fig. 10b shows the impact of FLOP-aware eviction on TTFT distribution. Due to Marconi's lower hit rate for shorter sequences, it suffers a slight increase in TTFT at lower percentiles: Marconi's P5 TTFT is 6.3% worse than in SGLang+. However, because Hybrid models prefill short sequences quickly, the absolute latency reduction is minimal (2.1 ms, see the magnified area in Fig. 10b). This trade-off allows Marconi to achieve a lower TTFT at higher percentiles, reducing P50 and P95 TTFT by 13.4% and 22.0% (74.2 ms and 274.9 ms), respectively.

### 5.4 Microbenchmarks and Ablation Studies

In these microbenchmarks, we use different representative traces to dissect Marconi's performance improvements.

**Impact of cache contention.** In Fig. 11, we analyze how cache contention affects Marconi's benefits. We vary the cache size from 60 GB (high contention) to 140 GB (low contention). Across the five cache sizes, Marconi achieves token hit rate improvements of 24.3%, 51.5%, 68.3%, 30.0%, and 10.0% over SGLang+, respectively. The most significant performance gains occur under moderate contention, where eviction decisions are critical. In high contention scenarios, limited cache capacity prevents caching many useful prefixes (resulting in a token hit rate of less than

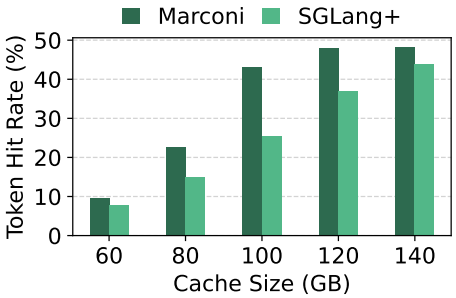
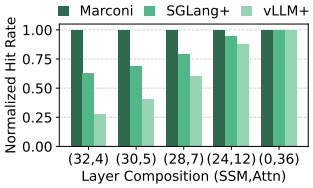
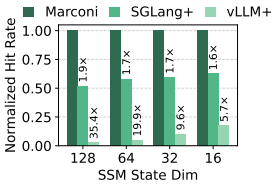



*Figure 11.* Impact of contention on FLOP-aware eviction's benefits. Marconi achieves the biggest win when the cache contention is moderate and judicious eviction decisions matter the most.




(a) Varying layer compositions.



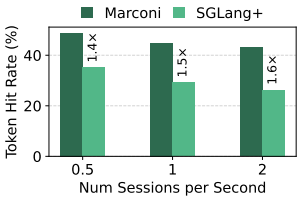

(b) Varying state dimensions.

*Figure 12.* Impact of model architecture on Marconi's performance. Marconi performs better for models with higher ratios of SSM layers and larger SSM state dimensions.

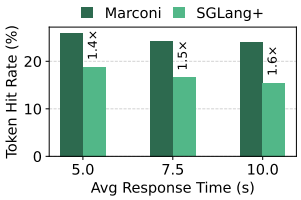
10%). Conversely, in low contention scenarios, the cache has sufficient space to store a larger number of prefixes, and thus FLOP-unaware eviction decisions have a smaller impact on the hit rate.

**Varying layer compositions.** As described in §3, the ratio of SSM layers directly affects the memory footprint of model states. In Fig. 12a, as we increase the Attention:SSM ratio from 1:2 to 1:4 to 1:8, Marconi's token hit rate improvement over vLLM+ and SGLang+ increases from 13.5% and 5.8% to 66.6% and 26.0% to 2.6× and 59.7%. When serving a pure Transformer, the three systems achieve the same performance. In order to get efficiency wins while preserving quality, the fundamental philosophy of Hybrid models is to use a *majority* of subquadratic layers with efficient computational properties while mixing in a small number of Attention layers to ensure model quality (Lieber et al., 2024; Glorioso et al., 2024; Waleffe et al., 2024). Therefore, we posit that Marconi will have better performance on most future Hybrid models.

**Varying SSM state dimensions.** As described in §3, the dimensionality of the recurrent SSM state directly affects the memory consumption of model states. Notably, the recent trend is for SSM state dimensionality to increase for better language modeling capability (Gu & Dao, 2023; Dao & Gu, 2024). In Fig. 12b, as we increase the state dimension of a Hybrid model from 16 (Mamba1) to 128 (Mamba2), Marconi's token hit rate improvement over vLLM+ grows from




(a) Varying session arrival rates.




(b) Varying request arrival rates.

*Figure 13.* Impact of request arrival pattern on Marconi's performance. Lower reusing opportunities, denoted by a higher session arrival rate (a) and longer time between requests within a session (b), reduce the token hit rate but improve Marconi's relative win due to higher contention between requests.

5.7× to 35.4×. A larger SSM state dimension increases the state sizes, exacerbating the issues in §3 and making Marconi's judicious admission more effective.

**Varying request arrival patterns.** Fig. 13 shows how request arrival patterns affect Marconi's performance. As the average number of request (chat) sessions per second increases from 0.5 to 2, Marconi's token hit rate decreases from 48.7% to 43.0%. Similarly, as the average response time between requests in a session increases from 5 s to 10 s, Marconi's token hit rate decreases from 25.9% to 24.1% due to reduced effectiveness of prefix caching from more sessions sharing the fixed cache capacity and longer delays between prefix reuses in a session. However, Marconi's relative improvement over SGLang+ grows from 1.4× to 1.6×, thanks to increased contention between requests across sessions.

## 6 Related Work

**(Hybrid) recurrent subquadratic models.** There has been a resurgence of recurrent/linear models in the recent years: RWKV (Peng et al., 2023), RetNet (Sun et al., 2023), GLA (Yang et al., 2023), Griffin (De et al., 2024), RecurrentGemma (Botev et al., 2024), xLSTM (Beck et al., 2024), Test-Time Training (TTT) (Sun et al., 2024), DeltaNet (Yang et al., 2024b;c), B'MOJO (Zancato et al., 2024), MambaFormer (Park et al., 2024), Titans (Behrouz et al., 2024), Lightning Attention (Li et al., 2025), Hunyuan-TurboS (hun, 2025), Nemotron-H (nem, 2025), etc. Importantly, although these models have different state updating rules (Yang et al., 2023; 2024c), they all update their (large) model states recurrently. As such, we only evaluate Mamba/SSMs, a representative architecture. The properties we summarize in §3 also apply to these recurrent layers (with slight variations on FLOP and state size), and Marconi can be extended to support prefix caching for all Hybrid models with recurrent layers.

**Prefix caching.** Many recent systems have been proposed to capitalize on prefix reusing opportunities in LLM inference.

InferCept (Abhyankar et al.) optimizes for KVs reusing in multi-turn chat scenarios. CachedAttention (Gao et al., 2024) and Pensieve (Yu & Li, 2023) maintain hierarchical caches to leverage memory/storage mediums. Preble (Srivatsa et al., 2024) performs cluster-level stateful caching that routes requests to the GPU with the longest prefix. PromptCache (Gim et al., 2024) and CacheBlend (Yao et al., 2024) reuse KVs but do approximations that lead to accuracy drops. All past work admits all tokens' KVs, whereas Marconi's philosophy of judicious admission is fundamentally different to handle SSM state entries' sparsity. Moreover, most prior work uses LRU for eviction and none of them factored in FLOP efficiency.

**Other LLM inference optimizations.** Continuous batching is easier to realize for SSM layers because the tensors in each step of decoding don't have a sequence dimension and can easily be batched even if the sequences have different lengths, unlike in Transformers where selective batching is needed for applying batching only to certain operations (Yu et al., 2022). The SSM states of Hybrid models don't require paged memory management (Kwon et al., 2023) as they are fix-sized and don't grow and shrink like KVs. However, the KVs of Attention layers still need to be managed by paging. Chunked prefill (Agrawal et al., 2024) for Hybrid models requires specialized kernels under development in many serving frameworks (Kwon et al., 2023). Hybrid models benefit from layer-specific optimizations like FlashAttention (Dao et al., 2022; Dao, 2023; Shah et al., 2024).

## 7 Conclusion

This paper proposes Marconi, the first prefix caching system designed to accommodate the unique characteristics of Hybrid models. Marconi proposes novel and judicious admission and eviction policies, achieving up to 34.4× higher token hit rates (71.1% or 617 ms lower TTFT) over extended versions of state-of-the-art systems.

## Acknowledgements

We thank Yinwei Dai, Neil Agarwal, Mike Wong, and many fellow interns at AWS for their helpful discussions and feedback.

## References

Introducing ChatGPT. https://openai.com/index/chatgpt/, 2022.

Github Copilot. https://github.com/features/copilot, 2022.

Perplexity. https://www.perplexity.ai/, 2023.

Rene: An Open-Source 1.3B SSM Language Model. https://cartesia.ai/blog/2024-08-27-on-device, 2024.

Optimizing AI Inference at Character.AI. https://research.character.ai/optimizing-inference/, 2024.

Prompt caching with Claude. https://www.anthropic.com/news/prompt-caching, 2024a.

Introducing the next generation of Claude. https://www.anthropic.com/news/claude-3-family, 2024b.

Needle In A Haystack - Pressure Testing LLMs. https://github.com/gkamradt/LLMTest_NeedleInAHaystack, 2024.

Prompt caching - OpenAI API. https://platform.openai.com/docs/guides/prompt-caching, 2024.

ShareGPT. https://sharegpt.com/, 2024.

Engine Arguments – vLLM. https://docs.vllm.ai/en/stable/models/engine_args.html, 2024.

Zamba2-7B. https://www.zyphra.com/post/zamba2-7b, 2024a.

Zamba2-mini (1.2B). https://www.zyphra.com/post/zamba2-mini, 2024b.

Reasoning Efficiency Redefined! Meet Tencent's 'Hunyuan-T1'—The First Mamba-Powered Ultra-Large Model. https://llm.hunyuan.tencent.com/#/blog/hy-t1?lang=en, 2025.

Nemotron-H: A Family of Accurate, Efficient Hybrid Mamba-Transformer Models . https://research.nvidia.com/labs/adlr/nemotronh/, 2025.

Abhyankar, R., He, Z., Srivatsa, V., Zhang, H., and Zhang, Y. Infercept: Efficient intercept support for augmented large language model inference. In *Forty-first International Conference on Machine Learning*.

Agrawal, A., Kedia, N., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B., Tumanov, A., and Ramjee, R. Taming throughput-latency tradeoff in llm inference with sarathi-serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 117–134, 2024.

Beck, M., Pöppel, K., Spanring, M., Auer, A., Prudnikova, O., Kopp, M., Klambauer, G., Brandstetter, J., and Hochreiter, S. xlstm: Extended long short-term memory. *arXiv preprint arXiv:2405.04517*, 2024.

Behrouz, A., Zhong, P., and Mirrokni, V. Titans: Learning to memorize at test time. *arXiv preprint arXiv:2501.00663*, 2024.

Botev, A., De, S., Smith, S. L., Fernando, A., Muraru, G.-C., Haroun, R., Berrada, L., Pascanu, R., Sessa, P. G., Dadashi, R., et al. Recurrentgemma: Moving past transformers for efficient open language models. *arXiv preprint arXiv:2404.07839*, 2024.

Brown, T. B. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Cherkasova, L. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories Palo Alto, CA, USA, 1998.

Dao, T. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

Dao, T. and Gu, A. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality. *arXiv preprint arXiv:2405.21060*, 2024.

Dao, T., Fu, D., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

De, S., Smith, S. L., Fernando, A., Botev, A., Cristian-Muraru, G., Gu, A., Haroun, R., Berrada, L., Chen, Y., Srinivasan, S., et al. Griffin: Mixing gated linear recurrences with local attention for efficient language models. *arXiv preprint arXiv:2402.19427*, 2024.

Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., Stoller, L., Hibler, M., Johnson, D., Webb, K., et al. The design and operation of cloudlab. In *2019 USENIX annual technical conference (USENIX ATC 19)*, pp. 1–14, 2019.

Gao, B., He, Z., Sharma, P., Kang, Q., Jevdjic, D., Deng, J., Yang, X., Yu, Z., and Zuo, P. Cost-efficient large language model serving for multi-turn conversations with cachedattention. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pp. 111–126, 2024.

Gim, I., Chen, G., Lee, S.-s., Sarda, N., Khandelwal, A., and Zhong, L. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems*, 6:325–338, 2024.

Glorioso, P., Anthony, Q., Tokpanov, Y., Whittington, J., Pilault, J., Ibrahim, A., and Millidge, B. Zamba: A compact 7b ssm hybrid model. *arXiv preprint arXiv:2405.16712*, 2024.

Gu, A. and Dao, T. Mamba: Linear-time sequence modeling with selective state spaces (2023). *arXiv preprint arXiv:2312.00752*, 2023.

Hendrycks, D., Burns, C., Basart, S., Zou, A., Mazeika, M., Song, D., and Steinhardt, J. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Khattab, O., Singhvi, A., Maheshwari, P., Zhang, Z., Santhanam, K., Vardhamanan, S., Haq, S., Sharma, A., Joshi, T. T., Moazam, H., et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Lee, W., Lee, J., Seo, J., and Sim, J. Infinigen: Efficient generative inference of large language models with dynamic kv cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 155–172, 2024.

Li, A., Gong, B., Yang, B., Shan, B., Liu, C., Zhu, C., Zhang, C., Guo, C., Chen, D., Li, D., et al. Minimax-01: Scaling foundation models with lightning attention. *arXiv preprint arXiv:2501.08313*, 2025.

Li, J., Wang, M., Zheng, Z., and Zhang, M. Loogle: Can long-context language models understand long contexts? *arXiv preprint arXiv:2311.04939*, 2023.

Lieber, O., Lenz, B., Bata, H., Cohen, G., Osin, J., Dalmedigos, I., Safahi, E., Meirom, S., Belinkov, Y., Shalev-Shwartz, S., et al. Jamba: A hybrid transformer-mamba language model. *arXiv preprint arXiv:2403.19887*, 2024.

Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., and Neubig, G. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35, 2023.

Park, J., Park, J., Xiong, Z., Lee, N., Cho, J., Oymak, S., Lee, K., and Papailiopoulos, D. Can mamba learn how to learn? a comparative study on in-context learning tasks. *arXiv preprint arXiv:2402.04248*, 2024.

Peng, B., Alcaide, E., Anthony, Q., Albalak, A., Arcadinho, S., Biderman, S., Cao, H., Cheng, X., Chung, M., Grella, M., et al. Rwkv: Reinventing rnns for the transformer era. *arXiv preprint arXiv:2305.13048*, 2023.

Qin, R., Li, Z., He, W., Zhang, M., Wu, Y., Zheng, W., and Xu, X. Mooncake: Kimi's kvcache-centric architecture for llm serving. *arXiv preprint arXiv:2407.00079*, 2024.

Ren, L., Liu, Y., Lu, Y., Shen, Y., Liang, C., and Chen, W. Samba: Simple hybrid state space models for efficient unlimited context language modeling. *arXiv preprint arXiv:2406.07522*, 2024.

Shah, J., Bikshandi, G., Zhang, Y., Thakkar, V., Ramani, P., and Dao, T. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608*, 2024.

Srivatsa, V., He, Z., Abhyankar, R., Li, D., and Zhang, Y. Preble: Efficient distributed prompt scheduling for llm serving. 2024.

Sun, Y., Dong, L., Huang, S., Ma, S., Xia, Y., Xue, J., Wang, J., and Wei, F. Retentive network: A successor to transformer for large language models. *arXiv preprint arXiv:2307.08621*, 2023.

Sun, Y., Li, X., Dalal, K., Xu, J., Vikram, A., Zhang, G., Dubois, Y., Chen, X., Wang, X., Koyejo, S., et al. Learning to (learn at test time): Rnns with expressive hidden states. *arXiv preprint arXiv:2407.04620*, 2024.

Team, G., Anil, R., Borgeaud, S., Wu, Y., Alayrac, J.-B., Yu, J., Soricut, R., Schalkwyk, J., Dai, A. M., Hauth, A., et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

Team, J., Lenz, B., Arazi, A., Bergman, A., Manevich, A., Peleg, B., Aviram, B., Almagor, C., Fridman, C., Padnos, D., et al. Jamba-1.5: Hybrid transformer-mamba models at scale. *arXiv preprint arXiv:2408.12570*, 2024.

Waleffe, R., Byeon, W., Riach, D., Norick, B., Korthikanti, V., Dao, T., Gu, A., Hatamizadeh, A., Singh, S., Narayanan, D., et al. An empirical study of mamba-based language models. *arXiv preprint arXiv:2406.07887*, 2024.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Wang, X., Li, B., Song, Y., Xu, F. F., Tang, X., Zhuge, M., Pan, J., Song, Y., Li, B., Singh, J., et al. Opendevin: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Wu, B., Liu, S., Zhong, Y., Sun, P., Liu, X., and Jin, X. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism. *arXiv preprint arXiv:2404.09526*, 2024.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024a.

Yang, S., Wang, B., Shen, Y., Panda, R., and Kim, Y. Gated linear attention transformers with hardware-efficient training. *arXiv preprint arXiv:2312.06635*, 2023.

Yang, S., Kautz, J., and Hatamizadeh, A. Gated delta networks: Improving mamba2 with delta rule. *arXiv preprint arXiv:2412.06464*, 2024b.

Yang, S., Wang, B., Zhang, Y., Shen, Y., and Kim, Y. Parallelizing linear transformers with the delta rule over sequence length. *arXiv preprint arXiv:2406.06484*, 2024c.

Yao, J., Li, H., Liu, Y., Ray, S., Cheng, Y., Zhang, Q., Du, K., Lu, S., and Jiang, J. Cacheblend: Fast large language model serving with cached knowledge fusion. *arXiv preprint arXiv:2405.16444*, 2024.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022.

Yu, L. and Li, J. Stateful large language model serving with pensieve. *arXiv preprint arXiv:2312.05516*, 2023.

Zancato, L., Seshadri, A., Dukler, Y., Golatkar, A., Shen, Y., Bowman, B., Trager, M., Achille, A., and Soatto, S. B'mojo: Hybrid state space realizations of foundation models with eidetic and fading memory. *arXiv preprint arXiv:2407.06324*, 2024.

Zheng, L., Chiang, W.-L., Sheng, Y., Li, T., Zhuang, S., Wu, Z., Zhuang, Y., Li, Z., Lin, Z., Xing, E., et al. Lmsys-chat-1m: A large-scale real-world llm conversation dataset. *arXiv preprint arXiv:2309.11998*, 2023a.

Zheng, L., Yin, L., Xie, Z., Huang, J., Sun, C., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., et al. Efficiently programming large language models using sglang. *arXiv preprint arXiv:2312.07104*, 2023b.

| | Attention | MLP | SSM |
|---|---|---|---|
| **FLOPs per layer** | $8LD^2 + 4L^2D$ | $16LD^2$ | $12LD^2 + 16LDN + 10L$ |
| **State size per layer (bytes)** | $4LD$ | N/A | $2DN$ |
| **FLOPs saved per byte** | $L + 2D$ | N/A | $L \cdot (6D/N + 8 + 5/DN)$ |
| **FLOPs saved per byte (7B model)** | $L + 8192$ | N/A | $200L$ |

*Table 1.* The FLOP efficiency of different layer types. As seen in the last two rows, the FLOP efficiency of SSM layers scales much more steeply compared to Attention layers. SSMs' state sizes are orders of magnitude ($N/2 = 64$ in this 7B Hybrid model, where $D = 4096$ and $N = 128$) bigger than the KVs of a single token.

| Notation | Description |
|---|---|
| L | Sequence length |
| D | Model dimension or $d_model$ |
| N | State/feature dimension or $d_state$ |

*Table 2.* Glossary of notation and terminology.

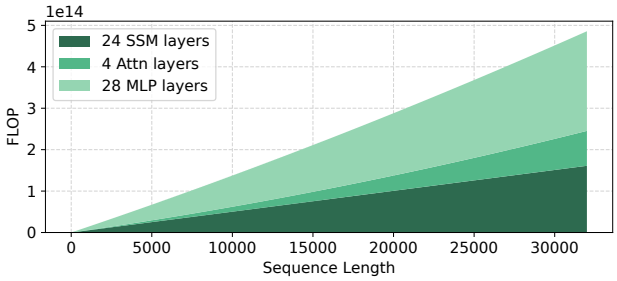



*Figure 14.* FLOP breakdown of a 7B Hybrid model.

# A Appendix

## A.1 FLOP Efficiency Analysis

In this section, we detail the math used for FLOP efficiency calculation. We list the notations used in Tab. 2.

**Memory footprint.** Assuming inference is performed in FP16, the memory size of the KVs in an Attention layer is calculated as 2 (K and V) · L · D · 2 bytes/parameter. In comparison, the memory size of an SSM layer's state is D · N · 2 bytes/parameter. Additionally, each SSM layer includes a `conv_1d` block with a state size of `in_channels` · `conv_kernel` · 2 bytes/parameter. Since the `conv_1d` states account for only a small fraction (6.1% for the 7B Hybrid model used throughout the paper) of the total state size, we omit them from Tab. 1 for simplicity, but they are included in all experiments in the main paper.

**FLOP efficiency of models with different layer compositions.** Fig.14 shows the FLOP distribution by layer type in a 7B hybrid model. Attention layers contribute fewer FLOPs for short sequences than SSMs and MLPs. However, as their quadratic computational complexity kicks in with longer sequences, they consume a significant portion of total FLOPs, even though they make up only 7.1% of the model's layers. Tab.1 provides the FLOP breakdown by layer for a 7B hybrid model, showing that the FLOP efficiency of SSM layers scales more sharply compared to Attention layers.

# B Artifact Appendix

## B.1 Abstract

In this artifact, we describe how to reproduce all experiments in this paper. The evaluation utilizes request arrival traces from LMSys, ShareGPT, and SWEBench, all tokenized using the `meta-llama/Llama-2-7b-hf` tokenizer for consistency. Key experiments involve running vLLM+, eviction policies V1 (SGLang+), V2 (Marconi), and V3 (offline-optimal, static-$\alpha$ oracle policy – not included in the paper) across various dataset/arrival rate/cache size combinations. This artifact supports easy customizations, including implementing additional eviction policies and evaluating new datasets and model configurations. For a detailed version of the artifact appendix, please see https://github.com/ruipeterpan/marconi/blob/main/artifact_evaluation.md.

## B.2 Artifact check-list (meta-information)

- **Program: Python**
- **Disk space required: ∼20 GB**
- **Time needed to prepare workflow: ∼1 hour**
- **Time needed to complete experiments: ∼12 hours**
- **Publicly available: Yes**
- **Code licenses: CC BY-NC**

- **Archived DOI:** 10.5281/zenodo.14970139

## B.3 Description

### *B.3.1 How delivered*

The artifact may be downloaded from zenodo at https://zenodo.org/records/14970139 or cloned from the GitHub repository at https://github.com/ruipeterpan/marconi. The request arrival traces (with token-level information) can be downloaded from Google Drive by following the documentation below.

### *B.3.2 Hardware dependencies*

We have tested Marconi on Cloudlab (Duplyakin et al., 2019) nodes with Ubuntu 22.04 and Python 3.11.9. Due to the request arrival traces containing token IDs for each request, they require a total disk size of ∼7 GB to house.

### *B.3.3 Software dependencies*

We have prepared a conda environment file that lists all dependencies and their versions.

### *B.3.4 Datasets*

To ensure consistency when evaluating the same trace on different model architectures, we use the same tokenizer to produce the request arrival traces. For convenience, we compress the processed traces (∼700M/6.3G pre/post compression) and host them on Google Drive.

## B.4 Installation

The software dependencies can be installed via `conda` using the environment configuration file provided in the artifact. The traces can be downloaded via this link.

## B.5 Experiment workflow

The sweep of all experiments (combinations of different cache sizes and request arrival patterns on different datasets) can be done by running `bash run_all_experiments.sh`, which invokes `policy_exploration.py` and does the following for each experiment configuration (dataset/arrival rate/cache size combination):

- Runs vLLM+
- Runs eviction policy V1, which represents SGLang+
- Runs eviction policy V2, which represents Marconi
- Runs eviction policy V3, which represents an offline-optimal, static-$\alpha$ oracle policy (the results weren't included in the paper). This policy sweeps over possible values of $\alpha$ and selects the one that maximizes the hit rate

Running `bash run_all_experiments.sh` creates three log files in `/logs`: `lmsys.txt`, `sharegpt.txt`, and `swebench.txt`. Each file contains the output log of all evaluations on the sweep of configurations for this dataset. Once the log files have been generated, plotting scripts can be run to analyze and plot the results.

## B.6 Evaluation and expected result

All plotting scripts are under `/plotting` and can be run once the sweep of experiments on all configurations has finished:

- Fig. 7: `token_hit_rate.py`
- Fig. 8: `sglang_comparison.py`
- Fig. 9: `ttft.py`
- Fig. 10: `fine_grained_analysis.py`
- Fig. 11: `microbenchmark_contention.py`
- Fig. 12a: `microbenchmark_layer_composition.py`
- Fig. 12b: `microbenchmark_dstate.py`
- Fig. 13a and 13b: `microbenchmark_arrivalrate.py`

## B.7 Experiment customization

- Additional eviction policies can be easily implemented in `radix_cache_hybrid.py` by adding a new `evict_policy_version`
- Additional model configurations can be applied by editing the default configurations (NVIDIA's Attention-Mamba2 7B Hybrid model) in `policy_exploration.py`
- Additional datasets can be evaluated by producing compatible traces as the ones generated by `/utils/generate_trace.py`

## B.8 Methodology

Submission, reviewing, and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging