A More detailed comparisons with existing baselines

This section provides the reader with a more in-depth comparison with similar architectures. We cover BRecT [20] in Section A.1 and GSS-Hybrid [24] in Section A.2.

A.1 Comparison with Block Recurrent Transformer (BRecT)

The Block Transformer layer (i.e Slide:12L) also processes keys and values from the previous window stored in a differentiable cache. This is implemented similarly to the sliding window attention pattern suggested in [20] and was originally introduced by Transformer-XL [8]. Using a causal mask, at every token inference step, the attention mechanism is applied to blocks of tokens of size W and is partially extended to the cached keys and values from the previous block with the sliding window. BRecT, as explained in [20], uses a non-differentiable cache that is carried from one sequence of size L to the next[2]. The last recurrent states of a sequence are stored in a non-differentiable cache and fed to the next training step on the following sequence in the document as a warm-start. We do not pass such a representation, since to compute the output of the convolution, we need access to the whole sequence. We believe that this is an one advantage that BRecT has over our method, especially for very long examples that split into ordered sequences of length L, since the cache carried from one sequence to the next can provide very useful long range information and (weak) access to the whole past. Since we need the whole sequence to compute SSM states, history beyond L may be lost in the process. We believe that BST can further be improved by adding non-differentiable sequence cache for very long documents.

While in other architectures, the history between blocks of tokens is not modeled, both BST and BRecT use a mechanism to model previous block context. The authors of BRecT experiment with various sequential gating mechanisms to condense the information from past blocks. With BST, we use SSM to provide context from previous blocks to the current block as explained in Section 3.2.

A.2 Comparison with the Transformer GSS-Hybrid

GSS-Hybrid [24] is a SSM-Transformer hybrid architecture that we first describe in Section 4.1. The architecture is significantly different from BRT. GSS-Hybrid is primarily composed of Gated State Space (GSS) layers and has a few interleaved Transformer layers at every 4th layer starting with the 2nd layer. BRT on the other hand is mainly composed of Block Transformer layers and has Block State Transformer layers at layer positions $\{1, 7, 9\}$ for the $\sim$200M model and $\{1, 5, 7, 9\}$ for the $\sim$400M model. Our hybrid does not stack SSM and Transformer layers like the GSS-Hybrid but rather replaces the recurrence in BRecT with an SSM such as S4. In BRT, the SSM generates states for each Block Transformer representations and we then use cross-attention to mix the states and the self-attention outputs. We also use a simpler SSM. The authors in [24] initially built GSS, a gated version of DSS [15], to (1) reduce SSM parameter dimensions, (2) stabilize training of the SSM and (3) allow better length generalization. However, when experimenting with SSMs such as S4 or DSS, we found that the gating was not necessary to achieve all three objectives stated above. We decided that using GSS's Gated Attention Unit [19] was therefore not needed when integrating SSM states into the attention mechanism. We also reiterate that the authors in [24] used hyperparameter search to get the best performance while we did not.

B Evaluating Length Generalization capabilities

We present our length generalization analysis and report perplexity in Figure 4. Our models and baselines all have $\sim$400M parameters, are trained on a sequence length of 4k and tested on sequences with *lower* and *higher* sequence lengths of $\{512, 16k, 65k\}$.

We notice that all models have similar perplexity for sequence lengths of 512. Both BST:SH:S4-L and GSS-Hybrid-L generalize well on 16k and 65k sequence lengths for PG19 and arXiv. For GitHub, GSS-Hybrid-L and BST:MF:unstruct-L perplexities increase drastically, potentially due to noise in the GitHub dataset. For GitHub again, BRecT:fixed:skip-L performs very well at higher sequence lengths. We hypothesize that the block recurrent model's access to the entire past, via non-differentiable cache

[2] In our work and in [20], a document is split into multiple sequences of size L and each sequence is split into multiple blocks of size W

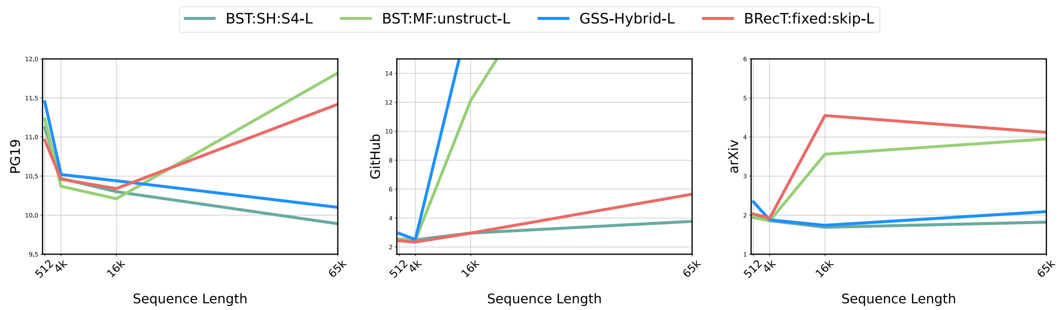

Figure 4: Length Generalization for sequence lengths {512, 16k, 65k} on PG19 (left), GitHub (middle) and arXiv (right). BST:SH:S4-L generalizes better than any other baselines, including GSS-Hybrid-L that uses GSS, a structured SSM. GSS-Hybrid-L numbers are from [24].

of representations across sequences, helps retain a "memory" of dependencies between each code file in the GitHub dataset. Interestingly, we also note that BST:MF:unstruct-L and BRecT:fixed:skip-L outperform other methods on PG19 up to a sequence length of 16K. Perplexity performance on PG19 is perhaps less reliant on long term relationships between tokens, which can explain the performance of models that have no explicit built-in mechanisms for length generalization.

The analysis also allows us to draw a clear distinction between *structured* and *unstructured* SSMs integrated in hybrid architectures. As previously mentioned in Section 3.1, SSMs such as GSS, DSS and S4 use a structured kernel K, built from learned matrices A, B and C for any sequence length L in Equation 3. Since K is extendable to any arbitrary sequence length L, both BST:SH:S4-L and GSS-Hybrid-L have a build-in mechanism for length generalization that the unstructured BST:MF:unstruct-L model does not. BST:MF:unstruct-L performs best on the training sequence of 4K and is on-par for 512 with perplexity increasing for unseen 16K and 65K sequence lengths. **BST:SH:S4-L has by far the best perplexity for 65K sequence lengths on PG19, GitHub and arXiv.**

C Ablation Studies

In the following section, we perform ablations to investigate (1) the placement of a *single* SSM layer in Table 2 in the overall architecture, (2) the effects of the number of SSM layers added in Table 3, and (3) the size D of the SSM state in Table 4. For the ablations, we use the $\sim$200M parameter BST:SH:S4, since it is the fastest model, and assess various configurations on PG19.

Table 2: A single BST at various layer index.

Layer index	Perplexity
3	12.41
7	11.92
9	11.88
12	12.03

Table 3: Multiple BST layers at various locations.

Num layers	Perplexity
2	11.69
3	11.57
4	11.21
5	11.20

Table 4: Increasing BST's S4 model state size D.

State Size	Perplexity	Step Time
8	11.95	$\times 0.7$
16	11.57	$\times 1.0$
32	11.55	$\times 1.8$
64	11.54	$\times 3.2$

In Table 2, we experiment adding a single BST layer at layer indices $3, 6, 9, 12$. We notice that a single BST layer with state size $D = 16$ located closer to the middle of the whole Block Transformer stack, at index $= 9$, has the greatest effect on perplexity. This finding is inline with findings in prior work [36, 20].

In Table 3, we test if adding multiple BST layers yields improvements on performance. We start with BST layers with state size $D = 16$ at indices $0, 9$. We follow by adding another BST layer at index 7 for a total of three BST layers and then another at index 5, followed by another at index 12. Adding more BST layers lowers perplexity. However, the results seem to plateau at 5 BST layers. We note also that there is a 3.5% training step time increase for each added layer.

In Table 4, we train our models with different state sizes D. For the state size ablation, we use three BST layers at indices $0, 7, 9$. We find that increasing D improves perplexity to the detriment of training speed (step time). For this reason, we chose $D = 16$ for Table 1 BST results.

D Limitations

While BST's SSM layer allows the model to unroll and parallelize the recurrence that models long-term context between blocks of tokens, the SSM variants are reliant on efficient FFT operations. We have found that the FFT operation is an important speed bottleneck on TPUs that needs to be resolved to better scale BST to multiple layers and larger models. While we are still investigating the reasons, we found that JAX FFT was x4 faster on GPUs. Further, new SSM variants such as S5 [30] bypass FFT operations using a binary associative operator[3]. Our implementation is modular enough that we can simply plug in S5 or use other FFT implementations.

One of our assumption is that BST's SSM layer is able to capture the right long-term dependency for each block. The SSM recurrence at step $T = t$ provides a summarized representation of previous steps for $T = 0$ to $T = t - 1$. However, a single vector representation may not be powerful enough to support all important long-term dependencies. Despite the perplexity improvements on long-range language modeling tasks, this assumption needs to be tested on other long range classification tasks such as Long Range Arena [32] as well. It is possible that our model can perform better if we feed to the attention layer W SSM representations that are chosen by a top-k retrieval operation, similar to the one in Memorizing Transformer [36].

E JAX Implementation of BST

Pseudocode 1 contains a function that implements convolution of multiple filters over the same input sequence using FFT and inverse FFT operations. Pseudocodes 2, 3 and 4 respectively implement context state collection of BST variants: Single-Head (SH), Multi-Head (MH) and Multi-Filter (MF). Finally, Pseudocode 5 runs the Block Transformer sublayer in parallel by feeding the context states to their corresponding block.

```python
"""Unstructured filters and convolutions."""

import jax
from jax import numpy as jnp
from einops import rearrange

win_length = 512    # (w)
seq_length = 4096   # (l)

def get_filters_unstruct(channels):
    """Returns trainable filters and biases.

    Args:
        channels: number of filters.

    Returns:
        h: filter of shape (seq_length, channels, dim)
        b: bias of shape (channels, dim)
    """
    t = jnp.linspace(0.0, 1.0, seq_length)
    h = jnp.exp(- alpha * t) * dense(positional_emb(t))
    b = get_bias()
    return h, b

def multichannel_convolution(u, h, b):
    """Multichannel convolution function.

    Args:
```

[3] In JAX, this is equivalent to using $jax.lax.associative\_scan$.

```
        u: input of shape (seq_length, dim)
        h: filters of shape (seq_length, channels, dim)
        b: bias of shape (channels, dim)
    """
    h = rearrange(h, "l c d -> c d l")

    fft_size = seq_length * 2
    u_f = jnp.fft.rfft(x, n=fft_size)
    h_f = jnp.fft.rfft(h, n=fft_size)

    y = jnp.fft.irfft(h_f * x_f, n=fft_size, norm="forward")[
            ..., :seq_length]       # (c, d, l)
    y = y + x * b[..., None]        # (c, d, l)
    y = rearrange(y, "c d l -> l d c")
    return y
```

Pseudocode 1: Unstructured filters and convolutions.

```
"""Context state collection for BST-SH variant."""

num_heads = 8       # (h)
num_states = 32     # (s)

# (SH): Single-Head
def SH_context_states(u):
    """Single-Head Context Collection."""
    h, b = get_filters_[unstruct/s4](channels=1)
    y_1 = multichannel_convolution(u, h, b)
    # y_1: (l, d, 1)

    # lift to multiple heads
    y_h = dense(y_1)
    # y_h: (l, d, h)

    context_states = jnp.split(
            y_h, seq_length // win_length, axis=0)
    return context_states # (l/w, w, d, h)
```

Pseudocode 2: Context state collection for BST-SH variants.

```
"""Context state collection for BST-MH variant."""

# (MH): Multi-Head
def MH_context_states(u):
    """Multi-Head Context Collection."""
    h, b = get_filters_[unstruct/s4](channels=num_heads)
    y_h = multichannel_convolution(u, h, b)
    # y_h: (l, d, h)

    context_states = jnp.split(
            y_h, seq_length // win_length, axis=0)
    return context_states # (l/w, w, d, h)
```

Pseudocode 3: Context state collection for BST-MH variants.

```
"""Context state collection for BST-MF variant."""

# (MF): Multi-Filter
def MF_context_states(u):
    """Multi-Filter Context Collection."""
    h, b = get_filters_[unstruct/s4](channels=num_states)
    y_s = multichannel_convolution(u, h, b)
```

```python
    # y_s: (l, d, s)
    context_states = jnp.split(
            y_s, seq_length // win_length, axis=0)
    # context_states: (l/w, w, d, s)

    # collect the last context states
    context_states = context_states[:, -1, ...] # (l/w, d, s)
    context_states = rearrange(
            context_states, "lw d s -> lw s d")

    # shift context states corresponding to windows
    context_states = jnp.roll(context_states, 1, axis=1)

    # replace the initial window with trainable weights
    init_context = get_init_context(num_states) # (d, s)
    context_states[0] = init_context

    # lift to multiple heads
    context_states = dense(context_states)

    return context_states # (l/w, s, d, h)
```

Pseudocode 4: Context state collection for BST-MF variants.

```python
"""Block-State Transformer Layer."""

# Block Transformers are non-recurrent and parallelizable.
block_transformer = jax.vmap(BRecT.nonrecurrent_cell)

def BST(u):
    """Block-State Transformer Layer."""
    global MF # True if Multi-Filter, False otherwise (SH/MH)

    # split inputs into windows (l/w, w, d)
    u = jnp.split(u, seq_length // win_length, axis=0)

    # collect context states from SSM outputs
    context_states = [SH/MH/MF]_context_states(u)

    # pass the contexts in place of recurrent states
    y = block_transformer(
            token_embeddings=u,
            recurrent_state=context_states,
            use_cross_attn_causal_mask=not MF,
            use_cross_positional_emb=MF, # context IDs
    )

    return rearrange(y, "lw w d -> (lw w) d") # (l, d)
```

Pseudocode 5: Block-State Transformer Layer.