Future-Seed: Cross-Layer State Feedback for In-Place Constraint Repair
(RWKV7-CUDA Technical Report)

yanghu819

February 15, 2026

Abstract

Fixed-memory recurrent language models often struggle with *in-place constraint repair*: filling masked spans without rewriting the whole sequence while maintaining global consistency (e.g., permutation validity, structural constraints, or Sudoku rules). A key difficulty is that early positions must depend on information that appears on the right ("future") side of the sequence, while naive parallel infill can easily create conflicts (duplicates, omissions, invalid structures). This report studies a minimal mechanism, **Future-Seed**, which initializes each layer with the previous layer's final state (cross-layer state feedback), creating a "re-read across depth" information pathway. On a single RTX 4090 using an RWKV7 `cuda_wind` kernel, we evaluate a small suite of hard synthetic tasks: **RIGHTCOPY/CONSTR** (future-aware sanity checks), **KVSORT** (in-place key–value permutation repair), **PERMFILL** (permutation filling with length extrapolation), and **4×4 Sudoku** (2D constraint repair). We observe consistent gains on sanity tasks; a sharp failure-to-success transition on KVSORT (FS=0 exact=0 vs. FS=1 exact=1 for keys36, $n = 20$); strong length extrapolation improvements on PERMFILL via a tiny **anchor** (keeping $k = 2$ ground-truth tokens inside the masked span); and a clear phase shift on Sudoku. These results support the interpretation of Future-Seed as a lightweight global "bookkeeping/constraint propagation" channel for in-place repair.

1 Problem Setting: In-Place Constraint Repair

We focus on tasks where the model must repair a fixed-length sequence *in-place*, rather than generating a new output sequence. A typical input format is:

```
P=… | M=___… | R=…
```

where R (right context) contains information needed to fill the masked span M, and training loss is computed only on masked positions. This setup stresses:

- **Future dependency**: correct predictions in M require information located in R.
- **Global consistency**: outputs must satisfy hard constraints (e.g., permutation validity, Sudoku rules).

2 Method: Future-Seed

Consider a model with L stacked recurrent layers. Let $s_T^{(l)}$ be the final recurrent state of layer l after processing a length-T sequence. **Future-Seed** sets the initial state of layer l to the previous layer's final state:

$$s_0^{(l)} \leftarrow s_T^{(l-1)}.$$

Intuitively, this allows the network to "re-read" the same sequence across depth, so later layers can condition early-token computation on information summarized from the whole sequence.

3 Tasks and Metrics

All tasks use the same in-place masked objective (loss on M only).

RIGHTCOPY / CONSTR (sanity). Minimal future-aware probes. Metric: masked token accuracy (acc).

KVSORT (key–value permutation repair). The right context provides a shuffled key–value listing; the masked span must output the sequence sorted by key. Metric: exact match (`exact`) on in-distribution (id) and out-of-distribution (ood) variants produced by the task generator.

PERMFILL (permutation fill with anchors). The model must reconstruct a permutation in the masked span from right-context information. We optionally add an **anchor**: keep k ground-truth tokens visible inside the masked span (here $k = 2$) to reduce alignment instability when everything is masked. Metric: exact/valid rate (identical in our implementation).

4×4 Sudoku (2D constraint repair). We generate valid 4×4 Sudoku solutions (digits 1–4; 2×2 blocks), then mask h cells. Metrics:

- **solve-rate**: predicted grid satisfies row/column/block constraints.
- **exact**: matches the hidden generated solution exactly.

4 Experimental Setup

We run all experiments on a single NVIDIA RTX 4090, using an RWKV7 `cuda_wind` kernel. The main entry point is `rwkv_diff_future_seed.py`. Tasks are toggled via environment variables (e.g., `KVSORT_TASK=1`) and the Future-Seed switch `FUTURE_SEED` (set to `0` or `1`).

5 Results

5.1 RIGHTCOPY / CONSTR: consistent gains

Across 3 random seeds, Future-Seed improves final masked-token accuracy (mean ± sd):

Table 1: RIGHTCOPY/CONSTR final accuracy (3 seeds, mean±sd).

task	FS=0	FS=1
rightcopy acc	0.1046 ± 0.0074	0.1550 ± 0.0176
constr acc	0.0977 ± 0.0028	0.2114 ± 0.0179

5.2 KVSORT: a failure-to-success transition

For keys36, $n = 20$, no-separator encoding, $L = 8$, and SEQ_LEN = 256, we observe:

Table 2: KVSORT exact match (keys36, $n = 20$, nosep, L8, SEQ_LEN=256).

metric	FS=0	FS=1
kvsort_id exact	0.0	1.0
kvsort_ood exact	0.0	1.0

5.3 Transformer attention baselines (negative)

To isolate whether KVSORT is simply an attention-visibility issue, we implemented several Transformer baselines in the same codebase: (i) bidirectional Transformer-MLM (`MODEL=transformer`), (ii) decoder-only causal Transformer (`MODEL=transformer_causal`), and (iii) an "attention-side Future-Seed" variant that passes a cross-layer global token summary (`ATTN_FS=1`, K=32). On KVSORT (keys36, n=20, nosep), all Transformer variants fail to recover the correct ordering (exact=0.0). Hungarian decoding can enforce permutation *validity* (no duplicates) but does not fix ordering. We also tested iterative refinement (`REFINE_STEPS=8`) and a structured Sinkhorn assignment loss (`PERM_SINKHORN=1`); neither resolves the failure.

Table 3: KVSORT Transformer baselines (keys36, $n = 20$, nosep). Hungarian decoding enforces validity but ordering remains 0.

model	decode	exact	key_valid	key_order
Transformer-MLM	argmax	0.0	0.0	0.0
Transformer-MLM	hungarian	0.0	1.0	0.0
Transformer-Causal	hungarian	0.0	1.0	0.0
Transformer-Causal + ATTN_FS (K=32)	hungarian	0.0	1.0	0.0
Transformer-MLM + Sinkhorn loss	hungarian	0.0	1.0	0.0

5.4 PERMFILL: anchors improve length extrapolation

Evaluating saved weights under length extrapolation (n_{test} sweep) shows that a tiny anchor plus higher training max length can substantially improve OOD generalization:

Table 4: PERMFILL length extrapolation (exact/valid). Anchor keeps k ground-truth tokens visible inside the masked span.

weights	anchor	n=24	n=28	n=32	n=36
permfill_n24_fs1_L12_seq256	off	1.000	0.000	0.000	0.000
permfill_anchor2_n24_fs1_L12_seq256	k=2	0.990	0.830	0.120	0.000
permfill_anchor2_n32_fs1_L12_seq256	k=2	1.000	1.000	1.000	0.935

5.5 4×4 Sudoku: a phase shift

Future-Seed significantly shifts the solve-rate phase transition to harder puzzles:

Table 5: 4×4 Sudoku in-place fill: solve-rate phase curve (trials=2000). FS=0 uses L4/e128; FS=1 uses L12/e128.

holes	FS=0 solve	FS=1 solve
4	0.3530	1.0000
6	0.1665	1.0000
8	0.0520	1.0000
10	0.0095	0.9510
12	0.0000	0.5510
14	0.0000	0.0000

We also tested a minimal soft constraint regularizer (row/col/block digit-count consistency), which improves holes=12 solve-rate (0.4795 → 0.5855) but does not solve the hardest setting (holes=14).

6 Limitations and Next Steps

This report is intentionally minimal and synthetic. Key next steps:

- **Inference protocol**: beyond one-shot fill, study iterative refinement under equal compute on harder constraints (e.g., larger Sudoku / structured text repairs).
- **Stronger baselines**: we tested Hungarian decoding and Sinkhorn permutation loss; future work includes pointer/ILP-style structured models and richer permutation tasks (with duplicates / instance-level matching).
- **More realistic repairs**: extend to hard-constraint structured text (JSON/brackets) and consistency editing tasks.

Reproducibility

Repository: `git@github.com:yanghu819/future-seed.git`. Main script: `rwkv_diff_future_seed.py`. Sudoku script: `rwkv-diff-future-seed/run_sudoku.sh`.