A. Architecture Details

Our model is implemented in PyTorch (Paszke et al., 2019) and is optimized with the Adam optimizer (Kingma & Ba, 2014).

Embeddings. In the full-sized model, to embed memory addresses and cache lines, we train a unique embedding for each unique memory address observed during training, sharing the same embedder across memory addresses and cache lines. We similarly embed PCs.

Concretely, we initialize an embedding matrix $W^m \in \mathbb{R}^{(n_m+1)\times d_m}$ via Glorot uniform initialization (Glorot & Bengio, 2010), where n_m is set to the number of unique memory addresses in the training set and d_m is the dimension of the embedding. Then, each unique memory address m is dynamically assigned a unique id 1 to n_m, and its embedding $e(m)$ is set to the i-th row of the embedding matrix W^m. At test time, all memory addresses unseen during training are mapped to a special *UNK* embedding, equal to the last row of W^m. We embed PCs with a similar embedding matrix $W^p \in \mathbb{R}^{(n_p+1)\times d_p}$.

Attention. After computing embeddings (with either the full-sized model or the byte embedder) $e(l_1), \ldots, e(l_W)$ for each line in the cache state and hidden states $[h_{t-H+1}, \ldots, h_t]$ representing the past H accesses, we compute a context g_w for each cache line by attending over the hidden states with the line embeddings as queries as follows:

1. Following Vaswani et al. (2017), we compute positional embeddings $e(-H+1), \ldots, e(0)$, where $e(\text{pos}) \in \mathbb{R}^{d_{\text{pos}}}$ and:

$$e(\text{pos})_{2i} = \sin\left(\frac{\text{pos}}{10000^{2i/d_{\text{pos}}}}\right)$$
$$e(\text{pos})_{2i+1} = \cos\left(\frac{\text{pos}}{10000^{2i/d_{\text{pos}}}}\right).$$

   We concatenate these positional embeddings with the hidden states to encode how far in the past each access is: $[(h_{t-H+1}; e(-H+1)), \ldots, (h_t; e(0))]$. Although in theory, the LSTM hidden states can encode positions, we found that explicitly concatenating a positional embedding helped optimization.
2. We apply General Attention (Luong et al., 2015) with each cache line embedding as the query and the concatenated hidden states and positional embeddings as keys:

$$\alpha_i = \text{softmax}(e(l_w)^T W_e h_{t-H+i})$$
$$g_w = \sum_{i=1}^{H} \alpha_i h_{t-H+i}.$$

   The matrix $W_e \in \mathbb{R}^{d_m \times (d_{\text{LSTM}}+d_{\text{pos}})}$ is learned and $g_1, \ldots, g_W$ can be computed in parallel (Vaswani et al., 2017).

B. Experimental Details

B.1. Detailed Results

To provide further insight into our learned policy, we also report results on two additional metrics:

- **Top-K Accuracy**: The percentage of the time that the optimal line to evict according to Belady's is in the top-K lines with highest probability of eviction under our learned policy. This indicates how frequently our learned policy is outputting decisions like those of Belady's. We report top-1 and top-5 accuracy. Note that since each cache set in the last-level cache can only hold 16 lines, the top-16 accuracy is 100%.
- **Reuse Distance Gap**: The average difference between the reuse distance of the optimal line to evict l^* and the reuse distance of the line evicted by PARROT, i.e., $d_t(l^*) - d_t(l_w)$, where $w = \arg\max_i \pi(i \mid s_t)$. This metric roughly captures how sub-optimal the decision made by PARROT is at each timestep, as evicting a line with a smaller reuse distance is more likely to lead to a cache miss. A policy with a reuse distance gap of 0 is optimal, while a high reuse distance gap leads to more cache misses.

We report results with of our full model with this metric in Table 2. In mcf, libquantum, and lbm, our replacement policy frequently chooses to evict the same cache line as Belady's with a top-1 accuracy of over 75% and a top-5 accuracy close to 100%. In other programs (e.g., omnetpp, Web Search), our replacement policy's top-1 and top-5 accuracies are significantly lower, even though its normalized cache hit rate in these programs is similar to its normalized cache hit rate in mcf. Intuitively, this can occur when several cache lines have similar reuse distances to the reuse distance of the optimal cache line, so evicting any of them is roughly equivalent. Thus top-K accuracy is an interesting, but imperfect metric. Note that this is the same intuition behind our ranking loss, which roughly measures the relative suboptimality of a line as a function of the suboptimality of its reuse distance.

The differences in the reuse distance gaps between the programs emphasize the differences in the program behaviors and roughly indicate how frequently cache lines are reused in each program. For example, PARROT achieves wildly different average reuse distance gaps, while maintaining similar normalized and raw cache hit rates (Table 1) in omnetpp and mcf, due to differences in their access patterns. In omnetpp, evicting a line with a reuse distance 100's of accesses smaller than the reuse distance of the optimal line some-

Table 2. Mean top-1/5 accuracy and reuse distance gap of PARROT, averaged over 3 seeds with single standard deviation intervals.

Program	Top-1 Acc. (%)	Top-5 Acc. (%)	Reuse Dist. Gap
astar	17.2 ± 1.1	50.8 ± 1.8	20065.8 ± 6433.1
bwaves	20.9 ± 3.2	49.1 ± 3.0	1356.9 ± 653.5
bzip	21.2 ± 0.9	44.3 ± 2.5	478.1 ± 39.5
cactusadm	91.3 ± 1.7	98.3 ± 0.2	2.0 ± 0.1
gems	15.0 ± 0.6	45.5 ± 2.4	46.4 ± 3.8
lbm	83.6 ± 2.4	98.0 ± 0.4	2.1 ± 0.3
leslie3d	88.5 ± 0.2	98.1 ± 0.2	1.9 ± 0.0
libquantum	83.2 ± 3.9	97.3 ± 0.2	2.2 ± 0.6
mcf	75.2 ± 0.5	87.6 ± 0.3	6.4 ± 0.4
milc	45.2 ± 2.2	65.6 ± 2.3	39.8 ± 5.1
omnetpp	35.0 ± 0.4	65.1 ± 0.4	533.4 ± 8.0
sphinx3	67.0 ± 2.0	88.4 ± 1.9	39.0 ± 8.5
xalanc	40.4 ± 2.2	93.4 ± 0.7	34805.1 ± 14760.2
Web Search	31.8 ± 0.8	77.5 ± 3.6	4012.5 ± 413.9

times does not lead to a cache miss, as both lines may eventually be evicted before being reused anyway. On the other hand, in mcf, evicting a line that is used only slightly earlier than the optimal line more frequently leads to cache misses.

B.2. Hyperparameters

The following shows the values of all the hyperparameters we used in all our final experiments. We ran our final experiments with the bolded values and tuned over the non-bolded values. These values were used in all of our final experiments, including the ablations, except the history length experiments (Section 5.4), where we varied the history length H.

- Learning rate: (**0.001**, 0.003)
- Address embedding dimension (d_m): **64**
- PC embedding dimension (d_p): **64**
- PC embedding vocab size (n_p): **5000**
- Position embedding dimension (d_{pos}): **128**
- LSTM hidden size (d_{LSTM}): **128**
- Frequency of recollecting B: (**5000**, 10000)
- History Length (H): (20, 40, 60, **80**, 100, 120, 140)

We used the same hyperparameter values in all 5 programs (omnetpp, mcf, libq, lbm, and Web Search) we evaluated on, where the address embedding vocab size n_m was set to the number of unique addresses seen in the train set of each program (see Table 3). For most hyperparameters, we selected a reasonable value and never changed it. We tuned the rest of the hyperparameters exclusively on the validation set of omnetpp.

B.3. Program Details

Table 3 reports the number of unique addresses and PCs contained in the train/test splits of each program, including the number of unique addresses and PCs in the test split that were not in the train split. Notably, in some programs, new addresses and PCs appear at test time, that are not seen during training, requiring the replacement policy to generalize.

In Table 3, we also report the total number of last-level cache accesses collected for each of the five programs in our 50s collection interval. These accesses were split into the 80% train, 10% validation, and 10% test sets. Since different programs exhibited varying levels of cacheability at the L1 and L2 cache levels, different numbers of last-level cache accesses resulted for each program. These varying numbers also indicate how different programs exhibit highly different behavior.

B.4. Randomly Chosen Cache Sets

We randomly chose 64 sets and collected accesses to those sets on the last-level cache. The 64 randomly chosen sets were: 6, 35, 38, 53, 67, 70, 113, 143, 157, 196, 287, 324, 332, 348, 362, 398, 406, 456, 458, 488, 497, 499, 558, 611, 718, 725, 754, 775, 793, 822, 862, 895, 928, 1062, 1086, 1101, 1102, 1137, 1144, 1175, 1210, 1211, 1223, 1237, 1268, 1308, 1342, 1348, 1353, 1424, 1437, 1456, 1574, 1599, 1604, 1662, 1683, 1782, 1789, 1812, 1905, 1940, 1967, and 1973.

Table 3. Program details in terms of the number of last-level cache accesses and unique addresses/PCs. 'Train' and 'Test' show the number of unique addresses and PCs appearing in each domain at train and test time, whereas 'Unseen Test' indicates the number of addresses and PCs appearing at test time, but not train time. The given percentages indicate what portion of all test accesses had unseen addresses or PCs.

		Train		Test		Unseen Test	
Program	Cache Accesses	Addresses	PCs	Addresses	PCs	Addresses	PCs
astar	879,040	13,047	25	8,235	9	46 (0.6%)	0 (0%)
bwaves	2,785,280	443,662	436	209,231	155	12 (0.1%)	38 (5.8%)
bzip	3,899,840	6,086	415	3,571	28	0 (0%)	0 (0%)
cactusadm	1,759,040	298,213	243	83,046	191	0 (0%)	0 (0%)
gems	7,298,880	423,706	5,305	363,313	1,395	1 (0%)	0 (0%)
lbm	10,224,960	206,271	44	206,265	34	0 (0%)	0 (0%)
leslie3d	6,956,160	38,507	1,705	38,487	1,663	0 (0%)	0 (0%)
libquantum	4,507,200	16,394	14	16,386	5	0 (0%)	0 (0%)
mcf	4,143,360	622,142	73	81,666	67	77,608 (21.6%)	0 (0%)
milc	3,048,000	203,504	254	90,933	82	0 (0%)	2 (0.5%)
omnetpp	3,414,720	16,912	402	14,079	315	275 (0.6%)	3 (1.0%)
sphinx3	2,372,800	20,629	528	4,586	199	0 (0%)	1 (0%)
xalanc	4,714,240	15,515	217	11,600	161	507 (1%)	5 (0%)
Web Search	3,636,800	241,164	32,468	66,645	15,893	10,334 (5.3%)	948 (6%)