Scalable RL for Systems via Offline Imitation from Multiple Baselines: A Case Study in Compiler Optimization

Teodor V. Marinov
Google Research
tvmarinov@google.com

Alekh Agarwal
Google Research
alekhagarwal@google.com

Mircea Trofin
Google
mtrofin@google.com

Abstract

From scheduling, to resource allocation to optimization of complex workflows, systems are replete with decision-making problems which are typically addressed with hand-designed heuristics. Recent literature studies pose these setups as Reinforcement Learning (RL) problems owing to a natural fit, with several successes in simulated benchmark environments. However, bringing the RL approach to any complex system in practice is full of challenges in integrating the system into the act-observe-learn paradigm of RL, which has limited the adoption of these techniques. In this work, we present an alternative approach which uses offline data collected using multiple existing baseline policies to simultaneously improve upon them. By repeating multiple iterations of this improvement process, including any learned policies into the set of baselines, we show how performance can be quickly bootstrapped using our approach. We demonstrate the practicality of our approach through evaluation in optimizing the inlining decisions for the LLVM compiler, and obtain significant improvements even over prior RL-based policies.

1 Introduction

Mapping an input from a user to its actual execution on hardware in any computer system typically involves a sequence of decisions, ranging from scheduling of operations and allocation of resources, to choosing among many equivalent reformulations of the input. The goal typically is to make decisions that improve some target metric of interest, such as running time, network utilization or memory consumption. In practice, exact optimization is typically intractable, and the decisions are often made through expert-defined heuristics, often refined over a long period of time. More recently, the research literature has investigated the use of Reinforcement Learning (RL), which is a principled paradigm for finding optimal decision sequences.

When applying Reinforcement Learning (RL) to real-world applications, two key challenges often prove to be critical blockers to adoption. First is that the online act-observe-learn loop in conventional RL poses a significant engineering overhead in most large-scale systems, that are more naturally designed to take a static Machine Learning (ML) model as a dependency and update this model only periodically in an offline manner. The difficulty arises due to coupling of the data collection and RL policy update steps which typically requires significantly more complex architecture [Mnih et al., 2016] to scale to domains where data collection is expensive. Our approach, on the other hand, simply requires interleaving standard supervised learning and batch data collection, which is quite desirable especially in the compiler application, where the ML training happens on GPUs, while the compilation happens on CPU machines. Further RL algorithms typically begin tabula rasa, that is, they only leverage the information they glean about the task at hand through these online interactions. Typical scenarios, where the use of RL is often preceded by prior attempts using rule-based or supervised ML approaches, come with a treasure trove of valuable data about desirable

Machine Learning for Systems Workshop at (NeurIPS 2024).

and undesirable behaviors, ignoring which leads to undesirable sample complexity of learning from scratch for RL. More important, the previously tried decision making policies, even when individually suboptimal, provide a valuable source of insight into the plausibly good choices in many scenarios. In this work, we study the question of leveraging such prior policies and any data collected using them, without necessarily relying on online policy updates. We propose an algorithm based on combining the available policies and apply it to the inlining for size problem in an LLVM compiler.

Inlining is the process of replacing a call to a function in a program by the body of the function. Inlining for size is the decision problem of which functions to inline with the goal of minimizing the size of the compiled binary. This fits nicely into the sequential decision making paradigm as during compilation steps there is a decision to be made whether to inline a called function or not. This decision in turn changes the call graph associated with the program and thus evolves the state for future inlining decisions.

2 Inlining for size as a RL problem

We omit most compilation details and just give a brief overview which should be sufficient for understanding the problem from a RL perspective. A detailed description can be found in Appendix D.1.

Following prior work Trofin et al. [2021], each binary target is broken up into multiple modules, where each module roughly corresponds to a C/C++ source file (after application of any preprocessor directives). The module in turn consists of multiple callsites in the LLVM Intermediate Representation (IR), and we consider whether to inline each callsite in the context of the call graph of the module in which it occurs. We note that the compiler is deterministic, so that we subsequently formulate the problem as a deterministic environment for RL.

To formulate the above as an RL problem we consider a contextual Markov Decision Process (MDP) where each context x corresponds to a single module. The state space, $\mathcal{S}$, consists of callsites in the IR of the module x, and the action set is $\{\texttt{inline}, \texttt{ don't inline}\}$ or $\{1, 0\}$ respectively. The initial state distribution is denoted by D and the sampled context (initial state) is $x \sim D$. Once the context is sampled, the transition kernel $\mathbb{P}_x$ is deterministic, that is $\mathbb{P}_x(\cdot|s, a)$ is a point-mass distribution. The reward function is $r_x(s, a)$ and we assume deterministic rewards. We note that both the transition kernel and reward kernel are context-dependent. We omit the context subscript from our notation whenever it does not introduce ambiguity. We work in the finite-horizon setting and denote the horizon as H.

Our goal is to find an inlining policy, which maps the pair (s, x) to an action, which is an inlining decision. We limit to policies which only depend on the state s and not the context x for simplicity of notation, with the understanding that any relevant parts of the context can always be included in the state without loss of generality. The goal is to learn a policy which maximizes the sum of rewards of the chosen actions: $\max_\pi \mathbb{E}_{x\sim D}[\sum_{h=1}^{H} r_x(s_h, a_h)|a_h = \pi(s_h)]$.

We conclude this section with noting some conceptual challenges of the setting, apart from the aforementioned engineering complexity of embedding the system in an RL algorithm's update loop. First, most systems applications involve sparse, trajectory-level rewards. For instance, after we choose a sequence of inlining decisions, we may compile the module and observe the binary size of the result. This does not give a precise signal about the quality of individual decisions. The sparse feedback is particularly challenging for RL algorithms which attempt to learn value functions, as these typically rely critically on intermediate feedback across the trajectory. We intentionally adopt a solution strategy that can handle such terminal feedback in our approach that we describe next.

3 Algorithm

We start with a set of K baselines policies $\mathcal{B} = \{\pi_i\}_{k\in[K]}$ together with n trajectories for each policy, which we denote by $\{\tau_{i,j}\}_{i\in[K],j\in[n]}$, where $x_j \sim D$. A trajectory for policy π consists of state-action pairs $\{(s_h, a_h)\}_{h\in[H]}$, with $a_h \sim \pi(\cdot|s_h)$. For an arbitrary policy π and module x we use $\tau_\pi(x)$ to denote the trajectory generated by following π on module x. We also assume that we see the total reward for each trajectory and policy, that is for all $i \in [K], j \in [n]$, we only observe $r(\tau_{i,j}) = \sum_{(s_{j,h},a_{j,h})\in\tau_{i,j}} r(s_{j,h}, a_{j,h})$, instead of observing a dense reward across all the time steps

of the trajectory. We assume that the rewards are bounded and non-negative, that is $r(\tau) \in [0, B]$ for all trajectories τ and some $B > 0$.

We have access to a policy class Π and seek to find a policy in Π which ideally competes with each of the baselines, and is able to combine their strengths.

Algorithm 1 `BC-MAX` for cloning best per-context baseline

Require: Base policies set $\mathcal{B} = \{\pi_k\}_{k\in[K]}$ and policy class Π. Max number of iterations N.
Ensure: Policy $\hat{\pi} \in \Pi$.

1: **for** $l \in [N]$ **do**
2: **for** $j \in [n]$ **do**
3: Sample module $x_j \sim D$
4: $r_{i,j}, \tau_{i,j} = \texttt{Collect trajectory}(\pi_i, x_j), \forall \pi_i \in \mathcal{B}$
5: Compute highest reward policy $\pi_{i_j} = \text{argmin}_{i\in[K]} \sum_{(s_{j,h},a_{j,h})\in\tau_{i,j}} r_{i,j}(s_{j,h}, a_{j,h})$
6: Compute module weights $\{w_j\}_{j\in[n]}$.
7: $\hat{\pi}_l = \text{argmin}_{\pi\in\Pi} - \sum_{j=1}^{n} w_j \sum_{(s_{j,h},a_{j,h})\in\tau_{i_j,j}} a_{j,h} \log(\pi(a_{j,h}|s_{j,h}))$
8: $\pi_l(s) = \text{argmax}_a \hat{\pi}_l(a|s), \forall s \in S$. Expand baseline policy set $\mathcal{B} = \mathcal{B} \bigcup \{\pi_l\}$.

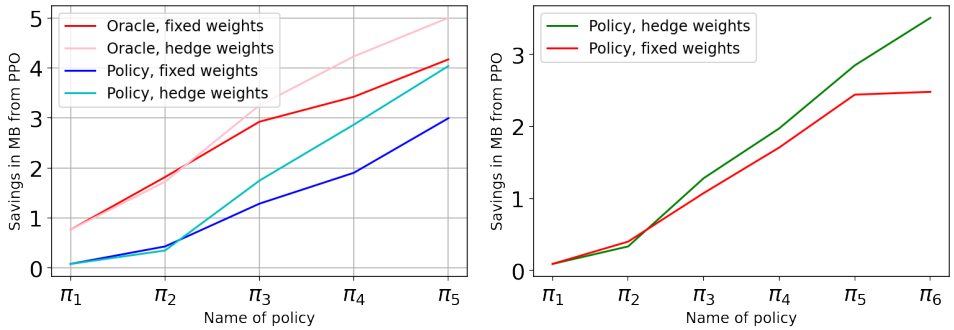
We describe our algorithm, `BC-MAX`, in Algorithm 1. Each iteration of our algorithm can be decomposed into two parts. First, select the best policy, among the set of baseline policies, $\mathcal{B}$, for each module in the corpus. For this, we collect the trajectory of each baesline policy (line 4), and record the highest reward policy π_{i_j} for module j (line 5). The second part in our algorithm learns a new policy to imitate this best per module policy. This is done by minimizing the weighted Behavior-Cloning (BC) loss in line 7. The newly learned policy is then added to the the set of baseline policies $\mathcal{B}$ in line 8 after which the algorithm begins a new iteration of compiling n modules and learning a new best policy. The maximum number of overall iterations is N. In Appendix B we provide a theoretical guarantee on the performance of the learned policy.

Computing the weights. The vanilla BC loss function weighs all module trajectories equally which leads to learning policies with small average error for the distribution of trajectories in the training dataset. When deployed, such policies may perform poorly on modules that have rarely been observed in the dataset. To combat this distribution shift problem we take a boosting style approach where we re-weight the BC loss so that we maximize the size savings over the worst case module in our dataset, that is we target small worst-case error rather than small average error. We do this as follows. At every iteration l of learning we know the performance of the last trained policy, $\hat{\pi}_{l-1}$, on every module which allows us to compute the relative performance of $\hat{\pi}_{l-1}$ to the best possible baseline policy for the respective module. The weights are then adjusted by increasing their value on modules where the relative performance of $\hat{\pi}_{l-1}$ is the weakest.

Online versus offline learning. Our theoretical setup frames the problem in an offline learning scenario, yet Algorithm 1 relies on our ability to interact with the environment in an adaptive manner. Note, however, that the modality of interaction used in our approach is quite different, and significantly more practical than full-fledged online RL. Each round of policy learning, i.e. solving the optimization problem in Equation 7, is fully offline. This process, which involves a large number ($10^5 - 10^6$) stochastic gradient steps, happens without any interaction with the environment, and is where the bulk of the learning happens. Subsequently, we form a new data collection policy for the next iteration, and this policy is applied to collect one trajectory per module. The data collection process does not involve any policy updates, and hence is massively parallelizable.

4 Empirical evaluation

We train and evaluate on two sets of binaries. In the first experiment we train on a proprietary search binary and evaluate the model on a different proprietary set of targets that are part of a cloud systems infrastructure. These targets need to be installed on a fixed size partition of each cloud machine and hence are size-constrained. In the second experiment we train and evaluate on the Chrome binary on Android. Here we only present results for Chrome on Android and defer the search application targets to Appendix D.3.



(a) Savings (MB) to PPO on sum of module sizes (b) Savings (MB) to PPO on binary size

Figure 1: Savings (MB) to PPO

The dataset collection begins by creating a corpus of IRs of modules which make up the final binary. The corpus creation follows the work of Trofin et al. [2021], Grossman et al. [2023], and the tools for extracting the corpus are available on GitHub[1]. The corpus is created at the beginning of training and remains the same throughout every iteration. Training begins under the assumption that there exists at least one base policy π_1. We track the improvements at each iteration of BC-MAX over π_1.

4.1 Chrome on Android

Training and test binaries are the same for this setting. The base policy with which we start is an RL policy trained using Proximal Policy Optimization (PPO[2]) [Schulman et al., 2017] as done in Trofin et al. [2021]. In practice our algorithm incorporates additional exploration when collecting trajectories in line 4, where we may deviate from the inlining action selected by a baseline policy at certain states. For details on how this is done we refer the reader to Algorithm 3 in the appendix. Further, we experiment with two types of weights, one is a fixed weighting of modules which does not depend on the performance of the current policy and the second is the boosting based approach called Hedge weights, which is described in the previous section. More details on how the weights are computed can be found in Appendix D.4. In Figures 1a and 1b, we plot the savings of our learned policies across iterations, relative to the initial PPO policy, measured in two different ways. For Fig 1a, we simply add up the sizes of the binaries produced by compiling each module in our training dataset. This is a clean metric, as the distribution shift between training and evaluation is small, and no artifacts from linker or post-inlining be optimizations are introduced in the evaluation. As we see, we improve rapidly beyond the PPO policy with the iterative applications of BC-MAX. Note that even the sum of module sizes suffers from the typical distribution shift between online and offline RL, since the data used from behavior cloning is collected using a different policy than the one we apply in evaluation. For the sum of module sizes metric, we can study the effect of this distribution shift rather carefully by also compiling with an *oracle* policy, which simply chooses the best baseline policy for each module, which is the target for training in BC-MAX. This oracle, shown in red in Figure 1a naturally provides larger gains relative to PPO than our learned policy as expected, but the gap reduces through the iterations of our process, indicating that the policies tend to stabilize through iterations, and the training data for later applications of BC-MAX is closer to on-policy data. We note that the oracle changes between different instantiations of our weights. This is because the i-th trained policy π_i depends on the choice of weights and so the oracle after the i-th iteration which chooses the best among $\{\pi_i\}_{\ell=1}^{i}$ also depends on the choice of weights. We also note that the gap between the learned and oracle policy's performance is smaller when we use the Hedge weights, and that the weighted version has a bigger gain relative to PPO, showing the efficacy of this approach. Finally, in Figure 1b we present the savings in size of the Chrome on Android binary, which is the actual yardstick. Here we cannot easily evaluate the size of the oracle, so we only compare our policies to PPO, and again observe impressive gains, with the Hedge-weighted variant doing better. The binary size when compiled with the PPO policy is approximately 213.32 MB.

[1] A detailed example can be found at https://github.com/google/ml-compiler-opt/blob/main/docs/inlining-demo/demo.md

[2] The policy can be found here: https://commondatastorage.googleapis.com/chromium-browser-clang/tools/mlgo_model2.tgz

References

André Barreto, Shaobo Hou, Diana Borsa, David Silver, and Doina Precup. Fast reinforcement learning with generalized policy updates. *Proceedings of the National Academy of Sciences*, 117 (48):30079–30087, 2020.

Ching-An Cheng, Andrey Kolobov, and Alekh Agarwal. Policy improvement via imitation of multiple oracles. *Advances in Neural Information Processing Systems*, 33:5587–5598, 2020.

Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

Aiden Grossman, Ludger Paehler, Konstantinos Parasyris, Tal Ben-Nun, Jacob Hegna, William Moses, Jose M Monsalve Diaz, Mircea Trofin, and Johannes Doerfert. Compile: A large ir dataset from production sources. *arXiv preprint arXiv:2309.15432*, 2023.

Teresa Johnson, Mehdi Amini, and Xinliang David Li. Thinlto: scalable and incremental lto. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 111–121. IEEE, 2017.

Aviral Kumar, Aurick Zhou, George Tucker, and Sergey Levine. Conservative q-learning for offline reinforcement learning. *Advances in Neural Information Processing Systems*, 33:1179–1191, 2020.

Nick Littlestone and Manfred K Warmuth. The weighted majority algorithm. *Information and computation*, 108(2):212–261, 1994.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

Dean A Pomerleau. Alvinn: An autonomous land vehicle in a neural network. *Advances in neural information processing systems*, 1, 1988.

Stéphane Ross and Drew Bagnell. Efficient reductions for imitation learning. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 661–668. JMLR Workshop and Conference Proceedings, 2010.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. Mlgo: a machine learning guided compiler optimizations framework. *arXiv preprint arXiv:2101.04808*, 2021.

Bernard Widrow. Pattern recognition and adaptive control. *IEEE Transactions on Applications and Industry*, 83(74):269–277, 1964.

Tengyang Xie, Ching-An Cheng, Nan Jiang, Paul Mineiro, and Alekh Agarwal. Bellman-consistent pessimism for offline reinforcement learning. *Advances in neural information processing systems*, 34:6683–6694, 2021.

Wenhao Zhan, Baihe Huang, Audrey Huang, Nan Jiang, and Jason Lee. Offline reinforcement learning with realizability and single-policy concentrability. In *Conference on Learning Theory*, pages 2730–2775. PMLR, 2022.

A Setting and Related work

We now define the problem setting formally, and then discuss some lines of prior work which are relevant to this setting.

A.1 Problem setting

Contextual MDP setting. We work in the finite-horizon setting and denote the horizon as H. The value function of a policy π is

$$V_\pi(x) = \sum_{h=1}^{H} \mathbb{E}_{a_h \sim \pi(\cdot|s_h)}[r_x(s_h, a_h)],$$

where s_h is the the state at step h s.t. $\mathbb{P}_x(s_h|s_{h-1}, a_{h-1}) = 1$ and $s_1 \equiv x$. Importantly, the policy class is such that the action distribution at state s depends only on s and not on the context, that is $\pi(\cdot|s, x) = \pi(\cdot|s)$. All of our algorithms and upper bounds can be extended to the setting where the policy depends on the context as well.

Goal. Let $V_i(x^1) = V_{\pi_i}(x^1)$ denote the expected cumulative reward of baseline i, conditioned on the context x. Then we seek to minimize the regret:

$$\mathrm{Reg}(\pi) := \mathbb{E}_{x \sim D}\left[\max_{i=1,2,\ldots,K} V_i(x) - V_\pi(x)\right]. \tag{1}$$

That is, we seek to compete with the best of the baselines for each individual context.

A.2 Related work

Learning setup. We assume access to the policy class Π, but *do not assume any other function approximators, such as for modeling value functions*. This is partly due to the fact that the typical training of value functions using Bellman backups is not feasible in our sparse-reward setting. Furthermore, standard actor-critic techniques make strong completeness and realizability assumptions on the value function class, which are not realistic with a restricted notion of state which we encounter in our motivating problem of compiler optimization. This necessitates the development of purely policy-based methods.

Vanilla behavior cloning Behavior cloning Widrow [1964], Pomerleau [1988] refers to the approach of learning a policy that matches the mapping from states to actions observed in the data. This is typically solved as a classification problem for deterministic policies, or by maximizing the log-likelihood of the chosen actions in the observed states for stochastic policies. It is unclear how to apply vanilla behavior cloning in the presence of multiple baselines. We will present a natural formulation to behavior clone the best baseline policy per context in the following section.

Value-based improvement upon multiple baselines (MAMBA) Cheng et al. [2020] show how to simultaneously improve upon multiple baseline policies to compete with the best policy *at each state* in the MDP, which is a significantly stronger notion that competing with the best baseline in each context only. However, this comes with two caveats. Their method requires value function estimation for the baselines and access to the MDP to execute trajectories under the learner's policy and/or baselines. Barreto et al. [2020] also study a problem which involves improving over multiple baseline policies, which they title General Policy Improvement. The policy improvement step again requires value function evaluation. We do not assume such access to additional function approximators or the MDP in this work.

Offline RL: Without access to the MDP, a natural approach is to consider offline reinforcement learning, with the data collection policy being a mixture of the baselines π_i, say chosen uniformly. Given the recent results on offline RL to compete with any policy that is covered by the data distribution [Kumar et al., 2020, Xie et al., 2021, Zhan et al., 2022], we can expect a favorable bound on the regret (1), since all the baselines have a good coverage under the uniform data collection policy. However, existing offline RL methods with theoretical guarantees are typically based on value function approximation, relying on actor-critic or Q-learning style approaches and on strong credit assignment using per timestep rewards rather than the aggregated reward of a trajectory. Applying these techniques using policy-based function approximation alone and with aggregated reward feedback is not feasible as we argue through a simple lower bound example.

B Vanilla BC-Max and Regret Bounds

We now describe our algorithm, BC-MAX, and give an upper bound on the regret it incurs to the best per-context baseline.

Algorithm. We describe BC-MAX in Algorithm 2. The basic idea of the algorithm is quite simple. For each context x_j in our dataset, we first choose the trajectory with the highest cumulative reward across all the baselines. Then we use a standard behavior cloning loss to mimic the choice of actions in this trajectory. For a context $x_j, j \in [n]$, we denote $i_j = \text{argmax}_{i \in [K]} r(\tau_{i,j})$, and BC-MAX tries to find a policy $\hat{\pi} \in \Pi$ that optimizes the following intuitive objective:

$$\hat{\pi} = \underset{\pi \in \Pi}{\text{argmin}} \sum_{j=1}^{n} \sum_{(s_{j,h}, a_{j,h}) \in \tau_{i_j,j}} \mathbf{1}(\pi(s_{j,h}) \neq a_{j,h}). \tag{2}$$

Algorithm 2 BC-MAX for cloning best per-context baseline.

Input: Base policies $\{\pi_i\}_{i \in [K]}$ and policy class Π.
Output: Policy $\hat{\pi} \in \Pi$.
  for $j \in [n]$ **do**
    Sample $x_j \sim D$, collect trajectories $\{\tau_{i,j}\}_{j \in [n]}$.
    Compute the highest reward policy $\pi_{i_j} = \text{argmax}_{i \in [K]} \sum_{(s_{j,h}, a_{j,h}) \in \tau_{i,j}} r(s_{j,h}, a_{j,h})$.
  $\hat{\pi} = \text{argmin}_{\pi \in \Pi} \sum_{j=1}^{n} \sum_{(s_{j,h}, a_{j,h}) \in \tau_{i_j,j}} \mathbf{1}(\pi(s_{j,h}) \neq a_{j,h})$.

One natural question at this point might be that if there are two trajectories with very similar high rewards in a context, can it help to leverage this information rather than only picking the one with the higher reward and cloning it. This is indeed a shortcoming of BC-MAX, and other behavior cloning style approaches. However, we note that we only have access to a trajectory-level reward, and hedging between two very different trajectories can create a very noisy learning setup for the algorithm. In situations where value-functions can be feasibly learned, such information is naturally modeled through the value function which assigns similar future rewards to similarly good actions, but we do not find a natural way for incorporating this information in our setup.

Performance guarantee for BC-Max. We now give a bound on the suboptimality of the policy learned by BC-MAX, relative to the best per-context baseline, in terms of the rewards. The analysis mirrors the standard results for behavior cloning algorithms [Ross and Bagnell, 2010]. We begin with a realizability assumption which governs how well the best per-context baseline can be approximated using the learner's policy class Π.

Assumption B.1. Let $\tau^*(x) = \text{argmax}_{\tau_{\pi_i}(x)} r(\tau_{\pi_i}(x))$ be the trajectory with maximum return over all policies $\pi_i, i \in [K]$. There exists $\pi^* \in \Pi$ such that $\mathbb{P}_{x \sim D}(\tau_{\pi^*}(x) \neq \tau^*(x)) \leq \epsilon$.

Here $\mathbb{P}_{x \sim D}(A)$ denotes the probability of an event A under the distribution D, which we recall is the distribution over the contexts x. The assumption is natural as BC-MAX cannot do a good job of approximating the best per-context baseline when no policy in the policy class has a small error in achieving this task. Note that the assumption does not take rewards into account as BC-MAX only matches the actions of $\tau^*(x)$, and does not reason about the reward sub-optimality of other actions, as is common in behavior cloning setups. Indeed this assumption is unavoidable in our problem setting as we illustrate in the next section.

Theorem B.2. *Under Assumption B.1, after collecting n trajectories from each of the K base policies, Algorithm 2 returns a policy $\hat{\pi}$ with regret at most*

$$\textit{Reg}(\hat{\pi}) \leq O\left(\epsilon H + \frac{H^2 \log(H|\Pi|/\delta)}{n}\right),$$

with probability at least $1 - \delta$.

Proof of Theorem B.2. Recall the definitions of $\tau^*(x)$ from Assumption B.1, and let $\pi^* = \text{argmin}_{\pi \in \Pi} \mathbb{E}_{x \sim D}(\sum_{h=1}^{H} \mathbf{1}(\pi(s(x)) \neq a(x)))$ where $(s_h(x), a_h(x))_{h=1}^{H} = \tau^*(x)$ form the

best trajectory for x among the baseline policies. Under Assumption B.1, we know that $\mathbb{E}_{x\sim D}(\sum_{h=1}^{H} \mathbf{1}(\pi(s(x)) \neq a(x))) \leq \epsilon H$. Let us define

$$\hat{A}(\pi) = \sum_{j=1}^{n}\sum_{h=1}^{H} \mathbf{1}(\pi(s_{j,h}) \neq a_{j,h}),$$
$$A(\pi) = \mathbb{E}_x\left(\sum_{h=1}^{H} \mathbf{1}(\pi(s_h(x)) \neq a_h(x))\right).$$

Clearly we have that $\mathbb{E}[\hat{A}(\pi)] = A(\pi)$ for any fixed policy π, and $\hat{A}(\pi) = \sum_{j=1}^{n} Z_j(\pi)$ with $Z_j(\pi) = \sum_{h=1}^{H} \mathbf{1}(\pi(s_{j,h}) \neq a_{j,h}) \geq 0$. We note that the Z_j are i.i.d., with $\mathbb{E}[Z_j(\pi)] = A(\pi)$ and $\mathbb{E}[Z_j(\pi)^2] \leq H\,\mathbb{E}[Z_j(\pi)] = H\,A(\pi)$. Then by Bernstein's inequality combined with a union bound over policies, we have with probability at least $1-\delta$, for all $\pi \in \Pi$:

$$\begin{aligned}|\hat{A}(\pi) - nA(\pi)| \leq& \sqrt{nH\,A(\pi)\log(2|\Pi|/\delta)} + H\log(2|\Pi|/\delta)\\ \leq& \frac{nA(\pi)}{2} + \frac{3}{2}H\log(2|\Pi|/\delta).\end{aligned}$$

Applying the inequality with $\pi = \hat{\pi}$ and $\pi = \pi^*$, we obtain

$$\begin{aligned}A(\hat{\pi}) \leq& \frac{2}{n}\hat{A}(\hat{\pi}) + \frac{3H\log(2|\Pi|/\delta)}{n}\\ \frac{1}{n}\hat{A}(\pi^*) \leq& \frac{3}{2}A(\pi^*) + \frac{3}{2}\frac{H\log(2|\Pi|/\delta)}{n}.\end{aligned}$$

Scaling the second inequality by 2 and adding them yields

$$A(\hat{\pi}) \leq 3A(\pi^*) + \frac{6H\log(2|\Pi|/\delta)}{n} \leq 3\epsilon + \frac{6H\log(2|\Pi|/\delta)}{n}, \tag{3}$$

where the second inequality follows by Assumption B.1.

Now we note that for any policy π:

$$\begin{aligned}\mathrm{Reg}(\pi) =& \mathbb{E}_x[\max_i V_i(x) - V_\pi(x)] = \mathbb{E}_x[r(\tau(x)) - r(\tau_\pi(x))]\\ \leq& \sum_{h=1}^{H}(H-h)\,\mathbb{E}_x[\mathbf{1}(\pi(s_h(x)) \neq a_h(x))|].\end{aligned}$$

Plugging the bound from Equation 3 into the inequality above completes the proof. □

C Necessity of the comparator choice

In this section, we show an example which explains why it is necessary to restrict the comparator class to the max over baseline policies.

Let us consider a contextual multi-armed bandit problem, meaning that we fix $H = 1$. For any ϵ, we choose the context space $\mathcal{S} = [M]$ for $M = \lceil \frac{1}{\epsilon} \rceil$, and choose D to be the uniform distribution on $\mathcal{S}$. We fix $\mathcal{A} = \{a_1, a_2, a_3\}$, and $K = 1$, with the data collection policy π_1 choosing $a = a_1$ for each context $x \in \mathcal{S}$. We consider two possible environments, defined through rewards r_1, r_2. For a_1, we have $r_1(x, a_1) = r_2(x, a_1) = 1$. For the other two actions, we have $r_1(x, a_2) = 0$ and $r_1(x, a_3) = 1$, while the second environment has $r_2(x, a_2) = 1$ and $r_2(x, a_3) = 0$. We design a policy class π with two policies $\{\pi_1, \pi_2\}$ such that $\pi_1(x) = \pi_2(x) = a_1$ for all $x \neq 1$ and $\pi_1(1) = a_2$, $\pi_2(1) = a_3$. Clearly this policy class satisfies Assumption B.1. But it also contains an optimal policy for both the rewards r_1, r_2 with a regret equal to 0. However, since the data contains no information about which one of r_1 or r_2 generated the data, the best we can do is to pick between π_1 and π_2 uniformly at random, and incur a regret of at least 0.5ϵ. This argument shows that we cannot replace the $0-1$ loss for measuring the accuracy of a policy in Assupmtion B.1, with a more reward-aware quantity. It is also evident from the example that we cannot avoid incurring a regret of $\Omega(\epsilon)$.

D Case study: Optimizing a compiler's inlining policy

D.1 The inlining for size problem

In short the inlining problem which we study in our experiments consists of deciding to inline or not to inline a callsite in a program with the goal of minimizing the size of the final program binary. We omit most compilation details and just give a brief overview which should be sufficient for understanding the problem from a RL perspective. In our specific scenario, compilation is split into a frontend (`fe`) and a backend (`be`). The frontend consists of translating the program into an Intermediate Representation (IR), doing some frontend optimizations, then a (thin) link step follows, which re-organizes functions in the various modules to improve inlining opportunities. For more details on the linking step see Johnson et al. [2017]. The backend compilation follows the thin link step and is applied on the updated modules. It consists of further optimizations, including inlining decisions, final linking and lowering the IR to machine code, e.g., x86, ARM, etc. The IRs with which our RL algorithms work with are post frontend linking and pre backend optimization, that is we work only on backend optimizations. We note that a program is made up of multiple *modules*. In the `fe`, a module corresponds to a single C/C++ source file, after all the preprocessor directives have been applied. In the `be`, the module consists of a mix of IRs from different `fe` modules. The inlining decisions are taken at callsites in the IRs of each module, where the callee is also in the same module. We note that each module is ultimately compiled to a machine code-specific binary, with its own size, that is further linked into the final executable. Hence we treat each module as a context x in our contextual MDP setting, and the value $V_i(x)$ of the baseline policy π_i is the size of the binary we get for module x, when we make inlining decisions for the module according to π_i.

$$\begin{aligned} &\texttt{Program} \rightarrow \texttt{IRs} \rightarrow \texttt{fe optimizations} \\ \rightarrow &\texttt{ThinLinking}^{3} \xrightarrow{\texttt{Collecting IRs}} \texttt{be optimization} \\ \rightarrow &\texttt{Final linking} \rightarrow \texttt{x86} \end{aligned} \tag{4}$$

In Equation 4 we outline the compilation process, together with the step at which we collect IRs from the respective modules to be used in our RL algorithms. It is important to note that the learned RL policy makes inlining decisions both at the `fe` optimization and `be` optimization parts in Equation 4, however, the IRs for training are only collected after the `fe` optimization step.

The contextual MDP setting can now be tied together with the compilation process as follows. Each context is a module as mentioned above, with state-space defined by its IR. Each state corresponds to a callsite in the IR of the module x, and the action set is $\{\texttt{inline}, \texttt{ don't inline}\}$ or $\{1, 0\}$ respectively. Each π_i is some base inlining policy and $V_i(x)$ is defined by the size of the compiled stand-alone module x. It is important to note that there is a mismatch between trying to maximize $\mathbb{E}_{x\sim D}[V_\pi(x)]$ and the overall goal of minimizing the binary size, as it is not necessarily true that the sum of module sizes equals the size of the binary. In fact, part of the post-inlining `be` and linker optimizations may introduce a significant distribution shift between the sum of module sizes and the size of the final binary. In our experiments, we try to minimize this distribution shift by turning off certain optimizations. For more details on the compilation pipeline we refer to Trofin et al. [2021].

We note that the entire process is fully deterministic, as we assumed in our theoretical setup, since the compiler is a deterministic program.

D.2 Dataset collection

We train and evaluate on two sets of binaries. In the first experiment we train on a proprietary search binary and evaluate the model on a different proprietary set of targets that are part of a cloud systems infrastructure. These targets need to be installed on a fixed size partition of each cloud machine and hence are size-constrained. In the second experiment we train and evaluate on the Chrome binary on Android. Training proceeds in two separate steps, which we repeat over several iterations. The two steps can be summarized as follows, first we collect a training dataset which consists of trajectories with smallest size over all base policies available at the current iteration. Next, we train a new base model using the objective defined in Equation 2. This conceptually applies Algorithm 2 repeatedly, where the set of baseline policies is updated at each iteration to include the new policy obtained from the previous iteration. We now describe each step carefully.

[3] See Johnson et al. [2017]

Training dataset collection. The dataset collection begins by creating a corpus of IRs of modules which make up the final binary. The corpus creation follows the work of Trofin et al. [2021], Grossman et al. [2023], and the tools for extracting the corpus are available on GitHub[4]. The corpus is created at the beginning of training and remains the same throughout every iteration. Training begins under the assumption that there exists at least one base policy. In the first iteration a training dataset is collected from this initial base policy π_1. Next, π_1 is behavior cloned by solving the optimization problem in Equation 2, where the setting of example weights is described shortly. Let $\hat{\pi}_2$ denote the resulting policy. This policy is non-deterministic and so we construct the base policy π_2 by setting $\pi_2(s) = \text{argmax}_{a\in\{0,1\}} \hat{\pi}_2(s, a)$, that π_2 always plays the most likely action according to $\hat{\pi}_2$. This concludes the first iteration. More generally, if we have a larger initial set of baseline policies than just a singleton$\{\pi_1\}$, the iterations proceed similarly. However, instead of just using π_1, we use the full set $\{\pi_i\}_{i\in[K]}$ of baseline policies at every iteration at the first iteration.

Proceeding this way, at the t-th iteration the set of base policies is taken as a subset of $\{\pi_1, \ldots, \pi_{t-1}\}$ which always contains π_1 (or the larger set of all initial baselines). Then we again invoke BC-MAX with these baseline policies, and obtain a new randomized policy $\hat{\pi}_t$, and we refer to π_t as the corresponding deterministic greedy policy. When collecting a new training dataset we not only collect trajectories with the chosen subset of base policies but we also may force exploration by using $\hat{\pi}_{t-1}$ in the way discussed next.

Exploration in training dataset collection. For a fixed module x and a policy π, we choose a ceiling on the number of exploration steps as a hyper parameter, which is a fraction of the length of the trajectory $|\tau_{\pi_1}(x)|$. The call-sites at which exploration occurs are selected as follows. The first exploration call-site is selected as $\tilde{h} = \text{argmin}_h\{|\hat{\pi}(s_h)(0) - \hat{\pi}(s_h)(1)|\}_{s_h\in\tau_\pi(x)}$, as the call-site where the exploration policy $\hat{\pi}$ is the least confident about the action to choose. The exploration step is then played at $s_{\tilde{h}}$ by taking the action $1-\pi(s_{\tilde{h}})$ (recall that $\pi(s) \in \{0, 1\}$), and the remaining steps in the trajectory are completed by playing according to π. Let $\hat{\tau}$ denote this exploration trajectory. In the following exploration round, the exploration step is selected as the step h at which the gap, $|\hat{\pi}(s_h)(0) - \hat{\pi}(s_h)(0)|_{s_h\in\hat{\tau}}$, is smallest among all $h > \tilde{h}$, where $\tilde{h}$ is the exploration step in the previous round. Once the maximum number of exploration rounds is reached or the exploration step reaches the end of the trajectory, we return the trajectory which results in the smallest module size among all explored trajectories. Pseudo-code is presented in Algorithm 3.

Algorithm 3 Explore module

Input: Base policy π, exploration policy $\hat{\pi}$, module x, maximum exploration steps T.
Output: Compilation trajectory $\tau_\pi(x)$ with reward $r_{\pi,x}$.

```
Compute vanilla trajectory τ_π(x) by compiling with π and receive reward r^1_{π,x}
t = 1, τ̂_1 = τ_π(x), h̃_1 = argmin_h{|π̂(s_h)(0) − π̂(s_h)(1)|}_{s_h∈τ̂_1}
while t ≤ T do
    Replay τ̂_t until h̃_t,
    Play 1 − π(s_{h̃_t}) at h̃_t. Complete trajectory τ̂^{t+1} by playing π. Receive reward r^{t+1}_{π,x}
    if h̃_t < |τ̂_{t+1}| then
        h̃_{t+1} = argmin_{h>h̃_t} {|π̂(s_h)(0) − π̂(s_h)(1)|}_{s_h∈τ̂^{t+1}}
    else
        break
t* = argmax_t r^t_{π,x}, r_{π,x} = r^{t*}_{π,x}, τ_π(x) = τ̂_{t*}
```

Online versus offline learning. Our theoretical setup frames the problem in an offline learning scenario, yet Algorithm 3 and the iterative procedure do rely on our ability to interact with the environment in an adaptive manner. Note, however, that the modality of interaction used in our approach is quite different, and significantly more practical than full-fledged online RL. Each round of policy learning, which happens using BC-MAX, is fully offline. This process, which involves a large number ($10^5 - 10^6$) stochastic gradient steps, happens without any interaction with the environment, and is where the bulk of the learning happens. Subsequently, we form a new data

[4]A detailed example can be found at https://github.com/google/ml-compiler-opt/blob/main/docs/inlining-demo/demo.md

collection policy for the next iteration, and this policy is applied to collect one trajectory per module. The data collection process does not involve any policy updates, and hence is massively parallelizable with no interlocking bottlenecks with the learning process. In online RL, on the other hand, data collection and policy updating go hand-in-hand, which typically requires significantly more complex architecture [Mnih et al., 2016] to scale to domains where data collection is expensive. Our approach, on the other hand, simply requires interleaving standard supervised learning and batch data collection, which is quite desirable especially in the compiler application, where the ML training happens on GPUs, while the compilation happens on CPU machines.

D.3 Search application targets

Similarly to Trofin et al. [2021] we collect a corpus for training purposes from a search application binary with approximately 30000 modules. The initial base policy is an RL model trained using an Evolutionary Strategy (ES[5]) as in Trofin et al. [2021]. For the training dataset with the ES policy, the distribution of sizes of modules is fairly non-uniform, with few modules having very large sizes or very small sizes and majority of modules being somewhere in-between. Because we expect that the actions of the behavior cloning policy taken on larger size modules are more important for size saving we upweight the actions in such trajectories. The weights used in training are computed as follows. Let $\texttt{size}(x, \pi_1)$ denote the size of module x from the collected trajectory under policy π_1 (or in the case of multiple baseline policy under the best baseline policy). The modules are partitioned into buckets according to their sizes where the limits of the buckets are taken to be on exponentially scaling grid, that is the first bucket contains all modules with size $\texttt{size}(x, \pi_1) \in [0, 2^0)$, the second bucket all modules such that $\texttt{size}(x, \pi_1) \in [2^0, 2^1)$ etc., up to the final bucket with size $[2^{M-1}, 2^M)$. Let $b_m = \{x : \texttt{size}(x, \pi_1) \in [2^{m-1}, 2^m)\}$ denote the m-th bucket and let $m(x)$ be the m for which $x \in b_{m(x)}$. The weight w_x for module x is computed as follows $w_x = \frac{\max_m |b_m|}{|b_{m(x)}|}$.

Algorithm 4 `BC-Max` for cloning best per-context baseline with exploration

Input: Base policy π_1 and policy class Π. Max exploration steps T. Max number of iterations N.
Output: Policy $\hat{\pi} \in \Pi$.

$l = 1$
while $l \leq N$ **do**
  for $j \in [n]$ **do**
    Sample $x_j \sim D_1$
    $r_{i,j}, \tau_{i,j} = \texttt{Algorithm 3}(\pi_i, \hat{\pi}_\ell, x_j, T), \forall i \in \{\pi_s\}_{s \leq l}$
    Compute highest reward policy $\pi_{i_j} = \text{argmin}_{i \in [K]} \sum_{(s_{j,h}, a_{j,h}) \in \tau_{i,j}} r_{i,j}(s_{j,h}, a_{j,h})$.
  Compute module weights $\{w_j\}_{j \in [n]}$ (See Sect. D.3,4.1).

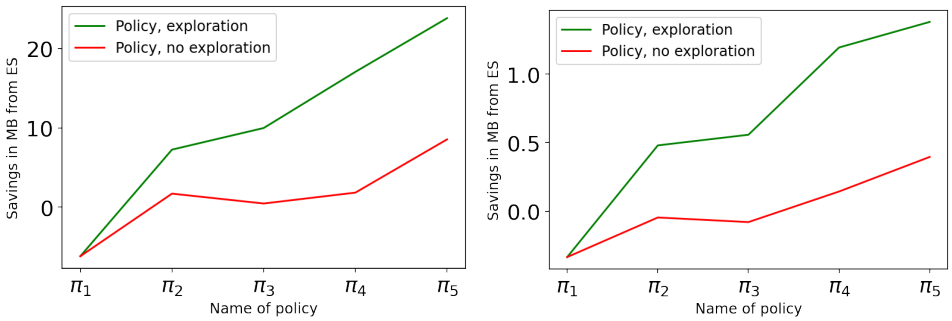
$$\hat{\pi}_l = \underset{\pi \in \Pi}{\text{argmin}} - \sum_{j=1}^{n} w_j \sum_{(s_{j,h}, a_{j,h}) \in \tau_{i_j,j}} a_{j,h} \log(\pi(a_{j,h}|s_{j,h}))$$

$\pi_l(s) = \text{argmax}_a \hat{\pi}_l(a|s), \forall s \in S$

We train two sets of policies, one set is trained without exploration and is precisely in line with Algorithm 2. For full pseudo-code, which includes the exploration step, we refer the reader to Algorithm 4. The second set is trained with exploration as described in Section D.2. This way, the first policy provides an ablation for the value of the exploration strategy.

In Figure 2a we show savings of the trained policies to π_1, which is the ES policy, on the search binary from which the training dataset is collected. In Figure 2b we show the savings on a *different test binary*, which is part of a cloud systems infrastructure. On the x-axis of the figures we show the size savings of the policy π_i learned at each iteration i, with and without exploration respectively, where bc_0 is the behavior cloned policy from ES. Both figures demonstrate the success of our approach in improving significantly beyond the initial baseline, as well as the benefits from multiple iterations of the process. Furthermore, the gap between the lines with and without exploration highlights the benefits of the added exploration.

[5]The policy can be found here: https://github.com/google/ml-compiler-opt/releases/tag/inlining-Oz-v1.1



(a) Savings in MB to ES on training binary (b) Savings in MB to ES on test binary

Figure 2: Savings in MB to ES

We note that the compilation for both the training and test binaries is carried out in the following way to minimize the distribution shift – the `fe` optimizations are carried out by ES, while the `be` optimizations are carried out by the trained policies. If we were to use the trained policies in both `fe` and `be`, this might lead to significant distribution shift, as Algorithm 2 works only on trajectories collected after the `fe` optimizations for which ES is always used. That is, if any of the trained policies, bc_i, act very differently on the `fe`, compared to ES, the resulting IRs before the `be` optimization might be completely different from the training set IRs, and hence the trained policy might take very sub-optimal actions.

D.4 Chrome on Android

In our second set of experiments we train an RL policy for Chrome on Android. The training and test binaries are the same in this case. The base policy with which we start is an RL policy trained using Proximal Policy Optimization (PPO[6]) [Schulman et al., 2017] as done in Trofin et al. [2021]. There are two differences in training from Section D.3. First, we focus only on the setting where we do exploration. The second difference in training is how the weights for the objective in Equation 2 are formed. The approach for computing the weights used here is inspired by the fact that we want to improve on PPO in each module and not just on the sum of module sizes. That is we want to maximize the size savings over the worst case module in our dataset. The following approach is natural when such max-min guarantees are desired.

Reusing notation from Section D.3 we let $p^1_x = \frac{|b_{m(x)}|}{\sum_m |b_m|}, w^1_x = \frac{\max_m p^1_x}{p^1_{m(x)}}$, be the weights in the first iteration of training. In following iterations the weights are set as $w^t_x = \frac{\max_m p^t_x}{p^t_{m(x)}}$, where p^t is updated using the Hedge algorithm [Littlestone and Warmuth, 1994], and is inspired by connections with boosting [Freund and Schapire, 1997]. The update uses the sum of sizes in each bucket as losses, normalized by the ℓ-infinity norm, that is

$$\tilde{L}^t_m = \sum_{x \in b_m} \texttt{size}(x, \pi_t), L^t_m = \frac{\tilde{L}^t_m}{\|L^t\|_\infty},$$

where L^t_m denotes the m-th coordinate of the loss vector L^t. The Hedge update is then

$$\tilde{p}^{t+1}_m = p^t_m \exp(-\eta L^t_m), p^{t+1}_m = \frac{\tilde{p}^t_m}{\sum_m \tilde{p}^t_m}.$$

In Figures 1a and 1b, we plot the savings of our learned policies across iterations, relative to the initial PPO policy, measured in two different ways. For Fig 1a, we simply add up the sizes of the binaries produced by compiling each module in our training dataset. This is a clean metric, as the distribution shift between training and evaluation is small, and no artifacts from linker or post-inlining

[6]The policy can be found here: https://commondatastorage.googleapis.com/chromium-browser-clang/tools/mlgo_model2.tgz

be optimizations are introduced in the evaluation. As we see, we improve rapidly beyond the PPO policy with the iterative applications of BC-MAX. Note that even the sum of module sizes suffers from the typical distribution shift between online and offline RL, since the data used from behavior cloning is collected using a different policy than the one we apply in evaluation. For the sum of module sizes metric, we can study the effect of this distribution shift rather carefully by also compiling with an *oracle* policy, which simply chooses the best baseline policy for each module, which is the target for training in BC-MAX. This oracle, shown in red in Figure 1a naturally provides larger gains relative to PPO than our learned policy as expected, but the gap reduces through the iterations of our process, indicating that the policies tend to stabilize through iterations, and the training data for later applications of BC-MAX is closer to on-policy data. We note that the oracle changes between different instantiations of our weights. This is because the i-th trained policy π_i depends on the choice of weights and so the oracle after the i-th iteration which chooses the best among $\{\pi_i\}_{\ell=1}^{i}$ also depends on the choice of weights. We also note that the gap between the learned and oracle policy's performance is smaller when we use the Hedge weights, and that the weighted version has a bigger gain relative to PPO, showing the efficacy of this approach.

Finally, in Figure 1b we present the savings in size of the Chrome on Android binary, which is the actual yardstick. Here we cannot easily evaluate the size of the oracle, so we only compare our policies to PPO, and again observe impressive gains, with the Hedge-weighted variant doing better. The binary size when compiled with the PPO policy is approximately 213.32 MB.