A Proposal for Policy Decisions and Phase Definitions

Ariel Hoffman

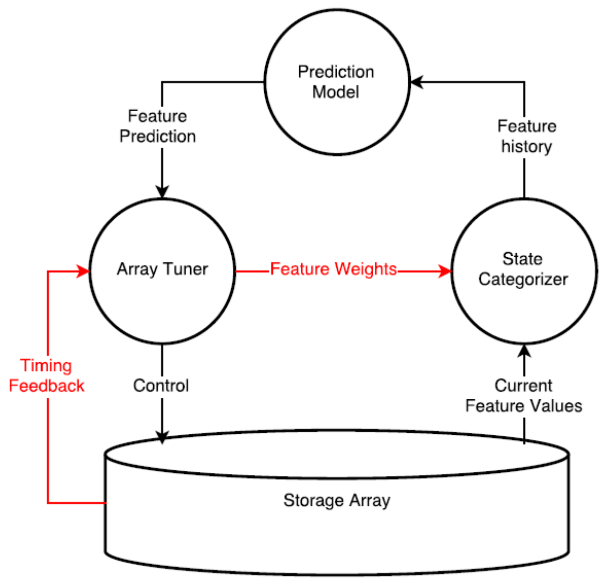
1 The Big Picture

The main objective of my project, as I understand it, is to determine a way to dynamically optimize a storage array by predicting workload patterns. The system we have been talking about until now is described in Figure 1.1 (sans the red lines). Basically, the current state of the storage array is evaluated by the state categorizer (using something like K-means clustering or LoadIQ). The state categorizer tells the prediction model the current state. The prediction model stores previous states and the current state and then, using some prediction scheme (like C-Miner, Markhov Chains, or some combination) it guesses about what the next state's workload will look like. It then passes that information to the array tuner as a set of "features." The array tuner uses the predicted features to determine how it should tune the storage array. Currently the set of "features" is exclusively the LBA access pattern in the form of a histogram.

This paper makes two proposals:

1. A proposal for an unsupervised storage array tuner that uses Temporal Difference Learning to determine what the best policies should be by accounting for live timing feedback.
2. A proposal for an autonomous state categorizer that uses feedback from the array tuner in order to categorize states based on the features that actually matter most to performance.

Figure 1.1: Optimizer Diagram



The key to understanding this proposal is to remember that the features stay the same throughout this diagram, but the abstract definition of "state" can get better as the array tuner provides better and better feedback about the importance of each feature. The array tuner decides what features are important based on the red feedback line and then it sends that information to the state categorizer. The state categorizer then uses all the information it has collected about previous states and it redefines "distance" based on the feature weights provided by the array tuner. As a result, the prediction model will get better at providing data to the array tuner (because its states will be defined based on the important features) which will, in turn, become more precise.

The main reason I think we might want to pursue this avenue of thought is that the current system only examines the LBA access histogram in order to create the states on which everything is based. I think there may be other features that would be worth evaluating besides the LBA access histogram when it comes to tuning the array. The mechanism outlined herein allows us to naively give the system data and then the system itself can be smart enough to determine the most important features.

If this works, the main benefits would be as follows:

1. A low-maintenence learning system that can determine on its own what features are performance sensitive and what features are not. It can tune itself so that the whole system is sensitive to those features as well.
2. Possibly better performance than only looking at LBA histogram.

The main risks that I can see currently are as follows:

1. One risk is that perhaps the LBA histogram *is* the only thing that matters. If that is the case then we won't get any big performance improvement.
2. It might turn out that the training phase takes too long. That said, our current setup is essentially this system with a single feature. Therefore, I believe that this system will not be any worse than what we would be producing otherwise.
3. The main proponent of this idea (myself) is still in school and has significantly less experience than most people in this field. Therefore, there is a significant possibility that there are aspects of the problem that haven't been accounted for.

The rest of this paper will be devoted to detailing the technical portions of the idea. Section 2 will provide a brief introduction to the math behind temporal difference learning (which is what I plan to use in the array tuner). Section 3 will show how that math applies to the cache by defining the stage and the cost in context and giving some examples of features. Section 4 is a brief non-technical discussion on providing feedback to the state categorizer and why that is important. Section 5 is a psuedocode description of the most important algorithms.

2 A Brief Introduction to Temporal Difference Learning

I learned about temporal difference learning in the context of the game Tetris. We used it to train a program to play the game when it became too complicated to solve using standard dynamic programming.

Normally in dynamic programming, we use the optimal "cost to go" in order to determine what the optimal next state should be. We can do this because the total cost will be the optimal "cost to go" from the next state plus the transition cost from this state.

$$J_k^*(x_k) = g(x_k, u_k) + J_{k+1}^*(x_{k+1}) \tag{1}$$

We can get the minimal cost by starting from the end states and working backwards. Our end state must be associated with a set cost because there are no more actions to be had. Therefore,

$J^*_N(x_N) = g(x_N)$. If the number of possible x_N is small enough, we can run through this program for all possible values and work backwards for each of them. However, in games like Tetris and in our system, the reward (-cost) to go is impossible to calculate because the state space is too large. That said, we can create an estimated reward to go by using features and weightings.

If I am trying to create features for Tetris, I might use things like the height of the highest column or the difference between the height of the highest column and the hight of the lowest column. When we're evaluating the state of the cache simulator, we will have to make features that include information about the policies and the contents of the trace. If we're feeling particularly adventuresome, we might even include information about the cache contents itself, though that seems unnecessary at this point.

Supposing we create a number of these features, we can store them in a vector. We can call this vector ϕ^T. Then, in order to approximate $J(x_k)$ (the cost or reward to go) we can use the following equation where $\phi^T(x_k)$ represents the feature vector for the state, x_k and r represents a vector of predefined feature weights:

$$J(i) = \phi^T(i)r \tag{2}$$

Naturally, the first model we pick will not necessarily be the best model we come up with. Once we have our initial definition of the cost to go, we can start using real temporal difference learning to modify that model. First, we calculate the temporal difference. The temporal difference is simply the difference between our estimated transition cost (the estimated cost to go at the current state minus the estimated cost to go at the next state) $\phi^T(x_k)r_k - \phi^T(x_{k+1})r_k$ and our real transition cost, $g(x_k, u_k)$.

$$d_k = g(x_k, u_k) + \phi^T(x_{k+1})r_k - \phi^T(x_k)r_k \tag{3}$$

(Note well, if we had a perfect model, this value would always be zero.) Once we have calculated the temporal difference, we merely need to push the feature weights in the correct direction to make the model more accurate. We do this by taking the unit direction of the feature vector as follows:

$$z_t = \frac{\phi(x_t)}{||\phi(x_t)||} \tag{4}$$

We use the following equation to modify the feature weight vector:

$$r_{t+1} = r_t + \gamma_t d_k z_t \tag{5}$$

Where γ_t is simply a tunable decaying constant. We don't want the later errors to have as large an impact as the earlier ones. Often times this decaying constant is $\frac{1}{\sqrt{k)}}$, where k is the stage count.

When we're done training a system, we should end up with a set of r values (feature weights) and a bunch of predefined features that allow us to quickly decide what the next best state should be (and therefore, which policies we should implement.)

3 Applications in Caching

3.1 Stages

Every stage can be defined as an interval of commands. For now, lets assume the number of commands in each stage is q. Thus, the commands issued between 0 and q will be part of stage 0 and the commands issued between q and $2q$ will belong to stage 1 and so on. We will refer to the current stage as stage k. If c represents total number of commands issued until now, the current stage can therefore be calculated as follows:

$$k = floor(\frac{c}{q}) \tag{6}$$

3.2 Features

The features can be a number of things we think might be important multiplied by the setting on the policy we think it might impact. In the real system, I think we should have a lot of features, but for the purposes of this illustration, I'll stick to two.

Suppose we have the option to turn the read cache on or off. One thing that might decide whether that will be a good idea is the read percentage. If we suppose that h is the read ratio, p is either 1 or 0 depending on whether the cache is on or not, we could define our feature as follows:

$$f_1 = hp \tag{7}$$

In a real scenario, we might guess that the write percentage also has an impact on whether we should turn on that feature. By adding another feature specifically for the write percentage, we are now able to balance them against one and other. Thus, our next feature will be

$$f_2 = mp \tag{8}$$

3.3 Cost

Because we are working directly with a real system, we can define cost as the time it takes to complete the phase. If u_k is our selected action, then that means

$$g(x_k, u_k) = time \tag{9}$$

3.4 Example Iteration

If we have trained LoadIQ correctly, we will have preconceived notions about how many hits or misses are in the next phase. Suppose LoadIQ predicts that our next phase will have 70% reads and 30% writes. Using policy control, we have two possible next states: either we turn on the read cache or we turn off the read cache. We can calculate an approximate time to completion using our features and our weightings as follows: $J_k^* = \phi(x_{k+1}) = \min_p([f_1, f_2]^T * [r_1, r_2])$
So, the algorithm makes the following decision:

$$p = \begin{cases} 1 & hpr_1 + mpr_2 > 0 \\ 0 & hpr_1 + mpr_2 \leq 0 \end{cases} \tag{10}$$

It is worth noting again that the above solution does not necessarily produce the optimal value. However, by tuning the algorithm, we can hopefully get closer to the optimal value. We can tune the algorithm using feedback from the system once we get it. This process involves three steps:

1. Calculate $\phi^T(x_{k+1})$: because we time has passed and we now know what x_{k+1} is with certainty, we can calculate the feature vector for it.
2. Record $g(x_k, u_k)$: this is simply the time to complete the phase.
3. Calculate d_k: we have already computed $\phi^T(x_k)$, so computing d_k is merely a matter of using equation 3.
4. Calculate z_t using equation 4.
5. Calculate the new r_{t+1} using equation 5.

We can do this process again and again as we get more and more phases until we get a final solution.

4 Providing Feedback to the State Categorizer

The implementation of the state categorizer doesn't matter as much to me. It could be using K-Means clustering or it could be using the SVMs. All that matters to me at this point is that it be autonomous and that the definition of distance incorporates the feature weights calculated by the temporal difference learning algorithm. Features that are more important for determining performance should be given greater weight in terms of the distance calculation.

So, for instance, suppose we initially provided our temporal difference learning algorithm access to the number of times our workload accessed a specific data point. However, suppose that our algorithm decided that the feature didn't actually matter and it assigned the value a weight of 0. Then, if we've defined "distance" so that it depends on the weight, then two stages which have identical feature sets except that one feature would be defined as the same state because the distance would not depend on that feature anymore. This would be valuable down the line because the predictive algorithm would more accurately predict other more important features and less accurately predict that feature. Then, because the tuner has access to better data, it should be able to make better decisions on tuning.

5 Big Picture Pseudocode

This section provides a clearer insight to the big picture system. This psuedocode is not about the temporal difference learning, but rather the overall mechanism described in figure 1.1.

```
// Initial values
// My guess is that these will be a vector of 0s and 1s (i.e. on or off)
policyChoices = array of vectors of policy choices
// see the discussion of features for a more detailed description
featureMatrix = matrix of features associated with each policy
expertWeightings = initial value
phaseModel = initial value
phasePredictionModel = initial value

// This function makes a guess about what the optimal next policy should be.
// The current feature matrix is known - it comes from the state
// of the cache or the state of the array.
// The estimated next feature matrix will have to be determined by
// our predictive algorithm based on our phase definition.
function makeGuess(currentFeatureMatrix, estimatedNextFeatureMatrix):
        // This is deterministic - it's based on known features of the
        // cache and the trace we've already seen.
        currentCostToGo = calculateCurrentFeatureMatrix(this.cacheState)* \
                this.cacheState.currentPolicies

        // We want to find our minimum cost estimate.
        minimumEstimate = MAX_NUM
        for policyChoice in policyChoices:
                expertGuesses = estimatedNextFeatureMatrix*policyChoices
                estimatedCostToGo = expertGuesses*expertWeightings
                // Store whichever is smaller.
                if(estimatedCostToGo < minimumEstimate):
                        minimumEstimate = estimatedCostToGo
                        minimumPolicy = policyChoice
        assert minimumPolicy

        // Now that we have our minimum policy, we reinitialize
        // the cache accordingly
        initializeCache(minimumPolicy);

// In order to get our estimated next feature matrix, we need to use
// our model generated in C-miner or using standard Markhov chains
// to estimate what the next set of features will be. Basically we'll
// store a bunch of phases (like we're doing now) and those phases
// will have ideal feature matrix footprints (which is essentially what
// we're doing now, except with different features).
function getEstimatedNextFeatureMatrix(phaseModel, recentHistory):
        return phaseModel.getNextFeatureMatrix(recentHistory)

// This is the main training algorithm.
function trainModel:
        stateHistory = []
        while workloadRunning:
                // Here I choose 1000 as a reasonable number of phases to
                // run, but this is tunable.
                for i in xrange(1000):
                        // We wait for our current phase to be available
                        wait(cacheState.newPhaseInterrupt())

                        // Get our current feature matrix
                        currentFeatureMatrix = \
                                calculateCurrentFeatureMatrix(this.cacheState)

                        // keep a history of up to 1000 states
                        // Note: this is not all the data; this is just the
                        // data from the current feature matrix.
                        // This will help us when we're remodelling stuff later.
                        stateHistory = [stateHistory, currentFeatureMatrix]

                        // Based on the model loadIQ created during its last run,
                        // we sort this into a phase. This is analogous to what
                        // we were doing with the histogram, except instead of
                        // a histogram, we're dealing with a broader set of
                        // featurs.
                        currentPhase = \
                                phaseModel.getCurrentPhase(currentFeatureMatrix);

                        // Now, using the prediciton model, we guess about the
                        // next phase. This could be a call to C-Miner or
                        // Markhov chains or something
                        nextPhaseGuess = \
                                phasePredicitonModel.getNextPhaseGuess(currentFeatureMatrix);

                        // Now that we have an idea of what our next phase
                        // will be, we can figure out a good set of policies
                        // using the algorithm described in the function makeGuess.
                        makeGuess(currentFeatureMatrix, nextPhaseGuess.featureMatrix)

                // This should give us a model of "phases" so that during our next
                // round, we can start evaluating things based on what phase they
                // should belong to. Thinking about it now, this probably shouldn't
                // be loadIQ. This should be some other, independent algorithm. Take
                // careful note of the fact that we're feeding back our feature
                // weightings into this algorithm. We do this because we want to
                // refine our definitons of the states so that they are more useful
                // when we're trying to decide on policies.
                phaseModel = loadIQ(stateHistory, featureWeightings)

                // Because we've recreated our phase model, we need to resort our phases
                // based on their feature matrices. (This is the reason we stored them
                // away before.) This means that all the labels change. Because all the
                // labels have changed, we need to recreate our predictive model. This
                // resorts all the phases and adjusts our predictive model accordingly.
                phasePredictionmodel = cminer(stateHistory, phaseModel)
```